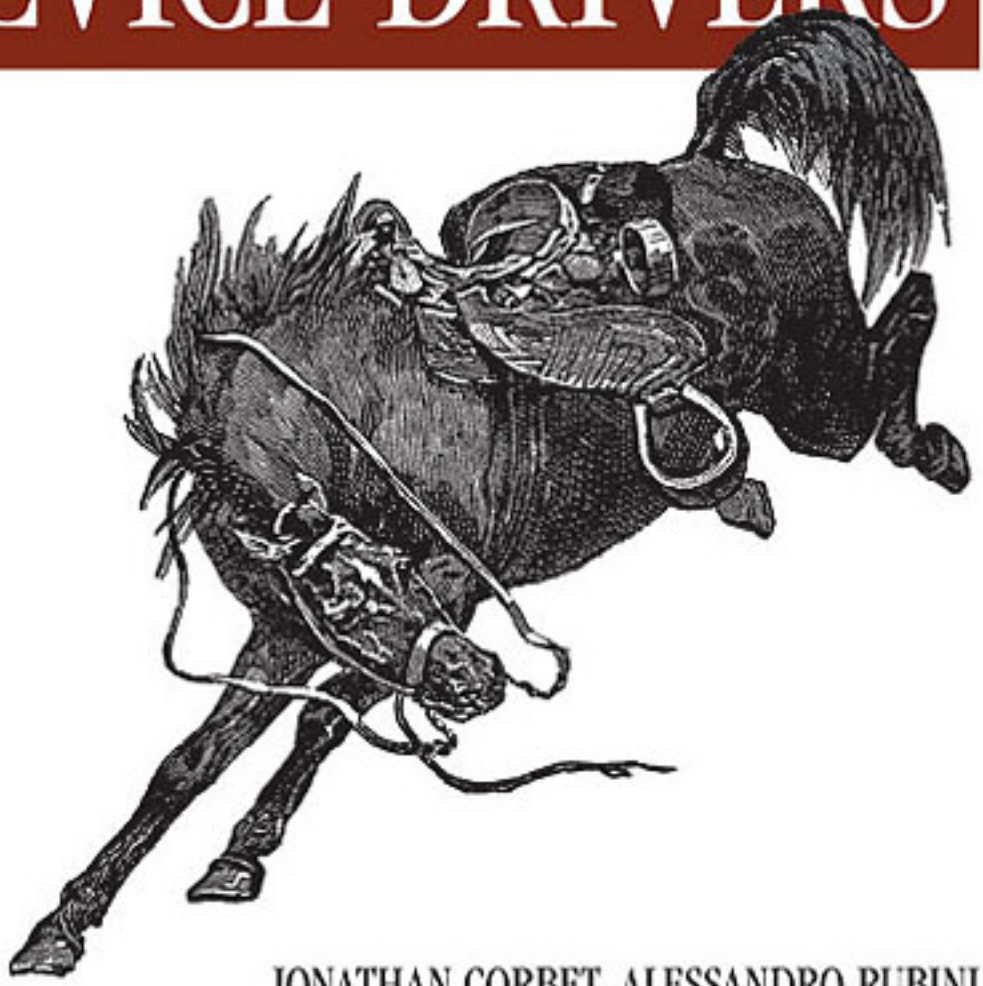
WHERE THE KERNEL MEETS THE HARDWARE

3rd Edition

LINUX DEVICE DRIVERS



JONATHAN CORBET, ALESSANDRO RUBINI
& GREG KROAH-HARTMAN

O'REILLY®

Перевод глав

Драйверы Устройств Linux, 3-я редакция

V*D*V

Оглавление

Драйверы Устройств Linux, Третья Редакция

http://lwn.net/Kernel/LDD3/
http://oreilly.com/catalog/linuxdrive3/errata/
ftp://ftp.ora.com/pub/examples/linux/drivers/
ftp://ar.linux.it/pub/ldd3/
https://github.com/martinezjavier/ldd3/

Linux Device Drivers, Third Edition
by Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman

Опубликовано O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Книги O'Reilly можно приобрести для образования, бизнеса или продажи в рекламных целях. Для большинства книг также доступны Интернет издания (safari.oreilly.com). Для получения дополнительной информации свяжитесь с нашим корпоративным/институциональным отделом продаж: (800) 998-9938 или corporate@oreilly.com.

Редактор: Andy Oram
Производственный редактор: Matt Hutchinson
Услуги по производству: Octal Publishing, Inc.
Разработка обложки: Edie Freedman
Разработка внутреннего оформления: Melanie Wang

История печати:
Февраль 1998: Первая редакция.
Июнь 2001: Вторая редакция.
Февраль 2005: Третья редакция.

ISBN: 0-596-00590-3

Глава 1, Введение в драйверы устройств

Одно из главных преимуществ свободных операционных систем, таких как Linux, это то, что их внутренности открыты для просмотра всем. Эти операционные системы, когда-то тёмная и мистическая область, чей код был доступен только небольшому числу программистов, могут быть теперь легко изучены, поняты и модифицированы кем угодно, если он обладает необходимым уровнем знаний. Linux помог демократизировать операционные системы. Ядро Linux, тем не менее, представляет собой большой и сложный набор кода, и потенциальные исследователи ядра нуждаются в точке входа, где они могут обращаться к этому коду не будучи подавленными его сложностью. Часто такую точку входа создают драйверы устройств.

Драйверам устройств отводится особая роль в ядре Linux. Это прекрасные “чёрные ящики”, которые заставляют специфичную часть оборудования соответствовать строго заданному программному интерфейсу; они полностью скрывают детали того, как работает устройство. Действия пользователя сводятся к выполнению стандартизированных вызовов, которые не зависят от специфики драйвера; перевод этих вызовов в специфичные для данного устройства операции, которые исполняются реальным оборудованием, является задачей драйвера устройства. Этот программный интерфейс таков, что драйверы могут быть собраны отдельно от остальной части ядра и подключены в процессе работы, когда это необходимо. Такая модульность делает драйверы Linux простыми для написания, так что теперь доступны сотни драйверов.

Есть много причин быть заинтересованным в написании драйверов устройств для Linux. Количество нового оборудования, которое становится доступным (и устаревает!) гарантирует, что создатели драйверов будут заняты в обозримом будущем. Отдельные люди могут нуждаться в знаниях о драйверах, чтобы получить необходимый уровень доступа к устройству, которое представляет для них интерес. Разработчики оборудования, создавая драйвер для Linux для своей продукции, могут добавить большое и растущее количество пользователей Linux в качестве потенциальных покупателей. А открытая природа системы Linux означает, что если автор драйвера пожелает, исходник драйвера может быть быстро распространён среди миллионов пользователей.

Эта книга научит вас, как писать ваши собственные драйверы и как исследовать необходимые области ядра. Мы избрали независимый от устройства подход; предлагаемая техника программирования и интерфейсы представлены, где возможно, без привязки к какому-либо конкретному устройству. Каждый драйвер уникален; как разработчику драйвера, вам

необходимо хорошо знать работу своего устройства. Но большинство принципов и основных техник одинаковы для всех драйверов. Эта книга не может научить вас работать с вашим устройством, но она даст ту необходимую информацию, на основе которой вы сможете сделать ваше устройство рабочим. Поскольку вы учитесь писать драйверы, вы узнаете много вообще о ядре Linux; это может помочь понять вам, как работает ваш компьютер и почему всё происходит не так быстро, как вы ожидаете, или выполняется не совсем так, как вы хотите. Мы вводим новые идеи постепенно, начиная с очень простых драйверов и основываясь на них; каждое новое понятие сопровождается примером кода, который не нуждается в каком-то специальном оборудовании для тестирования.

В этой главе нет ничего по написанию кода. Тем не менее, мы вводим некоторые основные концепции ядра Linux, и позже, когда мы начнём программирование, вы будете рады, что познакомились с ними.

Роль драйвера устройства

Как программист, вы в состоянии делать свои выборы в своём драйвере и сделать приемлемый выбор между требуемым временем на программирование и гибкостью результата. Хотя может показаться странным, что мы называем драйвер "гибким", нам нравится это слово, потому что это подчёркивает, что роль драйвера устройства в обеспечении ***механизма***, а не ***политики*** (создании жёстких правил).

Различие между механизмом и политикой - одна из лучших идей, стоящих за проектом Unix. Большинство проблем программирования в действительности может быть разделено на две части: "какие возможности будут обеспечены" (это механизм) и "как эти возможности могут быть использованы" (это политика, правила). Если две проблемы адресованы разным частям программы, или даже разным программам в целом, программный пакет много легче разработать и адаптировать к специфическим требованиям.

Например, управление в Unix графическим дисплеем разделено между X-сервером, который знает оборудование и предлагает унифицированный интерфейс для пользовательских программ, менеджерами окна и сессий, которые осуществляют индивидуальную политику, не зная что-либо об оборудовании. Люди могут использовать тот же оконный менеджер на разном оборудовании и разные пользователи могут запускать разные конфигурации на той же рабочей станции. Даже полностью различные настольные среды, такие как **KDE** и **GNOME**, могут сосуществовать в одной системе. Другим примером является многоуровневая сетевая структура TCP/IP: эта операционная система предлагает абстракцию сокета, которая не осуществляет политики в отношении передаваемых данных, в то время как разные серверы управляют сервисами (и связанными с ними политиками). Более того, сервера, наподобие ***ftpd***, обеспечивают механизм передачи файлов, а пользователи могут использовать любого клиента, которого пожелают; существуют и клиенты, управляемые через командную строчку и через графический интерфейс, и кто угодно может написать новый пользовательский интерфейс для передачи файлов.

Применительно к драйверам используется то же самое разделение механизма и политики. Драйвер дисковода свободен от правил - его задача только показать дискету как непрерывный массив блоков данных. Более высокие уровни системы обеспечивают правила, такие как, кто может иметь доступ к дисководу, можно ли обращаться к нему напрямую или только через файловую систему, могут ли пользователи монтировать файловую систему дисковода. Так как различное окружение обычно нуждается в использовании оборудования разными способами, важно быть по возможности свободными от правил.

При написании драйверов программист должен уделить внимание фундаментальной концепции: написать код ядра для доступа к оборудованию, но не оказать давление частными правилами на пользователя, так как разные пользователи имеют разные потребности. Ваш драйвер должен обеспечивать доступ к оборудованию, оставляя задачи ***как*** использовать оборудование приложениям. Таким образом, драйвер гибок, если обеспечивает доступ к оборудованию без ограничений. Иногда, тем не менее, некоторые ограничения должны иметь место. К примеру, драйвер ввода/вывода может предоставлять только побайтный доступ к оборудованию, чтобы избежать написания дополнительного кода, необходимого для передачи отдельных битов.

Можно также рассматривать драйвер в другой перспективе: это программный слой, который находится между приложениями и реальным устройством. Эта привилегированная роль драйвера позволяет программисту драйвера точно выбрать, как устройство должно быть представлено: разные драйверы могут предлагать разные возможности даже для одного и того же устройства. Фактически, драйвер должен быть разработан так, чтобы обеспечивать баланс между разными соображениями. Например, за использование одного устройства могут конкурировать одновременно несколько программ и автор драйвера имеет полную свободу решать, как обслуживать конкурирующие запросы. Вы могли бы сделать распределение памяти на устройстве независимым от возможностей оборудования или могли бы предложить пользовательскую библиотеку, чтобы помочь программистам, пишущим приложения, создать новые правила поверх предложенных простых и так далее. Одно главное соображение - необходимо выбрать между желанием предоставить пользователю максимум возможностей и временем, которое вы должны затратить на написание драйвера, а также необходимостью сохранить код максимально простым, чтобы туда на вкрались ошибки.

"Гибкие" драйверы имеют типичные характеристики. Они включают поддержку синхронных и асинхронных операций, возможность быть открытыми множество раз, возможность максимально полного использования оборудования и отсутствие лишних программных уровней, чтобы оставаться простыми и свободными от ограничивающих операций. Драйверы такого сорта не только работают лучше у конечных пользователей, но также оказываются проще в написании и сопровождении. Быть свободными от ограничений - общая цель для разработчиков программного обеспечения.

Более того, многие драйверы устройств поставляются вместе с пользовательскими программами, чтобы помочь с конфигурированием и доступом к целевому устройству. Такие программы могут быть и простыми утилитами и графическими приложениями. В качестве примера можно привести программу ***tunelp***, которая регулирует работу драйвера параллельного порта, и графическую утилиту ***cardctl***, входящую в состав пакета PCMCIA драйвера. Часто предоставляются также клиентские библиотеки, которые обеспечивают возможности, которые нет необходимости иметь как часть самого драйвера.

Областью этой книги является ядро, таким образом мы пытаемся не иметь дело с проблемами политик (ограничений) или с пользовательскими программами или вспомогательными библиотеками. Иногда мы говорим о различных ограничениях и как реализовать их поддержку, но мы не будем погружаться в детали о программах, использующих устройство, или ограничений, которые они налагают. Вы должны понимать, однако, что пользовательские программы являются неотъемлемой частью пакета программного обеспечения и что даже гибкие пакеты распространяются с файлами конфигурации, которые определяют поведение по умолчанию для нижележащих механизмов.

Строение ядра Linux

В системе Unix несколько параллельных ***процессов*** обслуживают разные задачи. Каждый процесс запрашивает системные ресурсы, будь то энергия, память, сетевое подключение, или какие-то другие ресурсы. ***Ядро*** - это большой кусок исполняемого кода, отвечающего за обработку всех таких запросов. Хотя границы между разными задачами ядра не всегда ясно определены, роль ядра может быть разделена (как показано на Рисунке 1-1) на следующие части:

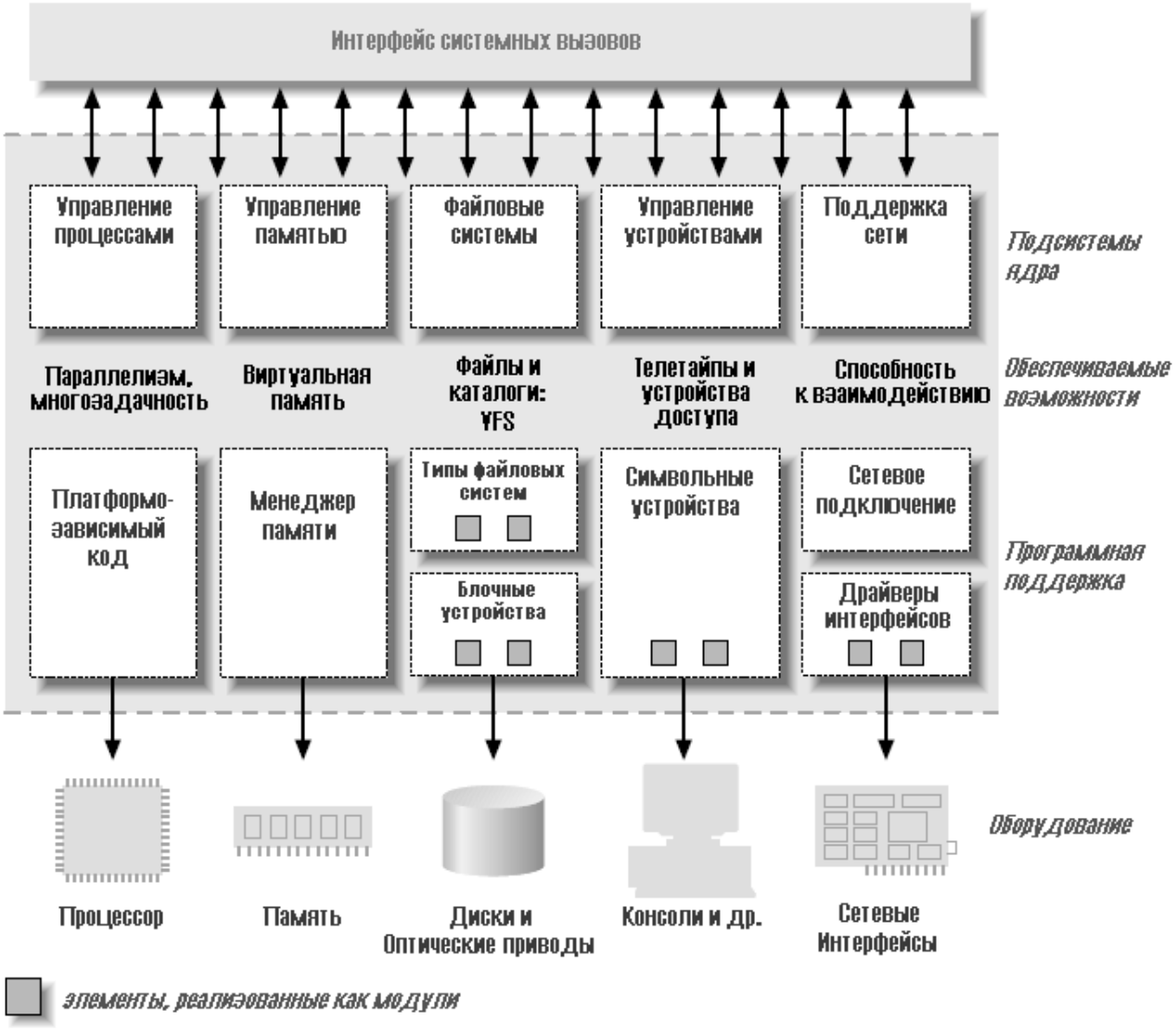


Рисунок 1-1. Строение ядра Linux

Управление процессами

Ядро отвечает за создание и уничтожение процессов и обеспечение их взаимодействия с внешним миром (ввод и вывод). Взаимодействие между разными процессами (через сигналы, каналы или примитивов межпроцессных взаимодействий) является основой общей функциональности системы и также возложена на ядро. Дополнительно, планировщик, который распределяет время процессора, тоже является частью системы управления процессами. В общих словах, деятельность процессов управления ядра

создаёт абстракцию нескольких процессов поверх одного или нескольких процессоров.

Управление памятью

Память компьютера - главный ресурс и способ управления ей особенно важен для производительности системы. Ядро создаёт виртуальное адресное пространство для каждого процесса поверх имеющихся ограниченных ресурсов. Разные части ядра взаимодействуют с подсистемой управления памятью через набор функциональных вызовов, начиная от простой пары ***malloc***/***free*** до много более развитой функциональности.

Файловые системы

Unix очень сильно связана с концепцией файловой системы; почти всё в Unix может быть обработано как файл. Ядро строит структурированную файловую систему поверх неструктурированного оборудования и полученная файловая абстракция интенсивно используется всей системой. В дополнение Linux поддерживает множество типов файловых систем, то есть различные способы организации данных на физическом носителе. К примеру, диски могут быть отформатированы в стандартной для Linux файловой системе ext3, часто используемой файловой системе FAT или некоторых других.

Управление устройствами

Почти каждая системная операция в конечном счёте связывается с физическим устройством. За исключением процессора, памяти и очень немногих других объектов, каждая операция управления устройством выполняются кодом, специфичным для данного адресуемого устройства. Этот код называется ***драйвером устройства***. Ядро должно иметь встроенный драйвер устройства для каждой периферии, существующей в системе, от жёсткого диска до клавиатуры и ленточного накопителя. Этот аспект функциональности ядра и является нашим основным интересом в этой книги.

Сетевое подключение

Сетевое подключение должно управляться операционной системой, потому что большинство сетевых операций не зависят от процессов: входящие пакеты - это асинхронные события. Эти пакеты должны быть собраны, распознаны и распределены перед тем, как они будут переданы другому процессу для обработки. Система отвечает за доставку пакетов данных между программой и сетевыми интерфейсами и должна управлять исполнением программ в зависимости от их сетевой активности. Дополнительно в ядро встроены все задачи маршрутизации и разрешение адресов.

Загружаемые модули

Одной из хороших особенностей Linux является способность расширения функциональности ядра во время работы. Это означает, что вы можете добавить функциональность в ядро (и убрать её), когда система запущена и работает.

Часть кода, которая может быть добавлена в ядро во время работы, называется ***модулем***. Ядро Linux предлагает поддержку довольно большого числа типов (или классов) модулей, включая, но не ограничиваясь, драйверами устройств. Каждый модуль является подготовленным объектным кодом (не слинкованным для самостоятельной работы), который может быть динамически подключен в работающее ядро программой ***insmod*** и отключен программой ***rmmod***.

Рисунок 1-1 определяет разные классы модулей, отвечающих за специфические задачи - как говорят, модуль принадлежит к определённому классу в зависимости от предлагаемой функциональности. Картина из модулей, показанная на Рисунке 1-1, охватывает все наиболее важные классы, но далеко не полная, потому что много большая функциональность Linux модулизирована.

Классы устройств и модулей

Способ видения устройств в Linux разделяется на три фундаментальных типа. Каждый модуль обычно реализован как один из этих типов и таким образом классифицируется как ***символьный модуль***, ***блочный модуль***, или ***сетевой модуль***. Такое разделение модулей на разные типы или классы не является жёстким; программист может при желании создавать большие модули, содержащие разные драйверы в одном куске кода. Хорошие программисты, тем не менее, обычно создают разные модули для каждой новой функциональности, потому что декомпозиция является ключом к масштабируемости и расширяемости.

Этими тремя классами являются:

Символьные устройства

Символьное устройство - это такое устройств, к которому можно обращаться как к потоку байтов (так же как к файлу); драйвер символьного устройства отвечает за реализацию такого поведения. Такой драйвер обычно, по крайней мере, поддерживает системные вызовы ***open***, ***close***, ***read*** и ***write***. Текстовый экран (***/dev/console***) и последовательные порты (***/dev/ttyS0*** и подобные) являются примерами символьных устройств, так как они хорошо представлены абстракцией потока. Для обращения к символьным устройствам используют узлы (node) файловой системы, такие как ***/dev/tty1*** и ***/dev/lp0***. Единственное важное отличие между символьными устройствами и обычными файлами - вы всегда можете двигаться вперед и назад в обычном файле, в то время как большинство символьных устройств - это только каналы данных, к которым вы можете обращаться только последовательно. Существуют, однако, символьные устройства, которые выглядят как области данных, и вы можете двигаться по ним назад и вперёд; к примеру, это обычно используется в грабберах экрана, где приложения могут получать доступ ко всему полученному изображению используя ***mmap*** или ***lseek***.

Блочные устройства

Так же как символьные устройства, блочные устройства доступны через узлы файловой системы в директории ***/dev***. Блочное устройство - это устройство (например, диск) который может содержать файловую систему. В большинстве систем Unix блочное устройство может поддерживать только операции ввода-вывода, которые передают один или более целых блоков, обычно равных 512 байт (или большей степени числа два). Linux, однако, разрешает приложению читать и писать в блочное устройство, так же как и в символьное устройство - это позволяет передавать любое число байт за раз. В результате, блочные и символьные устройства отличаются только способом управления данными внутри ядра и, соответственно, программным интерфейсом в ядре/драйвере. Как и символьное устройство, каждое блочное устройство доступно через узел файловой системы, так что различия между ними не видны пользователю. Блочные драйверы имеют интерфейс для ядра полностью отличный от символьных устройств.

Сетевые интерфейсы

Любой сетевой обмен данными делается через интерфейс, то есть устройство, которое в

состоянии обменяться данными с другими узлами сети. Обычно, ***интерфейс*** - это аппаратное устройство, но также он может быть чисто программным устройством, наподобие интерфейса loopback (локальное петлевое устройство). Сетевой интерфейс отвечает за отсылку и приём пакетов данных, управляемых подсистемой сети в ядре, без знания кому предназначены передаваемые пакеты.
Многие сетевые соединения (особенно использующие TCP) являются поточно-ориентированными, но сетевые устройства обычно разработаны для передачи и приёма пакетов. Сетевой драйвер ничего не знает об отдельных соединениях; он только обрабатывает пакеты. Не будучи поточно-ориентированным устройством, сетевой интерфейс нелегко представить как узел в файловой системе наподобие ***/dev/tty1***. Unix всё же обеспечивает доступ к интерфейсам через назначение им уникальных имён (таких как **eth0**), но это имя не имеет соответствующего элемента в файловой системе. Обмен между ядром и сетевым устройством сильно отличается от используемого в символьных и блочных драйверах. Вместо ***read*** и ***write*** ядро вызывает функции, относящиеся к передаче пакетов.

Есть другие пути классификации модулей драйверов, которые по-другому подразделяют устройства. Вообще, некоторые типы драйверов работают с дополнительными наборами функций ядра для данного типа устройств. К примеру, можно говорить о модулях универсальной последовательной шины (USB), последовательных модулях, модулях SCSI, и так далее. Каждое USB устройство управляется модулем USB, который работает с подсистемой USB, но само устройство представлено в системе или как символьное устройство (последовательный порт USB, к примеру), или как блочное устройство (USB устройство чтения карт памяти), или как сетевой интерфейс (например, сетевой USB интерфейс).

В последнее время в ядро были добавлены другие классы драйверов устройств, включающие драйверы FireWire и I2C. Таким же образом, как они добавили поддержку драйверов USB и SCSI, разработчики ядра собрали особенности всего класса и передали их разработчикам драйверов, чтобы избежать двойной работы и ошибок, упростив и стабилизировав таким образом процесс написания этих драйверов.

В дополнение к драйверам устройств в ядре в виде модулей реализованы и другие функциональные возможности, включающие и аппаратные средства и программное обеспечение. Общий пример - файловые системы. Тип файловой системы определяет, как организована информация на блочном устройстве, чтобы показать дерево файлов и директорий. Это не драйвер устройства, здесь нет какого-либо устройства, связанного со способом размещения информации; вместо этого, тип файловой системы - это программный драйвер, потому что он отображает структуры данных нижнего уровня на структуры данных верхнего уровня. Он и является файловой системой, которая определяет, какой длины может быть имя файла и какая информация о каждом файле хранится в записи каталога. Модуль файловой системы должен осуществить самый низкий уровень системных вызовов, которые обращаются к каталогам и файлам, отображая имена файла и пути (так же как другую информацию, такую как режимы доступа) к структурам данных, сохранённым в блоках данных. Такой интерфейс полностью независим от фактической передачи данных на и от диска (или другого носителя), что достигнуто с помощью драйвера блочного устройства.

Если вы подумаете о том, как сильно система Unix зависит от нижележащей файловой системы, то вы поймете, что такое программное понятие жизненно важно для функционирования системы. Способность декодировать информацию файловой системы остаётся на самом низком уровне иерархии ядра и имеет предельно важное значение; даже если вы напишете блочный драйвер для своего нового CD-ROM, это будет бесполезно, если

вы не в состоянии выполнить команды ***ls*** или ***cp*** для данных этого устройства. Linux поддерживает понятие модуля файловой системы, программный интерфейс которого декларирует различные операции, которые могут быть выполнены с индексом файловой системы (inode), каталогом, файлом и суперблоком. Вряд ли в действительности программисту потребуется написать модуль файловой системы, потому что официальное ядро уже включает код для самых важных типов файловых систем.

Вопросы безопасности

Безопасность - всё более и более важная проблема в наше время. Мы обсудим связанные с безопасностью проблемы, поскольку они встречаются в книге повсюду. Однако, есть несколько общих понятий, которые заслуживают внимания сейчас. Любая проверка безопасности в системе выполняется кодом ядра. Если у ядра есть бреши в защите, то и у системы в целом есть бреши. В официально распространяемом ядре только авторизованный пользователь может загрузить модуль в ядро; системный вызов ***init_module*** проверяет, разрешено ли вызывающему процессу загрузить модуль в ядро. Таким образом, когда работает официальное ядро, только суперпользователь (* Технически, только кто-то с разрешением CAP_SYS_MODULE может выполнить эту операцию. Мы обсуждаем разрешения в Главе 6 128.), или злоумышленник, который смог получить эту привилегию, может использовать мощность привилегированного кода. Когда возможно, авторы драйверов должны избегать реализации политики безопасности в своем коде. Безопасность - результат ограничений, которые часто лучше всего обрабатываются на более высоких уровнях ядра, под управлением системного администратора. Однако, всегда есть исключения.

Как автор драйвера устройства, вы должны знать о ситуациях, в которых некоторые способы доступа к устройству могли бы неблагоприятно затронуть систему в целом и должны обеспечить адекватный контроль. Например, операции устройства, которые затрагивают глобальные ресурсы (такие как установка линии прерывания), которые могут повредить аппаратные средства (загрузку встроенного программного обеспечения, например), или могли бы затронуть других пользователей (таких как установка заданного по умолчанию размера блока на ленточном накопителе), обычно доступны только для достаточно привилегированных пользователей, и эта проверка должна быть осуществлена в драйвере непосредственно.

Конечно, авторы драйверов должны также быть внимательными, чтобы избежать внедрения ошибок безопасности. Язык программирования Си позволяет легко делать некоторые типы ошибок. Много текущих проблем безопасности созданы, например, ошибками ***переполнения буфера***, когда программист забывает проверять, сколько данных записано в буфер, и данные продолжают записываться после окончания буфера, поверх совершенно других данных. Такие ошибки могут поставить под угрозу всю систему и их надо избегать. К счастью, обычно относительно просто избежать этих ошибок в контексте драйвера устройства, в котором интерфейс для пользователя чётко определен и строго контролируем.

Стоит иметь в виду и некоторые другие общие идеи безопасности. Любые данные, полученные от пользовательских процессов, должны быть обработаны с большим подозрением; никогда не доверяйте им, пока они не проверены. Будьте внимательны с неинициализированной памятью; любая память, полученная от ядра, должна быть обнулена или проинициализирована другим способом прежде, чем стать доступной пользовательскому процессу или устройству. Иначе, результатом может быть утечка информации (раскрытие данных, паролей и так далее). Если ваше устройство обрабатывает данные, посланные в него, убедитесь, что пользователь не может послать ничего, что могло бы поставить под угрозу систему. Наконец, думайте о возможном эффекте операций устройства; если есть определённые операции (например, перезагрузка встроенного программного обеспечения на

плате адаптера или форматирование диска), которые могли бы затронуть систему, эти операции должны почти наверняка быть разрешены только привилегированным пользователям.

Будьте внимательны также, получая программное обеспечение от третьих лиц, особенно когда затрагивается ядро: потому что любой имеет доступ к исходному тексту, любой может сломать и перекомпилировать части. Хотя вы можете обычно доверять предварительно откомпилированным ядрам в ваших дистрибутивах, вы должны избегать запуска ядра, откомпилированного незнакомыми людьми - если вы избегаете запуска предварительно откомпилированного ядра как root, тогда вам лучше не запускать откомпилированное ядро. Например, злонамеренно изменённое ядро могло бы позволить любому загружать модуль, открывая таким образом неожиданную лазейку через ***init_module***. Заметьте, что ядро Linux может быть откомпилировано так, чтобы вообще не поддерживать модули, закрывая таким образом любые связанные с модулем бреши в защите. Конечно, в этом случае все необходимые драйверы должны быть встроены непосредственно в само ядро. Отключение загрузки модулей ядра после начальной загрузки системы через соответствующий механизм стало возможно для версий, начиная с 2.2.

Нумерация версий

Прежде, чем углубиться в программирование, мы должны прокомментировать схему нумерации версий, используемую в Linux, и какие версии охвачены этой книгой. Прежде всего, отметьте, что у каждого пакета программ, используемого в системе Linux, есть свой собственный номер выпуска и часто они взаимозависимы: вы нуждаетесь в определённой версии одного пакета, чтобы запустить определённую версию другого пакета. Создатели дистрибутивов Linux обычно учитывают проблему совместимости пакетов и пользователь, который устанавливает подготовленный дистрибутив, не сталкивается с этой проблемой. С другой стороны, те, кто заменяет и модернизирует системное программное обеспечение сами решают эту проблему. К счастью, почти все современные дистрибутивы поддерживают обновление отдельных пакетов, проверяя межпакетные зависимости; менеджер дистрибутивных пакетов вообще не позволит обновиться, пока зависимости не удовлетворены.

Для запуска примеров, которые мы приводим во время обсуждения, вы не будете нуждаться в особых версиях какого-то пакета помимо того, что требуется ядро версии 2.6; для запуска наших примеров может использоваться любой современный дистрибутив Linux. Мы не будем детализировать определённые требования, потому что файл ***Documentation/Changes*** в ваших исходниках ядра - лучший источник такой информации, если вы испытываете какие-то проблемы. Возвращаясь к теме ядра, чётно пронумерованные версии ядра (то есть, 2.6.x) являются устойчивыми, которые предназначены для общего распространения. Нечётные версии (такие как 2.7.x), напротив, являются рабочими копиями и весьма неустойчивы; последняя из них представляет текущее состояние разработки, но становится устаревшей через несколько дней или около этого.

Эта книга охватывает версию ядра 2.6. Мы постарались показать все возможности, доступные для авторов драйверов устройств в версии 2.6.10, текущей версии во время написания книги. Этот выпуск книги не охватывает предыдущие версии ядра. Для тех, кому это интересно, вторая редакция книги подробно охватывала версии от 2.0 до 2.4. Та редакция всё ещё доступна на http://lwn.net/Kernel/LDD2/. Программисты ядра должны знать, что процесс разработки изменился с версии 2.6. Ядра серии 2.6 теперь принимают изменения, которые ранее считали бы слишком большими для "устойчивого" ядра. Между прочим, это означает, что внутренние программные интерфейсы ядра могут измениться, потенциально делая, таким

образом, части этой книги устаревшими; по этой причине код примеров, сопровождающий текст, как известно, работает на версии 2.6.10, но некоторые модули не компилируют под более ранними версиями. Программистам, желающим не отставать от изменений в программировании ядра, стоит присоединиться к почтовым рассылкам и использовать веб-сайты, перечисленные в библиографии. Есть также интернет-страница, созданная на http://lwn.net/Articles/2.6-kernel-api/, которая содержит информацию об изменениях API, которые произошли после публикации этой книги.

Этот текст особенно не говорит о версиях ядра с нечётным номером. У обычных пользователей никогда нет причины использовать ядра, находящиеся в разработке. Однако, разработчики, экспериментирующие с новыми возможностями, стремятся запускать последние выпускаемые релизы. Они обычно обновляются до новой версии, чтобы получить исправления ошибок и новые реализованные возможности. Отметьте, однако, что нет никакой гарантии на экспериментальных ядрах (* Примечание, нет никакой гарантии также и на чётно пронумерованных ядрах, если вы не полагаетесь на коммерческого провайдера, который предоставляет его собственную гарантию), и никто не поможет вам, если есть проблемы из-за ошибки в нетекущем ядре с нечётным номером. Те, кто запускает версии с нечётным номером ядра, обычно достаточно квалифицированы, чтобы углубиться в код без потребности в учебнике, что является другой причиной, почему мы не говорим здесь о разработках ядра. Другая особенность Linux - это то, что это платформо-независимая операционная система, больше не только “клон Unix для клонов PC”: в настоящее время поддерживается примерно 20 архитектур. Эта книга платформо-независимая в максимально возможной степени и все примеры кода были проверены по крайней мере на платформах x86 и x86-64. Поскольку код был проверен и на 32-х разрядных и на 64-х разрядных процессорах, он должен компилироваться и работать на всех других платформах. Как вы могли бы ожидать, примеры кода, которые полагаются на специфические аппаратные средства, не работают на всех поддерживаемых платформах, но это всегда заявляется в исходном тексте.

Лицензионное соглашение

Linux лицензируется по версии 2 GNU General Public License (GPL), документа, разработанного для проекта GNU Фондом бесплатного программного обеспечения. GPL позволяет любому распространять и даже продавать продукт, покрытый GPL, пока получатель имеет доступ к исходнику и в состоянии реализовать те же самые права. Дополнительно, любой программный продукт, произошедший от продукта, покрытого GPL, если он вообще распространяется, должен быть выпущен под GPL.

Основная цель такой лицензии состоит в том, чтобы позволить рост знания, разрешая всем изменять программы по желанию; в то же самое время, люди, продающие программное обеспечение общественности, могут всё ещё делать свою работу. Несмотря на эту простую цель, есть бесконечное обсуждение GPL и её использования. Если вы хотите прочитать лицензию, вы можете найти её в нескольких местах в вашей системе, включая главный каталог дерева исходников ядра в файле ***COPYING***.

Продавцы часто спрашивают, могут ли они распространить модули ядра только в бинарной форме. Ответ на этот вопрос преднамеренно оставили неоднозначным. Дистрибьюция бинарных модулей, пока они придерживаются опубликованного интерфейса ядра, допускается пока. Но авторские права на ядро удерживаются многими разработчиками и не все они соглашаются, что модули ядра - вторичные продукты. Если вы или ваш работодатель желаете распространять модули ядра согласно небесплатной лицензии, вы действительно должны обсудить ситуацию со своим юридическим представителем. Пожалуйста отметьте также, что разработчики ядра не боятся поломать работу бинарных модулей между выпусками ядра,

даже если это середина стабильной серии ядер. Если это вообще возможно, и вы и ваши пользователи находятся в лучшем положении, если вы выпускаете свой модуль как бесплатное программное обеспечение.

Если вы хотите, чтобы ваш код вошёл в основную ветку ядра, или если ваш код требует исправлений в ядре, вы ***должны*** использовать GPL-совместимую лицензию, как только вы выпускаете код. Хотя личное использование ваших изменений не применяет GPL для вас, если вы распространяете свой код, вы должны включать исходный текст в дистрибутивы - людям, приобретающим ваш пакет, должно быть разрешено пересобрать бинарный файл при желании в будущем.

По отношению к этой книге, большая часть кода имеет свободное распространение и в исходных текстах и в бинарном виде, и ни мы, ни O'Reilly не сохраняем права на любые вторичные работы. Все программы доступны на ftp://ftp.ora.com/pub/examples/linux/drivers/ (примеры находятся на ftp://ar.linux.it/pub/ldd3/, обновление версий примеров для сборки на новых ядрах можно найти на https://github.com/martinezjavier/ldd3), а точные сроки действия лицензии заявлены в файле ***LICENSE*** в том же каталоге.

Присоединение к сообществу разработчиков ядра Linux

Поскольку вы начинаете писать модули для ядра Linux, вы становитесь частью большего семейства разработчиков. Внутри этого сообщества вы можете найти не только людей, занятых аналогичной работой, но также и группу чрезвычайно преданных инженеров, старающихся сделать Linux лучшей системой. Эти люди могут быть источником помощи, идей и критики тоже - они будут первыми людьми к которым вы, вероятно, обратитесь, когда будете искать тестеров для нового драйвера.

Центральный сборочный пункт для разработчиков ядра Linux - список почтовой рассылки ***linux-kernel***. Все главные разработчики ядра, начиная от Линуса Торволдса, подписаны на эту рассылку. Предупреждаем, однако, что эта рассылка не для слабонервных: трафик может достигать 200 сообщений в день или больше. Тем не менее, чтение этой рассылки полезно для тех, кто интересуется разработкой ядра; она также может быть высококачественным источником для тех, кто нуждается в помощи разработчиков ядра.

Чтобы присоединиться к рассылки linux-kernel, выполните инструкции на странице вопросов и ответов рассылки http://www.tux.org/lkml.Прочитайте и остальные вопросы и ответы, ведь вы собираетесь заниматься этим; там есть много полезной информации. Разработчики ядра Linux - занятые люди и они намного более склонны помочь людям, которые сначала полностью сделали свою домашнюю работу.

Обзор книги

Отсюда мы входим в мир программирования ядра. Глава 2 14 знакомит с модулями, объясняя тайны искусства и показывая код работающих модулей. Глава 3 39 рассказывает о символьных драйверах и показывает законченный код драйвера устройства, работающего с памятью, из которого можно читать и писать для развлечения. Использование памяти вместо аппаратного устройства позволяет любому запускать код примера без необходимости приобретения специального оборудования.

Техники отладки - жизненно важные средства для программиста и описываются в Главе 4 69 . Одинаково важными для тех, кто будет исследовать современные ядра, являются способы управления исполнением и конкурентным доступом к ресурсам. Глава 5 101 интересуется проблемами, возникающими при параллельном доступе к ресурсам, и описывает механизмы

Linux для управления конкурентным доступом.

С отладкой и навыками управления конкурентных процессов вместе, мы двигаемся в дополнительные особенности символьных драйверов, такие как блокирующие операции, использование ***select*** и важный вызов ***ioctl***; эти темы - предмет Главы 6 128.

Прежде чем иметь дело с управлением оборудованием, мы анализируем ещё несколько программных интерфейсов ядра: Глава 7 174 показывает, как управляют временем в ядре, а Глава 8 203 объясняет распределение памяти.

Затем вы сфокусируемся на оборудовании. Глава 9 224 описывает управление портами ввода-вывода и буферами памяти, имеющимися на устройстве; после этого переходим к обработке прерываний в Главе 10 273. К сожалению, не каждый сможет запустить код примера этих глав, потому что ***необходимо иметь*** некоторое оборудование, чтобы протестировать программный интерфейс прерываний. Мы старались изо всех сил, чтобы свести к минимуму требования к оборудованию, но вам всё-таки необходимо такое простое оборудование, как стандартный параллельный порт, чтобы поработать с кодом примера для этих глав.

Глава 11 286 охватывает использование типов данных в ядре и написание переносимого кода.

Вторая половина книги посвящена более сложным темам. Мы начинаем углубляться в работу оборудования, в частности, функционирование специфичных шин для периферии. Глава 12 288 охватывает детали написания драйверов для устройств PCI, а Глава 13 344 рассматривает API для работы с устройствами USB.

Понимая работу периферийных шин, мы сможем бросить детальный взгляд на модель устройств в Linux, которая является уровнем абстракции, используемым ядром, чтобы описывать аппаратные и программные ресурсы, которыми оно управляет. Глава 14 392 - восходящий взгляд на инфраструктуру модели устройства, начинающуюся с типа **kobject** и построенную на нём. Она описывает интеграцию модели устройства с реальными аппаратными средствами; затем используем это знание, чтобы затронуть такие темы, как устройства, подключаемые без выключения системы (hot-plugged devices) и управление питанием.

В Главе 15 395 мы разбираемся в управлении памятью в Linux. Эта глава показывает, как отобразить память ядра в пространство пользователя (системный вызов ***mmap***), отображение пользовательской памяти в пространство ядра (с помощью ***get_user_pages***) и как отобразить любой вид памяти в пространство устройства (чтобы выполнять операции прямого доступа к памяти [DMA]).

Наше понимание памяти будет полезно в следующих двух главах, которые описывают другие главные классы драйверов. Глава 16 445 вводит блочные драйверы и показывает, как они отличаются от символьных драйверов, с которыми мы работали до этого. Затем Глава 17 478 рассказывает о написании сетевых драйверов. Мы финишируем разговором о последовательных драйверах (Глава 18 525) и библиографией.

Глава 2, Сборка и запуск модулей

Время для начала программирования почти пришло. Эта глава вводит все основные понятия о программировании ядра и о модулях. На этих немногих страницах мы соберём и запустим законченный (хотя и относительно бесполезный) модуль и рассмотрим некоторые части базового кода, используемого всеми модулями. Получение такого опыта является важной основой для создания любого вида модульного драйвера. Чтобы не вводить слишком много понятий сразу, эта глава говорит только о модулях, не относящихся к какому-то определённому классу устройства. Все элементы ядра (функции, переменные, заголовочные файлы и макроопределения), которые введены здесь, описаны в разделе справочной информации в конце главы.

Установка вашей тестовой системы

Начиная с этой главы, для демонстрации концепций программирования мы представляем примеры модулей. (Все эти примеры доступны на FTP сайте O'Reilly, адрес его можно найти в Главе 1.) Сборка, загрузка и модификация этих примеров является хорошим способом улучшить ваше понимание того, как драйверы работают и взаимодействуют с ядром. Модули примеров должны работать практически с любым ядром версии 2.6.x, в том числе и распространяемым поставщиками дистрибутивов. Тем не менее, мы рекомендуем вам получить "основное" ядро непосредственно с группы сайтов kernel.org и установить его на вашей системе. Ядра поставщиков могут быть сильно изменены внутри и отличаться от основных; иногда поставщики патчей могут изменять API ядра, видимый драйверами устройств. Если вы пишете драйвер, который должен работать на определённом дистрибутиве, вам наверняка придётся собрать и протестировать его с соответствующими ядрами. Но с целью изучения написания драйверов стандартное ядро лучше. Независимо от происхождения ядра, сборка модулей для 2.6.x требует, чтобы вы имели сконфигурированное и собранное дерево ядра на своей системе. Это требование является изменением относительно предыдущей версии ядра, когда текущего набора заголовочных файлов было достаточно. Модули в версии 2.6 связаны с объектными файлами в дереве исходных текстов ядра, в результате получается более надёжный загрузочный модуль, но это также требует, чтобы эти объектные файлы были доступны. Так что ваши первые шаги в деле - получить дерево исходных кодов ядра (либо из сети kernel.org или из пакета исходных кодов ядра вашего дистрибьютора), собрать новое ядро и установить его на вашей системе. По причинам, которые мы рассмотрим позже, жизнь, как правило, упрощается, если вы на самом деле

работаете на том же самом ядре, для которого собираете модули, хотя это и не обязательно.

Вы должны также подумать о том, где вы проводите с модулем экспериментирование, разработку и тестирование. Мы сделали всё от нас зависящее, чтобы сделать наши примеры модулей безопасными и корректными, но возможность ошибок всегда остаётся. Ошибки в коде ядра могут привести к прекращению процесса пользователя или, иногда, всей системы. Они обычно не создают более серьёзных проблем, таких, как повреждение диска. Тем не менее, желательно проводить ваши эксперименты с ядром на системе, которая не содержит данных, которые вы не можете позволить себе потерять, и которая не занята выполнением основных задач. Исследователи ядра обычно держат "жертвенную" систему для тестирования нового кода.

Итак, если вы ещё не имеете подходящей системы с настроенным и собранным деревом исходных текстов ядра на диске, сейчас самое время, чтобы сделать это. Мы подождём. После выполнения этой задачи вы будете готовы начать играть с модулями ядра.

Модуль Hello World

Многие книги по программированию начинаются с примера “hello world”, как способа создания простейшей программы. Эта книга посвящена созданию модулей ядра, а не программам; так что, нетерпеливый читатель, следующий код представляет собой завершённый модуль “hello world”:

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");

static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}

static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}

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

В этом модуле определены две функции, одна вызывается, когда модуль загружается в ядро (***hello_init***), другая, когда модуль удаляется (***hello_exit***). Строчки с ***module_init*** и ***module_exit*** используют специальные макросы ядра, чтобы задать роль этих двух функций. Другой специальный макрос (***MODULE_LICENSE***) использован, чтобы сказать ядру, что этот модуль имеет свободную лицензию; без такой декларации при загрузке модуля ядро выводит предупреждение.

Функция ***printk*** определена в ядре Linux и доступна модулям; она ведёт себя аналогично стандартной библиотечной функции языка Си ***printf***. Ядру необходима своя функция печати, поскольку оно работает само по себе, без помощи библиотек Си. Модуль может вызвать ***printk***, потому что после того, как ***insmod*** загрузила его, модуль связан с ядром и может

получить доступ к публичным символам ядра (функциям и переменным, это объясняется в следующем разделе). Определение **KERN_ALERT** - это приоритет сообщения. (* Приоритет - это просто строка, например, <1>, которая добавляется в начало строки форматирования *printk*. Обратите внимание на отсутствие запятой после KERN_ALERT; добавление запятой здесь - это частая и раздражающая опечатка (которая, к счастью, улавливается компилятором.))

Мы определили высокий приоритет в этом модуле, так как сообщения с приоритетом по умолчанию может быть не показано, это зависит от версии работающего ядра, версии демона ***klogd*** и ваших настроек. Вы можете проигнорировать сейчас этот вопрос; мы объясним это в Главе 4 69.

Вы можете протестировать модуль утилитами ***insmod*** и ***rmmod***, как показано ниже. Обратите внимание, что только суперпользователь может загружать и выгружать модуль.

```
% make
make[1]: Entering directory `/usr/src/linux-2.6.10'
 CC [M] /home/ldd3/src/misc-modules/hello.o
 Building modules, stage 2.
 MODPOST
 CC /home/ldd3/src/misc-modules/hello.mod.o
 LD [M] /home/ldd3/src/misc-modules/hello.ko
make[1]: Leaving directory `/usr/src/linux-2.6.10'
% su
root# insmod ./hello.ko
Hello, world
root# rmmod hello
Goodbye cruel world
root#
```

Пожалуйста, обратите внимание ещё раз, чтобы указанная выше последовательность команд заработала, вы должны иметь правильно сконфигурированное и собранное дерево ядра там же, где Makefile будет искать его (***/usr/src/linux-2.6.10*** в показанном примере). Мы углубимся в детали сборки модулей в разделе "Компиляция и загрузка" 21.

В зависимости от механизма, который используется в вашей системе для отображения сообщений, результат может быть другим. В частности, предыдущий снимок экрана был взят из текстовой консоли; если вы запускаете ***insmod*** и ***rmmod*** в эмуляторе терминала под управлением оконной системы, то вы не увидите ничего на экране. Сообщения отправятся в один из системных лог-файлов, таких как ***/var/log/messages*** (реальное имя файла разнится в дистрибутивах Linux). Механизм, используемый для доставки сообщений ядра описан в Главе 4 69.

Как вы можете видеть, написание модулей не так сложно, как можно было ожидать, по крайней мере, пока от модуля не требуется делать что-то полезное. Трудной частью является понимание работы устройства и как добиться максимальной производительности. Мы углубимся дальше в модуляризацию в этой главе и оставим вопросы специфики устройств следующим главам.

Отличия между модулями ядра и приложениями

Прежде чем идти дальше, следует подчеркнуть разнообразные отличия между модулями ядра и приложениями.

Хотя большинство малых и средних приложений выполняют от начала до конца одну задачу, каждый модуль ядра просто регистрирует себя для того, чтобы обслуживать в будущем запросы, и его функция инициализации немедленно прекращается. Иными словами, задача функции инициализации модуля заключается в подготовке функций модуля для последующего вызова; это как будто модуль сказал: "Вот я и вот что я могу делать". Функция выхода модуля (***hello_exit*** в примере) вызывается только непосредственно перед выгрузкой модуля. Она сообщает ядру: "Меня больше нет; не просите меня сделать что-нибудь ещё". Такой подход к программированию подобен программируемой обработке событий, но пока не все приложения управляются событиями, как модули ядра. Другое сильное отличие между событийно-управляемым приложением и кодом ядра в функции выхода: в то время как приложение, которое прекращает работу, может быть ленивым при высвобождении ресурсов или избегать очистки всего, функция выхода модуля должна тщательно отменить все изменения, сделанные функцией инициализации, или эти куски останутся вокруг до перезагрузки системы.

Кстати, возможность выгрузить модуль является одной из тех особенностей подхода модуляризации, которую вы больше всего оцените, потому что это помогает сократить время разработки; можно тестировать последовательные версии новых драйверов, не прибегая каждый раз к длительному циклу выключения/перезагрузки.

Как программист, вы знаете, что приложение может вызывать функции, которые не определены: стадия линковки разрешает (определяет) внешние ссылки, используя соответствующие библиотечные функции. ***printf*** является одной из таких вызываемых функций и определена в ***libc***. Модуль, с другой стороны, связан только с ядром и может вызывать только те функции, которые экспортированы ядром, нет библиотек для установления связи. Например, функция ***printk***, использованная ранее в ***hello.c***, является версией ***printf***, определённой в ядре и экспортированной для модулей. Она ведёт себя аналогично оригинальной функции с небольшими отличиями, главным из которых является отсутствие поддержки плавающей точки. Рисунок 2-1 показывает, как используются в модуле вызовы функций и указатели на функции, чтобы добавить ядру новую функциональность.

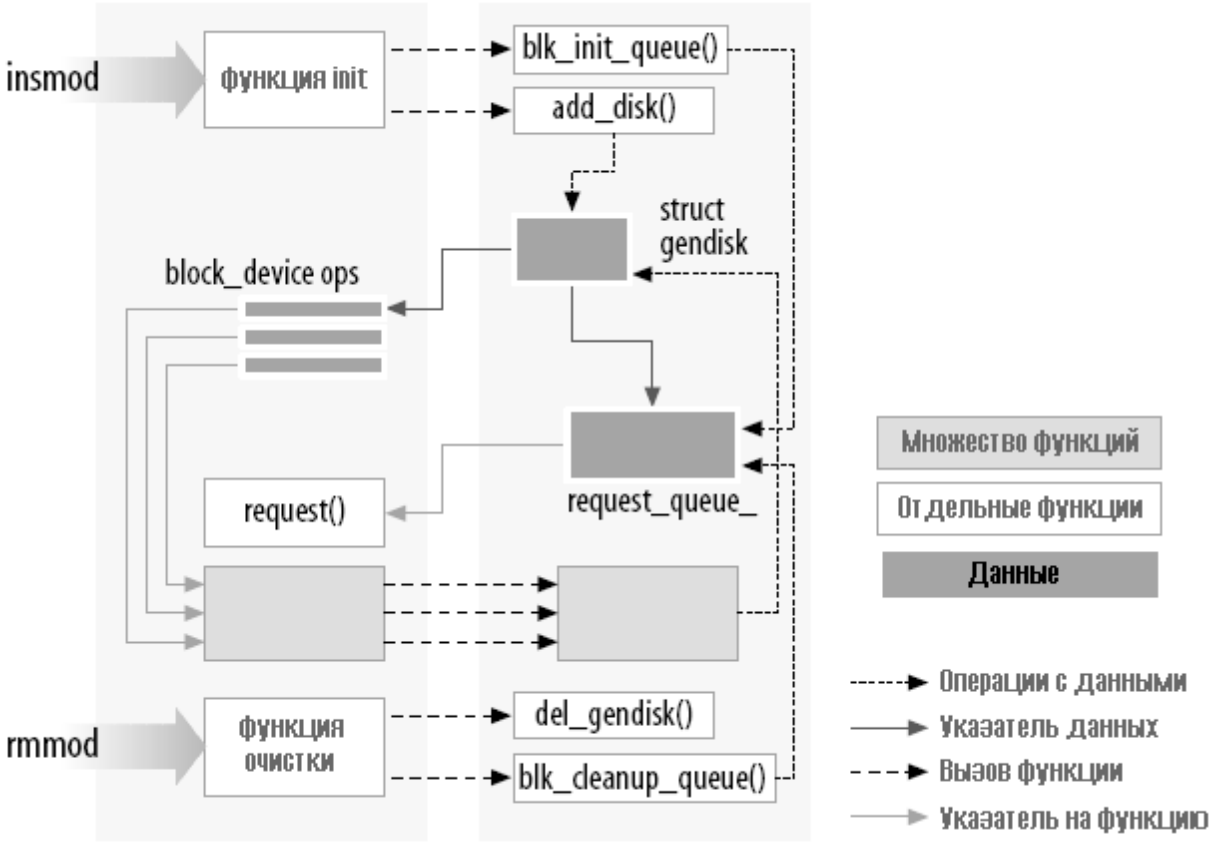


Рисунок 2-1. Связи модуля в ядре

Файлы исходников никогда не должны подключать обычные заголовочные файлы, потому что нет библиотеки, связанной с модулями, ***<stdarg.h>*** и очень специальные ситуации будут только исключениями. Только функции, которые фактически являются частью самого ядра, могут быть использованы в модулях ядра. Всё относящееся к ядру объявлено в заголовках, находящихся в дереве исходных текстов ядра, которое вы установили и настроили; наиболее часто используемые заголовки живут в ***include/linux*** и ***include/asm***, но есть и другие подкаталоги в папке ***include*** для содержания материалов, связанных со специфичными подсистемами ядра.

Роль каждого отдельного заголовка ядра объясняется в книге по мере того, как каждый из них становится необходим.

Ещё одно важное различие между программированием ядра и прикладным программированием в том, как каждое окружение обрабатывает ошибки: в то время, как ошибка сегментации является безвредной при разработке приложений и всегда можно использовать отладчик для поиска ошибки в исходнике, ошибка ядра убивает по крайней мере текущий процесс, если не всю систему. Мы покажем, как искать ошибки в ядре в Главе 4 69.

Пространство пользователя и пространство ядра

Модули работают в пространстве ядра, в то время как приложения работают в пользовательском пространстве. Это базовая концепция теории операционных систем.

На практике ролью операционной системы является обеспечение программ надёжным доступом к аппаратной части компьютера. Кроме того, операционная система должна обеспечивать независимую работу программ и защиту от несанкционированного доступа к ресурсам. Решение этих нетривиальных задач становится возможным, только если процессор обеспечивает защиту системного программного обеспечения от прикладных программ.

Каждый современный процессор позволяет реализовать такое поведение. Выбранный подход заключается в обеспечении разных режимов работы (или уровней) в самом центральном процессоре. Уровни играют разные роли и некоторые операции на более низких уровнях не допускаются; программный код может переключить один уровень на другой только ограниченным числом способов. Unix системы разработаны для использования этой аппаратной функции с помощью двух таких уровней. Все современные процессоры имеют не менее двух уровней защиты, а некоторые, например семейство x86, имеют больше уровней; когда существует несколько уровней, используются самый высокий и самый низкий уровни. Под Unix ядро выполняется на самом высоком уровне (также называемым режимом супервизора), где разрешено всё, а приложения выполняются на самом низком уровне (так называемом пользовательском режиме), в котором процессор регулирует прямой доступ к оборудованию и несанкционированный доступ к памяти.

Мы обычно упоминаем о режимах исполнения, как о пространстве ядра и пространстве пользователя. Эти термины включают в себя не только различные уровни привилегий, присущие двум режимам, но также тот факт, что каждый режим может также иметь своё собственное отображение памяти, своё собственное адресное пространство.

Unix выполняет переход из пользовательского пространства в пространство ядра, когда приложение делает системный вызов или приостанавливается аппаратным прерыванием. Код ядра, выполняя системный вызов, работает в контексте процесса - он действует от имени вызывающего процесса и в состоянии получить данные в адресном пространстве процесса. Код, который обрабатывает прерывания, с другой стороны, является асинхронным по отношению к процессам и не связан с каким-либо определённым процессом.

Ролью модуля является расширение функциональности ядра; код модулей выполняется в пространстве ядра. Обычно драйвер выполняет обе задачи, изложенные ранее: некоторые функции в модуле выполняются как часть системных вызовов, а некоторые из них отвечают за обработку прерываний.

Конкуренция в ядре

Направлением, в котором программирование ядра существенно отличается от обычного прикладного программирования является вопрос конкуренции. Большинство приложений, за исключением многопоточных приложений, как правило, работают последовательно, от начала до конца, без необходимости беспокоиться о том, что может произойти что-то ещё и изменить окружающую их среду. Код ядра не работает в таком простом мире и даже простейший модуль ядра должен быть написан с идеей, что одновременно могут происходить многие события.

Есть несколько источников конкуренции в программировании ядра. Естественно, система Linux запускает множество процессов и более чем один из них может пытаться использовать драйвер в то же самое время. Большинство устройств способны вызвать прерывание процессора; обработчики прерываний запускаются асинхронно и могут быть вызваны в то же время, когда ваш драйвер пытается сделать что-то другое. Несколько программных абстракций (например, таймеры ядра 187, описанные в Главе 7 174) тоже работают асинхронно. И конечно, Linux может работать на симметричных многопроцессорных (SMP) системах, в

результате чего ваш драйвер может выполняться одновременно на более чем одном процессоре. Наконец, в версии 2.6 код ядра был сделан вытесняемым; это изменение вызывает даже на однопроцессорных системах многие из тех же вопросов конкуренции, что и на многопроцессорных системах.

В результате, код ядра Linux, включая код драйвера, должен быть повторно-входимым - он должен быть способен работать в более чем одном контексте одновременно. Структуры данных должны быть тщательно продуманы, чтобы сохранять раздельным исполнение нескольких потоков, и код должен заботиться о том, чтобы получать доступ к общим данным таким образом, который предотвращает повреждение данных. Написание кода, который обрабатывает конкурентные запросы и избегает состояния гонки(ситуации, в которых неудачный порядок выполнения является причиной нежелательного поведения), требует размышлений и искусства. Правильное управление конкуренцией требует написания корректного кода ядра; по этой причине каждый драйвер примеров в этой книге был написан держа в уме конкуренцию. Используемые техники объясняются, когда мы доходим до них; Глава 5 101 также посвящена этой проблеме и примитивам ядра для управления конкуренцией.

Общей ошибкой, совершаемой программистами драйверов, является предположение, что конкуренция не является проблемой, пока определённый сегмент кода не отправляется спать (или "блокирован"). Даже в предыдущих ядрах (которые были не вытесняющими), это предположение не действительно на многопроцессорных системах. В версии 2.6, код ядра никогда (почти) не может считать, что он может держать процессор на данном участке кода. Если вы не пишете код имея в виду конкуренцию, будут случаться катастрофические отказы, которые могут быть чрезвычайно трудными для отладки.

Текущий процесс

Хотя модули ядра не выполняются последовательно, как приложения, большинство действий, выполняемых ядром, делаются от имени определённого процесса. Код ядра может обратиться к текущему процессу через глобальный объект ***current***, определённый в ***<asm/current.h>***, который даёт указатель на структуру ***task_struct***, определённую в ***<linux/sched.h>***. Указатель ***current*** ссылается на процесс, который выполняется в настоящее время. Во время выполнения системного вызова, например, ***open*** или ***read***, текущим процессом является тот, который сделал вызов. Код ядра может получать процессо-зависимую информацию используя ***current***, если это необходимо сделать. Пример этой техники приводится в Главе 6 128.

На самом деле ***current*** не является подлинно глобальной переменной. Необходимость поддержки SMP систем вынудила разработчиков ядра разработать механизм, который ищет текущий процесс на соответствующем процессоре. Этот механизм должен быть также быстрым, поскольку ссылки на ***current*** происходят часто. Результатом является архитектурно-зависимый механизм, который, как правило, скрывает указатель на структуру ***task_struct*** на стеке ядра. Детали реализации остаются скрытыми для других подсистем ядра и драйвер устройства может только подключить ***<linux/sched.h>*** и сослаться на текущий процесс. Например, следующая команда печатает идентификатор процесса и название команды текущего процесса через доступ к соответствующим полям в структуре ***task_struct***:

```
printk(KERN_INFO "The process is \"%s\" (pid %i)\n",
                current->comm, current->pid);
```

Название команды сохраняется в ***current->comm*** и является базовым именем файла программы (обрезается до 15 символов, если это необходимо), которая в настоящее время выполняется текущим процессом.

Несколько дополнительных деталей

Программирование ядра во многих отношениях отличается от программирования пользовательского пространства. Мы будем узнавать об этом, когда будем доходить до них в течение этой книги, но есть несколько основных вопросов, которые, хотя и не требуют своего отдельного раздела, заслуживают упоминания. Итак, вы углубляетесь в ядро и должны иметь в виду следующие соображения. Приложения размещаются в виртуальной памяти с очень большой областью стека. Стек, естественно, используется для хранения истории вызовов функций и всех автоматических переменных, создаваемых активной в данный момент функцией. Ядро, напротив, имеет очень маленький стек, он может быть так же мал, как одна 4096 байтовая страница. Ваши функции должны делить стек со всей цепочкой вызовов пространства ядра. Таким образом, никогда не является хорошей идеей объявление больших автоматических переменных; если вам необходимы более крупные структуры, вы должны выделять им память динамически во время вызова.

Часто при просмотре API ядра вы будете наталкиваться на имена функций, начинающиеся с двойного подчеркивания (__). Отмеченные так функции являются, как правило, низкоуровневым компонентом интерфейса и должны использоваться с осторожностью. По существу, двойное подчёркивание говорит программисту: "Если вы вызываете эту функцию, убедитесь, что знаете, что делаете".

Код ядра не может делать арифметики с плавающей точкой. Разрешение плавающей точки требовало бы, чтобы ядро сохраняло и восстанавливало состояние процессора с плавающей точкой при каждом входе и выходе в пространстве ядра, по крайней мере, на некоторых архитектурах. Учитывая, что в действительности нет необходимости в плавающей точке в коде ядра, не стоит иметь дополнительных накладных расходов.

Компиляция и загрузка

Пример “hello world” в начале этой главы включает краткую демонстрацию сборки модуля и загрузки его в систему. Существует, конечно, гораздо больше, чем тот процесс, который мы видели до сих пор. Этот раздел содержит более подробную информацию о том, как автор модуля превращает исходный код в работающую подсистему внутри ядра.

Компиляция модулей

В первую очередь нам необходимо узнать немножко о том, как должны собираться модули. Процесс сборки модулей существенно отличается от используемого для приложений пользовательского пространства; ядро - это большая автономная программа с подробными и точными требованиями о том, как кусочки собирать вместе. Процесс сборки также отличен от того, как это делалось в предыдущих версиях ядра; новая система сборки проще в использовании и даёт более правильный результат, но он очень отличается от того, что использовался раньше. Система сборки ядра представляет собой сложного зверя и мы посмотрим только на крошечный кусочек. Файлы, находящиеся в директории ***Documentation/kbuild*** исходных кодов ядра, являются обязательными для чтения любым желающим понять всё, что на самом деле происходит под поверхностью. Есть некоторые предварительные условия, которые необходимо выполнить, прежде чем вы сможете собрать модули ядра. Во-первых, убедитесь, что вы имеете достаточно свежие версии компилятора, утилит модулей и других необходимых утилит. Файл ***Documentation/changes*** в каталоге документации ядра всегда содержит список необходимых версий утилит; вы должны проверить это перед тем, как двигаться дальше. Попытка построить ядро (и его модули) с неправильными версиями утилит

может привести к бесконечным тонким, сложным проблемам. Заметим, что иногда слишком новая версия компилятора может быть столь же проблематичной, как и слишком старая; исходный код ядра делает много предположений о компиляторе и новые релизы могут иногда сломать что-то на некоторое время.

Если вы всё ещё не имеете под рукой дерева ядра или это ядро ещё не сконфигурировано и не собрано, теперь пора это сделать. Вы не сможете собрать загружаемый модуль для ядра версии 2.6 без этого дерева в своей файловой системе. Также полезно (но не обязательно), чтобы работающим ядром было то, для которого вы будете собирать модули.

После того как вы всё установили, создать Makefile для модуля просто. Фактически, для примера "hello world", приведённого ранее в этой главе, достаточно будет одной строчки:

```
obj-m := hello.o
```

Читателям, знакомым с ***make***, но не с системой сборки ядра 2.6, вероятно, будет интересно, как работает этот Makefile. В конце концов, приведённая выше строчка выглядит не как обычный Makefile. Ответ конечно же в том, что всё остальное делает система сборки ядра. Определение выше (которое использует расширенный синтаксис, предоставляемый GNU make) заявляет, что требуется собрать один модуль из объектного файла ***hello.o***. Результирующий модуль после сборки из объектного файла именуется как ***hello.ko***.

Если вместо этого вы имеете модуль под названием ***module.ko***, который создаётся из двух исходных файлов (называемых, скажем, ***file1.c*** и ***file2.c***), правильными магическими строчками будут:

```
obj-m := module.o
module-objs := file1.o file2.o
```

Чтобы Makefile, подобный приведённому выше, заработал, он должен быть вызван в более широком контексте системы сборки ядра. Если ваше дерево исходных текстов ядра находится, скажем, в каталоге ***~/kernel-2.6***, команда ***make***, необходимая для сборки модуля (набранная в каталоге, содержащем исходник модуля и Makefile), будет:

```
make -C ~/kernel-2.6 M=`pwd` modules
```

Эта команда начинается со смены своего каталога на указанный опцией **-C** (то есть на ваш каталог исходных кодов ядра). Там она находит Makefile верхнего уровня ядра. Опция **M=** заставляет Makefile вернуться обратно в директорию исходников вашего модуля, прежде чем попытаться построить целевой модуль. Эта задача, в свою очередь, ссылается на список модулей, содержащихся в переменной **obj-m**, который мы определили в наших примерах как ***module.o***. Ввод предыдущей команды ***make*** может стать через некоторое время утомительным, так что разработчики ядра разработали своего рода идиому Makefile, которая делает жизнь легче для модулей, собираемых вне дерева ядра. Весь фокус в том, чтобы написать Makefile следующим образом:

```
# Если KERNELRELEASE определён, значит вызов сделан из
# системы сборки ядра и можно использовать её язык.
ifneq ($(KERNELRELEASE),)
 obj-m := hello.o
# Иначе вызов сделан прямо из командной
# строки; вызвать систему сборки ядра.
```

```
else
 KERNELDIR ?= /lib/modules/$(shell uname -r)/build
 PWD := $(shell pwd)
default:
 $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

И снова мы видим расширенный синтаксис GNU ***make*** в действии. Этот Makefile читается при типичной сборке дважды. Когда Makefile вызывается из командной строки, он замечает, что переменная **KERNELRELEASE** не установлена. Он определяет расположение каталога исходных текстов ядра, воспользовавшись тем, что символическая ссылка ***build*** в директории установленных модулей указывает обратно на дерево сборки ядра. Если у вас на самом деле не работает ядро, для которого вы делаете сборку, можно добавить параметр **KERNELDIR=** в командную строчку, установив переменную **KERNELDIR** среды окружения или переписать строчку, которая задаёт **KERNELDIR** в Makefile. После того, как дерево исходных текстов ядра было найдено, Makefile запускает цель **default:**, которая запускает вторую команду ***make*** (параметризованную как **$(MAKE)** в Makefile), чтобы вызвать систему сборки ядра, как описано выше. При втором чтении Makefile устанавливает **obj-m** и о реальном создании модуля заботятся Makefile-ы ядра.

Этот механизм создания модулей может вам показаться немного громоздким и расплывчатым. Однако, когда вы привыкнете к нему, вы, скорее всего, оцените возможности, которые были запрограммированы в системе сборки ядра. Обратите внимание, что текст, приведённый выше, не является полным Makefile; реальный Makefile включает обычные набор целей по очистке от ненужных файлов, установке модулей и так далее. Для полноценной иллюстрации посмотрите Makefile-ы в каталоге с исходными кодами примеров.

Загрузка и выгрузка модулей

После сборки модуля следующим шагом является загрузка его в ядро. Как мы уже говорили, эту работу делает для вас ***insmod***. Программа загружает код модуля и данные в ядро, которое, в свою очередь, выполняет функцию, аналогичную выполняемой ***ld***, которой она связывает неразрешённые символы в модуле с таблицей символов ядра. В отличие от линкера (компоновщика), ядро, однако, не изменяет файл модуля на диске, а делает это с копией в памяти. ***insmod*** поддерживает несколько параметров командной строки (подробности смотрите на странице справки) и она может присваивать значения параметрам модуля до линковки его с текущим ядром. Таким образом, если модуль разработан правильно, он может быть сконфигурирован во время загрузки; конфигурация во время загрузки даёт пользователю больше гибкости, чем конфигурация во время компиляции, которая всё ещё иногда используется. Конфигурирование во время загрузки объясняется в разделе "Параметры модуля" 33 далее в этой главе.

Заинтересованные читатели могут захотеть посмотреть, как ядро поддерживает ***insmod***: это опирается на системный вызов, определённый в ***kernel/module.c***. Функция ***sys_init_module*** выделяет память ядра для удержания модуля (эта память выделяется через ***vmalloc***; смотрите раздел "vmalloc и друзья" 214 в Главе 8 203); затем копирует текст модуля в эту область памяти, разрешает (определяет) ссылки в модуле на ядро через таблицу символов ядра и вызывает функцию инициализации модуля, чтобы дать выполнить остальное.

Если реально посмотреть исходный код ядра, вы обнаружите, что имена системных вызовов имеют префикс **sys_**. Это верно для всех системных вызовов, но не других функций; это полезно иметь в виду, когда делается быстрый поиск (греппинг, grepping от grep-служебной

программы в Unix) системных вызовов в исходниках. Достойна быстрого упоминания утилита ***modprobe***. ***modprobe***, как и ***insmod***, загружает модуль в ядро. Она отличается тем, что просматривает модуль, который необходимо загрузить, ссылается ли он на любые символы, которые в настоящее время не определены в ядре. В случае обнаружения любых таких ссылок, ***modprobe*** ищет другие модули, в которых определены соответствующие символы, по текущему пути поиска модулей. Когда ***modprobe*** находит такие модули (которые необходимы загружаемому модулю), она также загружает их в ядро. Если вы в этой ситуации взамен используете ***insmod***, команда даст ошибку с сообщением “unresolved symbols” ("неизвестные символы"), оставленным в системном лог-файле.

Как упоминалось ранее, модули могут быть удалены из ядра утилитой ***rmmod***. Обратите внимание, что удаление модуля завершится неудачей, если ядро считает, что модуль всё ещё используется (например, какая-то программа всё ещё имеет открытый файл для устройства, экспортированного модулями), или если ядро было сконфигурировано, чтобы запрещать удаление модулей. Можно сконфигурировать ядро, чтобы разрешить "принудительное" удаление модулей, даже если они выглядят занятыми. Однако, если вы достигли точки, где вы рассматриваете использовании этой опции, что-то, вероятно, пошло неправильно достаточно сильно, так что перезагрузка может быть лучшим способом действия.

Программа ***lsmod*** выдаёт список модулей, загруженных в данный момент в ядро. Также предоставляется некоторая другая информация, такая как любые другие модули, использующие определённый модуль. ***lsmod*** работает путём чтения виртуального файла ***/proc/module***. Информацию о загруженных в данный момент модулях можно также найти в виртуальной файловой системе **sysfs** под ***/sys/module***.

Зависимость от версии

Имейте в виду, что код вашего модуля должен быть перекомпилирован для каждой версии ядра, это связано, по крайней мере, с отсутствием **modversions**, не рассматриваемыми здесь, поскольку они больше для дистрибьютеров, чем для разработчиков. Модули сильно привязаны к структурам данных и прототипам функций, определённым в конкретной версии ядра; интерфейс, видимый модулю, может значительно меняться от одной версии ядра к другому. Это, конечно, особенно верно в отношении разрабатываемых ядер.

Ядро не только предполагает, что данный модуль был собран для подходящей версии ядра. Одним из шагов в процессе сборки является ссылка вашего модуля на файл (названный ***vermagic.o***) из текущего дерева ядра; этот объект содержит достаточно много информации о ядре для которого был собран модуль, включая целевую версию ядра, версию компилятора и значения ряда важных параметров конфигурации. При попытке загрузить модуль эта информация может быть проверена на совместимость с работающим ядром. Если данные не совпадают, модуль не загружается, вместо этого вы увидите что-то вроде:

```
# insmod hello.ko
Error inserting './hello.ko': -1 Invalid module format
```

Просмотр системных лог-файлов (***/var/log/messages*** или того, что ваша система настроена использовать) выявит те проблемы, которые привели к невозможности загрузки модуля. Если вам необходимо скомпилировать модуль для определённой версии ядра, необходимо использовать систему сборки и дерево исходных текстов данной версии. Простое изменение переменной **KERNELDIR** в показанном ранее примере Makefile делает этот трюк. Интерфейсы ядра часто меняются между релизами. Если вы пишете модуль, который предназначен для работы с множеством версий ядра (особенно если он должен работать с основными

релизами), вам, скорее всего, придётся использовать макросы и конструкции **#ifdef**, чтобы сделать код правильно собираемым. Данное издание этой книги рассказывает только об одной основной версии ядра, так что вы не часто встретите проверку версии в наших примерах кода. Но потребность в них иногда возникает. В таких случаях вы должны использовать определения, содержащиеся в ***linux/version.h***. В этом заголовочном файле определены следующие макросы:

UTS_RELEASE
: Этот макрос заменяется на строку, которая описывает версию дерева. Например, "2.6.10".

LINUX_VERSION_CODE
: Этот макрос заменяется на бинарное представление версии ядра, один байт для каждой части номера версии. Например, код для 2.6.10 это 132618 (то есть, 0x02060a). (* Это позволяет использовать до 256 разрабатываемых версий между стабильными версиями.) С этой информацией вы можете (почти) легко определить, с какой версией ядра вы имеете дело.

KERNEL_VERSION(major,minor,release)
: Этот макрос используется для построения целого числа кода версии из отдельных номеров, которые представляют собой номер версии. Например, KERNEL_VERSION (2,6,10) заменяется на 132618. Этот макрос очень полезен, когда необходимо сравнить текущую версию и заданное контрольное значение.

Большинство зависимостей, базирующихся на версии ядра, могут быть обработаны условиями препроцессора, используя **KERNEL_VERSION** и **LINUX_VERSION_CODE**. Не следует, однако, делать проверку зависимостей от версии, покрывая код драйвера беспорядочным волосатыми условиями **#ifdef**; лучшим способом обращения с несовместимостями является скрытие их в специфичных заголовочных файлах. По общему правилу код, который зависит от версии (или платформы), должен быть скрыт за низкоуровневым макросом или функцией. Высокоуровневый код может просто вызывать эти функции, не заботясь о низкоуровневых деталях. Код, написанный таким образом, как правило, легче читать и более надёжен.

Зависимость от платформы

Каждая компьютерная платформа имеет свои особенности и разработчики ядра могут свободно использовать все особенности для достижения наилучшей производительности в результирующем объектном файле. В отличие от разработчиков приложений, которые должны линковать свой код с предварительно скомпилированными библиотеками и придерживаться конвенций о передачи параметров, разработчики ядра могут определить некоторым регистрам процессора определённые роли и они сделали это. Кроме того, код ядра может быть оптимизирован для определённого процессора в семействе процессоров, чтобы получить лучшее от целевой платформы: в отличие от приложений, которые часто распространяются в бинарном формате, заказная компиляции ядра может быть оптимизирована для определенного набора компьютеров.

Например, архитектура IA32 (x86) была разделена на несколько разных типов процессоров. Старый 80386 процессоров до сих пор поддерживается (пока), хотя его набор команд по современным меркам довольно ограничен. Более современные процессоры этой архитектуры ввели целый ряд новых возможностей, включая быстрые инструкции входа в ядро,

межпроцессорную блокировку, копирование данных и так далее. Новые процессоры могут также при эксплуатации в соответствующем режиме использовать 36-х разрядные (или больше) физические адреса, что позволяет им адресовать более 4 ГБ физической памяти. Другие семейства процессоров предоставляют аналогичные усовершенствования. В зависимости от различных параметров настройки ядро может быть собрано так, чтобы использовать эти дополнительные возможности.

Очевидно, что если модуль предназначен для работы с данным ядром, он должен быть построен с таким же знанием целевого процессора, как и ядро. Вновь в игру вступает объект ***vermagic.o***. При загрузке модуля ядро проверяет процессорно-зависимые параметры конфигурации для модуля и убеждается, что они соответствуют работающему ядру. Если модуль был скомпилирован с другими параметрами, он не загружается. Если вы планируете написать драйвер для общего распространения, вы вполне можете быть удивлены, как можно поддержать все эти различные варианты. Наилучший ответ, конечно, выпустить драйвер под GPL-совместимой лицензией и внести его в основное ядро. В противном случае, самым лучшим вариантом может быть распространение драйвера в форме исходников и набора скриптов для компиляции его на компьютере пользователя. Некоторые производители выпустили утилиты для упрощения этой задачи. Если вы должны распространять свой драйвер в бинарном виде, необходимо посмотреть на разные ядра, для которых вы будете распространять драйвер, и обеспечить версию модуля для каждого. Не забудьте принять во внимание ошибки ядер, которые могут быть исправлены после момента начала дистрибьюции. Затем существуют вопросы лицензирования, которые обсуждались в разделе "Лицензионное соглашение" 11 в Главе 1 2. Как правило, дистрибьюция в исходной форме - наиболее простой способ проложить свой путь в мире.

Символьная таблица ядра

Мы видели, как ***insmod*** разрешает неопределённые символы на таблицу публичных символов ядра. В таблице содержатся адреса глобальных объектов ядра - функций и переменных, которые необходимы для выполнения модуляризованных драйверов. При загрузке модуля любой символ, экспортируемый модулем, становится частью таблицы символов ядра. В обычном случае модуль предоставляет свою функциональность без необходимости экспортировать любые символы вообще. Однако, вам необходимо экспортировать символы, если другие модули могут получить выгоду от их использования.

Новые модули могут использовать символы, экспортированные вашим модулем, и вы можете накладывать новые модули поверх других модулей (то есть создавать стек модулей). Исходный код основного ядра также использует наложение модулей: файловая система ***msdos*** опирается на символы, экспортируемые модулем ***fat***, а каждое устройство ввода модуля USB опирается на модули ***usbcore*** и ***input***. Стек модулей полезен в сложных проектах. Если новые абстракции реализованы в виде драйвера устройства, он может предложить подключение для конкретной аппаратной реализации. Например, набор драйверов video-for-linux разделён внутри общего модуля, который экспортирует символы, используемые более низкоуровневыми драйверами устройств для конкретного оборудования. Вы загружаете общий видео модуль и специфичные модули для вашего установленного оборудования в соответствии с настройками. Поддержка параллельных портов и широкий спектр подключаемых устройств обрабатывается таким же образом, как и в подсистеме USB ядра. Стек подсистемы параллельного порта показан на Рисунке 2-2; Стрелки показывают связи между модулями и с интерфейсом программирования ядра.

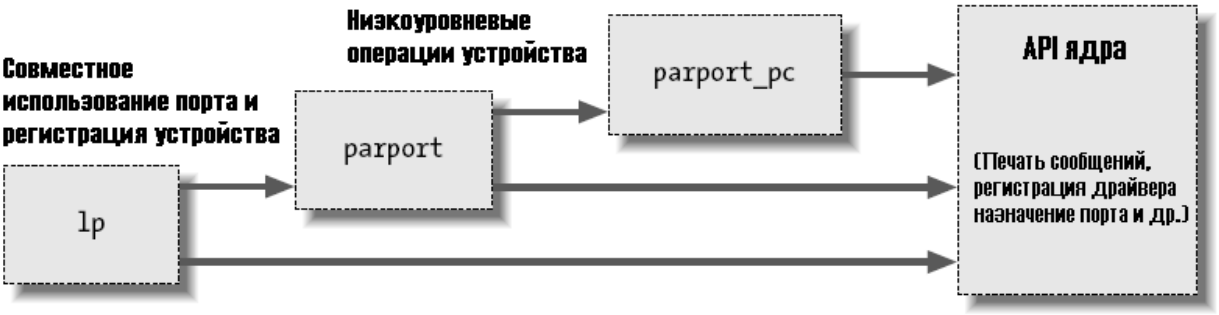


Рисунок 2-2. Стек модулей драйвера параллельного порта

При использовании стековых модулей полезно использовать утилиту ***modprobe***. Как описано выше, функции ***modprobe*** во многом такие же, как у ***insmod***, но она также загружает любые другие модули, которые необходимы для модуля, который вы хотите загрузить. Таким образом, одна команда ***modprobe*** может иногда заменить несколько вызовов ***insmod*** (хотя ***insmod*** всё равно необходима при загрузке собственных модулей из текущего каталога, так как ***modprobe*** просматривает только стандартные каталоги для установленных модулей). Использование стека, чтобы разделить модули на несколько слоёв, может помочь сократить время разработки за счёт упрощения каждого слоя. Это похоже на разделение между механизмом и политикой, которое мы обсуждали в Главе 1.

Файлы ядра Linux обеспечивают удобный способ управления видимостью ваших символов, уменьшая тем самым загрязнение пространства имён (заполнение пространства имён именами, которые могут конфликтовать с теми, которые определены в другом месте в ядре) и поощряя правильное скрытие информации. Если ваш модуль должен экспортировать символы для использования другими модулями, необходимо использоваться следующие макросы:

```
EXPORT_SYMBOL(name);
EXPORT_SYMBOL_GPL(name);
```

Каждый из вышеописанных макросов делает данный символ доступным за пределами модуля. Версия **_GPL** делает символ доступным только для GPL-лицензированных модулей. Символы должны быть экспортированы в глобальной части файла модуля, вне какой-либо функции, потому что макросы заменяются на объявление переменной специального назначения, которая, как ожидается, будет доступна глобально. Эта переменная запоминается в специальной части исполняемого модуля (“секции ELF“), которая используется ядром во время загрузки, чтобы найти переменные, экспортируемые модулем. (Заинтересованные читатели могут заглянуть в ***<linux/module.h>*** для деталей, хотя детали не нужны, чтобы всё работало.)

Предварительные замечания

Мы приближаемся к просмотру некоторого фактического кода модуля. Но сначала нам необходимо взглянуть на некоторые другие вещи, которые должны появиться в ваших исходных файлах модуля. Ядро является уникальной окружающей средой и это накладывает свои требования на код, который будет с ним взаимодействовать.

Большинство кода ядра приходит к включению довольно большого числа заголовочных файлов для получения определения функций, типов данных и переменных. Мы будем

рассматривать эти файлы по мере подхода к ним, но есть несколько, которые являются специфическими для модулей, и должны появиться в каждом загружаемом модуле. Таким образом, почти все коды модулей имеют следующее:

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

module.h содержит большое количество определений символов и функций, необходимых загружаемым модулям. Вам необходим ***init.h***, чтобы указать ваши функции инициализации и очистки, как мы видели в вышеприведённом примере “hello world”, и к которому мы вернёмся в следующем разделе. Большинство модулей также подключают ***moduleparam.h***, чтобы разрешить получение параметров модуля во время загрузки; мы будем делать это в ближайшее время. Это не является строго необходимым, но ваш модуль действительно должен указать, под какой лицензией распространяется код. Делается это просто включением строки **MODULE_LICENSE**:

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

Определёнными лицензиями, признаваемыми ядром являются “GPL” (для любой версии GNU General Public License), “GPL v2” (для GPL версии 2 только), “GPL and additional rights” ("GPL и дополнительные права"), “Dual BSD/GPL” (“Двойная BSD/GPL“), “Dual MPL/GPL” (“Двойная MPL/GPL”) и “Proprietary” (“Частная собственность”). Если ваш модуль не помечен явно находящимся под свободной лицензией, признающейся ядром, предполагается, что он будет частным, а ядро "испорчено", когда модуль загружен. Как уже упоминалось в разделе "Лицензионное соглашение" 11 в Главе 1 2, разработчики ядра имеют склонность относиться без энтузиазма к помощи пользователям, у которых возникают проблемы после загрузки собственных модулей.

Другие описательные определения, которые могут содержаться внутри модуля включают **MODULE_AUTHOR** (где указывается, кто написал модуль), **MODULE_DESCRIPTION** (описание для человека, что делает модуль), **MODULE_VERSION** (номер ревизии кода; смотрите комментарии в ***<linux/module.h>*** для соглашений, используемых при создании строки версии), **MODULE_ALIAS** (другое имя, под которым этот модуль может быть известен) и **MODULE_DEVICE_TABLE** (говорит пространству пользователя, какие устройства поддерживает модуль). Мы обсудим **MODULE_ALIAS** в Главе 11 286 и **MODULE_DEVICE_TABLE** в Главе 12 288.

Различные декларации **MODULE_** могут появиться в любом месте вашего исходного файла вне функции. Однако, по относительно недавнему соглашению, в коде ядра эти декларации размещаются в конце файла.

Инициализация и выключение

Как уже говорилось, функция инициализации модуля регистрирует все средства, предлагаемые модулем. Под ***средствами*** мы имеем в виду новую функциональность, будь то целый драйвер или новая абстракция программного обеспечения, которые могут быть доступны приложению. Само определение функции инициализации всегда выглядит следующим образом:

```
static int __init initialization_function(void)
{
    /* Здесь размещается код инициализации */
```

```
}
module_init(initialization_function);
```

Функции инициализации должны быть объявлены статическими, так как они не предназначены быть видимыми за пределами определённого файла; тем не менее, нет жёсткого правила по этому поводу, так как ни одна функция не экспортируется в остальную часть ядра, если это не указано явно. Признак **__init** в определении может показаться немного странным; это подсказка ядру, что данная функция используется только во время инициализации. Загрузчик модуля отбрасывает функцию инициализации после загрузки модуля, делая память доступной для другого использования. Существует аналогичный атрибут (**__initdata**) для данных, используемых только в процессе инициализации. Использование **__init** и **__initdata** является необязательным, но стоит побеспокоиться об этом. Просто убедитесь, что не используете их для функций (или структуры данных), которые будут использоваться после завершения инициализации. Вы можете встретить в исходном коде ядра **__devinit** и **__devinitdata**, они транслируются к **__init** и **__initdata** только тогда, когда ядро не было сконфигурировано для поддержки устройств, подключаемых без выключения системы. Мы будем рассматривать поддержку "горячей" коммутации в Главе 14 347.

Использование ***module_init*** является обязательным. Этот макрос добавляет специальный раздел к объектному коду модуля там, где будет находится функция инициализации модуля. Без этого определения ваша функция инициализации никогда не вызовется. Модули могут зарегистрировать много типов средств, включая разные виды устройств, файловых систем, криптографические преобразования и многое другое. Для каждого средства есть определённые функции ядра, которые осуществляют такую регистрацию. Аргументы, передаваемые функциям регистрации ядра, как правило, указатели на структуры данных, описывающие новые регистрируемые средства и их имена. Структура данных обычно содержит указатели на функции модуля, которые определяют, как надо вызывать функции в теле модуля.

Число объектов, которые могут быть зарегистрированы, превышает список типов устройств, упомянутых в Главе 1 2. Среди прочего, они включают последовательные порты, разнообразные устройства, ***sysfs*** записи, ***/proc*** файлы, исполняемые домены и дисциплину линий связи. Многие из этих регистрируемых объектов поддерживают функции, не связанные непосредственно с аппаратурой, а остаются в виде "программных абстракций". Эти объекты могут быть зарегистрированы, потому что они, тем не менее, интегрированы в функциональность драйвера (например, ***/proc*** файлы и дисциплина линий связи).

Есть и другие средства, которые могут быть зарегистрированы в качестве дополнений для определённого драйвера, но их использование является настолько специфичным, что не стоит говорить о них; они используют технику стека, как описано в разделе "Символьная таблица ядра" 26. Если вы хотите исследовать поглубже, вы можете поискать **EXPORT_SYMBOL** в исходниках ядра и найти точки входа, предлагаемые разными драйверами. Большинство функций регистрации имеют префикс **register_**, так что ещё одним возможным способом является поиск в исходниках ядра по **register_**.

Функция очистки

Каждый нетривиальный модуль также требует функцию очистки, которая отменяет регистрацию интерфейсов и возвращает все ресурсы системе, прежде чем модуль удаляется. Эта функция определена следующим образом:

```
static void __exit cleanup_function(void)
{
```

```
    /* Здесь размещается код очистки */
}

module_exit(cleanup_function);
```

Функция очистки не имеет возвращаемого значения и объявляется как **void**. Признак **__exit** указывает, что этот код будет только выгружать модуль (чтобы компилятор поместил его в специальный раздел ELF). Если ваш модуль собирается прямо в ядре или если ваше ядро сконфигурировано не разрешать выгрузку модулей, функции, отмеченные **__exit**, просто отбрасываются. По этой причине функции, отмеченные **__exit**, могут быть вызваны только при выгрузке модуля или время завершения работы системы; любое другое использование является ошибкой. И снова, декларация ***module_exit*** необходима, чтобы позволить ядру найти функцию очистки.

Если модуль не определяет функцию очистки, ядро не позволит ему быть выгруженным.

Перехват ошибок во время инициализации

Одним обстоятельством, которое вы всегда должны иметь в виду при регистрации объектов ядром, является то, что регистрация может провалиться. Даже простейшие действия часто требуют выделения памяти, а необходимая память может быть недоступна. Так что код модуля должен всегда проверять возвращаемые значения и быть уверенным, что запрошенная операция в действительности удалась.

В случае возникновения ошибок при регистрации утилит в любом случае стоит первым делом решить сможет ли модуль продолжить инициализацию сам. Зачастую в случае необходимости модуль может продолжить работу после неудачной регистрации с уменьшенной функциональностью. Когда это возможно, ваш модуль должен продвигаться вперед и после неудавшихся операций предоставить то, что может.

Если выяснится, что ваш модуль просто не может загрузиться после ошибки определённого типа, необходимо отменить все зарегистрированные операции, выполненные до ошибки. Linux не хранит для каждого модуля реестр средств, которые были зарегистрированы, поэтому если в некоторой точке инициализация не удаётся, модуль должен вернуть всё сам. Если вы когда-нибудь потерпите неудачу при разрегистрации уже сделанного, ядро останется в нестабильном состоянии; оно содержит указатели на внутренний код, который больше не существует. В таких ситуациях единственным выходом, как правило, является перезагрузка системы. Вам действительно стоит хотеть заботиться сделать правильные вещи, когда возникают ошибки инициализации.

Возвращаемые ошибки иногда лучше обрабатывать инструкцией **goto**. Мы обычно ненавидим использовать **goto**, но на наш взгляд это та ситуация, в которой она полезна. Осторожное использование **goto** в ошибочных ситуациях может устранить большую, сложную, высоко-вложенную, "структурированную" логику. Таким образом, как показано здесь, в ядре **goto** часто используется для исправления ошибки.

Следующий пример кода (использующий фиктивные функции регистрации и разрегистрации) ведёт себя правильно, если инициализация в любой момент даёт ошибку:

```
int __init my_init_function(void)
{
    int err;
```

```
    /* регистрация использует указатель и имя */
    err = register_this(ptr1, "skull");
    if (err) goto fail_this;
    err = register_that(ptr2, "skull");
    if (err) goto fail_that;
    err = register_those(ptr3, "skull");
    if (err) goto fail_those;

    return 0; /* успешно */

fail_those: unregister_that(ptr2, "skull");
fail_that: unregister_this(ptr1, "skull");
fail_this: return err; /* передать ошибку для наследования */
}
```

Этот код пытается зарегистрировать три (фиктивных) средства. Инструкция **goto** используется в случае ошибки, чтобы разрегистрировать только средства, которые были успешно зарегистрированы до того, как дела пошли плохо.

Ещё одним вариантом, не требующим волосатых инструкций **goto**, является отслеживание того, что было успешно зарегистрировано, и вызов функции очистки вашего модуля в случае любой ошибки. Функция очистки откатывает только те шаги, которые были успешно выполнены. Однако, этот вариант требует больше кода и больше процессорного времени, так что для быстрых решений всё ещё прибегают к **goto**, как лучшему инструменту обработки ошибок.

Возвращаемым значением ***my_init_function*** является **err**, код ошибки. В ядре Linux коды ошибок - это отрицательные числа, принадлежащие к определённым в ***<linux/errno.h>***. Если вы хотите сгенерировать собственные коды ошибок взамен возвращаемых другими функциями, вы должны подключить ***<linux/errno.h>***, чтобы использовать символические значения, такие как **-ENODEV**, **-ENOMEM** и так далее. Возврат соответствующих кодов ошибок это всегда хорошая практика, потому что пользователь программы может превратить их в значимые строки используя ***perror*** или аналогичные средства.

Очевидно, что функция очистки модуля должна отменить регистрацию всего, что выполнила функция инициализации, и привычно (но обычно не обязательно), чтобы разрегистрация средств выполнялась в порядке, обратном использованному для регистрации:

```
void __exit my_cleanup_function(void)
{
    unregister_those(ptr3, "skull");
    unregister_that(ptr2, "skull");
    unregister_this(ptr1, "skull");
    return;
}
```

Если ваша инициализация и очистка являются более сложными, чем работа с несколькими объектами, подход через **goto** может стать трудным для управления, потому что весь код очистки должен быть повторен в функции инициализации, перемежаясь несколькими метками. Следовательно, иногда оказывается более удобной другая структура кода. То, что необходимо сделать, чтобы минимизировать дублирование кода и сохранять всё упорядоченным, это вызывать функцию очистки в функции инициализации, когда возникает ошибка. Функция

очистки затем должна проверить состояние каждого объекта перед отменой его регистрации. В своей простейшей форме код выглядит следующим образом:

```
struct something *item1;
struct somethingelse *item2;
int stuff_ok;

void my_cleanup(void)
{
    if (item1)
        release_thing(item1);
    if (item2)
        release_thing2(item2);
    if (stuff_ok)
        unregister_stuff( );
    return;
}

int __init my_init(void)
{
    int err = -ENOMEM;

    item1 = allocate_thing(arguments);
    item2 = allocate_thing2(arguments2);
    if (!item1 || !item2)
        goto fail;
    err = register_stuff(item1, item2);
    if (!err)
        stuff_ok = 1;
    else
        goto fail;
    return 0; /* успешно */

fail:
    my_cleanup( );
    return err;
}
```

Как показано в этом коде, могут требоваться или не требоваться внешние флаги, чтобы отметить успешность шага инициализации, в зависимости от семантики вызываемых функций регистрации/размещения. Независимо от того, нужны или нет флаги, такой способ инициализации хорошо масштабируется на большое число объектов и часто лучше, чем техника, показанная ранее. Однако, следует отметить, что функция очистки не может быть отмечена **__exit**, когда её вызывают не в коде, выполняемом при выходе, как в предыдущем примере.

Гонки при загрузке модуля

До сих пор наша дискуссия обходила один важный аспект загрузки модулей: условия гонок. Если вы не будете осторожны в том, как вы пишете функцию инициализации, вы можете создать ситуации, которые могут поставить под угрозу стабильность системы в целом. Мы будем обсуждать условия гонок позже в этой книге, ибо теперь будет достаточно нескольких небольших замечаний.

Во-первых, вы всегда должны помнить, что некоторые другие части ядра могут использовать любые средства, которые вы регистрируете, сразу же после завершения регистрации. Другими словами, вполне возможно, что ядро будет делать вызовы в вашем модуле в то время, пока ваша функция инициализации всё ещё работает. Так что ваш код должен быть подготовлен, чтобы быть вызванным, как только завершится первая регистрация. Не регистрируйте любые средства, пока все ваши внутренние инициализации, необходимые для поддержки этого средства, не будут завершены.

Вы должны также учесть, что произойдёт, если ваша функция инициализации решит, что необходимо прервать работу, но некоторые части ядра уже используют зарегистрированные средства модуля. Если такая ситуация в вашем модуле возможна, вы должны серьёзно подумать, не рухнет ли инициализация вообще. В конце концов, модуль достигает успеха экспортом чего-то полезного. Если инициализация должна закончиться неудачей, она должна тщательно обходить любые возможные операции, идущие в другие части ядра, до завершения тех операций (видимо, речь идёт об операциях обработки ошибок).

Параметры модуля

Некоторые параметры, которые драйвер должен знать, могут меняться от системы к системе. Они могут варьироваться от номера устройства для использования (как мы увидим в следующей главе) до многочисленных аспектов, как драйвер должен работать. Например, драйверы для адаптеров SCSI часто имеют параметры контроля за использованием маркированной очереди комманд (tagged command queuing), а драйвер, интегрированной в устройство электроники (Integrated Device Electronics, IDE), позволяет пользователю управлять операциями DMA. Если ваш драйвер управляет старым оборудованием, он может также нуждаться в точном указании, где найти порты ввода/вывода для оборудования или адреса памяти ввода/вывода. Ядро поддерживает эти потребности, делая возможным для драйвера указать параметры, которые могут быть изменены при загрузке модуля драйвера.

Значения параметрам могут быть заданы во время загрузки через ***insmod*** или ***modprobe***; последняя также можете прочитать значение параметра из своего файла конфигурации (***/etc/modprobe.conf***). Команды принимают спецификацию из нескольких типов значений в командной строке. В качестве способа демонстрации этой возможности представьте себе столь необходимое улучшение в модуле “hello world” (названном ***hellop***), показанного в начале данной главы. Мы добавляем два параметра: целое число, называемое **howmany**, и символьную строку, названную **whom**. Наш гораздо более функциональный модуль во время загрузки приветствует **whom** не раз, а **howmany** раз. Такой модуль мог бы быть загружен с помощью командной строчки, такой как:

```
insmod hellop howmany=10 whom="Mom"
```

Будучи загруженным таким образом, ***hellop*** сказал бы “Hello, Mom” 10 раз.

Однако, прежде чем ***insmod*** сможет изменить параметры модуля, модуль должен сделать это возможным. Параметры объявляются макросом **module_param**, который определён в ***moduleparam.h***. **module_param** принимает три параметра: имя переменной, её тип и маску разрешений, которые будут использоваться для сопровождения записи в sysfs. Макрос должен быть размещён вне какой-либо функции и, как правило, около заголовка файла с исходным текстом. Таким образом ***hellop*** объявил бы свои параметры и сделал их доступными для ***insmod*** следующим образом:

```
static char *whom = "world";
```

```
static int howmany = 1;
module_param(howmany, int, S_IRUGO);
module_param(whom, charp, S_IRUGO);
```

Для параметров модуля поддерживаются нижеперечисленные 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, и соответствующим образом устанавливается указатель.

int
long
short
uint
ulong
ushort
Базовые целые величины разной размерности. Версии, начинающиеся с **u**, являются беззнаковыми величинами.

Также модулем загрузчика поддерживаются массивы параметров, где значения поставляются как список значений, разделённых запятыми. Чтобы объявить массив параметров, используйте:

```
module_param_array(name,type,nump,perm);
```

Где **name** - это имя массива (и этого параметра), **type** - это тип элементов массива, **nump** - указатель на целочисленную переменную, а **perm** является обычным значением параметра разрешения доступа. Если во время загрузки задаётся массив параметров, **nump** задаёт число получаемых переменных. Загрузчик модуля отказывается принимать больше величин, чем может поместиться в массиве.

Если вам действительно необходим такой тип, который не фигурирует в списке выше, существуют приёмы в коде модуля, которые позволяют определить их; смотрите ***moduleparam.h*** для подробной информации о том, как это сделать. Всем параметрам модуля должны быть присвоены значения по умолчанию; ***insmod*** изменяет значение переменной только если это явно указано пользователем. Модуль может определить, какие параметры были заданы, сравнивая их значения со значениями по умолчанию.

Последнее поле **module_param** является значением разрешения доступа; вы должны использовать определения, содержащиеся в ***<linux/stat.h>***. Это значение определяет, кто имеет доступ к представлению параметра модуля в sysfs. Если **perm** установлен в 0, вообще нет никакого доступа к записи в sysfs, в противном случае он появится в каталоге ***/sys/module*** (* Однако, на момент написания есть разговоры о переносе куда-то в sysfs.) с заданным набором разрешений. Для параметра, который может читаться кем угодно, но не может быть изменён, используйте **S_IRUGO**; **S_IRUGO | S_IWUSR** позволяют root-у изменить параметр. Заметим, что если параметр изменяется через sysfs, значение этого параметра, видимого

модулем, изменяется, но ваш модуль не уведомляется об этом любым другим способом. Вы, вероятно, не сделаете параметры модуля записываемыми, если вы не готовы детектировать изменения и реагировать соответствующим образом.

Работа в пространстве пользователя

Программист Unix, который сталкивается с программированием ядра в первый раз, может нервничать при написании модуля. Написать пользовательскую программу, которая читает и пишет прямо в порты устройства, может быть проще.

Действительно, есть некоторые аргументы в пользу программирования в пространстве пользователя и иногда написание так называемого драйвера пользовательского пространства - мудрая альтернатива доскональному изучению ядра. В этом разделе мы рассмотрим некоторые причины, почему вы можете написать драйвер в пользовательском пространстве. Эта книга о драйверах пространства ядра, поэтому мы не пойдём дальше этого вводного обсуждения.

Преимущества драйверов пространства пользователя:

- Можно подключить полную библиотеку Си. Драйвер может выполнять множество экзотических задач, не прибегая к внешним программам (это полезные программы, обеспечивающие выполнение пользовательских политик, которые обычно распространяются вместе с самим драйвером).
- Программист может запустить обычный отладчик для кода драйвера без необходимости прохождения искривлений, чтобы отлаживать работающее ядро.
- Если драйвер пространства пользователя завис, вы можете просто убить его. Проблемы с драйвером вряд ли подвесят всю систему, если только оборудование управляется уж совсем неправильно.
- Пользовательская память переключаемая, в отличие от памяти ядра. Редко используемые устройства с большим драйвером не будут занимать оперативную память (RAM), которую могли бы использовать другие программы, за исключением момента, когда они действительно используются.
- Хорошо продуманная программа драйвера может ещё, как и драйверы пространства ядра, позволять конкурентный доступ к устройству.
- Если вы должны написать драйвер с закрытым исходным кодом, вариант пользовательского пространства позволяет вам избежать двусмысленных ситуаций лицензирования и проблемы с изменением интерфейсов ядра.

Например, USB драйверы могут быть написаны для пространства пользователя, смотрите (по-прежнему молодой) проект libusb на libusb.sourceforge.net и "gadgetfs" в исходных кодах ядра. Другим примером является X сервер: он точно знает, что оборудование может делать и чего оно не может, и предлагает графические ресурсы всем X клиентам. Однако, следует отметить, что существует медленный, но неуклонный дрейф в сторону базирующихся на кадровом буфере графических сред, где X сервер для фактической манипуляции графикой действует только в качестве сервера на базе реального драйвера пространства ядра.

Как правило, автор драйвера пространства пользователя обеспечивает выполнение серверного процесса, перенимая от ядра задачу быть единственным агентом, отвечающим за управление аппаратными средствами. Клиентские приложения могут затем подключаться к серверу для выполнения фактического взаимодействия с устройством; таким образом, процесс драйвера с развитой логикой может позволить одновременный доступ к устройству. Именно так работает X сервер.

Но подход к управлению устройством в пользовательском пространстве имеет ряд недостатков. Наиболее важными из них являются:

- В пользовательском пространстве не доступны прерывания. На некоторых платформах существуют методы обхода этого ограничения, такие как системный вызов ***vm86*** на архитектуре IA32.
- Прямой доступ к памяти возможен только через ***mmapping /dev/mem*** и делать это может только привилегированный пользователь.
- Доступ к портам ввода-вывода доступен только после вызова ***ioperm*** или ***iopl***. Кроме того, не все платформы поддерживают эти системные вызовы и доступ к ***/dev/port*** может быть слишком медленным, чтобы быть эффективным. Оба системных вызовов и файл устройства зарезервированы для привилегированных пользователей.
- Время отклика медленнее, так как требуется переключение контекста для передачи информации или действий между клиентом и аппаратным обеспечением.
- Что ещё хуже, если драйвер был перемещён (засвопирован) на диск, время отклика является неприемлемо долгим. Использование системного вызова ***mlock*** может помочь, но обычно требуется заблокировать много страниц памяти, потому что программы пользовательского пространства зависят от многих библиотек кода. ***mlock*** тоже ограничен для использования только привилегированными пользователями.
- Наиболее важные устройства не могут оперировать в пользовательском пространстве, в том числе, но не только, сетевые интерфейсы и блочные устройства.

Как видите, в конце концов, драйверы пользовательского пространства не могут делать многое. Интересные приложения, тем не менее, существуют: например, поддержка для устройств SCSI сканера (осуществляет пакет ***SANE***) и программы записи CD (осуществляется ***cdrecord*** и другими утилитами). В обоих случаях драйверы устройства пользовательского уровня зависят от драйвера ядра “SCSI generic”, который экспортирует для программ пользовательского пространства низкоуровневую функциональность SCSI, так что они могут управлять своим собственным оборудованием.

Случаем, в котором работа в пространстве пользователя может иметь смысл, является тот, когда вы начинаете иметь дело с новым и необычным оборудованием. Таким образом вы можете научиться управлять вашим оборудованием без риска подвешивания системы в целом. После того, как вы сделали это, выделение этого программного обеспечения в модуль ядра должно быть безболезненной операцией.

Краткая справка

Этот раздел суммирует функции ядра, переменные, макросы и ***/proc*** файлы, которых мы коснулись в этой главе. Он предназначен выступать как справочная информация. Каждый объект указывается после соответствующего заголовочного файла, если он необходим. Аналогичный раздел появляется в конце почти каждой главы, начиная с этой, суммируя новые обозначения, включённые в эту главу. Записи в этом разделе будет появляться в том же порядке, в котором они были введены в разделе:

insmod
modprobe
rmmod
Утилиты пространства пользователя для загрузки модулей в работающее ядро и удаления их.

#include <linux/init.h>

module_init(init_function);
module_exit(cleanup_function);
Макросы, которыми помечают функции модуля инициализации и очистки.

__init
__initdata
__exit
__exitdata
Маркеры для функций (***__init*** и ***__exit***) и данных (***__initdata*** и ***__exitdata***), которые используются только при инициализации модуля или во время очистки. Объекты, отмеченные для инициализации, могут быть отброшены после завершения инициализации; объекты выхода могут быть отброшены, если выгрузка модулей не была сконфигурирована в ядре. Эти маркеры работают, вызывая соответствующие объекты, находящиеся в специальной секции ELF исполняемого файла.

#include <linux/sched.h>
Один из наиболее важных заголовочных файлов. Этот файл содержит определения многих API ядра, используемых драйвером, в том числе функции для бездействия (sleeping) и декларацию числовых переменных.
struct task_struct *current;
Указатель на текущий процесс.
current->pid
current->comm
ID процесса и имя команды для текущего процесса.
obj-m
Символ в Makefile, используемый системой сборки ядра, чтобы определить, какие модули должны быть построены в текущем каталоге.

/sys/module
/proc/modules
/sys/module является иерархией каталогов ***sysfs***, содержащих информацию о загруженных в данный момент модулях. ***/proc/modules*** является старейшей однофайловой версией этой информации. Записи содержат имя модуля, объём занимаемой каждым модулем памяти и счётчик использования. Чтобы указать флаги, которые в настоящее время активны для модуля, к каждой записи добавляются дополнительные строки.
vermagic.o
Объектный файл из каталога исходных текстов ядра, который описывает окружающую среду, для которой был построен модуль.

#include <linux/module.h>
Обязательный заголовок. Он должен быть подключен в исходник модуля.

#include <linux/version.h>
Заголовочный файл, содержащий информацию о версии ядра, для которого будет собираться модуль.
LINUX_VERSION_CODE
Целочисленный макрос, полезный для построения зависимостей от версии через **#ifdef**.
EXPORT_SYMBOL (symbol);
EXPORT_SYMBOL_GPL (symbol);
Макросы, используемые для экспорта символа в ядро. Первая форма экспортирует без

использования информации о версиях, а вторая ограничивает этот экспорт только областью GPL-лицензированных модулей.

MODULE_AUTHOR(author);
MODULE_DESCRIPTION(description);
MODULE_VERSION(version_string);
MODULE_DEVICE_TABLE(table_info);
MODULE_ALIAS(alternate_name);

Размещение документации о модуле в объектном файле.

MODULE_LICENSE(license);

Объявляет лицензию, контролирующую данный модуль.

#include <linux/moduleparam.h>
module_param(variable, type, nump, perm);

Макрос, который создаёт параметр модуля, который может быть изменён пользователем, когда модуль загружается (или во время запуска для встроенного кода). Тип может быть одним из: bool, charp, int, invbool, long, short, ushort, uint, ulong или intarray.

#include <linux/kernel.h>
int printk(const char * fmt, ...);

Аналог ***printf*** для кода ядра.

Глава 3, Символьные драйверы

Цель этой главы - написать полный символьный драйвер устройства. Мы разрабатываем символьный драйвер, потому что этот класс предназначен для самых простых устройств. Символьные драйверы также легче понять, чем блочные или сетевые драйверы (с которыми мы познакомимся в последующих главах). Наша конечная цель заключается в написании модулизированного символьного драйвера, но мы не будем говорить в этой главе о вопросах модуляризации.

На протяжении всей главы мы представляем фрагменты кода, извлечённые из реального драйвера устройства: ***scull*** (Simple Character Utility for Loading Localities, Простую Символьную Утилиту для Загрузки Местоположений, "череп", так же созвучно school, школа). ***scull*** является символьным драйвером, который оперирует с областью памяти, как будто это устройство. В этой главе мы используем слово ***устройство*** наравне с "область памяти, используемая ***scull***", потому что это особенность ***scull***.

Преимуществом ***scull*** является аппаратная независимость. ***scull*** просто работает с некоторой областью памяти, выделенной ядром. Любой пользователь может скомпилировать и запустить ***scull***, и ***scull*** переносим на различные компьютерные архитектуры, на которых работает Linux. С другой стороны, устройство не делает ничего "полезного", кроме демонстрации интерфейса между ядром и символьными драйверами и возможности пользователю запускать некоторые тесты.

Дизайн scull

Первым шагом написания драйвера является определение возможностей (механизма), которые драйвер будет предлагать пользовательским программам. Так как наше "устройство" является частью памяти компьютера, мы свободны делать всё, что хотим. Это может быть устройство с последовательным или случайным доступом, одно устройство или много, и так далее.

Чтобы сделать ***scull*** полезным в качестве шаблона для написания настоящих драйверов для реальных устройств, мы покажем вам, как реализовать несколько абстракций устройств поверх памяти компьютера, каждая со своими особенностями.

Исходный код ***scull*** реализует следующие устройства. Каждый вид устройства реализуется

модулем соответствующего ***типа***.

scull0 ... scull3
Четыре устройства, каждое содержит область памяти, которая одновременно и глобальная и стойкая. Глобальная означает, что если устройство открыто несколько раз, данные, содержащиеся в устройстве, является общими для всех файловых дескрипторов, которые открыли его. Стойкое означает, что если устройство закрыть и вновь открыть, данные не потеряются. С этим устройством может быть интересно поработать, потому что оно может быть доступно и проверено с помощью обычных команд, таких как ***cp***, ***cat*** и перенаправления ввода/вывода командной оболочки.

scullpipe0 ... scullpipe3
Четыре FIFO (first-in-first-out, первый вошёл-первый вышел) устройства, которые работают как трубы. Один процесс читает то, что другой процесс пишет. Если несколько процессов читают одно устройство, они состязаются за данные. Внутренности ***scullpipe*** покажут, как блокирующие и неблокирующие чтение и запись могут быть реализованы без необходимости прибегать к прерываниям. Хотя реальные драйверы синхронизируются с их устройствами используя аппаратные прерывания, тема блокирующих и неблокирующих операций сама по себе является важной и рассматривается отдельно от обработки прерываний (рассматриваемых в Главе 10 246).

sculsingle
scullpriv
sculluid
scullwuid
Эти устройства похожи на ***scull0***, но с некоторыми ограничениями на разрешение открытия. Первое (***scullsingle***) разрешает использование драйвера только одному процессу, в то время как ***scullpriv*** является единственным для каждой виртуальной консоли (или X терминальной сессии), так как процессы каждой консоли/терминала получают разные области памяти. ***sculluid*** и ***scullwuid*** могут быть открыты несколько раз, но только одним пользователем за раз; первый возвращает ошибку "устройство занято", если другой пользователь блокирует устройство, в то время как второй реализует блокировку открытия. Эти вариации ***scull***, казалось бы, путают политику и механизмы, но они заслуживают рассмотрения, потому что некоторые устройства в реальной жизни требуют такого рода управления.

Каждое из устройств ***scull*** демонстрирует различные функции драйвера и представляет различные трудности. Эта глава охватывает внутреннее строение от ***scull0*** до ***scull3***; более совершенные устройства будут рассмотрены в Главе 6 128. ***scullpipe*** описан в разделе "Пример блокирующего ввода/вывода" 145, а остальные описаны в "Контроль доступа к файлу устройства" 164.

Старший и младший номера устройств

Символьные устройства доступны в файловой системе через имена. Эти имена называются специальными файлами или файлами устройств или просто узлами дерева файловой системы; они обычно находятся в каталоге ***/dev***. Специальные файлы для символьных драйверов идентифицируются по “c” в первом столбце вывода по команде ***ls -l***. Блочные устройства также представлены в ***/dev***, но они идентифицируются по “b”. В центре внимания этой главы символьные устройства, но большая часть следующей информации относится так же и к блочными устройствами.

Если вы введёте команду ***ls -l***, то увидите два числа (разделённые запятой) в каждой записи файла устройства перед датой последней модификации файла, где обычно показывается длина. Эти цифры являются старшим и младшим номером устройства для каждого из них. Следующая распечатка показывает нескольких устройств, имеющихся в типичной системе. Их старшие номера: 1, 4, 7 и 10, а младшие: 1, 3, 5, 64, 65 и 129.

```
crw-rw-rw- 1 root root    1,   3 Apr 11  2002 null
crw------- 1 root root   10,   1 Apr 11  2002 psaux
crw------- 1 root root    4,   1 Oct 28 03:04 tty1
crw-rw-rw- 1 root tty     4,  64 Apr 11  2002 ttys0
crw-rw---- 1 root uucp    4,  65 Apr 11  2002 ttyS1
crw--w---- 1 vcsa tty     7,   1 Apr 11  2002 vcs1
crw--w---- 1 vcsa tty     7, 129 Apr 11  2002 vcsa1
crw-rw-rw- 1 root root    1,   5 Apr 11  2002 zero
```

Традиционно, старший номер идентифицирует драйвер, ассоциированный с устройством. Например, и ***/dev/null*** и ***/dev/zero*** управляются драйвером 1, тогда как виртуальные консоли и последовательные терминалы управляются драйвером 4; аналогично, оба устройства ***vcs1*** и ***vcsa1*** управляются драйвером 7. Современные ядра Linux позволяют нескольким драйверам иметь одинаковые старшие номера, но большинство устройств, которые вы увидите, всё ещё организованы по принципу один-старший-один-драйвер.

Младший номер используется ядром, чтобы точно определить, о каком устройстве идёт речь. В зависимости от того, как написан драйвер (как мы увидим ниже), вы можете получить от ядра прямой указатель на своё устройство или вы можете использовать младший номер самостоятельно в качестве индекса в местном массиве устройств. В любом случае, само ядро почти ничего не знает о младших номерах кроме того, что они относятся к устройствам, управляемым вашим драйвером.

Внутреннее представление номеров устройств

Для хранения номеров устройств, обоих, старшего и младшего, в ядре используется тип **dev_t** (определённый в ***<linux/types.h>***). Начиная с версии ядра 2.6.0, **dev_t** является 32-х разрядным, 12 бит отведены для старшего номера и 20 - для младшего. Ваш код, конечно, никогда не должен делать никаких предположений о внутренней организации номеров устройств; наоборот, он должен использовать набор макросов, находящихся в ***<linux/kdev_t.h>***. Для получения старшей или младшей части **dev_t** используйте:

```
MAJOR(dev_t dev);
MINOR(dev_t dev);
```

Наоборот, если у вас есть старший и младший номера и необходимость превратить их в **dev_t**, используйте:

```
MKDEV(int major, int minor);
```

Заметим, что ядро версии 2.6 может вместить большое количество устройств, в то время как предыдущие версии ядра были ограничены 255-ю старшими и 255-ю младшими номерами. Предполагается, что такого широкого диапазона будет достаточно в течение довольно продолжительного времени, но компьютерная область достаточно усеяна ошибочными предположениями. Таким образом, вы должны ожидать, что формат **dev_t** может снова измениться в будущем; однако, если вы внимательно пишете свои драйверы, эти изменения не

будут проблемой.

Получение и освобождение номеров устройств

Одним из первых шагов, который необходимо сделать вашему драйверу при установке символьного устройства, является получение одного или нескольких номеров устройств для работы с ними. Необходимой функцией для выполнения этой задачи является ***register_chrdev_region***, которая объявлена в ***<linux/fs.h>***:

```
int register_chrdev_region(dev_t first, unsigned int count, char *name);
```

Здесь **first** - это начало диапазона номеров устройств, который вы хотели бы выделить. Младшее число **first** часто 0, но не существует никаких требований на этот счёт. **count** - запрашиваемое общее число смежных номеров устройств. Заметим, что если число **count** большое, запрашиваемый диапазон может перекинуться на следующей старший номер, но всё будет работать правильно, если запрашиваемый диапазон чисел доступен. Наконец, **name** - это имя устройства, которое должно быть связано с этим диапазоном чисел; оно будет отображаться в ***/proc/devices*** и sysfs.

Как и в большинстве функций ядра, возвращаемое значение ***register_chrdev_region*** будет 0, если выделение была успешно выполнено. В случае ошибки будет возвращён отрицательный код ошибки и вы не получите доступ к запрашиваемому региону. ***register_chrdev_region*** работает хорошо, если вы знаете заранее, какие именно номера устройств вы хотите. Однако, часто вы не будете знать, какие старшие номера устройств будут использоваться; есть постоянные усилия в рамках сообщества разработчиков ядра Linux перейти к использованию динамически выделяемых номеров устройств. Ядро будет счастливо выделить старший номер для вас "на лету", но вы должны запрашивать это распределение, используя другую функцию:

```
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int
count, char *name);
```

В этой функции **dev** является только выходным значением, которое при успешном завершении содержит первый номер выделенного диапазона. **firstminor** должен иметь значение первого младшего номера для использования; как правило, 0. Параметры **count** и **name** аналогичны ***register_chrdev_region***.

Независимо от того, как вы назначили номера устройств, вы должны освободить их, когда они больше не используются. Номера устройств освобождаются функцией:

```
void unregister_chrdev_region(dev_t first, unsigned int count);
```

Обычное место для вызова ***unregister_chrdev_region*** будет в функции очистки вашего модуля.

Вышеприведённые функции выделяют номера устройств для использования вашим драйвером, но ничего не говорят ядру, что вы в действительности будете делать с этими номерами. Перед тем, как какая-либо программа из пространства пользователя сможет получить доступ к одному из этих номеров устройств, вашему драйверу необходимо подключить их к своим внутренним функциям, которые осуществляют операции устройства. Мы скоро будем описывать, как это осуществляется, но сначала необходимо сделать несколько необходимых отступлений.

Динамическое выделение старших номеров

Некоторые старшие номера устройств для наиболее распространённых устройств выделены статически. Перечень этих устройств можно найти в ***Documentation/devices.txt*** в дереве исходных текстов ядра. Однако шансы, что статический номер уже был назначен перед использованием нового драйвера, малы и новые номера не назначаются (видимо, не будет назначен, если всё-таки совпадёт?). Так что, автор драйвера, у вас есть выбор: вы можете просто выбрать номер, который кажется неиспользованным, или вы можете определить старшие номера динамическим способом. Выбор номера может работать; пока вы единственный пользователь вашего драйвера; если ваш драйвер распространяется более широко, случайно выбранный старший номер будет приводить к конфликтам и неприятностям.

Таким образом, для новых драйверов мы настоятельно рекомендуем использовать динамическое выделение для получения старшего номера устройства, а не выбирать номер случайно из числа тех, которые в настоящее время свободны. Иными словами, ваш драйвер почти наверняка должен использовать ***alloc_chrdev_region*** вместо ***register_chrdev_region***.

Недостатком динамического назначения является то, что вы не сможете создать узлы устройств заранее, так как старший номер, выделяемый для вашего модуля, будет меняться. Вряд ли это проблема для нормального использования драйвера, потому что после того, как номер был назначен, вы можете прочитать его из ***/proc/devices***. (* Еще большая информация об устройстве обычно может быть получена из sysfs, обычно смонтированной в /sys в системах, базирующихся на ядре 2.6. Обучить *scull* экспортировать информацию через sysfs выходит за рамки данной главы, однако, мы вернёмся к этой теме в Главе 14 347.)

Следовательно, чтобы загрузить драйвер, использующий динамический старший номер, вызов ***insmod*** может быть заменён простым скриптом, который после вызова ***insmod*** читает ***/proc/devices*** в целью создания специального файла (ов).

Типичный файл ***/proc/devices*** выглядит следующим образом:

```
Символьные устройства:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  6 lp
  7 vcs
 10 misc
 13 input
 14 sound
 21 sg
180 usb

Блочные устройства:
  2 fd
  8 sd
 11 sr
 65 sd
 66 sd
```

Следовательно, чтобы извлечь информацию из ***/proc/devices*** для создания файлов в каталоге ***/dev***, скрипт загрузки модуля, которому был присвоен динамический номер, может быть написан с использованием такого инструмента, как ***awk***.

Следующий скрипт, ***scull_load***, является частью дистрибутива ***scull***. Пользователь драйвера, который распространяется в виде модуля, может вызывать такой сценарий из системного файла ***rc.local*** или запускать его вручную каждый раз, когда модуль становится необходим.

```
#!/bin/sh
module="scull"
device="scull"
mode="664"

# вызвать insmod со всеми полученными параметрами
# и использовать имя пути, так как новые modutils не просматривают . по
умолчанию
/sbin/insmod ./$module.ko $* || exit 1

# удалить давно ненужные узлы
rm -f /dev/${device}[0-3]

major=$(awk "\$2==\"$module\" {print \$1}" /proc/devices)

mknod /dev/${device}0 c $major 0
mknod /dev/${device}1 c $major 1
mknod /dev/${device}2 c $major 2
mknod /dev/${device}3 c $major 3

# назначьте соответствующую группу/разрешения, и измените группу.
# не все дистрибутивы имеют "staff", некоторые вместо этого используют
"wheel".
group="staff"
grep -q '^staff:' /etc/group || group="wheel"

chgrp $group /dev/${device}[0-3]
chmod $mode /dev/${device}[0-3]
```

Скрипт может быть адаптирован для других драйверов путём переопределения переменных и корректировке строчек с ***mknod***. Скрипт просто показывает создание четырёх устройств, потому что в исходниках ***scull*** по умолчанию используется число четыре.

Последние несколько строчек скрипта могут показаться неясными: зачем менять группы и режим работы устройства? Причина в том, что скрипт должен запускаться суперпользователем (superuser), так что вновь созданные специальные файлы принадлежат root-у. По умолчанию биты разрешения установлены так, что только root имеет доступ на запись, остальные могут получать доступ на чтение. Как правило, узлы устройств требуют иной политики доступа, так что тем или иным путём права доступа должны быть изменены.

Установки в нашем скрипте предоставляют доступ группе пользователей, но ваши потребности могут отличаться. В разделе "Контроль доступа к файлу устройства" 164 в Главе 6 128 код ***sculluid*** демонстрирует, как драйвер может реализовать свой собственный вид авторизации для доступа к устройству.

Для очистки каталога */dev* и удаления модуля так же доступен скрипт ***scull_unload***.

В качестве альтернативы использования пары скриптов для загрузки и выгрузки вы могли бы написать скрипт инициализации, готовый для размещения в каталоге вашего дистрибутива, используемого для таких скриптов. (* Linux Standard Base указывает, что скрипты инициализации должны быть размещены в */etc/init.d*, но некоторые дистрибутивы всё ещё размещают их в другом месте. Кроме того, если ваш скрипт будет работать во время загрузки, вам необходимо сделать ссылку на него из соответствующей директории уровня загрузки (run-level) (то есть *.../rc3.d*).) Как часть исходников ***scull***, мы предлагаем достаточно полный и настраиваемый пример скрипта инициализации, названного ***scull.init***; он принимает обычные аргументы: старт (start), стоп (stop) и перезагрузка (restart) и выполняет роль как ***scull_load***, так и ***scull_unload***.

Если неоднократное создание и уничтожение узлов */dev* выглядит как излишество, есть полезный обходной путь. Если вы загружаете и выгружаете только один драйвер, после первого создания специальных файлов вашим скриптом вы можете просто использовать ***rmmod*** и ***insmod***: динамические номера не случайны (не рандомизированы) (* Хотя некоторые разработчики ядра угрожали сделать это в будущем.) и вы можете рассчитывать, что такие же номера выбираются каждый раз, если вы не загружаете какие-то другие (динамические) модули. Избегание больших скриптов полезно в процессе разработки. Но очевидно, что этот трюк не масштабируется более чем на один драйвер за раз.

Лучшим способом присвоения старших номеров, на наш взгляд, является использование по умолчанию динамического присвоения, оставив себя возможность указать старший номер во время загрузки или даже во время компиляции. ***scull*** выполняет работу таким образом; он использует глобальную переменную **scull_major**, чтобы сохранить выбранный номер (есть также **scull_minor** для младшего номера). Переменная инициализируется **SCULL_MAJOR**, определённым в ***scull.h***. Значение по умолчанию **SCULL_MAJOR** в распространяемых исходниках равно 0, что означает "использовать динамическое определение". Пользователь может принять значение по умолчанию или выбрать специфичный старший номер либо изменив макрос перед компиляцией, либо указав значение для **scull_major** в командной строки ***insmod***. Наконец, с помощью скрипта ***scull_load*** пользователь может передать аргументы ***insmod*** в командной строке ***scull_load***. (* Инициализационный скрипт *scull.init* не принимает параметры драйвера в командной строке, но он поддерживает конфигурационный файл, потому что разработан для автоматического использования во время запуска и выключения.)

Для получения старшего номера в исходниках ***scull*** используется такой код:

```
if (scull_major) {
    dev = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region(dev, scull_nr_devs, "scull");
} else {
    result = alloc_chrdev_region(&dev, scull_minor, scull_nr_devs, "scull");
    scull_major = MAJOR(dev);
}
if (result < 0) {
    printk(KERN_WARNING "scull: can't get major %d\n", scull_major);
    return result;
}
```

Почти все примеры драйверов, используемых в этой книге, для назначения своего старшего

номера используют аналогичный код.

Некоторые важные структуры данных

Как вы можете себе представить, регистрация номера устройства является лишь первой из многих задач, которые должен выполнять код драйвера. В скором времени мы рассмотрим другие важные компоненты драйвера, но одно отступление необходимо в первую очередь. Большинство основных операций драйвера включает три важных структуры данных ядра, называемые **file_operations**, **file** и **inode**. Требуется базовое знакомство с этими структурами, чтобы быть способным делать много всего интересного, так что сейчас мы бросим быстрый взгляд на каждую из них, прежде чем углубляться в подробности того, как осуществляются основные операции драйвера.

Файловые операции

До этого мы зарезервировали некоторые номера устройств для нашего использования, но мы ещё не подключили никаких своих драйверных операций на эти номера. Структура **file_operations** это то, как символьный драйвер устанавливает эту связь. Структура определена в ***<linux/fs.h>***, это коллекция указателей на функции. Каждое открытие файла (представленное внутри структуры **file**, которую мы рассмотрим в ближайшее время) связано с собственным набором функций (включая поле, названное **f_op**, которое указывает на структуру **file_operations**). Операции в основном отвечают за осуществление системных вызовов и, таким образом, названы ***open*** (открыть), ***read*** (читать) и так далее. Мы можем рассматривать файл как "объект" и функции, работающие с ним, будут его "методами", используя терминологию объектно-ориентированного программирования для обозначения действий, декларируемых исполняющим их объектом. Это первый признак объектно-ориентированного программирования, видимого нами в ядре Linux, и мы увидим больше в последующих главах.

Обычно, структура **file_operations** или указатель на неё называется **fops** (или как-то похоже). Каждое поле структуры должно указывать на функцию в драйвере, которая реализует соответствующие операции, или оставить **NULL** для неподдерживаемых операций. Точное поведение ядра, когда задан указатель **NULL**, отличается для каждой функции, список показан позже в этом разделе.

В приведенном ниже списке приводятся все операции, которые приложение может вызывать на устройстве. Мы постарались сохранить список кратким, поэтому он может быть использован в качестве справки, только кратко описывающим каждую операцию и поведение ядра по умолчанию, когда используется указатель **NULL**.

Прочитав список методов **file_operations**, вы заметите, что некоторое число параметров включает строку **__user**. Эта аннотация является одной из форм документации, отмечая, что указатель является адресом пространства пользователя, который не может быть разыменовываться непосредственно. Для нормальной компиляции **__user** не имеет никакого эффекта, но может быть использована внешним проверяющим программным обеспечением, чтобы найти злоупотребление адресами пользовательского пространства.

Остальная часть главы после описания некоторых других важных структур данных, объясняет роль наиболее важных операций и предлагает подсказки, предостережения и реальные примеры кода. Мы отложим обсуждение наиболее сложных операций для последующих глав, потому что мы ещё совсем не готовы углубиться в такие темы, как

управление памятью, блокирующие операции, а также асинхронные уведомления.

struct module *owner
Первое поле **file_operations** вовсе не операция, это указатель на модуль, который "владеет" структурой. Это поле используется для предотвращения выгрузки модуля во время использования его операций. Почти всегда оно просто инициализируется **THIS_MODULE**, макрос определён в ***<linux/module.h>***.

loff_t (*llseek) (struct file *, loff_t, int);
Метод ***llseek*** используется для изменения текущей позиции чтения/записи в файле, а новая позиция передаётся как (положительное) возвращаемое значение. Параметром **loff_t** является "длинное смещение" и он, по крайней мере, 64 бита даже на 32-х разрядных платформах. Об ошибках сигнализируют отрицательные возвращаемые значения. Если указатель на эту функцию **NULL**, вызовы поиска позиции будут изменять счётчик позиции в структуре **file** (описанной в разделе "Структура file" 50) потенциально непредсказуемым образом.

ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
Используется для получения данных от устройства. Указатель **NULL** в этой позиции заставляет системный вызов ***read*** вернуться с ошибкой **-EINVAL** ("Invalid argument", "Недопустимый аргумент"). Неотрицательное возвращаемое значение представляет собой число успешно считанных байт (возвращаемое значение является типом "размер со знаком", обычно родным (нативным) целочисленным типом для целевой платформы).

ssize_t (*aio_read)(struct kiocb *, char __user *, size_t, loff_t);
Начинает асинхронное чтение - операцию чтения, которая может быть не завершена перед возвратом из функции. Если этот метод **NULL**, все операции будут обрабатываться (синхронно) вместо неё функцией ***read***.

ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
Отправляет данные на устройство. Если **NULL**, **-EINVAL** возвращается в программу, сделавшую системный вызов ***write***. Возвращаемое значение, если оно неотрицательное, представляет собой число успешно записанных байт.

ssize_t (*aio_write)(struct kiocb *, const char __user *, size_t, loff_t *);
Начинает асинхронную операции записи на устройство.

int (*readdir) (struct file *, void *, filldir_t);
Это поле должно быть **NULL** для файлов устройства; оно используется для чтения каталогов и используется только для файловых систем.

unsigned int (*poll) (struct file *, struct poll_table_struct *);
Метод ***poll*** (опрос) является внутренним для трёх системных вызовов: ***poll***, ***epoll*** и ***select***, все они используются для запросов чтения или записи, чтобы заблокировать один или несколько файловых дескрипторов. Метод ***poll*** должен вернуть битовую маску, показывающую, возможны ли неблокирующие чтение или запись, и, возможно, предоставить ядру информацию, которая может быть использована вызывающим процессом для ожидания, когда ввод/вывод станет возможным. Если драйвер оставляет метод ***poll*** **NULL**, предполагается, что устройство читаемо и записываемо без блокировки.

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

Системный вызов ***ioctl*** (управление вводом-выводом) открывает путь к выполнению зависящих от устройства команд (например, форматирование дорожки гибкого диска, которая и не чтение, и не запись). Кроме того, несколько команд ***ioctl*** распознаются ядром без ссылки на таблицу **fops**. Если устройство не обеспечивает метод ***ioctl***, системный вызов возвращает ошибку на любой запрос, который не является предопределенным (**-ENOTTY**, “No such ioctl for device”, "Нет такого управления вводом-выводом для устройства").

int (*mmap) (struct file *, struct vm_area_struct *);

mmap (отобразить в память) используется, чтобы запросить отображение памяти устройства на адресное пространство процесса. Если этот метод **NULL**, системный вызов ***mmap*** возвращает **-ENODEV**.

int (*open) (struct inode *, struct file *);

Хотя это всегда первая операция, выполняющаяся с файлом устройства, драйверу не требуется декларировать соответствующий метод. Если этот параметр **NULL**, открытие устройства всегда успешно, но ваш драйвер не уведомляется.

int (*flush) (struct file *);

Операция ***flush*** (сбросить на диск) вызывается, когда процесс закрывает свою копию файла дескриптора устройства; она должна выполнить (и ожидает это) любую ожидающую выполнения операцию на устройстве. Не следует путать это с операцией ***fsync***, запрашиваемой программами пользователя. В настоящее время ***flush*** используется в очень редких драйверах; например, драйвер ленточного накопителя SCSI использует её, чтобы гарантировать, что все данные записаны на ленту, прежде чем устройство закроется. Если ***flush*** **NULL**, ядро просто игнорирует запрос пользовательского приложения.

int (*release) (struct inode *, struct file *);

Эта операция (отключение) вызывается, когда файловая структура освобождается. Как и ***open***, ***release*** может быть **NULL**. (* Обратите внимание, что *release* не вызывается каждый раз, когда процесс вызывает *close*. Когда файловая структура используется несколькими процессами (например, после fork или dup), *release* не будет вызван, пока не будут закрыты все копии. Если вам необходимо при завершении копирования сбросить на диск ожидающие данные, вы должны реализовать метод *flush*.)

int (*fsync) (struct file *, struct dentry *, int);

Этот метод является внутренним системным вызовом ***fsync***, который пользователь вызывает для сброса на диск любых ожидающих данных. Если этот указатель **NULL**, системный вызов возвращает **-EINVAL**.

int (*aio_fsync)(struct kiocb *, int);

Это асинхронная версия метода ***fsync***.

int (*fasync) (int, struct file *, int);

Эта операция используется, чтобы уведомить устройство, изменив его флаг **FASYNC**. Асинхронные уведомления - сложная тема и она описана в Главе 6 128. Поле может быть **NULL**, если драйвер не поддерживает асинхронные уведомления.

int (*lock) (struct file *, int, struct file_lock *);

Метод ***lock*** (блокировка) используется для реализации блокировки файла; блокировка является неотъемлемой функцией для обычных файлов, но почти никогда не реализуется драйверами устройств.

ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
Эти методы осуществляют разбросанные (scatter/gather, разборка/сборка) операции чтения и записи. Приложениям иногда необходимо сделать одну операцию чтения или записи с участием нескольких областей памяти; эти системные вызовы позволяют им сделать это без необходимости дополнительных операций копирования данных. Если эти указатели на функции остаются **NULL**, вместо них вызываются методы ***read*** и ***write*** (возможно, несколько раз).

ssize_t (*sendfile)(struct file *, loff_t *, size_t, read_actor_t, void *);
Этот метод реализует читающую сторону системного вызова ***sendfile*** (послать файл), который перемещает данные из одного файлового дескриптора на другой с минимумом копирования. Используется, например, веб-сервером, которому необходимо отправить содержимое файла из сети наружу. Драйверы устройств обычно оставляют ***sendfile*** **NULL**.

ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
sendpage (послать страницу) - это другая половина ***sendfile***, она вызывается ядром для передачи данных, одну страницу за один раз, в соответствующий файл. Драйверы устройств, как правило, не реализуют ***sendpage***.

unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
Цель этого метода - найти подходящее место в адресном пространстве процесса, чтобы отобразить в сегменте памяти нижележащее устройство. Эта задача обычно выполняется кодом управления памятью; этот метод существует, чтобы разрешить драйверу реализовать любое согласование требований, которое может иметь специфичное устройство. Большинство драйверов может оставить этот метод **NULL**.

int (*check_flags)(int)
Этот метод позволяет модулю проверить флаги, передаваемые вызову ***fcntl(F_SETFL...)***.

int (*dir_notify)(struct file *, unsigned long);
Этот метод вызывается, когда приложение использует ***fcntl*** для запроса уведомлений об изменении директории. Это полезно только для файловых систем; драйверам нет необходимости реализовывать ***dir_notify***.

Драйвер устройства ***scull*** реализует только самые важные методы устройства. Его структура **file_operations** инициализируется следующим образом:

```
struct file_operations scull_fops = {
    .owner   = THIS_MODULE,
    .llseek  = scull_llseek,
    .read    = scull_read,
    .write   = scull_write,
    .ioctl   = scull_ioctl,
    .open    = scull_open,
    .release = scull_release,
```

```
};
```

Эта декларация использует синтаксис инициализации стандартной маркированной структуры языка Си. Этот синтаксис является предпочтительным, поскольку делает драйвера более переносимыми через изменения в определениях структуры и, возможно, делает код более компактным и читабельным.

Маркированная инициализация разрешает переназначение членов структуры; в некоторых случаях были реализованы существенные улучшения производительности путём размещения указателей к наиболее часто используемым членам в одной строке кэша оборудования.

Структура file

Структура **file**, определённая в ***<linux/fs.h>***, является второй наиболее важной структурой данных, используемой драйверами устройств. Обратите внимание, что **file** не имеет ничего общего с указателями **FILE** программ пространства пользователя. **FILE** определён в библиотеке Си и никогда не появляется в коде ядра. Структура **file**, с другой стороны, это структура ядра, которая никогда не появляется в пользовательских программах.

Структура **file** представляет ***открытый файл***. (Это не специфично для драйверов устройств; каждый открытый файл в системе имеет ассоциированную структуру **file** в пространстве ядра.) Она создаётся ядром при ***открытии*** и передаётся в любую функцию, которая работает с файлом, до последнего ***закрытия***. После закрытия всех экземпляров файла ядро освобождает структуру данных.

В исходных текстах ядра указатель на структуру **file** обычно назван или **file** или **filp** ("указатель на файл"). Мы будем использовать название указателя **filp** для предотвращения путаницы с самой структурой. Таким образом, **file** относится к структуре, а **filp** - указатель на структуру.

Здесь показаны наиболее важные поля структуры **file**. Как и в предыдущем разделе, список может быть пропущен при первом чтении. Тем не менее, позже в этой главе, когда мы столкнёмся с некоторым реальным кодом Си, мы обсудим эти поля более подробно.

mode_t f_mode;
: Определяет режим файла как или читаемый или записываемый (или и то и другое) с помощью битов **FMODE_READ** и **FMODE_WRITE**. Вы можете захотеть проверить это поле для разрешения чтения/записи в своей функции ***open*** или ***ioctl***, но вам не нужно проверять разрешения для ***read*** и ***write***, потому что ядро проверяет это перед вызовом вашего метода. Попытка чтения или записи, если файл не был открыт для этого типа доступа, отклоняется, так что драйвер даже не узнаёт об этом.

loff_t f_pos;
: Текущая позиция чтения или записи. **loff_t** является 64-х разрядным значением на всех платформах (**long long** в терминологии ***gcc***). Драйвер может читать это значение, чтобы узнать текущую позицию в файле, но обычно не должен изменять его; ***read*** и ***write*** должны обновить позицию с помощью указателя, который они получают в качестве последнего аргумента, вместо воздействия на **filp->f_pos** напрямую. Единственным исключением из этого правила является метод ***llseek***, целью которого является изменение позиции файла.

unsigned int f_flags;
Существуют флаги файлов, такие как **O_RDONLY**, **O_NONBLOCK** и **O_SYNC**. Драйвер должен проверить флаг **O_NONBLOCK**, чтобы увидеть, была ли запрошена неблокирующая операция (мы обсудим неблокирующий ввод/вывод в разделе "Блокирующие и неблокирующие операции" 143 в Главе 6 128); другие флаги используются редко. В частности, разрешение на чтение/запись должно быть проверено с помощью **f_mode**, а не **f_flags**. Все флаги определены в заголовке ***<linux/fcntl.h>***.

struct file_operations *f_op;
Операции, связанные с файлом. Ядро присваивает указатель как часть своей реализации ***open***, а затем считывает его, когда необходимо выполнить любые операции. Значение в **filp->f_op** никогда не сохраняется ядром для дальнейшего использования; это означает, что вы можете изменять файловые операции, связанные с вашим файлом, и новые методы будут действовать после вашего возвращения к вызвавшей программе. Например, код для ***open***, связанный со старшим номером 1 (***/dev/null***, ***/dev/zero***, и так далее), заменяет операции в **filp->f_op** в зависимости от открытого младшего номера. Эта практика позволяет реализовать несколько поведений под тем же основным номером без дополнительных накладных расходов при каждом системном вызове. Возможность заменить файловые операции является в ядре эквивалентом "переопределения метода" в объектно-ориентированном программировании.

void *private_data;
Системный вызов ***open*** устанавливает этот указатель в **NULL** для драйвера перед вызовом метода ***open***. Вы можете сделать своё собственное использование поля или игнорировать его; вы можете использовать поле как указатель на выделенные данные, но тогда вы должны помнить об освобождении памяти в методе ***release*** до уничтожения структуры **file** ядром. **private_data** является полезным ресурсом для сохранения информации о состоянии через системные вызовы и используется в большинстве наших примеров модулей.

struct dentry *f_dentry;
Структура элемента каталога (directory entry, **dentry**), связанного с файлом. Авторам драйверов устройств обычно не требуется заботиться о структуре **dentry**, помимо доступа к структуре **inode** через **filp->f_dentry->d_inode**.

Реальная структура имеет несколько больше полей, но они не являются полезными для драйверов устройств. Мы можем смело игнорировать эти поля, поскольку драйверы никогда не создают структуры файлов; они только обращаются к структурам, созданным другими.

Структура inode

Структура **inode** (индексный дескриптор) используется ядром внутренне для представления файлов. Поэтому она отличается от файловой структуры, которая представляет открытый файловый дескриптор. Может быть большое число файловых структур, представляющих собой множество открытых дескрипторов одного файла, но все они указывают на единственную структуру **inode**.

Структура **inode** содержит большое количество информации о файле. По общему правилу только два поля этой структуры представляют интерес для написания кода драйвера:

dev_t i_rdev;

Это поле содержит фактический номер устройства для индексных дескрипторов, которые представляют файлы устройств.

struct cdev *i_cdev;
Структура **cdev** является внутренней структурой ядра, которая представляет символьные устройства; это поле содержит указатель на ту структуру, где **inode** ссылается на файл символьного устройства.

Тип **i_rdev** изменился в течение серии разработки ядра версии 2.5, поломав множество драйверов. В качестве средства поощрения более переносимого программирования разработчики ядра добавили два макроса, которые могут быть использованы для получения старшего и младшего номера **inode**:

```
unsigned int iminor(struct inode *inode);
unsigned int imajor(struct inode *inode);
```

В интересах не быть пойманными следующими изменениями, вместо манипулирования **i_rdev** напрямую должны быть использованы эти макросы.

Регистрация символьных устройств

Как уже упоминалось, ядро использует структуры типа **cdev** для внутреннего представления символьных устройств. Перед тем, как ядро вызовет операции устройства, вы должны выделить и зарегистрировать одну или больше этих структур. (* Существует старый механизм, который избегает применения структур cdev (которые мы обсудим в разделе "Старый способ" 54). Однако, новый код должен использовать новую технику.) Чтобы сделать это, ваш код должен подключить ***<linux/cdev.h>***, где определены структура и связанные с ней вспомогательные функции. Есть два способа создания и инициализации каждой из этих структур. Если вы хотите получить автономную структуру **cdev** во время выполнения, вы можете сделать это таким кодом:

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

Однако, скорее всего, вы захотите вставлять свои собственные устройство-зависимые структуры **cdev**; это то, что делает ***scull***. В этом случае вы должны проинициализировать те структуры, что уже созданы с помощью:

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

В любом случае есть ещё одно поле структуры **cdev**, которое необходимо проинициализировать. Так же как в структуре **file_operations**, структура **cdev** имеет поле **owner** (владелец), которое должно быть установлено в **THIS_MODULE**.

Последний шаг после создания структуры **cdev** - сказать об этом ядру вызовом:

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

Здесь, **dev** является структурой **cdev**, **num** - первый номер устройства, на который реагирует данное устройство, а **count** - количество номеров устройств, которые должны быть связаны с устройством. Часто **count** единица, но бывают ситуации, когда имеет смысл иметь более одного номера устройства, соответствующего определённому устройству. Рассмотрим,

например, драйвер ленточного SCSI накопителя, который позволяет пользовательскому пространству выбирать режимы работы (например, плотность) путём назначения нескольких младших номеров для каждого физического устройства.

При использовании ***cdev_add*** необходимо иметь в виду несколько важных вещей. Первым является то, что этот вызов может потерпеть неудачу. Если он вернул код ошибки, ваше устройство не было добавлено в систему. Однако, это почти всегда удаётся, что вызывает другой момент: как только ***cdev_add*** возвращается, устройство становится "живым" и его операции могут быть вызваны ядром. Вы не должны вызывать ***cdev_add***, пока драйвер не готов полностью проводить операции на устройстве.

Чтобы удалить символьное устройство из системы, вызовите:

```
void cdev_del(struct cdev *dev);
```

Очевидно, что вы не должны обращаться к структуре **cdev** после передачи её в ***cdev_del***.

Регистрация устройства в scull

Внутри ***scull*** представляет каждое устройство структурой типа **scull_dev**. Эта структура определена как:

```
struct scull_dev {
    struct scull_qset *data; /* Указатель, установленный на первый квант */
    int quantum;             /* размер текущего кванта */
    int qset;                /* размер текущего массива */
    unsigned long size;      /* количество данных, хранимых здесь */
    unsigned int access_key; /* используется sculluid и scullpriv */
    struct semaphore sem;    /* семафор взаимного исключения */
    struct cdev cdev;        /* структура символьного устройства */
};
```

Мы обсуждаем различные поля этой структуры, когда приходим к ним, но сейчас мы обращаем внимание на **cdev**, структуру типа **cdev**, которая является интерфейсами нашего устройства к ядру. Эта структура должна быть проинициализирована и добавлена в систему, как описано выше; код ***scull***, который решает эту задачу:

```
static void scull_setup_cdev(struct scull_dev *dev, int index)
{
    int err, devno = MKDEV(scull_major, scull_minor + index);

    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Терпите неудачу изящно, если это необходимо */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

Поскольку структуры **cdev** внедрены в структуру **scull_dev**, чтобы выполнить инициализацию этой структуры должна быть вызвана ***cdev_init*** .

Старый способ

Если вы пороетесь во многих драйверных кодах в ядре версии 2.6, то сможете заметить, что довольно много символьных драйверов не используют интерфейс **cdev**, который мы только что описали. То, что вы видите, является старым кодом, который ещё не был обновлён до интерфейса версии 2.6. Так как этот код работает, это обновление может не произойти в течение длительного времени. Для полноты картины мы описываем старый интерфейс регистрации символьных устройств, но новый код не должен его использовать; этот механизм, вероятно, уйдёт в будущем ядре.

Классический способ зарегистрировать символьный драйвер устройства:

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

Здесь **major** является запрашиваемым старшим номером, **name** - это имя драйвера (оно появляется в */proc/devices*) и **fops** является структурой по умолчанию **file_operations**. Вызов ***register_chrdev*** регистрирует младшие номера 0 - 255 для данного старшего и устанавливает для каждого структуру по умолчанию **cdev**. Драйверы, использующие этот интерфейс, должны быть готовы для обработки вызовов ***open*** по всем 256 младшим номерам (независимо от того, соответствуют ли они реальным устройствам или нет) и они не могут использовать старший или младший номера больше 255.

Если вы используете ***register_chrdev***, правильная функция удаления устройств(а) из системы:

```
int unregister_chrdev(unsigned int major, const char *name);
```

major и **name** должны быть такими же, как передаваемые в ***register_chrdev***, или вызов будет ошибочным.

open и release

Теперь, после быстрого взгляда на поля, мы начнём использовать их в реальных функциях ***scull***.

Метод open

Чтобы сделать любые инициализации при подготовке к поздним операциям для драйвера, предусмотрен метод ***open***. В большинстве драйверов ***open*** должен выполнять следующие задачи:

- Проверку зависимых от устройства ошибок (таких, как "устройство не готово" или аналогичных проблем с оборудованием);
- Проинициализировать устройство, если оно открывается в первый раз;
- Обновить в случае необходимости указатель **f_op**;
- Создать и заполнить любые структуры данных для размещения в **filp->private_data**;

Первым делом, однако, обычно определяется, какое устройство открывается в настоящий момент. Вспомните прототип метода ***open***:

```
int (*open)(struct inode *inode, struct file *filp);
```

Аргумент ***inode*** имеет информацию, которая нам необходима, в форме его поля **i_cdev**, в котором содержится структура **cdev**, которую мы создали раньше. Единственной проблемой является то, что обычно мы не хотим саму структуру **cdev**, мы хотим структуру **scull_dev**, которая содержит эту структуру **cdev**. Язык Си позволяет программистам использовать всевозможные трюки, чтобы выполнить такое преобразование, однако, программирование таких трюков чревато ошибками и приводит к коду, который труден для чтения и понимания другими. К счастью, в данном случае программисты ядра сделали сложную работу за нас, в виде макроса ***container_of***, определённого в ***<linux/kernel.h>***:

```
container_of(pointer, container_type, container_field);
```

Этот макрос получает указатель на поле под названием **container_field**, находящееся внутри структуры типа **container_type**, и возвращает указатель на эту структуру. В ***scull_open*** этот макрос используется, чтобы найти соответствующую структуру устройства:

```
struct scull_dev *dev; /* информация об устройстве */

dev = container_of(inode->i_cdev, struct scull_dev, cdev);
filp->private_data = dev; /* для других методов */
```

Как только он нашел структуру **scull_dev**, для облегчения доступа в будущем ***scull*** сохраняет указатель на неё в поле **private_data** структуры **file**.

Другим способом идентификации открываемого устройства является поиск младшего номера, сохранённого в структуре **inode**. Если вы регистрируете устройство с ***register_chrdev***, вы должны использовать эту технику. Обязательно используйте ***iminor***, чтобы получить младший номер из структуры **inode**, а также убедиться, что он соответствует устройству, которое ваш драйвер действительно готов обслужить.

(Немного упрощённый) код для ***scull_open***:

```
int scull_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev; /* информация об устройстве */

    dev = container_of(inode->i_cdev, struct scull_dev, cdev);
    filp->private_data = dev; /* для других методов */

    /* теперь установим в 0 длину устройства, если открытие было только для
записи */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY) {
        scull_trim(dev); /* игнорирование ошибок */
    }
    return 0; /* успешно */
}
```

Код выглядит очень небольшим, поскольку не делает какого-либо обращения к определённому устройству, когда вызывается ***open***. В этом нет необходимости, потому что устройство ***scull*** по дизайну является глобальным и стойким. В частности, нет никаких действий, таких как "инициализация устройств при первом открытии", поэтому мы не храним

счётчик открытий для устройств ***scull***.

Единственная реальная операция, выполняемая на устройстве, это усечение его длины до 0, когда устройство открывается для записи. Это выполняется, так как по дизайну перезапись устройства ***scull*** более коротким файлом образует более короткую область данных устройства. Это подобно тому, как открытие обычного файла для записи обрезает его до нулевой длины. Операция ничего не делает, если устройство открыто для чтения.

Мы увидим позже, как работает настоящая инициализация, когда посмотрим на код других вариантов ***scull***.

Метод release

Роль метода ***release*** обратна ***open***. Иногда вы обнаружите, что реализация метода называется ***device_close*** вместо ***device_release***. В любом случае, метод устройства должен выполнять следующие задачи:

- Освободить всё, что ***open*** разместил в **filp->private_data**;
- Выключить устройство при последнем закрытии;

Базовая форма ***scull*** не работает с аппаратурой, которую надо выключать, так что код необходим минимальный: (* Другие разновидности устройства закрываются другими функциями, потому что *scull_open* заменяет различные filp->f_op для каждого устройства. Мы будем обсуждать их при знакомстве с каждой разновидностью.)

```
int scull_release(struct inode *inode, struct file *filp)
{
    return 0;
}
```

Вы можете быть удивлены, что происходит, когда файл устройства закрывается большее число раз, чем открывается. В конце концов, системные вызовы ***dup*** и ***fork*** создают копии открытых файлов без вызова ***open***; каждая из этих копий затем закрывается при завершении программы. Например, большинство программ не открывают свой файл ***stdin*** (или устройство), но все они в конечном итоге его закрывают. Как драйвер узнает, когда открытый файл устройства ***действительно*** был закрыт?

Ответ прост: не каждый системный вызов ***close*** вызывает метод ***release***. Только вызовы, которые действительно освобождают структуру данных устройства, запускают этот метод, отсюда его название. Ядро хранит счётчик, сколько раз используется структура **file**. Ни ***fork***, ни ***dup*** не создают новую структуру **file** (только ***open*** делает это); они просто увеличивают счётчик в существующей структуре. Системный вызов ***call*** запускает метод ***release*** только тогда, когда счётчик для структуры **file** падает до 0, что происходит, когда структура уничтожена. Эта взаимосвязь между методом ***release*** и системным вызовом ***close*** гарантирует, что ваш драйвер видит только один вызов ***release*** для каждого ***open***.

Обратите внимание, что метод ***flush*** вызывается каждый раз, когда приложение вызывает ***close***. Однако, очень немногие драйверы реализуют ***flush***, потому что обычно нечего делать во время закрытия, пока не вызван ***release***.

Как вы можете себе представить, предыдущее обсуждение применяется даже тогда, когда приложение завершается без явного закрытия открытых файлов: ядро автоматически

закрывает все файлы во время завершения процесса внутренне с помощью системного вызова ***close***.

Использование памяти в scull

Перед знакомством с операциями ***read*** и ***write*** мы рассмотрим получше, как и почему ***scull*** выполняет выделение памяти. "Как" необходимо, чтобы глубоко понимать код, а "почему" демонстрирует варианты выбора, который должен делать автор драйвера, хотя ***scull***, безусловно, не типичен, как устройство.

Этот раздел имеет дело только с политикой распределения памяти в ***scull*** и не показывает навыки аппаратного управления, необходимые для написания реальных драйверов. Эти навыки будут введены в Главах 9 224 и 10 246. Таким образом, вы можете пропустить этот раздел, если вы не заинтересованы в понимании внутренней работы ориентированного на память драйвера ***scull***.

Область памяти, используемая ***scull***, также называемая устройством, имеет переменную длину. Чем больше вы пишете, тем больше она растёт; укорачивание производится перезаписью устройства более коротким файлом.

Драйвер ***scull*** знакомит с двумя основными функциями, используемыми для управления памятью в ядре Linux. Вот эти функции, определённые в ***<linux/slab.h>***:

```
void *kmalloc(size_t size, int flags);
void kfree(void *ptr);
```

Вызов ***kmalloc*** пытается выделить **size** байт памяти; возвращаемая величина - указатель на эту память или **NULL**, если выделение не удаётся. Аргумент **flags** используется, чтобы описать, как должна быть выделена память; мы изучим эти флаги подробно в Главе 8 203. Сейчас мы всегда используем **GFP_KERNEL**. Выделенная память должна быть освобождена ***kfree***. Вы никогда не должны передавать ***kfree*** что-то, что не было получено от ***kmalloc***. Однако, правомерно передать ***kfree*** указатель **NULL**.

kmalloc это не самый эффективный способ распределения больших областей памяти (смотрите Главу 8 203), поэтому сделанный для ***scull*** выбор не особенно умный. Исходный код для изящной реализации будет более трудным для восприятия, а целью этого раздела является показать ***read*** и ***write***, а не управление памятью. Вот почему код просто использует ***kmalloc*** и ***kfree***, без пересортировки выделенных целых страниц, хотя такой подход был бы более эффективным.

С другой стороны, мы не хотим ограничивать размер области "устройства" по философским и практическим соображениям. Философски, это всегда плохая идея - поставить произвольные ограничения на управляемые объекты данных. Практически, ***scull*** может быть использован для временного поедания всей памяти вашей системы в целях выполнения тестов в условиях малого количества памяти. Выполнение таких тестов могло бы помочь вам понять внутренности системы. Вы можете использовать команду ***cp /dev/zero /dev/scull0***, чтобы съесть всю реальную оперативную память с помощью ***scull***, и вы можете использовать утилиту ***dd***, чтобы выбрать, какой объём данных скопируется на устройство ***scull***.

В ***scull*** каждое устройство представляет собой связный список указателей, каждый из которых указывает на структуру **scull_qset**. По умолчанию каждая такая структура может ссылаться на более чем четыре миллиона байт через массив промежуточных указателей.

Реализованный исходник использует массив из 1000 указателей на области по 4000 байта. Мы называем каждую область памяти ***квантом*** (quantum), а массив (или его длину) - ***набором квантов*** (quantum set). Устройство ***scull*** и его области памяти показаны на Рисунке 3-1.

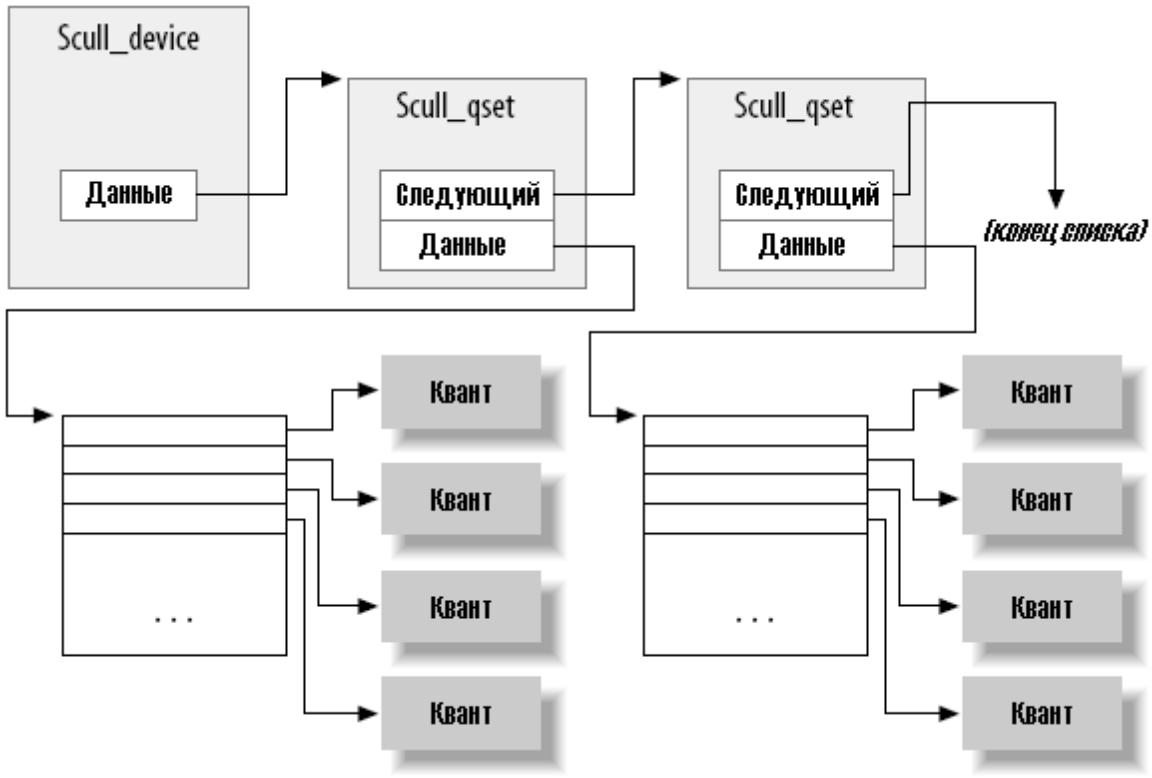


Рисунок 3-1. Схема устройства scull

Выбранные числа таковы, что запись одного байта в ***scull*** потребляет 8000 или 12000 байт памяти: 4000 для кванта и 4000 или 8000 для набора квантов (в зависимости от того, является ли на целевой платформе указатель 32-х разрядным или 64-х разрядным). Если, наоборот, вы пишете большое количество данных, накладные расходы на связный список не такие большие. Существует только один список элементов для каждых четырёх мегабайт данных и максимальный размер устройства ограничен объёмом памяти компьютера.

Выбор соответствующего значения для кванта и квантового набора - это вопрос политики, а не механизма, и их оптимальные размеры зависят от того, как используется устройство. Таким образом, драйвер ***scull*** не должен заставлять использовать какие-то определённые значения для размеров кванта и набора квантов. В ***scull*** пользователь может изменять эти значения несколькими способами: путём изменения макросов **SCULL_QUANTUM** и **SCULL_QSET** в ***scull.h*** во время компиляции, устанавливая целочисленные значения **scull_quantum** и **scull_qset**, во время загрузки модуля, или изменяя текущее значение и значение по умолчанию с помощью ***ioctl*** во время выполнения. Использование макроса и целой величины позволяют выполнять конфигурацию и во время компиляции и во время загрузки и напоминают выбор старшего номера. Мы используем эту технику для любого, произвольного или связанного с политикой, значения в драйвере.

Остался только вопрос, как были выбраны значения по умолчанию. В данном случае проблема состоит в нахождении лучшего соотношения между потерей памяти в результате

полузаполненного кванта и квантового набора, и накладных расходов на выделения, освобождения, и указатель связывания, что происходит, если кванты и наборы малы. Кроме того, должен быть принят во внимание внутренний дизайн ***kmalloc***. (Однако, мы не будем исследовать это сейчас; внутренности ***kmalloc*** рассматриваются в Главе 8 203.) Выбор чисел по умолчанию делается из предположения, что во время тестирования в ***scull*** могут быть записаны большие объёмы данных, хотя при обычном использовании устройства, скорее всего, будут передаваться только несколько килобайт данных.

Мы уже видели структуру **scull_dev**, которая является внутренним представлением нашего устройства. Это поля структуры **quantum** и **qset**, содержащие квант и размер квантового набора устройства, соответственно. Фактические данные, однако, отслеживаются другой структурой, которую мы назвали структурой **scull_qset**:

```
struct scull_qset {
    void **data;
    struct scull_qset *next;
};
```

Следующий фрагмент кода показывает на практике, как структуры **scull_dev** и **scull_qset** используются для хранения данных. Функция ***scull_trim*** отвечает за освобождение всей области данных и вызывается из ***scull_open***, когда файл открывается для записи. Это просто прогулка по списку и освобождение любого кванта и набора квантов при их нахождении.

```
int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset; /* "dev" является не-null */
    int i;
    for (dptr = dev->data; dptr; dptr = next) { /* все элементы списка */
        if (dptr->data) {
            for (i = 0; i < qset; i++)
                kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data = NULL;
        }
        next = dptr->next;
        kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    dev->data = NULL;
    return 0;
}
```

scull_trim также используется в функции очистки модуля, чтобы вернуть системе память, используемую ***scull***.

read и write

Методы ***read*** и ***write*** выполняют аналогичные задачи, то есть копирование данных из и в код приложения. Таким образом, их прототипы очень похожи и стоит представить их в одно время:

```
ssize_t read(struct file *filp, char __user *buff, size_t count, loff_t
*offp);
ssize_t write(struct file *filp, const char __user *buff, size_t count,
loff_t *offp);
```

Для обоих методов **filp** является указателем на **file**, а **count** - это размер запрашиваемой передачи данных. Аргумент **buff** указывает на пользовательский буфер данных, удерживающий данные, которые будут записаны, или пустой буфер, в котором должны быть размещены вновь считанные данные. Наконец, **offp** является указателем на объект "типом длинного смещения" (“long offset type”), который указывает на позицию файла, к которой обращается пользователь. Возвращаемое значение является "типом знакового размера" (“signed size type”); его использование будет рассмотрено позже.

Давайте повторим, что аргумент **buff** для методов ***read*** и ***write*** является указателем пространства пользователя. Поэтому он не может быть непосредственно разыменовываться кодом ядра. Есть несколько причин для этого ограничения:

- В зависимости от архитектуры на которой работает ваш драйвер и как было сконфигурировано ядро, указатель пользовательского пространства может совсем не быть действительным во время работы в режиме ядра. Может не быть ничего связанного с этим адресом, или он может указывать на какие-то другие, случайные данные.

- Даже если указатель означает что-то в пространстве ядра, память пользовательского пространства организована странично и память в запросе может не находиться в ОЗУ, когда сделан этот системный вызов. Попытка сослаться на память пользовательского пространства непосредственно может сгенерировать ошибку страничного доступа, это то, что код ядра не разрешает делать. Результатом будет "Ой" (“oops”), что приведёт к гибели процесса, который сделал этот системный вызов.

- Указатель в запросе поступает от пользовательской программы, которая могла бы быть ошибочной или злонамеренной. Если же ваш драйвер слепо разыменовывает переданный пользователем указатель, он представляет собой открытую дверь, позволяющую программе пользовательского пространства получить доступ или перезаписать память где угодно в системе. Если вы не хотите нести ответственность за ущерб безопасности системам ваших пользователей, вы никогда не можете разыменовывать указатель пользовательского пространства напрямую.

Очевидно, что ваш драйвер должен иметь возможность доступа в буфер пользовательского пространства для того, чтобы выполнить свою работу. Однако, чтобы быть безопасным, этот доступ должен всегда быть выполнен через специальные функции, поставляемые для того ядром. Приведём некоторые из этих функций (которые определены в ***<asm/uaccess.h>***) здесь, а остальные в разделе "Использование аргумента ioctl" 134 в Главе 6 128; они используют некоторую особую, архитектурно-зависимую магию, чтобы гарантировать, что передача данных между ядром и пользовательским пространством происходит безопасным и правильным способом.

Коду для ***read*** и ***write*** в ***scull*** необходимо скопировать весь сегмент данных в или из адресного пространства пользователя. Эта возможность предоставляется следующими функциями ядра, которые копируют произвольный массив байтов и находятся в основе большинства реализаций ***read*** и ***write***:

```
unsigned long copy_to_user(void __user *to,
```

```
                                const void *from,
                                unsigned long count);
unsigned long copy_from_user(void *to,
                                  const void __user *from,
                                  unsigned long count);
```

Хотя эти функции ведут себя как нормальные функции ***memcpy***, следует применять немного дополнительной осторожности, когда из кода ядра адресуется пространство пользователя. Адресуемые пользовательские страницы могут не быть в настоящее время в памяти и подсистема виртуальной памяти может отправить процесс в спячку, пока страница не будет передана на место. Это происходит, например, когда страница должна быть получена из свопа. Конечный результат для автора драйвера в том, что любая функция, которая осуществляет доступ к пространству пользователя, должна быть повторно-входимой, должна быть в состоянии выполняться одновременно с другими функциями драйвера, и, в частности, должна быть в состоянии, когда она может законно спать. Мы вернемся к этому вопросу в Главе 5 124.

Роль этих двух функций не ограничивается копированием данных в и из пространства пользователя: они также проверяют, является ли указатель пользовательского пространства действительным. Если указатель является недействительным, копирование не выполняется, с другой стороны, если неверный адрес встречается во время копирования, копируется только часть данных. В обоих случаях возвращается значение объёма памяти, которое всё же скопировано. Код ***scull*** смотрит на возвращаемую ошибку и возвращает **-EFAULT** пользователю, если значение не 0.

Тема доступа в пространство пользователя и недействительных указателей пространства пользователя более широка и обсуждается в Главе 6 128. Однако, стоит заметить, что если вам не требуется проверять указатель пользовательского пространства, вы можете вызвать взамен ***__copy_to_user*** и ***__copy_from_user***. Это полезно, например, если вы знаете, что уже проверили аргумент. Тем не менее, будьте осторожны, если в действительности вы не проверяете указатель пользовательского пространства, который вы передаёте в эти функции, вы можете создать аварии ядра и/или дыры в системе безопасности.

Что же касается самих методов устройства, задача метода ***read*** - скопировать данные из устройства в пространство пользователя (используя ***copy_to_user***), а метод ***write*** должен скопировать данные из пространства пользователя на устройство (с помощью ***copy_from_user***). Каждый системный вызов ***read*** или ***write*** запрашивает передачу определённого числа байт, но драйвер может свободно передать меньше данных - точные правила немного отличаются для чтения и записи и описаны далее в этой главе.

Независимо от количества переданных методами данных, обычно они должны обновить позицию файла в ***offp**, представляющего текущую позицию в файле после успешного завершения системного вызова. Затем когда требуется, ядро передаёт изменение позиции файла обратно в структуру **file**. Однако, системные вызовы ***pread*** и ***pwrite*** имеют различную семантику; они работают от заданного смещения файла и не изменяют позицию файла, видимую любым другим системным вызовом. Эти вызовы передают указатель на позицию, заданную пользователем, и отменяют изменения, которые делает ваш драйвер. Рисунок 3-2 показывает, как использует свои аргументы типичная реализация ***read***.

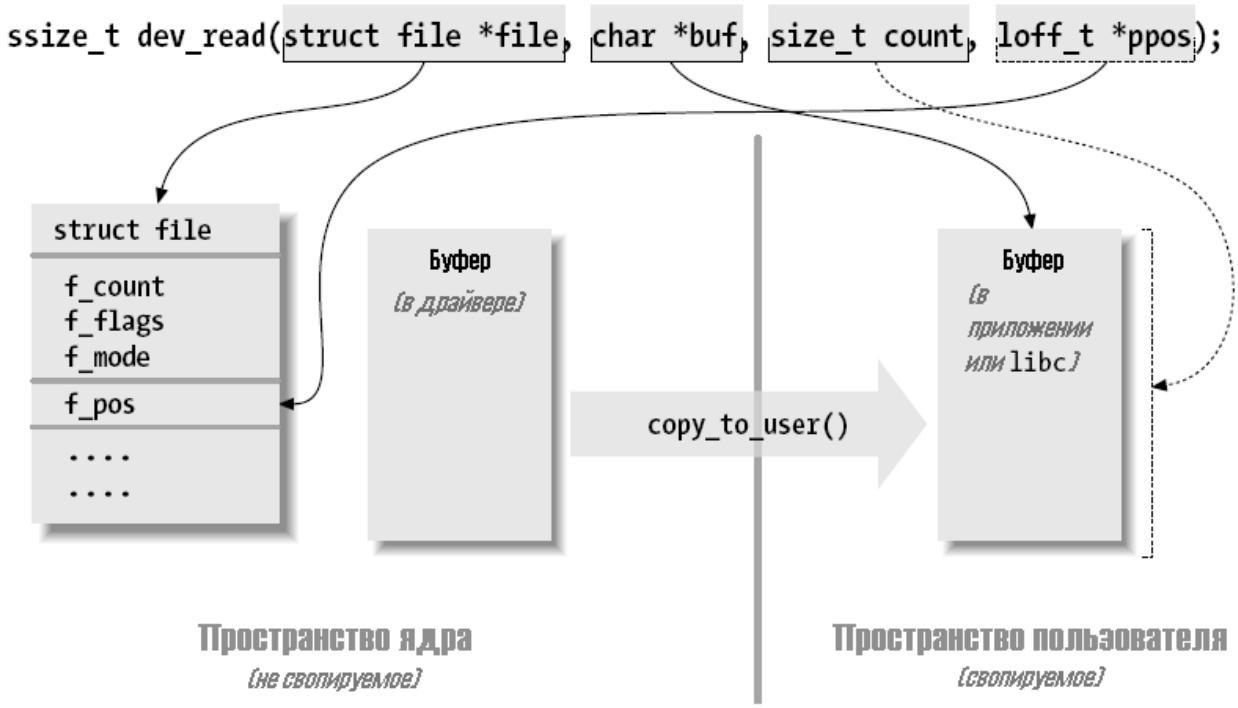


Рисунок 3-2. Аргументы для чтения

Оба метода, ***read*** и ***write***, если произошла ошибка, возвращают отрицательное значение. Вместо этого, возвращаемое значение большее или равное 0, говорит вызывающей программе, сколько байт было успешно передано. Если какие-то данные передаются правильно и затем происходит ошибка, возвращаемое значение должно быть числом успешно переданных байт и об ошибке не сообщается до следующего вызова функции. Конечно, выполнение этого правила требует, чтобы ваш драйвер помнил о том, что произошла ошибка, чтобы он мог вернуть статус ошибки в будущем.

Хотя функции ядра возвращают отрицательное число как сигнал об ошибке, а значение числа показывает вид произошедшей ошибки (как рассказывалось в Главе 2 14), программы, которые выполняются в пользовательском пространстве, всегда видят -1 в качестве возвращаемого ошибочного значения. Они должны получить доступ к переменной **errno**, чтобы выяснить, что случилось. Поведение в пользовательском пространстве диктуется стандартом **POSIX**, но этот стандарт не предъявляет требований к внутренним операциям ядра.

Метод read

Возвращаемое значение для ***read*** интерпретируется вызывающей прикладной программой:

- Если значение равно аргументу **count**, переданному системному вызову ***read***, передано запрашиваемое количество байт. Это оптимальный случай.

- Если число положительное, но меньше, чем **count**, только часть данных была передана. Это может произойти по ряду причин, в зависимости от устройства. Чаще всего прикладная программы повторяет чтение. Например, если вы читаете с помощью функции ***fread***, библиотечная функция повторяет системный вызов до завершения запрошенной передачи данных.

- Если значение равно 0, был достигнут конец файла (и данные не были прочитаны).

- Отрицательное значение означает, что произошла ошибка. Значение указывает, какая была ошибка согласно ***<linux/errno.h>***. Типичные возвращаемые значения ошибки включают **-EINTR** (прерванный системный вызов) или **-EFAULT** (плохой адрес).

Что отсутствует в этом списке, так это случай "нет данных, но они могут прийти позже". В этом случае системный вызов ***read*** должен заблокироваться. Мы будем работать с блокирующим вводом в Главе 6 128.

Код ***scull*** использует эти правила. В частности, он пользуется правилом частичного чтения. Каждый вызов ***scull_read*** имеет дело только с одним квантом данных, не создавая цикл, чтобы собрать все данные; это делает код короче и проще для чтения. Если читающая программа действительно хочет больше данных, она повторяет вызов. Если для чтения устройства используется стандартная библиотека ввода/вывода (например, ***fread***), приложение даже не заметит квантования при передаче данных.

Если текущая позиция чтения больше размера устройства, метод ***read*** драйвера ***scull*** возвращает 0, чтобы сигнализировать, что данных больше нет (другими словами, мы в конце файла). Эта ситуация может произойти, если процесс А читает устройство в то время, как процесс Б открывает его для записи, тем самым укорачивая размер устройства до 0. Процесс вдруг обнаруживает себя в конце файла и следующий вызов ***read*** возвращает 0.

Вот код для ***read*** (игнорируйте сейчас вызовы ***down_interruptible*** и ***up***, мы узнаем о них в следующей главе):

```
ssize_t scull_read(struct file *filp, char __user *buf, size_t count, loff_t
*f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr; /* первый списковый объект */
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset; /* как много байт в списковом объекте */
    int item, s_pos, q_pos, rest;
    ssize_t retval = 0;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    if (*f_pos >= dev->size)
        goto out;
    if (*f_pos + count > dev->size)
        count = dev->size - *f_pos;

    /* найти списковый объект, индекс qset, и смещение в кванте */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* следовать за списком до правой позиции (заданной где-то) */
    dptr = scull_follow(dev, item);

    if (dptr == NULL || !dptr->data || !dptr->data[s_pos])
        goto out; /* не заполнять пустые пространства */
```

```
    /* читать только до конца этого кванта */
    if (count > quantum - q_pos)
        count = quantum - q_pos;

    if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

out:
    up(&dev->sem);
    return retval;
}
```

Присвоения данных для 'quantum' и 'qset' (3-я строка) должны быть защищены семафором, чтобы избежать следующих (правда, маловероятных) состояний гонок:

- Данное устройство открыто для чтения каким-либо процессом.
- Кто-то изменяет значение 'quantum' с помощью ioctl().
- Начинается чтение из устройства, читается его значение 'quantum', но семафор ещё не захвачен.
- Устройство открывается вторым процессом в режиме O_WRONLY, вызывая его повторную инициализацию, что приводит к изменению значения 'quantum' устройства.
- Этот процесс пишет в устройство scull.
- Возобновление первого чтения.

Метод write

write, как и ***read***, может передавать меньше данных, чем было запрошено в соответствии со следующими правилами для возвращаемого значения:

- Если значение равно **count**, передано запрашиваемое количество байт.
- Если число положительное, но меньше, чем **count**, была передана только часть данных. Программа, скорее всего, повторит запись остальных данных.
- Если значение равно 0, ничего не записано. Этот результат не является ошибкой и нет никаких оснований для возвращения кода ошибки. И снова стандартная библиотека повторит вызов ***write***. Мы рассмотрим точное значение этого случая в Главе 6 128, где познакомимся с блокирующей записью.
- Отрицательное значение означает ошибку; как и для ***read***, допустимые значения ошибок определены в ***<linux/errno.h>***.

К сожалению, всё равно могут быть неправильные программы, которые выдают сообщение об ошибке и прерываются, когда производится частичная передача. Это происходит потому, что некоторые программисты привыкли к вызовам ***write***, которые либо ошибочны, либо полностью успешны, это происходит в большом количестве случаев и должно так же поддерживаться устройством. Это ограничение в реализации ***scull*** может быть исправлено, но мы не хотим усложнять код более, чем необходимо.

Код ***scull*** для ***write*** выполняется с одним квантом за раз, так же, как это делает метод ***read***:

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
loff_t *f_pos)
{
    struct scull_dev *dev = filp->private_data;
    struct scull_qset *dptr;
    int quantum = dev->quantum, qset = dev->qset;
    int itemsize = quantum * qset;
    int item, s_pos, q_pos, rest;
    ssize_t retval = -ENOMEM; /* значение используется в инструкции "goto
out" */

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    /* найти списковый объект, индекс qset, и смещение в кванте */
    item = (long)*f_pos / itemsize;
    rest = (long)*f_pos % itemsize;
    s_pos = rest / quantum; q_pos = rest % quantum;

    /* следовать за списком до правой позиции */
    dptr = scull_follow(dev, item);
    if (dptr = = NULL)
        goto out;
    if (!dptr->data) {
        dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
        if (!dptr->data)
            goto out;
        memset(dptr->data, 0, qset * sizeof(char *));
    }
    if (!dptr->data[s_pos]) {
        dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
        if (!dptr->data[s_pos])
            goto out;
    }
    /* записать только до конца этого кванта */
    if (count > quantum - q_pos)
        count = quantum - q_pos;
    if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
        retval = -EFAULT;
        goto out;
    }
    *f_pos += count;
    retval = count;

    /* обновить размер */
    if (dev->size < *f_pos)
        dev->size = *f_pos;

out:
    up(&dev->sem);
    return retval;
}
```

Функции readv и writev

Системы Unix уже давно поддерживают два системных вызова, названных ***readv*** и ***writev***. Эти "векторные" версии ***read*** и ***write*** берут массив структур, каждая из которых содержит указатель на буфер и значение длины. Вызов ***readv*** будет читать указанное количество в каждый буфер по порядку. ***writev***, вместо этого, будет собирать вместе содержимое каждого буфера и передавать их наружу как одну операцию записи.

Если драйвер не предоставляет методы для обработки векторных операций, ***readv*** и ***writev*** осуществляются несколькими вызовами ваших методов ***read*** и ***write***. Однако во многих случаях путём прямой реализации ***readv*** и ***writev*** достигается повышение эффективности.

Прототипами векторных операций являются:

```
ssize_t (*readv) (struct file *filp, const struct iovec *iov, unsigned long
count, loff_t *ppos);
ssize_t (*writev) (struct file *filp, const struct iovec *iov, unsigned long
count, loff_t *ppos);
```

Здесь аргументы **filp** и **ppos** такие же, как для ***read*** и ***write***. Структура **iovec**, определённая в ***<linux/uio.h>***, выглядит следующим образом:

```
struct iovec
{
    void __user *iov_base;
    __kernel_size_t iov_len;
};
```

Каждая **iovec** описывает одну порцию данных для передачи; она начинается в **iov_base** (в пространстве пользователя) и имеет длину **iov_len** байт. Параметр **count** говорит методу, сколько существует структур **iovec**. Эти структуры созданы приложением, но ядро копирует их в пространство ядра до вызова драйвера.

Простейшей реализацией векторных операций будет прямой цикл, который просто передаёт адрес и длину каждой **iovec** функциям драйвера ***read*** или ***write***. Часто, однако, эффективное и правильное поведение требует, чтобы драйвер делал что-то поумнее. Например, ***writev*** на ленточном накопителе должна сохранить содержимое всех структур **iovec** одной операцией записи на магнитную ленту.

Однако, многие драйверы не получили никакой выгоды от реализации самими этих методов. Таким образом, ***scull*** опускает их. Ядро эмулирует их с помощью ***read*** и ***write***, и конечный результат тот же.

Игра с новым устройством

Как только вы оснащены четырьмя вышеописанными методами, драйвер может быть собран и протестирован; он сохраняет любые данные, которые вы запишете в него, пока вы не перезапишете их новыми данными. Устройство действует как буфер данных, размер которого ограничен только реально доступной памятью. Для проверки драйвера вы можете попробовать использовать ***cp***, ***dd*** и перенаправление ввода/вывода.

Чтобы увидеть, как сжимается и расширяется объём свободной памяти в зависимости от того, как много данных записано в ***scull***, может быть использована команда ***free***.

Чтобы стать более уверенными при чтении и записи одного кванта за раз, вы можете добавить ***printk*** в соответствующую точку в драйвере и посмотреть, что происходит в то время, как приложение читает или записывает большие массивы данных. Альтернативно, используйте утилиту ***strace*** для мониторинга системных вызовов вместе с их возвращаемыми значениями, выполняемыми программой. Трассировка ***cp*** или ***ls -l > /dev/scull0*** показывает квантованные чтения и записи. Техники мониторинга (и отладки) подробно изложены в Главе 4 69.

Краткая справка

Эта глава ввела следующие символы и файлы заголовков. Списки полей в структуре **file_operations** и структуре **file** здесь не повторяются.

#include <linux/types.h>

dev_t

dev_t является типом, используемым для представления в ядре номера устройства.

int MAJOR(dev_t dev);

int MINOR(dev_t dev);

Макросы для получения старших и младших чисел из номера устройства.

dev_t MKDEV(unsigned int major, unsigned int minor);

Макрос, который строит элемент данных **dev_t** из старших и младших чисел.

#include <linux/fs.h>

Заголовок “filesystem” является заголовком, необходимым для написания драйверов устройств. Здесь декларируются многие важные функции и структуры данных.

int register_chrdev_region(dev_t first, unsigned int count, char *name)

int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int count, char *name)

void unregister_chrdev_region(dev_t first, unsigned int count);

Функции, которые позволяют драйверу выделять и освобождать диапазоны номеров устройств. ***register_chrdev_region*** должна быть использована, если старший желаемый номер известен заранее, для динамического выделения взамен используйте ***alloc_chrdev_region***.

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

Старая процедура (до 2.6) регистрации символьного устройства. Она эмулируется в ядре 2.6, но не должна быть использована для нового кода. Если старший номер не 0, он используется без изменений; в противном случае устройству присваивается динамический номер.

int unregister_chrdev(unsigned int major, const char *name);

Функция, которая отменяет регистрацию, сделанную с помощью ***register_chrdev***. Обе величины, **major** и строка **name**, должны содержать те же значения, которые были использованы для регистрации драйвера.

struct file_operations;

struct file;

struct inode;

Три важные структуры данных, используемые большинством драйверов устройств. Структура **file_operations** содержит методы символьного драйвера; структура **file**

представляет собой открытый файл, а структура **inode** представляет собой файл на диске.

#include <linux/cdev.h>
struct cdev *cdev_alloc(void);
void cdev_init(struct cdev *dev, struct file_operations *fops);
int cdev_add(struct cdev *dev, dev_t num, unsigned int count);
void cdev_del(struct cdev *dev);

Функции управления структурами **cdev**, которые представляют в ядре символьные устройства.

#include <linux/kernel.h>
container_of(pointer, type, field);

Удобный макрос, который может быть использован для получения указателя на структуру из указателя на другие структуры, содержащиеся в нём.

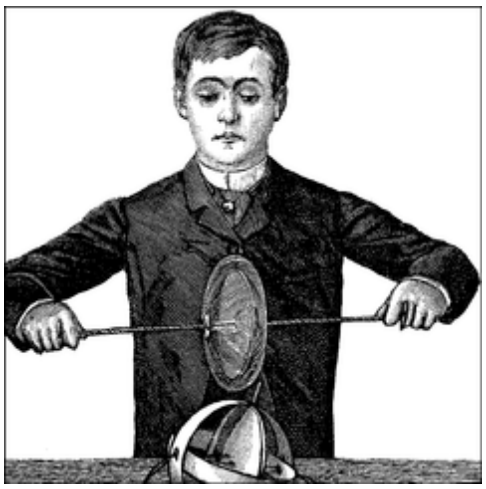
#include <asm/uaccess.h>

Подключает декларации файлов функций, используемых кодом ядра для перемещения данных в и из пространства пользователя.

unsigned long copy_from_user (void *to, const void *from, unsigned long count);
unsigned long copy_to_user (void *to, const void *from, unsigned long count);

Копирование данных между пользовательским пространством и пространством ядра.

Глава 4, Техники отладки



Программирования ядра приносит свои собственные, уникальные проблемы отладки. Код ядра не может быть легко выполнен под отладчиком, не может быть легко оттрассированным, потому что он представляет собой набор функциональных возможностей, не связанных с конкретным процессом. Ошибки кода ядра может быть чрезвычайно трудно воспроизвести и можно сломать ими всю систему, тем самым уничтожив большую часть данных, которые могли быть использованы для их поиска.

Эта глава представляет технологии, которые можно использовать для контроля кода ядра и поиска ошибок в таких сложных ситуациях.

Поддержка отладки в ядре

В Главе 2 14 мы рекомендовали вам собрать и установить собственное ядро, а не запускать поставляемое ядро, которое пришло с вашего дистрибутива. Одной из главных причин для запуска собственного ядра является то, что разработчики ядра встроили некоторые возможности отладки в само ядро. Эти функции могут создать дополнительный вывод и уменьшить производительность, поэтому они чаще всего не включены в ядрах, собранных дистрибьюторами. Как разработчик ядра, однако, вы имеете другие приоритеты и с радостью примете (минимальные) накладные расходы дополнительной поддержки отладки в ядре.

Здесь перечислены параметры конфигурации, которые должны быть включены для ядер, использующихся для разработки. Если не указано другое, все эти параметры находятся в меню “kernel hacking” ("взламывание ядра") любой утилиты конфигурации ядра. Заметим, что некоторые из этих параметров не поддерживаются всеми архитектурами.

CONFIG_DEBUG_KERNEL
Этот параметр просто доступными делает другие опции отладки; он должен быть включен, но сам по себе не включает какие-то возможности.

CONFIG_DEBUG_SLAB
Этот важнейший параметр включает несколько видов проверок в функциях выделения памяти ядра; при включении этих проверок можно обнаружить ошибки переполнения

памяти и отсутствие инициализации. Каждый байт выделенной памяти устанавливается в **0xa5** перед передачей вызывающему и устанавливается в **0x6b**, когда освобождается. Если вы когда-либо увидите эти повторяющиеся "ядовитые" шаблоны при выводе из вашего драйвера (или часто в распечатках Oops), вы будете точно знать, какую ошибку искать. При включенной отладке ядро также размещает особые защитные значения до и после каждой выделенной объекту памяти; если эти значения оказываются изменёнными, ядро знает, что кто-то вызвал переполнение памяти, и оно громко выражает недовольство. Так же включаются различные проверки на более малоизвестные ошибки.

CONFIG_DEBUG_PAGEALLOC
Полные страницы при освобождении удаляются из адресного пространства ядра. Эта опция может значительно замедлить ход событий, но она может также быстро указать на некоторые виды ошибок повреждения памяти.

CONFIG_DEBUG_SPINLOCK
Когда эта опция включена, ядро улавливает операции на неинициализированных спин-блокировках и другие различные ошибки (такие, как двойная разблокировка).

CONFIG_DEBUG_SPINLOCK_SLEEP
Эта опция позволяет проверить наличие попытки заснуть в момент удержания спин-блокировки. На самом деле, оно жалуется, если вы вызываете функцию, которая может потенциально заснуть, даже если вызов в запросе не будет спать.

CONFIG_INIT_DEBUG
Отмеченное **__init** (или **__initdata**) удаляется после инициализации системы или загрузки модуля. Эта опция позволяет проверить код, который пытается получить доступ к памяти, используемой во время инициализации, после завершения инициализации.

CONFIG_DEBUG_INFO
Эта опция заставляет ядро быть собранным с подключенной полной отладочной информацией. Вам необходима эта информация, если вы хотите отлаживать ядро с ***gdb***. Вы также можете захотеть включить **CONFIG_FRAME_POINTER**, если планируете использовать ***gdb***.

CONFIG_MAGIC_SYSRQ
Разрешает кнопку “системный SysRq” (“magic SysRq”). Мы рассмотрим эту кнопку позже в этой главе в разделе "Зависания системы" 92.

CONFIG_DEBUG_STACKOVERFLOW
CONFIG_DEBUG_STACK_USAGE
Эти опции могут помочь отследить переполнение стека ядра. Явным признаком переполнения стека является листинг Oops без какого-либо признака разумной обратной трассировки. Первая опция добавляет явные проверки переполнения в ядре; вторая заставляет ядро контролировать использование стека и делать доступной некоторую статистику, доступную через системную кнопку SysRq.

CONFIG_KALLSYMS
Этот параметр (в настройках “General setup/Standard features”, "Общая настройка/Стандартные возможности") заставляет информацию символов ядра быть встроенной в ядро; он включен по умолчанию. Информация о символах используется в контекстах

отладки; без неё листинг Oops может дать вам обратную трассировку ядра только в шестнадцатеричном виде, что не очень полезно.

CONFIG_IKCONFIG
CONFIG_IKCONFIG_PROC
Эти параметры (находятся в меню “General setup”, "Общая настройка") заставляют быть встроенным в ядро полное состояние конфигурации ядра и делают его доступным через */proc*. Большинство разработчиков ядра знают, какие конфигурации они использовали, и не нуждаются в этих опциях (которые делают ядро больше). Хотя они могут быть полезны, если вы пытаетесь найти проблему в ядре, собранном кем-то другим.

CONFIG_ACPI_DEBUG
Находится в “Power management/ACPI” ("Управление питанием/ACPI"). Эта опция включает подробную отладочную информацию ACPI (Advanced Configuration and Power Interface), которая может быть полезна, если вы подозреваете, что проблема связана с ACPI.

CONFIG_DEBUG_DRIVER
Находится в “Device drivers” ("Драйверы устройств"). Включает отладочную информацию в драйверном ядре, которая может быть полезной для отслеживания проблем в низкоуровневом коде поддержки. Мы рассмотрим драйверное ядро в Главе 14 347.

CONFIG_SCSI_CONSTANTS
Это параметр, находящийся в “Device drivers/SCSI device support” ("Драйверы устройств/ поддержка SCSI устройств"), встраивается в информацию для подробных сообщений об ошибках SCSI. Если вы работаете над драйвером SCSI, вы, вероятно, захотите включить эту опцию.

CONFIG_INPUT_EVBUG
Эта опция (в разделе “Device drivers/Input device support”, "Драйверы устройств/ Поддержка устройств ввода") включает подробное логирование входных событий. Если вы работаете над драйвером для устройства ввода, эта опция может быть полезной. Однако, будьте осведомлены о последствиях для безопасности: логируется всё, что вводится, включая ваши пароли.

CONFIG_PROFILING
Эта опция находится в разделе “Profiling support” ("Поддержка профилирования"). Профилирование обычно используется для настройки производительности системы, но оно также может быть полезно для отслеживания некоторых зависаний ядра и связанных с ними проблем.

Мы вернёмся к некоторым из вышеуказанных опций при рассмотрении различных способов отслеживания проблем с ядром. Но сначала мы рассмотрим классическую технику отладки: печать отчётов.

Отладка через печать

Наиболее общей техникой отладки является мониторинг, который в прикладном программировании осуществляется вызовом в соответствующих местах ***printf***. При отладке кода ядра вы можете достичь той же цели с помощью ***printk***.

printk

Мы использовали функцию ***printk*** в предыдущих главах с упрощающим предположением, что она работает как ***printf***. Теперь пришло время познакомиться с некоторыми отличиями.

Одним из отличий является то, что ***printk*** позволяет классифицировать сообщения в зависимости от их серьёзности, связывая разные ***уровни логирования*** (loglevels) или приоритеты, с сообщениями. Вы обычно указываете уровень логирования макросом. Например, **KERN_INFO**, который мы видели предшествующим чему-либо в приведённых раннее вызовах печати, является одним из возможных уровней логирования сообщений. Макрос уровня логирования заменяется на строку, которая объединяется с текстом сообщения во время компиляции; вот почему в следующих примерах нет запятой между приоритетом и строкой форматирования. Вот два примера команд ***printk***, отладочное сообщение и критическое сообщение:

```
printk(KERN_DEBUG "Here I am: %s:%i\n", __FILE__, __LINE__);
printk(KERN_CRIT "I'm trashed; giving up on %p\n", ptr);
```

Есть восемь возможных строк уровня логирования, определённых в заголовке ***<linux/kernel.h>***; мы перечисляем их в порядке убывания серьёзности:

KERN_EMERG
: Используется для аварийных сообщений, как правило, тем, которые предшествуют катастрофе.

KERN_ALERT
: Ситуации, требующей немедленных действий.

KERN_CRIT
: Критические условия, часто связанные с серьёзными аппаратными или программными сбоями.

KERN_ERR
: Используется, чтобы сообщить об ошибочных условиях; драйверы устройств часто используют **KERN_ERR** для сообщения об аппаратных проблемах.

KERN_WARNING
: Предупреждения о проблемных ситуациях, которые сами по себе не создают серьёзных проблем с системой.

KERN_NOTICE
: Ситуации, которые являются нормальными, но всё же достойны внимания. Различные обстоятельства, относящиеся к безопасности, сообщаются с этим уровнем.

KERN_INFO
: Информационные сообщения. Многие драйверы печатают информацию об оборудовании, которую они находят во время запуска с этим уровнем.

KERN_DEBUG
: Используется для отладочных сообщений.

Каждая строка (в макроподстановках) представляет собой целое число в угловых скобках. Целые числа в диапазоне от 0 до 7, чем меньше величина, тем больше приоритет.

Вызов ***printk*** без заданного приоритета использует **DEFAULT_MESSAGE_LOGLEVEL**, определённый в ***kernel/printk.c*** как целое число. В ядре версии 2.6.10, **DEFAULT_MESSAGE_LOGLEVEL** является **KERN_WARNING**, но это, как известно, менялось в прошлом.

На основании уровня логирования ядро может печатать сообщения в текущей консоли, будь то текстовый терминал, последовательный порт, или параллельный принтер. Если приоритет меньше целой переменной **console_loglevel**, сообщение будет доставлено на консоль по одной строке за раз (не посылается ничего, пока не получен завершающий символ "новая строка"). Если в системе работают и ***klogd*** и ***syslogd***, сообщения ядра добавляются в ***/var/log/messages*** (или другой, в зависимости от вашей конфигурации ***syslogd***), независимо от **console_loglevel**. Если ***klogd*** не работает, сообщение не дойдёт до пространства пользователя, пока вы не прочитаете ***/proc/kmsg*** (что часто проще всего сделать командой ***dmesg***). При использовании ***klogd*** вы должны помнить, что он не сохраняет одинаковые следующие друг за другом строки; он сохраняет только первую такую линию, а позднее - количество полученных повторов.

Переменная **console_loglevel** инициализируется с **DEFAULT_CONSOLE_LOGLEVEL** и может быть изменена с помощью системного вызова ***sys_syslog***. Один из способов изменить это - задать значение ключом ***-c*** при вызове ***klogd***, как указано в справке ***klogd***. Обратите внимание, что для изменения текущего значения, вы должны сначала убить ***klogd***, а затем перезапустить его с опцией ***-c***. Кроме того, вы можете написать программу для изменения уровня логирования консоли. Вы найдёте вариант такой программы в ***misc-progs/setlevel.c*** в исходных файлах, находящихся на FTP сайте O'Reilly. Новый уровень определяется как целое число между 1 и 8, включительно. Если он установлен в 1, консоли достигают только сообщения уровня 0 (**KERN_EMERG**), если он установлен в 8, отображаются все сообщения, включая отладочные.

Также можно читать и модифицировать уровень логирования консоли с помощью текстового файла ***/proc/sys/kernel/printk***. Файл содержит четыре целых величины: текущий уровень логирования, уровень по умолчанию для сообщений, которые не имеют явно заданного уровня логирования, минимально разрешённый уровень логирования и уровень логирования по умолчанию во время загрузки. Запись одного значения в этот файл изменяет текущий уровень логирования на это значение; так, например, вы можете распорядиться, чтобы все сообщения ядра появлялись в консоли, простым вводом:

```
# echo 8 > /proc/sys/kernel/printk
```

Сейчас должно быть понятно, почему пример ***hello.c*** имел маркеры **KERN_ALERT**; они применялись для уверенности, что сообщение появится в консоли.

Перенаправление сообщений консоли

Linux обеспечивает определённую гибкость в политиках консольного логирования, позволяя отправлять сообщения на заданную виртуальную консоль (если ваша консоль живёт на текстовом экране). По умолчанию, "консолью" является текущий виртуальный терминал. Чтобы выбрать для приёма сообщений другой виртуальный терминал, вы можете на любом

консольном устройстве вызвать **ioctl(TIOCLINUX)**. Следующая программа, ***setconsole***, может быть использована для выбора консоли, получающей сообщения ядра; она должна быть запущена суперпользователем и находиться в каталоге ***misc-progs***.

Ниже приводится программа в полном объёме. Вы должны вызывать её с одним аргументом, указывающим номер консоли, которая будет получать сообщения.

```
int main(int argc, char **argv)
{
    char bytes[2] = {11,0}; /* 11 - номер команды TIOCLINUX */

    if (argc==2) bytes[1] = atoi(argv[1]); /* выбранная консоль */
    else {
        fprintf(stderr, "%s: need a single arg\n",argv[0]); exit(1);
    }
    if (ioctl(STDIN_FILENO, TIOCLINUX, bytes)<0) { /* использовать stdin */
        fprintf(stderr,"%s: ioctl(stdin, TIOCLINUX): %s\n",
        argv[0], strerror(errno));
        exit(1);
    }
    exit(0);
}
```

setconsole использует специальную команду ***ioctl*** **TIOCLINUX**, которая реализует особые функции Linux. Для использования **TIOCLINUX** вы передаёте аргумент, который является указателем на массив байтов. Первый байт массива является числом, которое определяет запрошенную подкоманду, а следующий байт зависит от подкоманды. В ***setconsole*** используется подкоманда 11, а следующий байт (хранимый в **bytes[1]**) идентифицирует виртуальную консоль. Полное описание **TIOCLINUX** может быть найдено в исходных текстах ядра в ***drivers/char/tty_io.c***.

Как получить сообщения из лога

Функция ***printk*** записывает сообщения в круговой буфер размером **__LOG_BUF_LEN** байт: значение от 4 Кб до 1 Мб выбирается при конфигурации ядра. Эта функция затем будит любой процесс, который ожидает сообщений, то есть любой процесс, который является заснувшим при вызове системного вызова ***syslog*** или при чтении ***/proc/kmsg***. Эти два интерфейса движка логирования почти эквивалентны, но учтите, что чтение ***/proc/kmsg*** забирает данные из буфера журнала, а системный вызов ***syslog*** может произвольно возвращать данные протокола, оставляя их для других процессов. В общем, чтение файла ***/proc*** проще и является поведением по умолчанию для ***klogd***. Чтобы взглянуть на содержание буфера без сброса его на диск, может быть использована команда ***dmesg***; на деле команда возвращает в ***stdout*** всё содержимое буфера, независимо от того, читался ли он до этого.

Если вам случится читать сообщения ядра вручную, после остановки ***klogd***, вы увидите, что файл ***/proc*** выглядит как FIFO, в котором читатель блокируется, ожидая данных. Очевидно, что нельзя прочитать сообщения таким образом, если ***klogd*** или другой процесс уже читает эти же данные, так как вы будете соперничать.

Если круговой буфер заполнится, ***printk*** вернётся в начало и начнёт добавлять новые данные с начала буфера, перезаписывая старые данные. Таким образом, процесс протоколирования теряет старые данные. Эта проблема незначительна по сравнению с

преимуществами использования кругового буфера. Например, циклический буфер позволяет системе работать даже без процесса протоколирования, минимизируя потерю памяти и перезаписывая старые данные, которые никто не прочитает. Ещё одна особенность подхода Linux для обмена сообщениями, это то, что ***printk*** может быть вызвана из любого места, даже из обработчика прерываний, без ограничения на размер распечатываемых данных. Единственным недостатком является возможность потери некоторых данных.

Если процесс ***klogd*** запущен, он получает сообщения ядра и передаёт их ***syslogd***, который, в свою очередь, проверяет ***/etc/syslog.conf***, чтобы выяснить, как с ними поступать. ***syslogd*** делает различия между сообщениями в соответствии с видом приложения и приоритетом; допустимые значения видов и приоритетов определены в заголовочном файле ***<sys/syslog.h>***. Сообщения ядра протоколируются со значением типа **LOG_KERN** и соответствующим ему приоритетом, используемым в ***printk*** (например, **LOG_ERR** используется для сообщений **KERN_ERR**). Если ***klogd*** не работает, данные остаются в круговом буфере, пока кто-то не прочитает их или буфер не переполнится. Если вы хотите избежать затирания системного лога мониторинговыми сообщениями от вашего драйвера, вы можете либо указать опцию ***-f*** (файл) для ***klogd***, чтобы поручить ему сохранять сообщения в заданный файл, или изменить ***/etc/syslog.conf*** в соответствии с вашими требованиями. Ещё одна возможность использовать метод решения "в лоб": убить ***klogd*** и многословно печатать сообщения на неиспользуемых виртуальных терминалах (* Например, используйте setlevel 8; setconsole 10, чтобы установить терминал 10 для отображения сообщений.) или дать команду ***cat /proc/kmsg*** из неиспользуемого ***xterm***.

Включение и выключение сообщений

На ранних этапах разработки драйвера ***printk*** может оказать существенную помощь при отладке и тестировании нового кода. С другой стороны, когда вы официально опубликуете драйвер, вы должны удалить или, по крайней мере, отключить такую печать. К сожалению, вы, вероятно, обнаружите, что как только вы подумаете, что сообщения вам больше не нужны и удалите их, вы добавите новую функцию в драйвер (или кто-то найдёт ошибку) и вы захотите вернуть обратно по крайней мере одно сообщение. Есть несколько путей решения обоих вопросов: глобально разрешать или запрещать отладочные сообщения, и включать или выключать отдельные сообщения.

Здесь мы покажем один из способов кодирования вызовов ***printk*** так, что вы сможете включать и выключать их индивидуально или глобально; техника зависит от определения макроса, который разрешает вызов ***printk*** (или ***printf***), когда вы хотите, чтобы:

- Каждый оператор печати мог быть включен или отключен удалением или добавлением одной буквы к имени макроса.
- Все сообщения могли быть отключены сразу изменением значения переменной **CFLAGS** перед компиляцией.
- Такое же управление печатью могло бы быть использовано и в коде ядра и в коде пользовательского уровня, так что драйвер и тестовые программы могли бы управляться одним способом относительно дополнительных сообщений.

Эти возможности реализует следующий фрагмент кода, взятый прямо из заголовка ***scull.h***:

```
#undef PDEBUG /* уберём определение на всякий случай */
#ifdef SCULL_DEBUG
# ifdef __KERNEL__
    /* используется, если включена отладка и пространство ядра */
```

```
# define PDEBUG(fmt, args...) printk( KERN_DEBUG "scull: " fmt, ## args)
# else
    /* используется для пользовательского пространства */
# define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) /* не отлаживаемся: ничего */
#endif

#undef PDEBUGG
#define PDEBUGG(fmt, args...) /* ничего: это пустышка */
```

Символ **PDEBUG** определяется или не определяется, в зависимости от того, определён ли **SCULL_DEBUG**, и отображает информацию в форме, подходящей для той среды, где выполняется код: он использует вызов ядра ***printk*** в пространстве ядра и вызов ***fprintf*** библиотеки ***libc*** для стандартных ошибок при работе в пространстве пользователя. Символ **PDEBUGG**, наоборот, ничего не делает; он может быть использован, чтобы просто "закомментировать" строчки с печатью, не удаляя их полностью.

Чтобы упростить процесс ещё больше, добавьте в ваш Makefile следующие строчки:

```
# Закомментируйте/раскомментируйте следующие строки, чтобы запретить/
разрешить отладку
DEBUG = y

# Добавить ваш флаг отладки (или нет) к CFLAGS
ifeq ($(DEBUG),y)
 DEBFLAGS = -O -g -DSCULL_DEBUG # опция "-O" необходима для раскрытия
встраиваемых определений
else
 DEBFLAGS = -O2
endif

CFLAGS += $(DEBFLAGS)
```

Показанные в данном разделе макросы зависят от расширения ANSI Си препроцессора в ***gcc***, который поддерживает макросы с переменным числом аргументов. Эта зависимость ***gcc*** не должна быть проблемой, потому что ядро в любом случае сильно зависит от особенностей ***gcc***. Кроме того, Makefile зависит от версии утилиты GNU ***make***; раз ядро уже зависит от GNU ***make***, эта зависимость не является проблемой.

Если вы знакомы с препроцессором Си, вы можете расширить данные определения для реализации концепции "уровень отладки", определяя различные уровни и задавая целые значения (или битовые маски) для каждого уровня, чтобы задать, насколько подробными они должны быть. Но каждый драйвер имеет свои особенности и потребности мониторинга. Искусство хорошего программирования проявляется в выборе лучшего компромисса между гибкостью и эффективностью, и мы не можем сказать, что лучше для вас. Помните, что условия препроцессора (также как и постоянные выражения в коде) выполняются во время компиляции, и требуется перекомпиляция для включения или выключения сообщений. Возможной альтернативой является использование условий языка Си, которые будут выполнены во время выполнения программы и, следовательно, позволят вам включат и выключать сообщения во время исполнения программы. Это хорошая возможность, но она требует дополнительной обработки каждый раз при выполнении кода, который может повлиять

на производительность, даже если сообщения будут отключены. Иногда такая производительность является неприемлемой.

Макросы, показанные в этом разделе, зарекомендовали себя полезным во многих ситуациях, с единственным недостатком, требующим перекомпиляцию модулей после любого изменения их сообщений.

Ограничение скорости

Если вы не будете осторожны, то можете получить сгенерированные вами тысяч сообщений ***printk***, переполняющие консоль и, возможно, переполненный файл системного лога. При использовании медленного консольного устройства (например, последовательного порта), чрезмерный поток сообщений также может замедлить работу системы или просто сделать её не отвечающей. Может быть очень трудно получить управление, когда с системой что-то не так, когда консоль безостановочно извергает данные. Поэтому вы должны быть очень осторожны с тем, что вы печатаете, особенно в выпущенных версиях драйверов и особенно после завершения инициализации. Обычно, выпущенный код никогда не должен ничего печатать во время нормальной эксплуатации; печатный вывод должен быть индикацией исключительной ситуации, требующей внимания.

С другой стороны, вы можете захотеть выдать сообщение, если устройство, которым вы управляете, перестаёт работать. Но вы должны быть осторожны, чтобы не переусердствовать в этом. Неумный процесс, который продолжается вечно при встрече с проблемами, может генерировать тысяч попыток в секунду; если ваш драйвер печатает сообщение "моё устройство повреждено" каждый раз, он может создать огромное количество вывода и, возможно, подвесить процессор, если консольное устройство медленное - нет прерываний, которые могут быть использованы для управления консолью, даже если это последовательный порт или строчный принтер.

Во многих случаях лучшим поведением является установка флага говорящего: "я уже жаловался на это", а не печать любых дальнейших сообщений после установки флага. Хотя, с другой стороны, есть резон и в посылке периодического сообщения "устройство по-прежнему сломано". Ядро предлагает функцию, которая может быть полезна в таких случаях:

```
int printk_ratelimit(void);
```

Эта функция должна быть вызвана перед печатью сообщения, если вы считаете, что оно может повторяться часто. Если функция возвращает ненулевое значение, идите дальше и печатайте сообщение, в противном случае пропустите его. Таким образом, типичные вызовы выглядят следующим образом:

```
if (printk_ratelimit( ))
    printk(KERN_NOTICE "The printer is still on fire\n");
```

printk_ratelimit работает, отслеживая как много сообщений отправляются на консоль. Когда уровень вывода превышает порог, ***printk_ratelimit*** начинает возвращать 0 и вызывает пропуск сообщений.

Поведение ***printk_ratelimit*** можно настроить, изменяя ***/proc/sys/kernel/printk_ratelimit*** (количество секунд ожидания до повторного включения сообщений) и ***/proc/sys/kernel/printk_ratelimit_burst*** (количество сообщений, принимаемых до начала ограничения).

Печать номеров устройств

Иногда при печати сообщение от драйвера необходимо будет напечатать номер устройства, связанный с интересующим оборудованием. Не очень трудно печатать старшие и младшие номера, но в интересах обеспечения совместимости ядро предоставляет для этой цели несколько полезных макросов (определённых в ***<linux/kdev_t.h>***):

```
int print_dev_t(char *buffer, dev_t dev);
char *format_dev_t(char *buffer, dev_t dev);
```

Оба макроса кодируют номер устройства в данный буфер; отличие лишь в том, что ***print_dev_t*** возвращает количество напечатанных символов, а ***format_dev_t*** возвращает буфер, поэтому он может быть использован в качестве параметра при вызове ***printk*** напрямую, хотя надо помнить, что ***printk*** не сбросит данные на диск, пока не получит символ перевода строки. Буфер должен быть достаточно большим, чтобы содержать номер устройства; учитывая, что 64-х разрядные номера устройств являются определённо возможными в будущих версиях ядра, следует иметь буфер длиной, по крайней мере, 20 байт.

Отладка через запросы

В предыдущем разделе мы описали, как работает ***printk***, и как можно её использовать. Но пока ничего не говорили о её недостатках.

Массовое использование ***printk*** может заметно замедлить работу системы, даже если вы понизили ***console_loglevel*** во избежание загрузки консольного устройства, потому что ***syslogd*** поддерживает синхронизацию своих выходных файлов; таким образом, каждая печатаемая строка является причиной дисковой операции. Это правильно с точки зрения ***syslogd***. Он пытается записать всё на диск на случай падения системы сразу после печати сообщения; однако, вы не хотите замедлить работу системы только ради отладочных сообщений. Эта проблема может быть решена с помощью префикса с дефисом в имени вашего файла протокола, как это показано в ***/etc/syslogd.conf***. (* Дефис, или знак минус, это "магический" маркер syslogd для предотвращения сброса на диск каждого нового сообщения, документированный в syslog.conf (5), описание стоит прочитать.) Проблемой с изменением конфигурационного файла является то, что изменение, вероятно, останется там и после окончания отладки, хотя при нормальной работе системы вы хотите, чтобы сообщения сбрасывались на диск как можно скорее. Альтернативой такому постоянному изменению является работа другой программы вместо ***klogd*** (такой, как ***cat /proc/kmsg***, как предлагалось ранее), но это не может обеспечить благоприятные условия для нормального функционирования системы.

Чаще всего лучшим способом для получения соответствующей информации является запрос к системе когда вам нужна информация, а не постоянный вывод данных. По сути, каждая система Unix предлагает множество инструментов для получения системной информации: ***ps***, ***netstat***, ***vmstat*** и так далее.

Для разработчиков драйверов для запросов к системе доступны несколько методов: создание файла в файловой системе ***/proc***, используя метод драйвера ***ioctl***, и экспорт атрибутов через ***sysfs***. Использование ***sysfs*** требует некоторой информации о драйверной модели. Это обсуждается в Главе 14 347.

Использование файловой системы /proc

Файловая система ***/proc*** является специальной программно созданной файловой системой, которая используется ядром для экспорта информации в мир. Каждый файл в ***/proc*** связан с функцией ядра, которая порождает "содержимое" файла на лету, когда файл читается. Мы уже видели некоторые из этих файлов в действии; ***/proc/modules***, например, всегда возвращает список загруженных модулей.

/proc широко используется в системе Linux. Многие утилиты в современных дистрибутивах Linux, такие как ***ps***, ***top*** и ***uptime*** получают свою информацию из ***/proc***. Некоторые драйверы также экспортируют информацию через ***/proc*** и ваш может делать то же самое. Файловая система ***/proc*** динамическая, так что ваш модуль может добавлять или удалять записи в любое время. Полнофункциональные записи в ***/proc*** могут быть сложными созданиями; среди прочего, они могут быть записываемыми, также как и читаемыми. Однако, в большинстве случаев записи в ***/proc*** являются только читаемыми файлами. Этот раздел имеет дело с простым случаем "только для чтения". Те, кто заинтересован в осуществлении чего-то более сложного, могут посмотреть здесь основу; затем для полной картины можно посмотреть исходный код ядра.

Однако, прежде чем мы продолжим, следует отметить, что добавление файлов в каталоге ***/proc*** не приветствуется. Файловая система ***/proc*** рассматривается разработчиками ядра как немного неконтролируемый беспорядок, который вышел далеко за пределы своей первоначальной цели (которая была в предоставлении информации о процессах, запущенных в системе). Рекомендуемый способ предоставления информации в новом коде - через ***sysfs***. Однако, работа с ***sysfs*** требует понимания драйверной модели в Linux и мы не сможем сделать это, пока не дойдём до Главы 14 347. Между тем, файлы в ***/proc*** создать намного легче и они полностью подходят для отладки, поэтому мы и рассмотрим их здесь.

Работа с файлами в /proc

Все модули, которые работают с ***/proc***, должны для определения соответствующих функций подключать ***<linux/proc_fs.h>***.

Чтобы создать файл только для чтения в ***/proc***, ваш драйвер должен содержать функцию для предоставления данных, когда файл читается. Когда какой-то процесс читает файл (используя системный вызов ***call***), запрос достигает модуля с помощью этой функции. Мы рассмотрим эту функцию первой и далее в этом разделе доберёмся до регистрации интерфейса. Когда процесс читает из вашего файла в ***/proc***, ядро выделяет страницу памяти (то есть **PAGE_SIZE** байт), куда драйвер может записать данные для возврата в пространство пользователя. Этот буфер передаётся в вашу функцию, которая представляет собой метод, названный ***read_proc***:

```
int (*read_proc)(char *page, char **start, off_t offset, int count, int *eof,
void *data);
```

Указатель **page** является буфером, куда вы будете записывать ваши данные; **start** используется функцией, чтобы сказать, где на странице были записаны интересующие данные (подробнее об этом позже); **offset** и **count** имеют такое же значение, как и для метода ***read***. Аргумент **eof** указывает на целое число, которое должно быть установлено драйвером, чтобы сигнализировать, что у него нет больше данных для возвращения, в то время как **data** является указателем на специфические данные драйвера, которые можно использовать для

внутренней бухгалтерии.

Эта функция должна возвращать количество байт данных, фактически размещённых в буфере **page**, так же как делает метод ***read*** для других файлов. Другие выходные значения - это ***eof** и ***start**. **eof** - это простой флаг, а вот использование значения **start** является несколько более сложным; его целью является помощь в реализации больших (более одной страницы) ***/proc*** файлов.

Параметр **start** имеет несколько нетрадиционное использование. Его целью является указать, где (в пределах страницы) находятся данные, которые будут возвращены пользователю. Когда вызывается метод ***proc_read***, ***start** будет NULL. Если вы оставите его NULL, ядро предполагает, что данные были помещены на страницу, как если бы **offset** был 0; другими словами, оно предполагает простую версию ***proc_read***, которая размещает всё содержимое виртуального файла на страницу, не обращая внимания на параметр смещения. Вместо этого, если вы установите ***start** в значение не-NULL, ядро предполагает, что данные, на которые указывает ***start**, принимают во внимание **offset** и готовы быть возвращены непосредственно к пользователю. В общем, простые методы ***proc_read***, которые возвращают крошечные объёмы данных, просто игнорируют **start**. Более сложные методы устанавливают ***start** к **page** и размещают данные только начиная с запрошенного здесь смещения.

Давно существует другой важный вопрос с файлами ***/proc***, которые также призван решить **start**. Иногда между последовательными вызовами ***read*** изменяется ASCII представление структур данных ядра, так что читающий процесс может обнаружить противоречивые данные от одного вызова к другому. Если ***start** установлен как небольшое целое значение, вызывающий использует его для увеличения **filp->f_pos** независимо от объёма возвращаемых данных, тем самым делая **f_pos** внутренним номером записи вашей процедуры ***read_proc***. Если, например, ваша функция ***read_proc*** возвращает информацию из большого массива структур и пять из этих структур были возвращены в первом вызове, ***start** мог бы быть установлен в 5. Следующий вызов предоставляет то же значение, как **offset**; драйвер теперь знает, что надо начинать возвращать данные с шестой структуры в массиве. Это признается как "взлом" его авторами и это можно увидеть в ***fs/proc/generic.c***.

Отметим, что существует лучший путь для реализации больших ***/proc*** файлов; он называется ***seq_file*** и мы обсудим его в ближайшее время. Сейчас же настало время для примера. Вот простая (хотя и несколько некрасивая) реализация ***read_proc*** для устройства ***scull***:

```
int scull_read_procmem(char *buf, char **start, off_t offset, int count, int
*eof, void *data)
{
    int i, j, len = 0;
    int limit = count - 80; /* Не печатать больше, чем это */

    for (i = 0; i < scull_nr_devs && len <= limit; i++) {
        struct scull_dev *d = &scull_devices[i];
        struct scull_qset *qs = d->data;
        if (down_interruptible(&d->sem))
            return -ERESTARTSYS;
        len += sprintf(buf+len,"\nDevice %i: qset %i, q %i, sz %li\n",
                       i, d->qset, d->quantum, d->size);
        for (; qs && len <= limit; qs = qs->next) { /* сканируем список */
            len += sprintf(buf + len, " item at %p, qset at %p\n",
```

```
                                        qs, qs->data);
            if (qs->data && !qs->next) /* выводим только последний элемент */
                for (j = 0; j < d->qset; j++) {
                    if (qs->data[j])
                        len += sprintf(buf + len,
                                       " % 4i: %8p\n",
                                       j, qs->data[j]);
                }
        }
        up(&scull_devices[i].sem);
    }
    *eof = 1;
    return len;
}
```

Это довольно типичная реализация ***read_proc***. Она предполагает, что никогда не будет необходимо создать более одной страницы данных и, таким образом, игнорирует значения **start** и **offset**. Однако, на всякий случай, делает это осторожно, чтобы не переполнить буфер.

Устаревший интерфейс

Если вы почитаете исходный код ядра, то можете столкнуться с кодом реализующим ***/proc*** файлы со старым интерфейсом:

```
int (*get_info)(char *page, char **start, off_t offset, int count);
```

Все эти аргументы имеют тот же смысл, что и для ***read_proc***, но аргументы **eof** и **data** отсутствуют. Этот интерфейс всё ещё поддерживается, но он может уйти в будущем; новый код должен использовать взамен ему интерфейс ***read_proc***.

Создание вашего файла в /proc

После того, как функция ***read_proc*** определена, необходимо подключить её к записи в иерархии ***/proc***. Это делается с помощью вызова ***create_proc_read_entry***:

```
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);
```

Здесь, **name** это имя файла для создания, **mode** является маской защиты файла (может быть передана как 0 для общесистемного значения по умолчанию), **base** указывает каталог, в котором должен быть создан файл (если **base** равен NULL, файл создаётся в каталоге корне ***/proc***), **read_proc** является функцией ***read_proc***, которая реализует файлов и **data** игнорируется ядром (но передаётся в ***read_proc***). Вызов, использующийся ***scull***, чтобы создать функцию ***/proc***, доступную как ***/proc/scullmem***:

```
create_proc_read_entry("scullmem", 0 /* режим по умолчанию */,
                       NULL /* родительская директория */,
scull_read_procmem,
                       NULL /* клиентские данные */);
```

Здесь мы создаём файл с именем ***scullmem*** прямо в ***/proc***, с защитой со значением по умолчанию, для чтения всеми.

Указатель директории может быть использован для создания всей иерархии каталогов в ***/proc***. Однако, следует отметить, что запись может быть более легко помещена в подкаталог ***/proc*** просто указанием имени каталога в качестве части названия записи, если сам каталог уже существует. Например, (часто игнорирующееся) соглашение говорит, что записи в ***/proc***, связанные с драйверами устройств, должны вести в подкаталог ***driver/***; ***scull*** мог бы поместить свою запись там просто задав своё имя как ***driver/scullmem***.

Записи в ***/proc***, конечно же, должны быть удалены при выгрузке модуля. ***remove_proc_entry*** является функцией, которая отменяет уже сделанную ***create_proc_read_entry***:

```
remove_proc_entry("scullmem", NULL /* родительская директория */);
```

Сбой при удалении записей может привести к вызовам в нежелательное время, или, если ваш модуль был выгружен, аварийному отказу ядра.

При использовании файлов ***/proc***, как было показано, вы должны помнить несколько неудобств реализации - неудивительно, что его использование не рекомендуется в настоящее время.

Наиболее важная проблема связана с удалением записей в ***/proc***. Такое удаление может случиться во время использования файла, так как нет владельца, связанного с записями ***/proc***, поэтому их использование не воздействует на счётчик ссылок на модуль. Эта проблема, к примеру, может быть легко получена запуском ***sleep 100 < /proc/myfile*** перед удалением модуля.

Другой вопрос связан с регистрацией двух записей с одинаковыми именами. Ядро доверяет драйверу и не проверяет, зарегистрировано ли уже это имя, так что если вы не будете осторожны, то в можете доиграться до двух или более записей с тем же названием. Такие записи неразличимы и при доступе к ним, и когда вы вызываете ***remove_proc_entry***.

Интерфейс seq_file

Как уже отмечалось выше, реализация больших файлов в ***/proc*** немного затруднительна. С течением времени методы ***/proc*** стали пользоваться дурной славой из-за ошибочных реализаций, когда объём вывода сильно увеличивается. Для очистки кода ***/proc*** и облегчения жизни программистам ядра был добавлен интерфейс **seq_file**. Этот интерфейс предоставляет простой набор функций для реализации больших виртуальных файлов ядра.

Интерфейс **seq_file** предполагает, что вы создаёте виртуальный файл, который шагает через последовательность элементов, которые должны быть возвращены в пространство пользователя. Для использования **seq_file** вы должны создать простой объект "итератор", который может содержать позицию в последовательности, шагнуть вперёд и вывести один элемент последовательности. Это может показаться сложным, но по сути этот процесс довольно прост. Чтобы показать, как это делается, мы пройдём через создание файла ***/proc*** в драйвере ***scull***.

Первый неизбежным шагом является включение ***<linux/seq_file.h>***. Затем вы должны создать четыре итерационных метода, названных ***start***, ***next***, ***stop*** и ***show***.

Метод ***start*** всегда вызывается первым. Прототип этой функции:

```
void *start(struct seq_file *sfile, loff_t *pos);
```

Аргументом **sfile** почти всегда можно пренебречь. **pos** является целой позицией, указывающей, где должно начаться чтение. Интерпретация позиции полностью зависит от реализации; она не должна быть позицией байта в результирующем файле. В реализациях **seq_file** обычно шагающих через последовательность интересующих объектов, позиция часто интерпретируется как курсор, указывающий на следующий элемент последовательности. Драйвер ***scull*** интерпретирует каждое устройство как один элемент последовательности, так что входящее значение **pos** - просто индекс в массиве **scull_devices**. Таким образом, метод ***start***, используемый в ***scull***:

```
static void *scull_seq_start(struct seq_file *s, loff_t *pos)
{
    if (*pos >= scull_nr_devs)
        return NULL; /* Больше читать нечего */
    return scull_devices + *pos;
}
```

Возвращаемое значения, если не NULL, является внутренней величиной, которая может быть использована реализацией итератора.

Следующая функция должна сдвигать итератор на следующую позицию, возвращая NULL, если в последовательности ничего не осталось. Прототип этого метода:

```
void *next(struct seq_file *sfile, void *v, loff_t *pos);
```

Здесь, **v** является итератором, возвращённым предыдущим вызовом ***start*** или ***next***, а **pos** - текущая позиция в файле. **next** должен увеличить значение, указываемое **pos**; в зависимости от того, как работает ваш итератор, можно (хотя, вероятно, и нет) захотеть увеличить **pos** больше, чем на одно. Вот что делает ***scull***:

```
static void *scull_seq_next(struct seq_file *s, void *v, loff_t *pos)
{
    (*pos)++;
    if (*pos >= scull_nr_devs)
        return NULL;
    return scull_devices + *pos;
}
```

Когда итерации закончены, ядро вызывает для очистки ***stop***:

```
void stop(struct seq_file *sfile, void *v);
```

Реализация ***scull*** не выполняет работу по очистке, так что его метод ***stop*** пуст.

Стоит отметить, что код **seq_file** по своему дизайну не спит или не выполняет другие неатомарные задачи между вызовами ***start*** и ***stop***. Вам гарантировано также получить один вызов ***stop*** вскоре после вызова ***start***. Таким образом, она является безопасной для вашего метода ***start*** при использовании семафоров или спин-блокировок. Пока ваши другие методы **seq_file** атомарны, вся последовательность вызовов является атомарной. (Если этот параграф не имеет смысла для вас, вернитесь к нему после прочтения следующей главы).

Для фактического вывода чего-то интересного для пространства пользователя между этими вызовами ядро вызывает метод ***show***. Прототип этого метода:

```
int show(struct seq_file *sfile, void *v);
```

Этот метод должен создать вывод для элемента в последовательности, указанной итератором **v**. Однако, он не должен использовать ***printk***, вместо этого для вывода существует специальный набор функций **seq_file**:

int seq_printf(struct seq_file *sfile, const char *fmt, ...);
Это эквивалент ***printf*** для реализаций **seq_file**; он принимает обычную строку формата и дополнительные аргументы значений. Однако, вы также должны передать ей структуру **seq_file**, которая передаётся в функцию ***show***. Если ***seq_printf*** возвращает ненулевое значение, это означает, что буфер заполнен и вывод будет отброшен. Большинство реализаций, однако, игнорирует возвращаемое значение.

int seq_putc(struct seq_file *sfile, char c);
int seq_puts(struct seq_file *sfile, const char *s);
Эти эквиваленты функциям ***putc*** и ***puts*** пользовательского пространства.

int seq_escape(struct seq_file *m, const char *s, const char *esc);
Эта функция эквивалентна ***seq_puts*** с тем исключением, что любой символ в **s**, который также находится в **esc**, напечатается в восьмеричной форме. Общим значением для **esc** является " \t\n\\", которое предохраняет встроенное незаполненное пространство от беспорядка при выводе и, возможно, путаницы скриптов оболочки.

int seq_path(struct seq_file *sfile, struct vfsmount *m, struct dentry *dentry, char *esc);
Эта функция может быть использована для вывода имени файла, связанного с данной записью в каталоге. Она вряд ли будет полезна в драйверах устройств; мы включили её здесь для полноты.

Вернёмся назад к нашему примеру; метод ***show***, используемый в ***scull***:

```
static int scull_seq_show(struct seq_file *s, void *v)
{
    struct scull_dev *dev = (struct scull_dev *) v;
    struct scull_qset *d;
    int i;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;
    seq_printf(s, "\nDevice %i: qset %i, q %i, sz %li\n",
                (int) (dev - scull_devices), dev->qset,
                dev->quantum, dev->size);
    for (d = dev->data; d; d = d->next) { /* scan the list */
        seq_printf(s, " item at %p, qset at %p\n", d, d->data);
        if (d->data && !d->next) /* вывести только последний элемент */
            for (i = 0; i < dev->qset; i++) {
                if (d->data[i])
                    seq_printf(s, " % 4i: %8p\n",
                                i, d->data[i]);
```

```
            }
    }
    up(&dev->sem);
    return 0;
}
```

Здесь мы, наконец, интерпретировали наше значение "итератор", которое является просто указателем на структуру **scull_dev**.

Теперь, когда имеется полный набор итерационных операций, ***scull*** должен упаковать их и подключить их к файлу в ***/proc***. Первым шагом является заполнение структуры **seq_operations**:

```
static struct seq_operations scull_seq_ops = {
    .start = scull_seq_start,
    .next  = scull_seq_next,
    .stop  = scull_seq_stop,
    .show  = scull_seq_show
};
```

Вместе с этой структурой мы должны создать реализацию файла, которую поймёт ядро. Мы не используем метод ***read_proc***, описанный ранее; когда используется **seq_file**, лучше подключиться к ***/proc*** на немного более низком уровне. Это означает создание структуры **file_operations** (да, той же структуры, используемой для символьных драйверов), реализующей выполнение всех необходимых операций ядром для обработки чтения и позиционирования в этом файле. К счастью, эта задача проста. Первым шагом является создание метода ***open***, который подключает файл к операциям **seq_file**:

```
static int scull_proc_open(struct inode *inode, struct file *file)
{
    return seq_open(file, &scull_seq_ops);
}
```

Вызов к ***seq_open*** подключает структуру **file** с нашей последовательностью операций, определённых выше. Как оказалось, единственной файловой операцией, которую мы должны реализовать сами, является ***open***, поэтому сейчас мы можем создать нашу структуру **file_operations**:

```
static struct file_operations scull_proc_ops = {
    .owner   = THIS_MODULE,
    .open    = scull_proc_open,
    .read    = seq_read,
    .llseek  = seq_lseek,
    .release = seq_release
};
```

Здесь мы определяем наш собственный метод ***open***, но используем заранее подготовленные методы ***seq_read***, ***seq_lseek*** и ***seq_release*** для всего остального. Последний шаг заключается в фактическом создании файла в ***/proc***:

```
entry = create_proc_entry("scullseq", 0, NULL);
if (entry)
    entry->proc_fops = &scull_proc_ops;
```

Вместо того, чтобы использовать ***create_proc_read_entry***, мы вызываем низкоуровневую ***create_proc_entry***, которая имеет следующий прототип:

```
struct proc_dir_entry *create_proc_entry(const char *name,
                        mode_t mode,
                        struct proc_dir_entry *parent);
```

Аргументы такие же, как их аналоги в ***create_proc_read_entry***: имя файла, режим защиты, и родительская директория.

Используя приведённый выше код, ***scull*** имеет новую запись в ***/proc***, которая выглядит так же, как предыдущая. Она, однако, лучше, поскольку работает независимо от того, насколько большим становится вывод, она обрабатывает запросы должным образом, и, как правило, легче для чтения и поддержки. Мы рекомендуем использовать **seq_file** для реализации файлов, которые содержат больше, чем очень небольшое число строк вывода.

Метод ioctl

ioctl, использование которого мы покажем вам в Главе 6 128, это системный вызов, который действует на дескриптор файла; он получает номер, идентифицирующий команду, которая будет выполняться и (опционально) ещё один аргумент, как правило, указатель. В качестве альтернативы использованию файловой системы ***/proc*** вы можете реализовать несколько ***ioctl*** команд специально для отладки. Эти команды могут копировать соответствующие структуры данных из драйвера в пространство пользователя, где вы сможете их проверить.

Использование ***ioctl*** для получения информации является несколько более сложным способом, чем с помощью ***/proc***, потому что для вызова ***ioctl*** и отображения результатов вам необходима другая программа. Эта программа должна быть написана, скомпилирована и сохранять синхронизацию с модулем при тестировании. С другой стороны, код на стороне драйвера может быть проще, чем это необходимо для реализации ***/proc*** файла.

Бывают моменты, когда ***ioctl*** является лучшим способом получения информации, потому что работает быстрее, чем чтение ***/proc***. Если перед записью на экран с данными должна быть выполнена какая-то работа, получение данных в двоичной форме более эффективно, чем чтение текстового файла. Кроме того, ***ioctl*** не требует разделения данных на фрагменты меньшие, чем страница. Еще одним интересным преимуществом подхода ***ioctl*** является то, что информационно-поисковые команды могут быть оставлены в драйвере даже когда вся другая отладка будет отключена. В отличие от ***/proc*** файла, который будет виден всем, кто посмотрит в директории (и слишком многие люди, вероятно, удивятся "что это за странный файл"), недокументированные команды ***ioctl***, вероятно, останутся незамеченными. Кроме того, они всё ещё будут там, если с драйвером случится что-то странное. Единственный недостаток заключается в том, что модуль будет немного больше.

Отладка наблюдением

Иногда мелкие проблемы могут быть найдены путём наблюдения за поведением приложения в пространстве пользователя. Наблюдение за программами также может помочь в создании атмосферы доверия, что драйвер работает правильно. Например, мы могли чувствовать уверенность в ***scull*** после наблюдения за тем, как его реализация ***read*** реагировала на запросы чтения различных объёмов данных.

Существуют различные способы, чтобы наблюдать за работой программ пользовательского пространства. Вы можете запускать отладчик пошагово через свои функции, добавлять вывод на печать, или запускать программу под ***strace***. Здесь мы рассмотрим только последний метод, который является наиболее интересным, когда реальной целью является проверка кода ядра.

Команда ***strace*** представляет собой мощный инструмент, который показывает все системные вызовы, сделанные программой пространства пользователя. Она не только показывает вызовы, но может также показать аргументы вызовов и возвращаемые значения в символической форме. Когда системный вызов заканчивается неудачно, отображается символическое значение ошибки (например, **ENOMEM**) и соответствующая строка (**Out of memory**, не хватает памяти). ***strace*** имеет много параметров командной строки; наиболее полезной из которых являются ***-t*** для отображения времени, когда был выполнен каждый вызов, ***-T*** для отображения времени, проведённого в вызове, ***-e*** для ограничения типов трассируемых вызовов, и -o, чтобы перенаправить вывод в файл. По умолчанию ***strace*** печатает информацию трассировки в **stderr**.

strace получает информацию от самого ядра. Это означает, что программа может быть оттрассирована независимо от того, была ли она скомпилирована с поддержкой отладки (опция ***-g*** для ***gcc***) и была удалена или нет эта информация. Вы также можете подключить трассировку к работающему процессу, аналогично тому, как отладчик может подключиться к запущенному процессу и контролировать его.

Информация трассировки часто используется для подтверждения сообщений об ошибках, посылаемых разработчикам приложений, но это также бесценна для программистов ядра. Мы видели, как выполняется код драйвера, реагируя на системные вызовы; ***strace*** позволяет проверить соответствие входных и выходных данных каждого вызова.

Например, следующий снимок экрана показывает последние строки (большинство из них) работы команды ***strace ls /dev > /dev/scull0***:

```
open("/dev", O_RDONLY|O_NONBLOCK|O_LARGEFILE|O_DIRECTORY) = 3
fstat64(3, {st_mode=S_IFDIR|0755, st_size=24576, ...}) = 0
fcntl64(3, F_SETFD, FD_CLOEXEC) = 0
getdents64(3, /* 141 запись */, 4096) = 4088
[...]
getdents64(3, /* 0 записей */, 4096) = 0
close(3) = 0
[...]
fstat64(1, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
write(1, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 4096) = 4000
write(1, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 96) = 96
write(1, "b\nptyxc\nptyxd\nptyxe\nptyxf\nptyy0\n"..., 4096) = 3904
write(1, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 192) = 192
write(1, "\nvcs47\nvcs48\nvcs49\nvcs5\nvcs50\nvc"..., 673) = 673
close(1) = 0
exit_group(0) = ?
```

Как явствует из первого вызова ***write***, после того, как ***ls*** закончила просмотр целевого каталога, она попыталась записать 4 Кб. Странно (для ***ls***), только 4000 байт были записаны, и операция была повторена. Однако, мы знаем, что реализация ***write*** в ***scull*** пишет один квант за раз, так что мы могли бы ожидать частичную запись. Через несколько шагов все данные проталкиваются и программа успешно завершается. В качестве другого примера давайте

почитаем устройство ***scull*** (с помощью команды ***wc***):

```
[...]
open("/dev/scull0", O_RDONLY|O_LARGEFILE) = 3
fstat64(3, {st_mode=S_IFCHR|0664, st_rdev=makedev(254, 0), ...}) = 0
read(3, "MAKEDEV\nadmmidi0\nadmmidi1\nadmmid"..., 16384) = 4000
read(3, "b\nptywc\nptywd\nptywe\nptywf\nptyx0\n"..., 16384) = 4000
read(3, "s17\nvcs18\nvcs19\nvcs2\nvcs20\nvcs21"..., 16384) = 865
read(3, "", 16384) = 0
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
write(1, "8865 /dev/scull0\n", 17) = 17
close(3) = 0
exit_group(0) = ?
```

Как и ожидалось, ***read*** может получить только 4000 байт за раз, но общий объём данных тот же, что был записан в предыдущем примере. Интересно отметить, как, в отличие от предыдущей трассировки, организованы повторы в данном примере. ***wc*** оптимизирован для быстрого чтения и, таким образом, обходит стандартные библиотеки, пытаясь прочитать больше данных одним системным вызовом. Вы можете видеть из строк **чтения** в трассировке, как ***wc*** пытался прочитать 16 Кб за раз.

Специалисты по Linux могут найти много полезной информации в выводе ***strace***. Если вы убрали все символы, вы можете ограничить себя просмотром, как работают файловые методы (***open***, ***read*** и другие) с помощью флага **efile**.

Лично мы находим ***strace*** наиболее полезной для выявления ошибок в системных вызовах. Часто вызов ***perror*** в приложении или демо-программе не достаточно подробен, чтобы быть полезным для отладки, и возможность точно сказать, какие аргументы в каком системном вызове вызвали ошибку может быть большой помощью.

Система отладки неисправностей

Даже если вы использовали все техники мониторинга и отладки, иногда в драйвере остаются ошибки и система выдаёт ошибки при работе драйвера. Когда это происходит, важно уметь собрать столько информации, сколько возможно, чтобы решить эту проблему.

Отметим, что "ошибка", не означает "паника". Код Linux является достаточно прочным, чтобы корректно отреагировать на большинство ошибок: ошибки обычно приводят к уничтожению текущего процесса, в то время как система продолжает работать. Система может паниковать, и это может быть, если ошибка происходит за пределами контекста процесса или если скомпрометированы некоторые жизненно важные части системы. Но когда проблема вызвана ошибкой драйвера, это обычно приводит только к внезапной смерти процесса, которому не повезло использовать драйвер. Только неустранимые повреждения при разрушении процесса приводят к тому, что некоторый объём памяти, выделенной контексту процесса, теряется; например, могли бы быть утеряны динамические списки, выделенные драйвером через ***kmalloc***. Однако, поскольку ядро вызывает операцию ***close*** для любого открытого устройства, когда процесс умирает, драйвер может освободить то, что выделено методом ***open***.

Хотя даже Oops (ой) обычно не доводит до падения всей системы, вы можете посчитать, что необходима перезагрузка после того, как он случился. Драйвер с ошибками может оставить оборудование в непригодном для использования состоянии, оставить ресурсы ядра в неустойчивом состоянии, или, в худшем случае, в случайных местах повредить память ядра.

Часто после Oops вы можете просто выгрузить ваш ошибочный драйвер и попробовать ещё раз. Однако, если вы увидите что-то, что позволяет предположить, что система в целом не очень хороша, лучшим решением, как правило, является незамедлительная перезагрузка.

Мы уже говорили, что когда код ядра плохо себя ведёт, на консоль выводится информационное сообщение. Следующий раздел объясняет, как декодировать и использовать такие сообщения. Даже если они выглядят довольно неясными для новичков, процессор выводит всю интересную информацию, часто достаточную, чтобы определить ошибку в программе без необходимости проведения дополнительных тестов.

Сообщения Oops

Большинство ошибок показывают себя в разыменовании указателя **NULL** или других некорректных значений указателя. Обычным результатом таких ошибок является сообщение Oops.

Почти любой адрес, используемый процессором, представляет собой виртуальный адрес и сопоставляется с физическими адресами через сложную структуру таблиц страниц (исключениями являются физические адреса, используемые в самой подсистеме управления памятью). При разыменовании неверного указателя страничный механизм не может сопоставить указатель с физическим адресом и процессор сигнализирует об ***ошибке страницы*** (***page fault***) в операционной системе. Если адрес не является действительным, ядро не в состоянии загрузить страницу с несуществующим адресом; когда процессор находится в режиме супервизора, оно (обычно) генерирует Oops, если это произошло.

Oops отображает состояние процессора в момент ошибки, в том числе содержание регистров процессора и другую, казалось бы, непонятную информацию. Сообщение создаётся и отправляется с помощью ***printk*** в обработчике ошибок (***arch/*/kernel/traps.c***), как описано выше в разделе "printk" 72.

Давайте посмотрим на одно из таких сообщений. Вот какие результаты от разыменования указателя NULL на ПК под управлением ядра версии 2.6. Наиболее относящейся к делу информацией здесь является указатель команд (EIP), адрес ошибочной инструкции.

```
Unable to handle kernel NULL pointer dereference at virtual address 00000000
 printing eip:
d083a064
Oops: 0002 [#1]
SMP
CPU: 0
EIP: 0060:[<d083a064>] Not tainted
EFLAGS: 00010246 (2.6.6)
EIP is at faulty_write+0x4/0x10 [faulty]
eax: 00000000 ebx: 00000000 ecx: 00000000 edx: 00000000
esi: cf8b2460 edi: cf8b2480 ebp: 00000005 esp: c31c5f74
ds: 007b es: 007b ss: 0068
Process bash (pid: 2086, threadinfo=c31c4000 task=cfa0a6c0)
Stack: c0150558 cf8b2460 080e9408 00000005 cf8b2480 00000000 cf8b2460
cf8b2460
       fffffff7 080e9408 c31c4000 c0150682 cf8b2460 080e9408 00000005
cf8b2480
       00000000 00000001 00000005 c0103f8f 00000001 080e9408 00000005
00000005
```

```
Call Trace:
 [<c0150558>] vfs_write+0xb8/0x130
 [<c0150682>] sys_write+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb
Code: 89 15 00 00 00 00 c3 90 8d 74 26 00 83 ec 0c b8 00 a6 83 d0
```

Это сообщение было создано записью в устройство, принадлежащее модулю ***faulty***, модуль создан специально для демонстрации ошибки. Реализация метода ***write*** из ***faulty.c*** тривиальна:

```
ssize_t faulty_write (struct file *filp, const char __user *buf, size_t
count, loff_t *pos)
{
    /* создать простую ошибку, используя разыменование NULL указателя */
    *(int *)0 = 0;
    return 0;
}
```

Как вы можете видеть, мы делаем здесь разыменование **NULL** указателя. 0 никогда не является допустимым значением указателя, поэтому происходит ошибка, которое ядро превращает в сообщение Oops, показанное ранее. Вызывающий процесс затем убивается.

Модуль ***faulty*** имеет другое ошибочное условие в реализации ***read***:

```
ssize_t faulty_read(struct file *filp, char __user *buf, size_t count, loff_t
*pos)
{
    int ret;
    char stack_buf[4];

    /* Давайте попробуем переполнить буфер */
    memset(stack_buf, 0xff, 20);
    if (count > 4)
        count = 4; /* скопировать 4 байта пользователю */
    ret = copy_to_user(buf, stack_buf, count);
    if (!ret)
        return count;
    return ret;
}
```

Этот метод копирует строку в локальную переменную; к сожалению, строка больше, чем массив назначения. Переполнение буфера в результате приводит к Oops, когда функция возвращается. Так как инструкция **return** приносит никуда не указывающий указатель команд, этот вид ошибки гораздо труднее отследить и вы можете получить что-то похожее на это:

```
EIP: 0010:[<00000000>]
Unable to handle kernel paging request at virtual address ffffffff
 printing eip:
ffffffff
Oops: 0000 [#5]
SMP
CPU: 0
EIP: 0060:[<ffffffff>] Not tainted
EFLAGS: 00010296 (2.6.6)
```

```
EIP is at 0xffffffff
eax: 0000000c ebx: ffffffff ecx: 00000000 edx: bfffda7c
esi: cf434f00 edi: ffffffff ebp: 00002000 esp: c27fff78
ds: 007b es: 007b ss: 0068
Process head (pid: 2331, threadinfo=c27fe000 task=c3226150)
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00
ffffffff
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20
00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000
bfffda70
Call Trace:
 [<c0150612>] sys_read+0x42/0x70
 [<c0103f8f>] syscall_call+0x7/0xb
Code: Bad EIP value.
```

В этом случае мы видим только часть стека вызовов (***vfs_read*** и ***faulty_read*** отсутствуют) и ядро жалуется на "плохое значение EIP" (“bad EIP value”). Эта жалоба и адрес вызова (ffffffff), показанный в начале, оба намекают, что был повреждён стек ядра.

В общем, если вы столкнулись с Oops, первое, что нужно сделать, это посмотреть на место, где произошла проблема, которое обычно показывается отдельно от стека вызовов. В первом Oops, показанном выше, соответствующими строками являются:

```
EIP is at faulty_write+0x4/0x10 [faulty]
```

Здесь мы видим, что мы были в функции ***faulty_write***, которая находится в модуле ***faulty*** (который указан в квадратных скобках). Шестнадцатеричные цифры показывают, что указатель команд имел смещение 4 байта в функции, которая имеет размер 10 (hex) байтов. Часто этого достаточно, чтобы понять, в чём состоит проблема.

Если вам требуется больше информации, стек вызовов покажет вам, как вы попали туда, где всё распалось. Стек печатается в шестнадцатеричной форме; немного поработав, из листинга стека часто можно определить значения локальных переменных и параметров функции. Опытные разработчики ядра могут извлечь пользу из определенного количества распознанных здесь образов; например, если мы посмотрим на распечатку стека из ***faulty_read*** Oops:

```
Stack: ffffffff bfffda70 00002000 cf434f20 00000001 00000286 cf434f00
ffffffff
       bfffda70 c27fe000 c0150612 cf434f00 bfffda70 00002000 cf434f20
00000000
       00000003 00002000 c0103f8f 00000003 bfffda70 00002000 00002000
bfffda70
```

ffffffff на вершине стека является частью нашей строки, которая всё поломала. По умолчанию на архитектуре x86 стек пользовательского пространства начинается чуть ниже **0xc0000000**; таким образом, повторяющиеся значения **0xbfffda70**, вероятно, стековый адрес в пользовательском пространстве; это, фактически, адрес буфера, переданный системному вызову ***read***, повторенный каждый раз при передаче вниз по цепочке вызова ядра. На платформе x86 (опять же, по умолчанию), пространство ядра начинается с **0xc0000000**, так что значения выше этого - почти наверняка адреса пространства ядра, и так далее.

Наконец, при просмотре распечаток Oops всегда будьте бдительны к значениям "отравленных шаблонов" о которых рассказывалось в начале этой главы. Так, например, если вы получите от ядра Oops, в котором вызывающий адрес 0xa5a5a5a5, вы почти наверняка забываете где-то проинициализировать динамическую память.

Учтите, что вы видите символический стек вызовов (как показано выше), только если ваше ядро построено с включенной опцией **CONFIG_KALLSYMS**. В противном случае, вы увидите простой шестнадцатеричный список, который является гораздо менее полезным, пока вы не расшифровали его другими способами.

Зависания системы

Хотя большинство ошибок в коде ядра заканчивается сообщениями Oops, иногда они могут полностью завесить систему. Если система зависает, сообщение не печатается. Например, если код входит в бесконечный цикл, ядро останавливает планировщик (переключения процессов), (* На самом деле, многопроцессорные системы по-прежнему переключают процессы на других процессорах и даже однопроцессорная машина может переключать, если в ядре включено вытеснение. Однако, для наиболее распространенного случая (однопроцессорная с отключенным вытеснением), система вообще прекращает переключение.) и система не реагирует на любые действия, включая специальную комбинацию Ctrl-Alt-Del. Вы имеете два варианта действий с зависаниями системы - либо предотвратить их заранее или заниматься их поиском и устранением постфактум.

Вы можете предотвратить бесконечные циклы, вставив в важных точках вызов ***schedule***. Вызов ***schedule*** (как вы уже могли догадаться) вызывает планировщик и, следовательно, позволяет другим процессам украсть процессорное время у текущего процесса. Если процесс в пространстве ядра зациклился и-за ошибки в вашем драйвере, вызов ***schedule*** позволит вам убить процесс после трассировки произошедшего.

Вы должны знать, конечно, что любой вызов ***schedule*** может создать дополнительный источник повторных вызовов драйвера, так как он позволяет работать другим процессам. Этот повторный вход, как правило, не будет проблемой, если предположить, что вы использовали подходящую блокировку в вашем драйвере. Будьте уверены, однако, что не вызываете ***schedule*** в то время, когда ваш драйвер удерживает спин-блокировку.

Если ваш драйвер действительно висит в системе и вы не знаете, где вставить вызов ***schedule***, лучшим путём может быть добавление печати каких-нибудь сообщений и выводить их на консоль (изменяя значение **console_loglevel**, если это необходимо).

Иногда система может казаться зависшей, но это не так. Это может произойти, например, если клавиатура остаётся заблокированной каким-то странным образом. Эти ложные зависания можно обнаружить, глядя на вывод программы, который вы держите работающим только для этой цели. Часы или измеритель системой нагрузки на вашем дисплее - хороший монитор состояния; пока он продолжает обновление, планировщик работает.

Незаменимым инструментом для многих зависаний является "системная кнопка SysRq" (системный запрос), которая доступна на большинстве архитектур. Системный SysRq вызывается комбинацией клавиш Alt и SysRq на клавиатуре компьютера или с помощью других специальных клавиш на других платформах (смотрите для подробностей ***Documentation/sysrq.txt***) и также доступна на последовательной консоли. Третья кнопка, нажатая вместе с этими двумя, выполняет одно из перечисленных полезных действий:

r Отключает режим raw (сырой, возвращение скан-кодов) клавиатуры; полезно в ситуациях, когда порушенное приложение (такое, как X сервер) могло оставить вашу клавиатуру в странном состоянии.

k Вызывает функцию "ключ безопасного внимания", “secure attention key” (SAK). SAK убивает все процессы, запущенные на текущей консоли, оставив вас с чистым терминалом.

s Выполняет аварийную синхронизацию всех дисков.

u Размонтирование. Пытается перемонтировать все диски в режиме "только для чтения". Эта операция обычно вызывается сразу после s, можно сэкономить массу времени на проверку файловой системой в тех случаях, когда система находится в тяжелом состоянии.

b Загрузка. Сразу же перезагружает систему. Будьте уверены, что сначала засинхронизировали и перемонтировали диски.

p Печатает информацию о регистрах процессора.

t Печатает текущий список задач.

m Печатает информацию о памяти.

Существуют и другие функции системного SysRq; смотрите ***sysrq.txt*** в каталоге ***Documentation*** исходных текстов ядра для полного списка. Обратите внимание, что системный SysRq должен быть явно включен в конфигурации ядра и что большинство дистрибутивов его не разрешает по очевидным соображениям безопасности. Однако, в системе, используемой для разработки драйверов, разрешение системного SysRq стоит того, чтобы собрать себе новое ядро. Magic SysRq может быть отключен во время работы такой командой:

```
echo 0 > /proc/sys/kernel/sysrq
```

Вы должны предусмотреть его отключение для предотвращения случайного или намеренного ущерба, если непривилегированные пользователи могут получить доступ к вашей системной клавиатуры. Некоторые предыдущие версии ядра имели ***sysrq*** отключенным по умолчанию, так что для его разрешения вам необходимо во время выполнения записать 1 в тот же самый файл в ***/proc/sys***.

Операции sysrq чрезвычайно полезны, так что они делаются доступными для системных администраторов, которые не могут добраться до консоли. Файл ***/proc/sysrq-trigger*** является точкой "только для записи", где можно задать определённые действия ***sysrq***, записав соответствующий символ команды; потом вы сможете собрать любые выходные данные из логов ядра. Эта точка входа для ***sysrq*** работает всегда, даже если на консоли ***sysrq*** отключен.

Если вы столкнулись с "живым зависанием", в котором ваш драйвер застрял в цикле, но система в целом ещё функционирует, полезно знать несколько методов. Зачастую функция ***p*** SysRq прямо указывает на виновную процедуру. В противном случае, вы также можете использовать функции профилирования (протоколирования) ядра. Постройте ядро с включенным профилированием и запустите его с **profile=2** в командной строке. Сбросьте счётчики профиля утилитой ***readprofile***, а затем отправьте ваш драйвер в цикл. Через

некоторое время используйте ***readprofile*** снова, чтобы увидеть, на что ядро тратит своё время. Другой, более сложной альтернативой является ***oprofile***, которую вы также можете рассмотреть. Файл ***Documentation/basic_profiling.txt*** расскажет вам всё, что необходимо знать, чтобы начать работу с профайлерами.

Ценной предосторожностью при отладке системных зависаний является монтирование всех ваших дисков в режиме "только для чтения" (или их отключение). Если диски находятся в режиме "только для чтения" или демонтированы, нет риска повреждения файловой системы или оставить её в неустойчивом состоянии. Другая возможность заключается в использовании компьютера, который подключает все его файловые системы через NFS, сетевую файловую систему. В ядре должна быть включена возможность "NFS-Root" и во время загрузки должны быть переданы специальные параметры. В этом случае вы избежите повреждения файловой системы, даже не прибегая к SysRq, потому что согласованная файловая система находится под управлением сервера NFS, который не повреждается вашим драйвером устройства.

Отладчик и соответствующие инструменты

Последним средством при отладке модулей является использование отладчика для пошаговой отладки кода, наблюдая значения переменных и машинных регистров. Такой подход требует много времени и его надо по возможности избегать. Тем не менее, подробное исследование кода, которое достигается с помощью отладчика, порой бесценно.

Использование интерактивного отладчика в ядре является непростой задачей. Ядро работает в своём собственном адресном пространстве от имени всех процессов в системе. В результате, ряд общих возможностей, предоставляемых отладчиками пользовательского пространства, таких как точки останова и пошаговый режим, в ядре получить труднее. В этом разделе мы рассмотрим несколько способов отладки ядра; каждый из них имеет свои преимущества и недостатки.

Использование gdb

gdb может быть весьма полезным для просмотра внутренностей системы. Искусное использование отладчика на этом уровне требует определённое знание команд ***gdb***, некоторое понимание ассемблерного кода целевой платформы и способность сопоставлять исходный код и оптимизированный ассемблированный.

Отладчик должен быть вызван, как если бы ядро было приложением. В дополнение к указанию имени файла для ELF образа ядра, вам необходимо ввести в командной строке имя файла ядра. Для работающего ядра, где файл ядра является образом основного ядра, это ***/proc/kcore***. Типичный вызов ***gdb*** выглядит следующим образом:

```
gdb /usr/src/linux/vmlinux /proc/kcore
```

Первый аргумент является именем ELF (Executable and Linkable Format, формат исполняемых и компонуемых файлов) несжатого исполняемого ядра, а не ***zImage*** или ***bzImage***, или чего-то собранного специально для среды запуска.

Вторым аргументом в командной строке ***gdb*** является имя файла ядра. Как и любой файл в ***/proc***, ***/proc/kcore*** генерируется, когда его читают. Когда системный вызов ***call*** выполняется в файловой системой ***/proc***, он связывается с функцией генерации данных, а не извлечения данных; мы уже эксплуатировали эту возможность в разделе "Использование файловой системы /proc" 79 ранее в этой главе. ***kcore*** используется для представления ядра

"исполняемым" в формате файла ядра; это огромный файл, потому что он представляет собой целое адресное пространство ядра, которое соответствует всей физической памяти. Из ***gdb*** вы можете посмотреть переменные ядра стандартными командами ***gdb***. Например, **p jiffies** печатает количество тиков времени от загрузки системы до настоящего времени.

При печати с данных из ***gdb*** ядро является всё же работающим и различные элементы данных имеют разные значения в разное время; ***gdb***, однако, оптимизирует доступ к файлу ядра кэшируя данные, которые уже прочитаны. Если вы попытаетесь взглянуть на переменную **jiffies** ещё раз, то получите такой же ответ, как и прежде. Кэширование значений, чтобы избежать дополнительного доступа к диску, является правильным поведением для обычных файлов ядра, но является неудобным, когда используется "динамический" образ ядра. Решением является давать команду ***core-file /proc/kcore*** всякий раз, когда вы хотите очистить кэш ***gdb***; отладчик получает готовый к использованию новый файл ядра и отбрасывает всю старую информацию. Вам, однако, не требуется всегда давать ***core-file*** при чтении новых данных; ***gdb*** читает ядро кусками по несколько килобайт и кэширует только уже прочитанные куски.

Многочисленные возможности, обычно предоставляемые ***gdb***, не доступны, когда вы работаете с ядром. Например, ***gdb*** не сможет изменить данные ядра; он рассчитывает, что программа будет запущена для отладки под его контролем перед игрой с образом памяти. Также невозможно установить точки останова и точки наблюдения, или пошаговое выполнение функций ядра.

Заметим, что для того, чтобы для ***gdb*** была доступна информация о символах, вы должны скомпилировать ядро с установленной опцией **CONFIG_DEBUG_INFO**. В результате на диске получается гораздо больший образ ядра, но без этой информации копаться в переменных ядра почти невозможно.

Имея отладочную информацию вы можете многое узнать о том, что происходит внутри ядра. ***gdb*** весело распечатывает структуры, значения указателей и так далее. Однако, изучение модулей является трудной задачей. Поскольку модули не являются частью переданного ***gdb*** образа ***vmlinux***, отладчик ничего о них не знает. К счастью, на ядре с версии 2.6.7 можно научить ***gdb*** тому, что требуется знать для проверки загружаемых модулей.

Загружаемые модули Linux являются ELF-форматом исполняемых образов; как таковые, они разделены на множество секций. Типичный модуль может содержать дюжину или больше разделов, но есть как правило, три, которые имеют существенное значение в сеансе отладки:

.text
: Этот раздел содержит исполняемый код модуля. Отладчик должен знать, где этот раздел, чтобы быть в состоянии делать трассировку или установить точки останова. (Ни одна из этих операций не относится к запуску отладчика на ***/proc/kcore***, но они полезны при работе с ***kgdb***, описанной ниже).

.bss
.data
: Эти два раздела содержат переменные модуля. Любая переменная, которая не инициализируется во время компиляции находятся в **.bss**, а те, которые инициализированы, - в **.data**.

При работе ***gdb*** с загружаемыми модулями требуется информирование отладчика о том,

куда были загружены эти разделы модуля. Эта информация доступна в sysfs, в */sys/module*. Например, после загрузки модуля *scull*, каталог */sys/module/scull/sections* содержит файлы с именами, такими как *.text*; содержание каждого файла является базовым адресом для раздела.

Теперь мы в состоянии выдать команду *gdb*, рассказывающую ему о нашем модуле. Нужной нам командой является **add-symbol-file**; эта команда принимает в качестве параметров имя объектного файла модуля, базовый адрес в *.text*, а также ряд дополнительных параметров, описывающих, где находятся другие интересующие разделы. Порывшись в разделе данных модуля в sysfs, мы можем построить такую команду:

```
(gdb) add-symbol-file .../scull.ko 0xd0832000 \
       -s .bss 0xd0837100 \
       -s .data 0xd0836be0
```

Мы включили небольшой скрипт в исходник примера (*gdbline*), который может создать эту команду для данного модуля.

Теперь мы можем использовать *gdb* для изучения переменных в нашем загружаемом модуле. Вот короткий пример, взятый из сессии отладки *scull*:

```
(gdb) add-symbol-file scull.ko 0xd0832000 \
       -s .bss 0xd0837100 \
       -s .data 0xd0836be0
add symbol table from file "scull.ko" at
       .text_addr = 0xd0832000
       .bss_addr = 0xd0837100
       .data_addr = 0xd0836be0
(y or n) y
Reading symbols from scull.ko...done.
(gdb) p scull_devices[0]
$1 = {data = 0xcfd66c50,
       quantum = 4000,
       qset = 1000,
       size = 20881,
       access_key = 0,
       ...}
```

Здесь мы видим, что первое устройство *scull* в настоящее время занимает 20.881 байт. Если бы мы захотели, мы могли бы проследить данные цепочки, или посмотреть что-нибудь ещё интересное в модуле. Стоит знать ещё один полезный трюк:

```
(gdb) print *(address)
```

Введите адрес в шестнадцатиричном виде вместо **address**; вывод является файлом и номером строки кода, соответствующей этому адресу. Эта техника может быть полезна, например, чтобы выяснить, куда на самом деле указывают указатели функции.

Мы по-прежнему не можем выполнять типичные задачи отладки такие, как установку точек останова или изменение данных; для выполнения этих операций мы должны использовать такой инструмент, как *kdb* (описанный далее) или *kgdb* (с которым познакомимся в ближайшее время).

Отладчик ядра kdb

Многие читатели могут удивиться, почему ядро не имеет встроенных в него более развитых возможностей для отладки. Ответ довольно прост: Линус не доверяет интерактивным отладчикам. Он опасается, что они приведут к плохим исправлениям, которые залатают симптомы, а не устранят реальные причины проблем. Таким образом, встроенный отладчик отсутствует.

Однако, другие разработчики ядра периодически используют интерактивные средства отладки. Одним из таких инструментов является встроенный отладчик ядра ***kdb***, доступный как неофициальный патч от ***oss.sgi.com***. Для использования ***kdb*** вы должны получить патч (не забудьте получить версию, которая соответствует вашей версии ядра), применить его и пересобрать и переустановить ядро. Отметим, что на момент написания статьи ***kdb*** работает только на системах IA-32 (x86) (хотя некоторое время существовала версия для IA-64 в основной линии исходников ядра, пока не была удалена).

Когда вы работаете с ядром с включённым ***kdb***, есть несколько способов войти в отладчик. Отладчик запускает нажатие в консоли кнопки Pause (или Break). ***kdb*** также запускается, когда в ядре происходит Oops, или когда попадает на точку останова. В любом случае, вы увидите сообщение, которое выглядит примерно так:

```
Entering kdb (0xc0347b80) on processor 0 due to Keyboard Entry
[0]kdb>
```

Обратите внимание, что когда работает ***kdb***, всё в ядре останавливается. Ничто другое не должно быть запущено в системе, где вы вызываете ***kdb***; в частности, вы не должны иметь включенной сеть, если только, конечно, не отлаживаете сетевой драйвер. Обычно, если вы будете использовать ***kdb***, хорошей идеей является загрузка системы в однопользовательском режиме.

В качестве примера рассмотрим короткую сессию отладки ***scull***. Если предположить, что драйвер уже загружен, мы можем сказать ***kdb*** установить точку останова в ***scull_read*** следующим образом:

```
[0]kdb> bp scull_read
Instruction(i) BP #0 at 0xcd087c5dc (scull_read)
is enabled globally adjust 1
[0]kdb> go
```

Команда ***bp*** сообщает ***kdb***, что в следующий раз требуется остановиться, когда ядро входит в ***scull_read***. Затем введите **go** для продолжения выполнения. Затем направьте что-то в одно из устройств ***scull***, мы можем попытаться прочитать это, запустив ***cat*** под оболочкой на другом терминале, что приведёт к следующему:

```
Instruction(i) breakpoint #0 at 0xd087c5dc (adjusted)
0xd087c5dc scull_read: int3
Entering kdb (current=0xcf09f890, pid 1575) on processor 0 due to
Breakpoint @ 0xd087c5dc
[0]kdb>
```

Мы теперь позиционируемся в начале ***scull_read***. Чтобы увидеть, как мы туда попали, мы

можем сделать трассировку стека:

```
[0]kdb> bt
    ESP    EIP        Function (args)
0xcdbddf74 0xd087c5dc [scull]scull_read
0xcdbddf78 0xc0150718 vfs_read+0xb8
0xcdbddfa4 0xc01509c2 sys_read+0x42
0xcdbddfc4 0xc0103fcf syscall_call+0x7
[0]kdb>
```

kdb пытается распечатать аргументы для каждой функции при трассировке вызова. Однако, оказался запутанным приёмами оптимизации, используемыми компилятором. Таким образом, он не смог распечатать аргументы ***scull_read***.

Время взглянуть на некоторые данные. Команда ***mds*** манипулирует данными; мы можем запросить значение указателя ***scull_devices*** такой командой:

```
[0]kdb> mds scull_devices 1
0xd0880de8 cf36ac00 ....
```

Здесь мы запросили одно (4 байта) слово данных, начиная с местонахождения ***scull_devices***; ответ говорит нам, что наш массив устройств находится по адресу 0xd0880de8; сама первая структура устройство находится по адресу 0xcf36ac00. Чтобы посмотреть на структуру этого устройства, мы должны использовать этот адрес:

```
[0]kdb> mds cf36ac00
0xcf36ac00 ce137dbc ....
0xcf36ac04 00000fa0 ....
0xcf36ac08 000003e8 ....
0xcf36ac0c 0000009b ....
0xcf36ac10 00000000 ....
0xcf36ac14 00000001 ....
0xcf36ac18 00000000 ....
0xcf36ac1c 00000001 ....
```

Эти восемь строк здесь соответствуют начальной части структуры **scull_dev**. Таким образом, мы видим, что память первого устройства выделена по адресу 0xce137dbc, квант размером 4000 (hex fa0), квантовый набор имеет размер 1000 (hex 3e8) и в настоящее время насчитывается 155 (hex 9b) байт, хранящихся в устройстве .

kdb может также изменить данные. Предположим, что мы захотели изменить некоторые данные в устройстве:

```
[0]kdb> mm cf36ac0c 0x50
0xcf36ac0c = 0x50
```

Последующий ***cat*** на устройстве теперь возвращает меньше данных, чем раньше. ***kdb*** имеет ряд других возможностей, включая пошаговое выполнение (по инструкциям, а не строкам исходного кода Си), установку точек останова на доступ к данным, дизассемблирование кода, продвижение через связные списки, доступ к регистровым данным и многое другое. После того, как вы применили патч ***kdb***, полный набор страниц документации можно найти в каталоге ***Documentation/kdb*** в дереве исходных текстов ядра.

Патчи kgdb

Два интерактивных подхода к отладке, которые мы рассматривали до сих пор (с помощью ***gdb*** на ***/proc/kcore*** и ***kdb***), не отвечают условиям среды, к которым уже привыкли разработчики приложений пользовательского пространства. Разве не было бы хорошо, если бы там был настоящий отладчик для ядра, который поддерживает такие функции, как изменение переменных, точки останова и так далее?

Как оказалось, такое решение действительно существует. На момент написания статьи находятся в обращении два отдельных патча, которые позволяют ***gdb*** иметь полные возможности, которые будут работать с ядром. Как ни странно, оба эти патча называются ***kgdb***. Они работают путём разделения системы с запущенным тестовым ядром от системы с работающим отладчиком; как правило, они связываются с помощью последовательного кабеля. Таким образом, разработчик может запустить ***gdb*** на его или её стабильной настольной системе, работая с ядром, запущенным на жертвенном тестовом ящике. Настройка ***gdb*** в этом режиме занимает в самом начале немного времени, но инвестиции могут быстро окупиться, когда появится трудная ошибка.

Эти патчи находятся в состоянии сильного потока и даже могут быть объединены какой-то момент, поэтому мы избегаем многое о них говорить за пределами того, где они находятся и их основных возможностей. Заинтересованным читателям рекомендуется посмотреть и увидеть текущее положение дел. Первый патч ***kgdb*** в настоящее время находится в **-mm** дерева ядра - плацдарма для патчей на их пути в главную линию версии 2.6. Эта версия патча поддерживает архитектуры x86, SuperH, ia64, x86_64, SPARC и 32-х разрядную PPC. В дополнение к обычному режиму работы через последовательный порт, эта версия ***kgdb*** также может связываться через локальную сеть. Это просто вопрос включения режима Ethernet и загрузки с установленным параметром **kgdboe** для указания IP адреса, с которого могут приходить команды отладки. Как его установить, описывается в документации ***Documentation/i386/kgdb*** . (* Однако, она забывает указать, что вы должны иметь драйвер сетевого адаптера, встроенный в ядро, или отладчик не сможет найти его во время загрузки и выключит себя.)

В качестве альтернативы вы можете использовать патч ***kgdb***, находящийся на http://kgdb.sf.net/. Эта версия отладчика не поддерживает режим связи по сети (хотя, как говорят, находится в стадии разработки), но он имеет встроенную поддержку работы с загружаемыми модулями. Он поддерживает архитектуры x86, x86_64, PowerPC и S/390.

Вариант Linux для пользовательского режима

User-Mode Linux (UML, Linux пользовательского режима) является интересной концепцией. Он структурирован в виде отдельного варианта ядра Linux с собственным подкаталогом ***arch/um***. Однако, он не будет работать на новом типе оборудования; вместо этого он запускается на виртуальной машине, основанной на интерфейсе системных вызовов Linux. Таким образом, UML позволяет ядру Linux работать в качестве отдельного процесса в режиме пользователя на системе Linux.

Обладание копией ядра, работающей как процесс в режиме пользователя, приносит целый ряд преимуществ. Так как она запущена на ограниченном, виртуальном процессоре, ядро с ошибками не сможет повредить "реальной" системы. Различные аппаратные и программные конфигурации могут быть легко опробированы на том же компьютере. И, вероятно, что наиболее важно для разработчиков ядра, с ядром в пользовательском режиме можно легко манипулировать с помощью ***gdb*** или другого отладчика.

В конце концов, это просто другой процесс. UML, несомненно, располагает потенциалом для ускорения разработки ядра.

Однако, UML имеет большой недостаток с точки зрения авторов драйверов: в пользовательском режиме ядро не имеет доступа к установленному в системе оборудованию. Таким образом, хотя он может быть полезен для отладки большинства примеров драйверов в этой книге, UML ещё не полезен для отладки драйверов, которые имеют дело с реальным оборудованием. Для более подробной информации о UML смотрите http://user-mode-linux.sf.net/.

Linux Trace Toolkit

Linux Trace Toolkit (LTT) (пакет для трассировки Linux) является патчем для ядра и устанавливает соответствующие утилиты, которые позволяют трассировать события в ядре. Трассировка включает в себя информацию о времени и может создать довольно полную картину того, что произошло в течение определённого периода времени. Таким образом, он может быть использован не только для отладки, но и для отслеживания проблем с производительностью.

LTT, наряду с обширной документацией, можно найти на http://www.opersys.com/LTT.

Dynamic Probes

Dynamic Probes (или DProbes) (динамические датчики) является инструментом отладки, выпущенным (под GPL) IBM для Linux на архитектуре IA-32. Они позволяют размещение "зонда" практически в любом месте в системе, и в пространстве пользователя, и в пространстве ядра. Зонд состоит из некоторого кода (написанного на специальном, стек-ориентированном языке), который выполняется, когда управление попадает в заданную точку. Этот код может сообщить информацию обратно в пространство пользователя, изменить регистры, или же сделать ряд других вещей. Полезной особенностью DProbes является то, что после встраивания в ядро зонды могут быть вставлены в любое место в работающей системе без сборок ядра или перезагрузок. DProbes также могут работать с LTT, чтобы вставить новые события трассировки в произвольных местах.

Инструмент DProbes можно загрузить с сайта открытых исходников IBM: http://oss.software.ibm.com.

Глава 5, Конкуренция и состояния состязаний

До сих пор мы уделяли мало внимания проблеме конкуренции - то есть тому, что происходит, когда система пытается сделать больше, чем что-то одно одновременно. Управление конкуренцией является, однако, одной из основных проблем в программировании операционных систем. Ошибки, связанные с конкуренцией, одни из самых лёгких для создания и те, которые труднее всего найти. Даже опытные программисты ядра Linux в конечном итоге периодически допускают ошибки, связанные с конкуренцией.

В ранних ядрах Linux было относительно мало источников конкуренции. Симметричные многопроцессорные системы (Symmetric multiprocessing systems, SMP) не поддерживались ядром и единственной причиной одновременного исполнения было обслуживание аппаратных прерываний. Такой подход предлагает простоту, но это больше не работает в мире, который ценит производительность систем со всё большим и большим числом процессоров, и настаивает на том, чтобы система быстро реагировала на события. В ответ на эти требования современного оборудования и приложений ядро Linux превратилось в точку, где происходит одновременно много больших событий. Эта эволюция привела к гораздо более высокой производительности и масштабируемости. Это, однако, значительно усложняет программирование ядра. Программисты драйверов устройств должны теперь учитывать конкуренцию в их разработке с самого начала и они должны иметь чёткое понимание возможностей, предоставляемых ядром для управления конкуренцией.

Цель этой главы - начать процесс создания такого понимания. С этой целью мы вводим средства, которые сразу же применяются в драйвере ***scull*** из Главы 3 39. Другие средства, представленные здесь, не будут использоваться ещё некоторое время. Но сначала мы взглянем на то, что может пойти не так с нашим простым драйвером ***scull*** и как избежать этих потенциальных проблем.

Ловушки в scull

Давайте кратко рассмотрим фрагмент кода управления памятью в ***scull***. Глубоко внутри логики ***write***, ***scull*** должен решить, требуется ли выделение памяти или нет. Фрагмент кода, который решает эту задачу:

```
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
```

```
    if (!dptr->data[s_pos])
        goto out;
}
```

Предположим на минуту, что два процесса (назовём их "А" и "Б") независимо друг от друга пытаются записать по тому же самому смещению в то же самое устройство ***scull***. Каждый процесс доходит до проверки **if** в первой строке вышеприведённого фрагмента в одно и то же время. Если указатель в запросе является NULL, каждый процесс примет решение о выделении памяти, каждый в результате присвоит указателю значение для **dptr->data [s_pos]**. Поскольку оба процесса указывают на одно и то же место, ясно, что лишь одно из присвоений будет преобладать.

Что приведёт, конечно, к тому, что процесс, который завершает установку вторым и будет "победителем". Если процесс А делает присвоение первым, его назначение будет перезаписано процессом Б. В этот момент ***scull*** полностью забудет о памяти, которая выделяется А; он имеет только указатель на память процесса Б. Память, выделенная таким образом процессом А, будет потеряна и никогда не вернётся к системе.

Такая последовательность событий является демонстрацией ***гонок условий*** (или ***состояния гонок***). Состояния гонок являются результатом неконтролируемого доступа к общим данным. Неправильная модель доступа приводит к неожиданным результатам. Для гонки условий, обсуждаемой здесь, результатом является утечка памяти. Это плохо, но состояния гонок часто могут привести к падениям системы, повреждённым данным, а также к проблемам с безопасностью. Программисты могут поддаться искушению игнорировать состояния гонок как событие с крайне низкой вероятностью. Но в компьютерном мире события "одно на миллион" могут происходить раз в несколько секунд и последствия могут быть серьёзными.

В ближайшее время мы ликвидируем гонки условий в ***scull***, но сначала мы должны получить более общее представление о конкуренции.

Конкуренция и управление ей

В современной системе Linux существует множество источников конкуренции и, следовательно, возможны состояния гонок. Работают множество процессов в пользовательском пространстве и они могут получить доступ к вашему коду удивительной комбинацией способов. SMP системы могут выполнять ваш код одновременно на нескольких процессорах. Код ядра является вытесняемым; код вашего драйвера может лишиться процессора в любой момент, а процесс, который его заменит, также мог бы выполняться в вашем драйвере. Прерывания устройства являются асинхронными событиями, которые могут вызвать одновременное выполнение вашего кода. Ядро также обеспечивает различные механизмы для задержанного выполнения кода, такие как очереди задач (workqueues), микрозадачи (tasklets) и таймеры (timers), которые могут привести к запуску вашего кода в любое время способами, не связанными с тем, что делает текущий процесс. В современном мире "горячей" замены ваше устройство может просто исчезнуть в то время, как вы находитесь в середине работы с ним.

Избегание состояний гонок может оказаться весьма непростой задачей. В мире, где в любое время может произойти всё что угодно, каким образом программисту драйвера избежать создания абсолютного хаоса? Как оказалось, большинство состояний гонок можно избежать с помощью некоторого размышления, примитивов ядра для управления конкуренцией и применения нескольких базовых принципов. Мы начнём с принципов, а затем узнаем

специфику их применения.

Состояния гонок возникают в результате совместного доступа к ресурсам. Когда два потока исполнения (* Для целей применения в этой главе "поток" исполнения является любым описанием выполняющегося кода. Каждый процесс является, очевидно, потоком исполнения, а так же обработчик прерываний или другой код, который выполняется в ответ на асинхронные события ядра.) имеют причину поработать с одними и теми же структурами данных (или аппаратными ресурсами), всегда существует потенциал для смешения. Поэтому первым эмпирическим правилом, которое надо иметь в виду, когда вы разрабатываете свой драйвер, является избегание общих ресурсов, когда это возможно. Если нет одновременного доступа, не может быть никаких состояний гонок. Так что тщательно написанный код ядра должен иметь минимум общих ресурсов. Наиболее очевидное применение этой идеи заключается в отказе от использования глобальных переменных. Если вы помещаете ресурс в место, где его могут найти более одного потока исполнения, для этого должны быть веские основания.

Однако, неоспоримый факт, что такой обмен требуется часто. Ресурсы оборудования, которые по своей природе общие, а также программные ресурсы зачастую должны быть доступны более чем одному потоку. Имейте в виду также, что глобальные переменные далеко не единственный способ совместного использования данных; в любой момент ваш код передаёт указатель на другую часть ядра, это потенциально создаёт новую ситуацию доступа к общим данным. Общие ресурсы - это факт жизни.

Вот жёсткое правило разделения ресурсов: в любое время, когда ресурс аппаратного или программного обеспечения разделяется не одним потоком исполнения и существует вероятность того, что один поток может столкнуться с несоотвествующим видом этого ресурса, вы должны управлять доступом к этому ресурсу явно. В приведенном выше примере для ***scull*** мнение о ситуации процесса Б несостоятельно; незнание, что процесс А уже выделил память для (общего) устройства, он выполняет своё собственное выделение и перезаписывает работу А. В этом случае мы должны контролировать доступ к структуре данных ***scull***. Мы должны организовать вещи так, чтобы код либо видел память, которая была выделена, или знал, что память не выделена ***или будет выделена*** кем-то ещё. Обычная техника для управления доступом, названная ***блокировкой*** или ***взаимным исключением***, гарантирует, что в любое время общим ресурсом может манипулировать только один поток исполнения. Оставшаяся большая часть этой главы будет посвящена блокировке.

Прежде всего, однако, мы должны кратко рассмотреть одно важное правило. Когда код ядра создаёт объект, который будет использоваться совместно с любой другой частью ядра, такой объект должен продолжать своё существование (и нормально функционировать), пока не станет известно, что никаких внешних ссылок на него больше не существует . В момент, когда ***scull*** делает свои устройства доступными, он должен быть готов для обработки запросов на эти устройства. И ***scull*** должен продолжать быть в состоянии обрабатывать запросы на свои устройства, пока не узнает, что ссылок (таких, как открытые файлы в пространстве пользователя) на эти устройства больше не существует. Из этого правила следуют два требования: объект не может быть сделан доступным для ядра, пока он не находится в состоянии для правильного функционирования, и ссылки на такие объекты должны отслеживаться. В большинстве случаев вы обнаружите, что подсчёт ссылок для вас делает ядро, но всегда есть исключения.

После соблюдения вышеуказанных правил требуется планирование и тщательное внимание к деталям. Легко изумиться одновременным доступом к ресурсам, если вы не поняли, что они были общими. Однако, с некоторым усилием большинство состояний гонок можно предотвратить прежде, чем они укусят вас или ваших пользователей.

Семафоры и мьютексы

Давайте теперь посмотрим, каким образом мы можем добавить блокировку в ***scull***. Нашей целью является сделать в ***scull*** наши операции со структурами данных ***атомарными***, а это означает, что вся операция выполняется сразу, до того, как затребована другими потоками исполнения. В нашем примере с утечкой памяти нам необходимо обеспечить, чтобы если один поток считает, что должен быть выделен отдельный кусок памяти, он имел бы возможность выполнить это выделение перед тем, как любой другой поток сможет выполнить эту же проверку. Для этого мы должны создать ***критические секции***: код, который в любой данный момент времени может быть выполнен только одним потоком.

Не все критические секции похожи, так что для разных потребностей ядро предоставляет различные примитивы. В этом случае каждое обращение к структуре данных ***scull*** происходит в контексте процесса, как результат прямого запроса от пользователя; нет запросов, которые будут сделаны обработчиками прерываний или другими асинхронными контекстами. Нет никаких особых требований к латентности (времени отклика); прикладные программисты понимают, что запросы ввода/вывода обычно не удовлетворяются немедленно. Кроме того, ***scull*** не удерживает какие-то другие важные системные ресурсы во время доступа к его собственным структурам данных. Это всё означает то, что если драйвер ***scull*** засыпает в ожидании своей очереди на доступ к структурам данных, никого это не тревожит.

"Идти спать" в этом контексте является чётко определенным термином. Когда процесс в Linux достигает точки, где он не может выполнять любые дальнейшие действия, он переходит в спящий режим (или "блокируется"), уступая процессор кому-то другому, пока когда-то в будущем он не сможет снова начать работать. Процессы часто засыпают ожидая завершения ввода/вывода. По мере углубления в ядро мы будем сталкиваться с разными ситуациями, в которых мы не можем спать. Однако, метод ***write*** в ***scull*** - не одна из этих ситуаций. Так что мы можем использовать механизм блокировки, который может послать этот процесс в спячку в ожидании доступа к критической секции.

Важно, что мы будем выполнять операцию (распределение памяти с ***kmalloc***), которая может заснуть, так что в любом случае возможны бездействия. Чтобы наши критические секции работали должным образом, мы должны использовать блокирующий примитив, который работает, когда поток, который владеет блокировкой, спит. Где возможно засыпание, могут быть использованы не все механизмы блокировки (далее в этой главе мы увидим такие, которые не делают это). Однако, для наших нынешних потребностей механизм, который подходит лучше всего, это ***семафор***.

Семафоры - хорошо понимаемая концепция в компьютерной науке. По своей сути, семафор это одно целое значение в сочетании с парой функций, которые обычно называются ***P*** и ***V***. Процесс, желающий войти в критическую секцию, вызовет ***P*** на соответствующем семафоре; если в семафоре значение больше нуля, это значение уменьшается на единицу и этот процесс продолжается. Если, наоборот, в семафоре значение равно 0 (или меньше), процесс должен ждать, пока кто-нибудь другой освободит семафор. Разблокирование семафора осуществляется вызовом ***V***; эта функция увеличивает значение семафора и, если необходимо, будит ожидающие процессы.

Когда семафоры используются для ***взаимного исключения***, предохраняя множество процессов от одновременного выполнения в критической секции, их значение будет проинициализировано в 1. Такой семафор в любой данный момент времени может удерживаться только одним процессом или потоком. Семафор, используемый в этом режиме, иногда называют ***мьютекс (флаг)***, что, конечно же, расшифровывается как "взаимное

исключение" (mutex, mutual exclusion). Почти все семафоры, найденные в ядре Linux, используются для взаимного исключения.

Реализация семафоров в Linux

Ядро Linux обеспечивает реализацию семафоров, которая соответствует вышеприведённой семантике, хотя терминология немного отличается. Для использования семафоров код ядра должен подключить ***<asm/semaphore.h>***. Соответствующим типом является **struct semaphore**; фактические семафоров могут быть объявлены и проинициализированы несколькими способами. Одним является прямое создание семафора и инициализация его затем с помощью ***sema_init***:

```
void sema_init(struct semaphore *sem, int val);
```

где **val** является начальным значением для присвоения семафору.

Обычно, однако, семафоры используются в режиме мьютекса. Чтобы сделать использование этого общего случая немного легче, ядро обеспечивает набор вспомогательных функций и макросов. Таким образом, мьютекс может быть объявлен и проинициализирован одним из следующих действий:

```
DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
```

Здесь результатом является переменная семафора (названная **name**), которая инициализируется в **1** (в **DECLARE_MUTEX**) или в 0 (в **DECLARE_MUTEX_LOCKED**). В последнем случае мьютекс имеет начальное заблокированное состояние; он должен быть явным образом разблокирован до того, как какому-либо потоку будет разрешён доступ.

Если мьютекс должен быть проинициализирован во время исполнения (как в случае, если он создаётся динамически, например), используйте одно из следующих действий:

```
void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
```

В мире Linux ***P*** функция названа ***down*** или некоторой вариацией этого имени. Здесь, "вниз" (down) указывает на тот факт, что функция уменьшает значение семафора и возможность вызывающего заснуть после вызова на какое-то время для ожидания, пока семафор станет доступным и предоставит доступ к защищённым ресурсам. Есть три версии ***down***:

```
void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
```

down уменьшает значение семафора и ждёт столько, сколько необходимо. ***down_interruptible*** делает то же самое, но эта операция прерываемая. Прерываемая версия - это почти всегда та, которую вы будете хотеть; она позволяет процессу пространства пользователя, который ожидает на семафоре, быть прерванным пользователем. Как правило, вы не захотите использовать непрерываемые операции и сделаете это, если действительно нет альтернативы. Непрерываемые операции являются хорошим способом создания неубиваемых процессов (опасное "D state" ("состояние D") показываемое ***ps***) и раздражения

ваших пользователей. Однако, использование ***down_interruptible*** требует некоторой дополнительной заботы, если операция прервана, функция возвращает ненулевое значение и вызывающий не удерживает семафор. Правильное использование ***down_interruptible*** всегда требует проверки возвращаемого значения и соответствующего реагирования.

Последний вариант (***down_trylock***) никогда не засыпает; если семафор не доступен во время вызова, ***down_trylock*** немедленно возвращается с ненулевым значением.

После успешного вызова потоком одной из версий ***down***, он считается "удерживающим" семафор (или "получает" или "приобретает" семафор). Теперь этот поток имеет право на доступ к критической секции, защищаемой семафором. После того, как операции, требующие взаимного исключения, закончены, семафор должен быть возвращён. Эквивалентом ***V*** в Linux является ***Up***:

```
void up(struct semaphore *sem);
```

Как только ***up*** была вызвана, вызывающий больше не удерживает семафор.

Как и следовало ожидать, требуется, чтобы любой поток, который получает семафор, освободил его одним (и только одним) вызовом ***up***. Часто при обработке ошибок необходимо особое внимание; если во время удержания семафора произошла ошибка, перед возвратом статуса ошибки вызывающему этот семафор должен быть освобождён. Допустить ошибку с неосвобождением семафора легко; результат (процессы висят в кажущихся несвязанными местах) может быть трудно воспроизводимым и отслеживаемым.

Использование семафоров в scull

Семафорный механизм даёт ***scull*** инструмент, который можно использовать, чтобы избежать состояний гонок во время доступа к структуре данных **scull_dev**. Но на надо использовать этот инструмент правильно. Ключами к правильному использованию блокирующих примитивов являются точные указания, какие ресурсы должны быть защищены, и гарантия, что любой доступ к этим ресурсам использует правильную блокировку. В нашем примере драйвера всё представляющее интерес содержится в структуре **scull_dev**, так что она является логичной сферой применения для нашего режима блокировки.

Давайте взглянем ещё раз на эту структуру

```
struct scull_dev {
    struct scull_qset *data; /* Указатель, установленный на первый квант */
    int quantum;             /* размер текущего кванта */
    int qset;                /* размер текущего массива */
    unsigned long size;      /* количество данных, хранимых здесь */
    unsigned int access_key; /* используется sculluid и scullpriv */
    struct semaphore sem;    /* семафор взаимного исключения */
    struct cdev cdev;        /* структура символьного устройства */
};
```

В нижней части структуры имеется член, названный **sem**, который, конечно же, наш семафор. Мы решили использовать отдельный семафор для каждого виртуального устройства ***scull***. Также было бы правильно использовать единственный глобальный семафор. Однако, различные устройства ***scull*** не имеют общих ресурсов и нет оснований делать так, чтобы один процесс ожидал, пока другой процесс работает с другим устройством ***scull***. Использование

отдельного семафора для каждого устройства позволяет операциям на разных устройствах выполняться параллельно и, следовательно, повышает производительность.

Семафоры должны быть проинициализированы перед использованием. ***scull*** выполняет эту инициализацию при загрузке в данном цикле:

```
for (i = 0; i < scull_nr_devs; i++) {
    scull_devices[i].quantum = scull_quantum;
    scull_devices[i].qset = scull_qset;
    init_MUTEX(&scull_devices[i].sem);
    scull_setup_cdev(&scull_devices[i], i);
}
```

Обратите внимание, что семафор должен быть проинициализирован ***до того***, как устройство ***scull*** становится доступным остальной части системы. Таким образом, ***init_MUTEX*** вызывается перед ***scull_setup_cdev***. Выполнение этих операций в обратном порядке создаст состояние гонок, где семафор может быть доступен, когда ещё не подготовлен.

Далее, мы должны просмотреть код и убедиться, что нет обращений к структуре данных ***scull_dev***, производящихся без удерживания семафора. Так, например, ***scull_write*** начинается таким кодом:

```
if (down_interruptible(&dev->sem))
    return -ERESTARTSYS;
```

Обратите внимание на проверку возвращаемого значения ***down_interruptible***; если она возвращает ненулевое значение, операция была прервана. Обычной вещью в этой ситуации является возвращение **-ERESTARTSYS**. Увидев этот код возврата, высшие слои ядра либо заново сделают вызов или вернут ошибку пользователю. Если вы возвращаете **-ERESTARTSYS**, вы должны сначала отменить любые видимые пользователю изменения, которые могли быть сделаны, так чтобы при повторном системном вызове всё сработало правильно. Если вы не можете отменить что-то таким образом, вы должна вернуть взамен **-EINTR**.

scull_write должна освободить семафор независимо от того, было ли возможно успешно выполнять свои другие задачи. Если всё пойдёт хорошо, выполнение функции заканчивается в конце несколькими строчками:

```
out:
    up(&dev->sem);
    return retval;
```

Этот код освобождает семафор и возвращает вызывающему какой-то статус. В ***scull_write*** есть несколько мест, где дела могут пойти не так; сюда относятся ошибки при выделении памяти или ошибки при попытке скопировать данные из пространства пользователя. В этих случаях для обеспечения выполнения корректной очистки код выполняет **goto out**.

Чтение/Запись семафоров

Семафоры выполняют взаимное исключение для всех вызывающих, независимо от того, что каждый поток может захотеть сделать. Однако, многие задачи распадаются на два отдельных

типа работы: задачи, которым надо только прочитать защищаемые структуры данных и те, которые должны сделать изменения. Часто можно позволить несколько одновременных читателей, пока никто пытается сделать каких-то изменений. Это может значительно оптимизировать производительность; только читающие задачи могут выполнить свою работу параллельно, не дожидаясь, пока другой читатель покинет критическую секцию.

Для этой ситуации ядро Linux предоставляет специальный тип семафора, названный ***rwsem*** (или "семафор чтения/записи"). Семафоры ***rwsem*** используются в драйверах относительно редко, но они иногда полезны.

Код, использующий **rwsem**-ы, должен подключить ***<linux/rwsem.h>***. Соответствующим типом данных для семафоров чтения/записи является структура ***rw_semaphore***; ***rwsem*** должен быть явно проинициализирован во время работы с помощью:

```
void init_rwsem(struct rw_semaphore *sem);
```

Вновь проинициализированный **rwsem** доступен для последующих задач (читателя или писателя), использующих его. Интерфейс для кода, нуждающегося в доступе только для чтения:

```
void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);
```

Вызов ***down_read*** обеспечивает доступ к защищённым ресурсам только для чтения, возможный одновременно для разных читателей. Обратите внимание, что ***down_read*** может поместить вызывающий процесс в непрерываемый сон. ***down_read_trylock*** не будет ждать, если доступ на чтение невозможен; она возвращает ненулевое значение, если доступ был предоставлен, 0 в противном случае. Обратите внимание, что соглашение для ***down_read_trylock*** отличается от большинства функций ядра, где успех обозначается возвращаемым значением 0. **rwsem**, полученный от ***down_read***, должен в конечном итоге быть освобождён с помощью ***up_read***.

Интерфейс для записи аналогичен:

```
void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);
```

down_write, ***down_write_trylock*** и ***up_write*** все ведут себя так же, как и их коллеги читатели, за исключением, конечно, что они обеспечивают доступ для записи. Если у вас возникла ситуация, когда для быстрого изменения необходима блокировка записи с последующим длительным периодом доступа только на чтение, вы можете использовать ***downgrade_write***, чтобы разрешить работу другим читателям, как только вы завершили внесение изменений.

rwsem разрешает либо одного писателя, либо неограниченное число читателей для удержания семафора. Писатели получают приоритет; как только писатель пытается войти в критическую секцию, читатели не будут допускаться, пока все писатели не завершат свою работу. Эта реализация может привести к голоду читателя, когда читатели долгое время не имеют доступа, если у вас есть большое число писателей, борющихся за семафор. По этой

причине ***rwsem***-ы лучше всего использовать, когда доступ для записи требуется редко, и писатель удерживает доступ короткий период времени.

Завершения

Общепринятая модель в программировании ядра предполагает инициализацию какой-либо деятельности за пределами текущего потока, который ожидает завершения такой активности. Эта деятельность может быть созданием нового потока ядра или процесса в пользовательском пространстве, запрос к существующим процессам, или какой-то вид действий, связанных с оборудованием. В таких случаях может возникнуть соблазн использовать для синхронизации двух задач семафоры таким кодом:

```
struct semaphore sem;

init_MUTEX_LOCKED(&sem);
start_external_task(&sem);
down(&sem);
```

Внешняя задача может затем по завершении своей работы вызвать ***up(&sem)***.

Как оказалось, семафоры не лучший инструмент для использования в этой ситуации. При нормальном использовании код, пытающийся блокировать семафор, находит, что семафор доступен почти всё время; если есть значительные соперничество за семафор, страдает производительность и схема блокировки должна быть пересмотрена. Так что семафоры были сильно оптимизированы для "подходящих" случаев. Однако, при использовании для общения при завершении задачи показанным выше способом поток, вызывающий ***down***, почти всегда вынужден ждать; соответственно, пострадает производительность. При использовании этого способа семафоры могут также стать предметом (неприятного) состояния гонок, если они объявлены как автоматические переменные. В некоторых случаях семафор мог бы исчезнуть до того, как процесс, вызывающий ***up***, закончит с ним работу.

Эти опасения побудили добавить в ядре версии 2.4.7 интерфейс "завершения" (completions). Ожидание завершения является легковесным механизмом с одной задачей: позволить одному потока рассказать другому, что работа выполнена. Чтобы воспользоваться механизмом завершения, ваш код должен подключить ***<linux/completion.h>***. "Завершение" может быть создано с помощью:

```
DECLARE_COMPLETION(my_completion);
```

Или, если "завершение" должно быть создано и проинициализировано динамически:

```
struct completion my_completion;
/* ... */
init_completion(&my_completion);
```

Ожидание "завершения" является простым делом вызова:

```
void wait_for_completion(struct completion *c);
```

Обратите внимание, что эта функция выполняет непрерываемое ожидание. Если ваш код вызовет ***wait_for_completion*** и никто никогда не завершит задачу, результатом будет неубиваемый процесс. (* На момент написания, патчи, добавляющие прерываемые версии, были в обращении, но не были объединены с главной веткой.)

С другой стороны, о фактическом завершении события можно просигнализировать одним из следующих вызовов:

```
void complete(struct completion *c);
void complete_all(struct completion *c);
```

Эти две функции ведут себя иначе, если более, чем один поток ожидает того же сообщения завершения. ***complete*** будит только один из ожидающих потоков, а ***complete_all*** позволяет продолжиться всем. В большинстве случаев есть только один ожидающий и две функции будут давать идентичные результаты.

"Завершение", как правило, одноразовое устройство; оно используется однажды, а затем отбрасывается. Однако, вполне возможно повторное использование структур "завершения", если об этом правильно позаботиться. Если ***complete_all*** не используется, структура "завершения" может быть использована снова без каких-либо проблем, пока нет никакой двусмысленности в том, о каком событии в настоящее время просигнализировано. Но если вы используете ***complete_all***, вы должны переинициализировать структуру "завершения" перед повторным использованием. Этот макрос:

```
INIT_COMPLETION(struct completion c);
```

может быть использован для быстрого выполнения такой повторной инициализации.

В качестве примера того, как может быть использовано "завершение", рассмотрим модуль ***complete***, который включён в исходник примера. Этот модуль определяет устройство с простой семантикой: любой процесс, пытающийся прочитать из устройства будет ждать (используя ***wait_for_completion***), пока в устройство пишет какой-нибудь другой процесс. Код, который реализует это поведение:

```
DECLARE_COMPLETION(comp);

ssize_t complete_read (struct file *filp, char __user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
    current->pid, current->comm);
    wait_for_completion(&comp);
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* EOF */
}

ssize_t complete_write (struct file *filp, const char __user *buf, size_t
count, loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
    current->pid, current->comm);
    complete(&comp);
    return count; /* успешно, избегаем повтора */
}
```

Можно иметь несколько процессов "читающих" из этого устройства в одно и то же время. Каждая запись в устройство вызовет по завершении только одну операцию чтения, но нет

способа узнать, какую именно.

Типичным использованием механизма завершения является прекращение потока ядра во время завершения работы модуля. В случае прототипа, какое-то действие внутри драйвера выполняется потоком ядра в цикле **while (1)**. Когда модуль готов к очистке, функция выхода сообщает потоку, что завершается, а затем ожидает завершения. С этой целью, ядро включает в себя специфическую функцию, которая будет использоваться потоком:

```
void complete_and_exit(struct completion *c, long retval);
```

Спин-блокировки

Семафоры являются полезным инструментом для взаимного исключения, но они являются не единственным таким инструментом, предоставляемым ядром. Вместо этого, большинство блокировок осуществляется механизмом, названным ***спин-блокировкой***. В отличие от семафоров, спин-блокировки могут быть использованы в коде, который не может спать, таком, как обработчики прерываний. При правильном применении, спин-блокировки предлагают в целом более высокую производительность, чем семафоры. Они, однако, имеют другой набор ограничений на своё использование.

Спин-блокировки - простая концепция. Спин-блокировка является взаимным исключением устройства, которое может иметь только два значения: "заблокировано" и "разблокировано". Это обычно реализуется как один бит в целом числе. Код, желающий забрать какую-либо определённую блокировку, проверяет соответствующий бит. Если блокировка доступна, "блокирующий" бит устанавливается и этот код продолжает работу в критической секции. Вместо этого, если блокировка уже захвачена кем-то другим, код переходит в короткий цикл, где постоянно проверяет блокировку, пока она не станет доступна. Этот цикл является "спин" частью спин-блокировки.

Конечно, реальная реализация спин-блокировки немного более сложная, чем описание выше. Операция "проверить и установить" должна быть выполнена атомарным образом, так, чтобы только один поток смог получить блокировку, даже если несколько из них в любой момент времени вращаются в ожидании доступа. Также необходимо избегать взаимоблокировок на многопоточных (hyperthreaded) процессорах - чипах, которые реализуют несколько виртуальных процессоров на одном процессорном ядре и кэш-памяти. Так что реальная реализация спин-блокировки отличается для каждой архитектуры, поддерживаемой Linux. Тем не менее, основная концепция одинакова на всех системах, когда есть соперничество за спин-блокировку, процессоры, которые ждут, выполняют короткий цикл и совершают бесполезную работу.

Спин-блокировки, в силу их особенности, предназначены для использования на многопроцессорных системах, хотя однопроцессорная рабочая станция, работающая на вытесняющем ядре, ведёт себя в отношении конкуренции как SMP. Если бы невытесняющая однопроцессорная система когда-нибудь вошла в блокировке в цикл, она бы осталась там навсегда; никакой другой поток никогда не смог бы получить процессор для снятия блокировки. По этой причине операции спин-блокировки на однопроцессорных системах без разрешённого вытеснения оптимизированы, чтобы ничего не делать, за исключением изменения статуса маскировки прерываний. Из-за вытеснения, даже если вы никогда не ожидаете работы своего кода на SMP системах, вам всё равно придётся осуществлять корректную блокировку.

Знакомство с API спин-блокировки

Для использования примитивов спин-блокировки необходимо подключить файл ***<linux/spinlock.h>***. Фактическая блокировка имеет тип **spinlock_t**. Как и любые другие структуры данных, спин-блокировка должна быть проинициализирована. Эта инициализация может быть сделано во время компиляции следующим образом:

```
spinlock_t my_lock = SPIN_LOCK_UNLOCKED;
```

или во время работы:

```
void spin_lock_init(spinlock_t *lock);
```

Перед входом в критическую секцию ваш код должен получить необходимую блокировку:

```
void spin_lock(spinlock_t *lock);
```

Заметим, что все ожидания в спин-блокировках непрерываемы по своей природе. После вызова ***spin_lock*** вы будете крутиться в цикле, пока блокировка не станет доступной.

Для освобождения полученной блокировки передайте её в:

```
void spin_unlock(spinlock_t *lock);
```

Для спин-блокировки есть много других функций и мы скоро посмотрим на них все. Но никто из них не отходит от основной идеи, проиллюстрированной вышепоказанными функциями. Есть очень мало того, что можно сделать с блокировкой, кроме постановки и снятия. Однако, есть несколько правил, как необходимо работать со спин-блокировками. Отвлечёмся на минутку, чтобы посмотреть на них перед изучением полного интерфейса спин-блокировки.

Спин-блокировки и контекст атомарности

Представьте на мгновение, что ваш драйвер приобретает спин-блокировку и идёт делать своё дело в критическую секцию. Где-то посередине ваш драйвер теряет процессор. Возможно, это вызвано функцией (скажем, ***copy_from_user***), которая помещает процесс в сон. Или, возможно, вытеснение в ядре вышибает его и процесс с более высоким приоритетом выталкивает ваш код в сторону. Ваш код теперь удерживает блокировку, которую он не выпустит в обозримом будущем. Если какой-то другой поток попытается получить блокировку, он будет в лучшем случае ждать (крутясь в процессоре) в течение очень долгого времени. В худшем случае в системе может случиться взаимоблокировка.

Большинство читателей согласятся, что такого сценария лучше избегать. Таким образом, основное правило, которое применяется к спин-блокировкам - любой код, удерживающий спин-блокировку, должен быть атомарным. Он не может спать; фактически, он не может отдать процессор по какой-то причине, кроме обслуживания прерываний (а иногда даже и для этого).

Случай вытеснения в ядре обрабатывается самим кодом спин-блокировки. Каждый раз, когда код ядра удерживает спин-блокировку, на соответствующем процессоре вытеснение запрещается. Даже однопроцессорные системы должны запрещать вытеснение, чтобы таким образом избежать состояния гонок. Вот почему требуется правильное блокирование, даже если вы никогда не ожидаете запуска вашего кода на многопроцессорной машины.

Избегание сна во время удержания блокировки может быть довольно трудным; многие функции ядра могут засыпать и такое поведение не всегда хорошо документировано. Копирование данных в или из пользовательского пространства является наглядным примером: необходимая страница пространства пользователя может требовать обмена с диском перед тем, как копия сможет быть обработана, и такая операция явно требует засыпания. Практически любая операция, которая должна выделить память, может заснуть; ***kmalloc*** может решить отказаться от процессора и ожидать, пока станет доступно больше памяти, если прямо не сказать не делать этого. Засыпание может произойти и в неожиданных местах; написание кода, который будет выполняться со спин-блокировкой, требует обращения внимания на каждую функцию, которую вы вызываете.

Вот ещё один сценарий: ваш драйвер исполняется и только что забрал блокировку, которая контролирует доступ к его устройству. Пока удерживается блокировка, устройство вызывает прерывание, которая запускает ваш обработчик прерываний. Обработчик прерываний перед обращением к устройству также должен получить блокировку. Получение спин-блокировки в обработчике прерывания является законной вещью; это является одной из причин того, что операции спин-блокировки не засыпают. Но что произойдёт, если подпрограмма прерывания выполняется тем же процессором, что и код, который до этого забрал спин-блокировку? Пока обработчик прерываний вращается в цикле, непрерываемый код не будет иметь возможность запуститься для снятия блокировки. Этот процессор зациклится навсегда.

Избегание этой ловушки требует отключения прерываний (только на данном процессоре) во время удержания спин-блокировки. Существуют варианты функций спин-блокировки, которые отключают для вас прерывания (мы увидим их в следующем разделе). Однако, полное обсуждение прерываний должно подождать до Главы 10 246.

Последнее важное правило для использования спин-блокировок: спин-блокировки всегда должны удерживаться минимально возможное время. Чем дольше вы держите блокировку, тем больше другой процессор может вращаться в ожидании, пока вы освободите её, и вероятность получения цикла увеличивается. Длительное время удержания блокировки также отключает текущий процессор от планировщика задач, это означает, что процесс с более высоким приоритетом, который действительно должен иметь возможность получить процессор, возможно, вынужден ждать. Разработчики ядра приложили много усилий на сокращение латентности ядра (времени, которое процессу придётся ждать до переключения на него) во время разработки версий 2.5. Плохо написанный драйвер может уничтожить все эти достижения просто слишком долго удерживая блокировку. Чтобы избежать подобной проблемы, считайте обязательным для себя удерживать блокировку минимальное время.

Функции спин-блокировки

Мы уже видели две функции, ***spin_lock*** и ***spin_unlock***, которые манипулируют спин-блокировками. Однако, есть несколько других функций с похожими именами и назначением. Сейчас мы представим полный набор. Эта обсуждение даст нам основу для ещё нескольких глав, пока мы не будем в состоянии должным образом полностью разобраться в вопросе; полное понимание API спин-блокировки требует понимания обработки прерываний и связанных с ним понятий.

Фактически, есть четыре функции (* На самом деле это макросы, определённые в <linux/spinlock.h>, а не функции. Вот почему параметр flags в spin_lock_irqsave() не является указателем, как это можно было ожидать), которые могут блокировать спин-блокировку:

```
void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);
```

Мы уже видели, как работает ***spin_lock***. ***spin_lock_irqsave*** запрещает прерывания (только на местном процессоре) до снятия блокировки; предыдущее состояние прерывания запоминается во флагах. Если вы абсолютно уверены, что никто другой не запрещал прерывания на вашем процессоре (или, другими словами, вы уверены, что должны разрешить прерывания, когда освободите вашу спин-блокировку), вы можете использовать взамен ***spin_lock_irq*** и сохранять флаги. Наконец, ***spin_lock_bh*** перед получением блокировки запрещает программные прерывания, но оставляет включёнными аппаратные прерывания.

Если у вас есть спин-блокировка, которая может быть получена кодом, который работает в контексте (аппаратного или программного) прерывания, необходимо использовать одну из форм ***spin_lock***, которая запрещает прерывания. Если поступить иначе, рано или поздно в системе случится взаимоблокировка. Если вы не обращаетесь к вашей блокировке в обработчике аппаратного прерывания, а делаете это с помощью программных прерываний (например, в коде, который запускается микрозадачами, тема рассматривается в Главе 7 174), для безопасного избегания взаимоблокировок вы можете использовать ***spin_lock_bh***, разрешая в то же время обслуживание аппаратных прерываний.

Есть также четыре способа освободить блокировку; он должен соответствовать функции, которую вы использовали для получения блокировки:

```
void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);
void spin_unlock_bh(spinlock_t *lock);
```

Каждый вариант ***spin_unlock*** отменяет работу, выполненную соответствующей функцией ***spin_lock***. Аргумент **flags**, передаваемый в ***spin_unlock_irqrestore***, должен быть той же переменной, переданной в ***spin_lock_irqsave***. Вы должны также вызвать ***spin_lock_irqsave*** и ***spin_unlock_irqrestore*** в той же самой функции; в противном случае, на некоторых архитектурах ваш код может сломаться.

Существует также набор неблокирующих операций спин-блокировки:

```
int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);
```

Эти функции возвращают ненулевое значение при успехе (блокировка была получена), иначе - 0. Не существует версии "попробовать" ("try"), которая запрещает прерывания.

Чтение/Запись спин-блокировок

Ядро предоставляет формы для чтения/записи спин-блокировок, которые полностью аналогичны чтению/записи семафоров, рассмотренному ранее в этой главе. Эти блокировки разрешают в критической секции одновременно любое количество читателей, но писатели должны иметь эксклюзивный доступ. Блокировки чтения/записи имеют тип **rwlock_t**, определённый в ***<linux/spinlock.h>***. Они могут быть объявлены и проинициализированы двумя способами:

```
rwlock_t my_rwlock = RW_LOCK_UNLOCKED; /* Статический способ */

rwlock_t my_rwlock;
rwlock_init(&my_rwlock); /* Динамический способ */
```

Список доступных функций должен выглядеть похожим на уже знакомые функции. Для читателей доступны следующие функции:

```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```

Интересно, что нет ***read_trylock***.

Функции для записи аналогичны:

```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

Блокировки чтения/записи могут оставить читателей голодными, так же как и ***rwsem***-ы. Такое поведение является редкой проблемой; однако, если есть достаточно спорящих за блокировку, чтобы вызвать зависание, производительность в любом случае плохая.

Ловушки блокировок

Многолетний опыт работы с блокировками - опыт, который предшествует Linux - показал, что может быть очень трудно правильно работать с блокировками. Управление конкуренцией является по своей сути сложной задачей и существуют множество способов допустить ошибки. В этом разделе мы бросим быстрый взгляд на то, что может пойти не так.

Сомнительные правила

Как уже было сказано выше, правильная схема блокирования требует чётких и ясных правил. При создании ресурса, к которому может быть одновременный доступен, вы должны определить, какие блокировки будут контролировать этот доступ. Блокировка должна быть действительно заложена в начале; модификация кода потом может быть трудным делом. Время, потраченное в начале, обычно щедро возмещается во время отладки.

Когда вы пишете код, вам, несомненно, потребуются несколько функций, которые требуют доступа к структурам, защищённым определённой блокировкой. На этом этапе вы должны

быть осторожны: если одна функция приобретает блокировку и затем вызывает другую функцию, которая также пытается забрать эту же блокировку, ваш код взаимоблокируется. Ни семафоры, ни спин-блокировки не позволят получить блокировку второй раз; если вы делаете такую попытку, всё просто зависнет.

Чтобы сделать ваши блокировки работающими правильно, вам придётся написать несколько функций в предположении, что вызвавший их уже приобрёл соответствующую блокировку(ки). Как правило, только внутренние, статические функции могут быть написаны таким образом; функции, вызываемые снаружи, должны обработать блокировку в явном виде. Когда вы пишете внутренние функции, которые делают предположения о блокировке, создайте себе (и всем остальным, кто работает с вашим кодом) справку и документ с этими предположениями в явном виде. Может быть очень трудно вернуться месяцы спустя и выяснить, требуется ли удерживать блокировку при вызове данной функции или нет.

В случае со ***scull***, дизайнерским решением было принято требовать, чтобы все функции, вызываемые непосредственно системными вызовами, запрашивали для доступа семафор соответствующей структуры устройства. Все внутренние функции, которые только вызываются другими функциями ***scull***, могут считать, что семафор был получен правильно.

Правила очерёдности блокировки

В системах с большим количеством блокировок (и ядро становится такой системой), необходимость проведения более чем одной блокировки за раз не является необычной для кода. Если какие-то операции должны быть выполнены с использованием двух различных ресурсов, каждый из которых имеет свою собственную блокировку, часто нет альтернативы, кроме получения обоих блокировок. Однако, получение множества блокировок может быть опасным. Если у вас есть две блокировки, названных ***Lock1*** и ***Lock2***, и коду необходимо получить их в одно и то же время, вы имеете потенциальную взаимоблокировку. Только представьте, один поток блокирует ***Lock1***, а другой одновременного забирает ***Lock2***. Затем каждый поток пытается получить ту, которую не имеет. Оба потоки будут блокированы.

Решение этой проблемы, как правило, простое: если требуется получить несколько блокировок, они всегда должны быть получены в одинаковом порядке. Придерживаясь этого правила, простых взаимоблокировок, похожих на описанную выше, можно избежать. Однако, следовать правилам порядку блокирования может быть легче сказать, чем сделать. Очень редко, когда такие правила не используются на самом деле где-то ещё. Часто лучшее, что вы можете сделать, это посмотреть, что делает другой код.

Могут помочь несколько эмпирических правил. Если вам необходимо получить блокировку, которая является локальной для вашего кода (скажем, блокировка устройства), наряду с блокировкой, принадлежащей к более центральной часть ядра, сделайте вашу блокировку первой. Если у вас есть комбинация семафоров и спин-блокировок, вы должны, конечно, получить семафор(ы) в первую очередь; вызов ***down*** (которая может заснуть) во время удержания спин-блокировки является серьёзной ошибкой. Но прежде всего постарайтесь избегать ситуаций, когда вам необходимо более одной блокировки.

Точечное блокирование против грубого

Первым ядром Linux, поддерживающим многопроцессорные системы, было 2.0; оно содержало ровно одну спин-блокировку. ***Большая блокировка ядра*** оборачивала всё ядро в одну большую критическую секцию; только один процессор мог выполнять код ядра в любой момент времени. Эта блокировка решала проблему конкуренции достаточно хорошо, чтобы

позволить разработчикам ядра обратиться к решению всех других вопросов, связанных с поддержкой SMP. Но это не очень масштабируемо. Даже двухпроцессорная система могла бы потратить значительное количество времени просто в ожидании большой блокировки ядра. Производительность четырёхпроцессорной системы даже не приближалась к таковой у четырёх независимых машин.

Таким образом, последующие версии ядра включают более точечную блокировку. В версии 2.2 одна спин-блокировка контролировала доступ к блоку подсистемы ввода/вывода, другая работала для сети и так далее. Современное ядро может содержать тысячи блокировок, каждая защищает один небольшой ресурс. Такая точечная блокировка может быть хороша для масштабируемости; это позволяет каждому процессору выполнять свою собственную задачу не соперничая за блокировки, используемые другими процессорами. Мало кто пропустил большую блокировку ядра. (* Эта блокировка всё ещё существует в версии 2.6, хотя она охватывает сейчас очень малую часть ядра. Если вы случайно натолкнулись на вызов *lock_kernel*, вы нашли большую блокировку ядра. Однако, даже не думайте об её использовании в любом новом коде.)

Однако, точечное блокирование связано с определёнными затратами. В ядре тысячи блокировок, может быть очень трудно узнать, в каких блокировках вы нуждаетесь, и в каком порядке вы должны их получать для выполнения определённой операции. Помните, что ошибки блокировки может быть очень сложно найти; больше блокировок предоставляют больше возможностей для проползания в ядро действительно неприятных ошибок блокировки. Точечная блокировка может привести к такому уровню сложности, что в долгосрочном плане может иметь большие негативные последствия для сопровождения ядра.

Блокировка в драйвере устройства, как правило, сравнительно проста; вы можете иметь одну блокировку, охватывающую всё, что вы делаете, или вы можете создать одну блокировку для каждого управляемого устройства. Как правило, вам следует начать с относительно грубой блокировки, если есть реальные основания полагать, что конкуренция может быть проблемой. Боритесь с желанием оптимизировать преждевременно; реальные ограничения производительности часто появляются в неожиданных местах.

Если вы подозреваете, что соперничество за блокировку сказывается на производительности, вы можете найти полезным инструмент ***lockmeter***. Этот патч (доступен на http://oss.sgi.com/projects/lockmeter/) - инструменты ядра для измерения времени, затраченного на ожидание в блокировках. Глядя на отчёт, вы сможете быстрее определить, действительно ли соперничество за блокировку является проблемой или нет.

Альтернативы блокированию

Ядро Linux предоставляет ряд мощных блокирующих примитивов, которые могут быть использованы для предохранения ядра от спотыкания о собственные ноги. Но, как мы видели, разработка и реализация схемы блокирования не лишены недостатков. Нередко бывает, что нет альтернативы для семафоров и спин-блокировок; они могут быть единственным способом выполнить работу правильно. Однако, есть ситуации, где может быть установлен атомарный доступ без необходимости полной блокировки. В этом разделе рассматриваются другие способы ведения дел.

Свободные от блокировки алгоритмы

Иногда вы можете переделать свои алгоритмы, чтобы вообще избежать необходимость блокирования. Множество ситуаций чтения/записи, если есть только один писатель, часто

могут работать таким образом. Если писатель позаботится о том, чтобы представление структуры данных, видимое читателем, всегда являлось согласованным, можно создать свободную от блокировки структуру данных.

Структурой данных, которая часто может быть полезной для свободных от блокировок задач поставщиков/потребителей, является ***круговой буфер***. Этот алгоритм предлагает поставщику размещение данных в один конец массива, в то время как потребитель удаляет данные с другого. Когда достигнут конец массива, производитель автоматически возвращается в начало. Таким образом, круговой буфер требует массив и два индексных значения для указания, где находится следующее новое значение и какое значение должно быть удалено из буфера следующем.

Тщательно реализованный круговой буфер не требует блокирования в отсутствие нескольких поставщиков или потребителей. Поставщик является единственным потоком, которому разрешено изменять индекс записи и массив в том месте, куда он указывает. Пока писатель заносит новое значение в буфер до обновления индекса записи, читатель будет всегда видеть согласованные данные. Читатель, в свою очередь, является единственным потоком, который может получить доступ к индексу чтения и значению, которое он указывает. Немного позаботясь, чтобы эти два указателя не вышли за границы друг друга, поставщик и потребитель могут получать доступ к буферу одновременно без состояний гонок.

Рисунок 5-1 показывает круговой буфер в нескольких состояниях заполненности. Этот буфер был определён так, что пустое состояние обозначается одинаковыми указателями чтения и записи, а состояние заполненности - когда указатель записи имеет значение непосредственно за указателем чтения (будьте внимательны с учётом переноса по достижении границ буфера!). После тщательного программирования этот буфер может быть использован без блокировок.

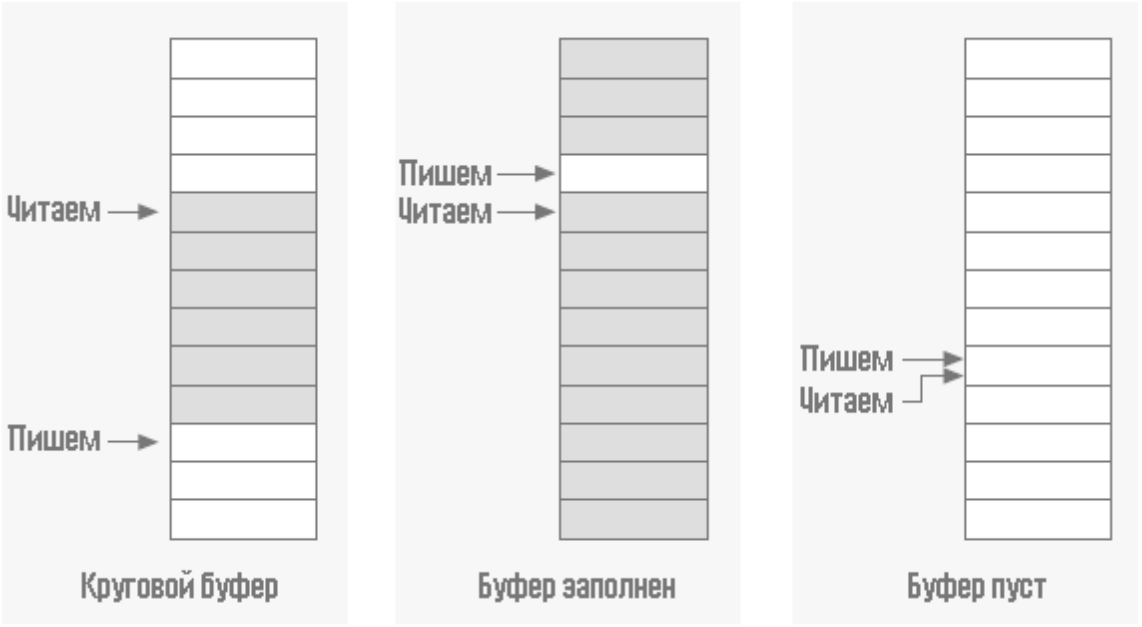


Рисунок 5-1. Круговой буфер

Круговые буферы появляются в драйверах устройств разумно часто. Сетевые адаптеры, в частности, часто используют круговые буферы для обмена данными (пакетами) с

процессором. Отметим, что в ядре версии 2.6.10 есть общая реализация кругового буфера, смотрите ***<linux/kfifo.h>*** для получения информации о том, как его использовать.

Атомарные переменные

Иногда общий ресурс представляет собой простую целую величину. Предположим, что драйвер поддерживает общую переменную **n_op**, которая сообщает, сколько операций устройства в настоящее время ожидает исполнения. Обычно, даже такие простые операции, как:

```
n_op++;
```

потребуют блокировки. Некоторые процессоры могут выполнять такого рода увеличение атомарным образом, но вы не можете на это рассчитывать. Но режим полной блокировки выглядит излишеством для простых целочисленных значений. В случаях, подобных этому, ядро обеспечивает атомарный целочисленный тип, названный **atomic_t** и определённый в ***<asm/atomic.h>***.

atomic_t на всех поддерживаемых архитектурах содержит значение **int**. Однако, из-за особенностей работы этого типа на некоторых процессорах, полный диапазон целого числа может быть не доступен; таким образом, вы не должны рассчитывать, что **atomic_t** содержит более чем 24 бита. Для этого типа определены следующие операции и они гарантированно будут атомарными в отношении всех процессоров SMP компьютера. Операции очень быстрые, потому что когда это возможно, они компилируются в одну машинную команду.

void atomic_set(atomic_t *v, int i);
atomic_t v = ATOMIC_INIT(0);
Задать атомарной переменной **v** целое значение **i**. Вы также можете инициализировать атомарные значения во время компиляции с помощью макроса **ATOMIC_INIT**.

int atomic_read(atomic_t *v);
Возвращает текущее значение **v**.

void atomic_add(int i, atomic_t *v);
Добавляет **i** к атомарной переменной, указываемой **v**. Возвращаемое значение является неопределённым, потому что существуют дополнительные расходы для возвращения нового значения и в большинстве случаев знать его нет необходимости.

void atomic_sub(int i, atomic_t *v);
Вычесть **i** из ***v**.

void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
Увеличение или уменьшение атомарной переменной.

int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
Выполнить указанные операции и проверить результат; если после операции атомарное значение равно 0, то возвращаемое значение является истинным (true); в противном случае, оно является ложным (false). Обратите внимание, что нет ***atomic_add_and_test***.

int atomic_add_negative(int i, atomic_t *v);

Добавить целую переменную **i** к ***v**. Возвращаемое значение является истиной (true), если результат отрицательный, ложным (false), в противном случае.

int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);

Ведут себя так же, как ***atomic_add*** и друзья, с тем исключением, что они возвращают новое значение атомарной переменной вызывающему.

Как отмечалось ранее, объекты данных **atomic_t** должны быть доступны только через эти функции. Если вы передаёте атомарный объект в функцию, которая ожидает целочисленный аргумент, вы получите ошибку компилятора.

Вы должны также иметь в виду, что значения **atomic_t** работают только когда их количество в запросе действительно атомарное. Операции, требующие несколько переменных **atomic_t**, всё ещё требуют некоторого другого вида блокировки. Рассмотрим следующий код:

```
atomic_sub(amount, &first_atomic);
atomic_add(amount, &second_atomic);
```

Существует определенный период времени, когда **amount** был вычтен из первой атомарной величины, но ещё не добавлен ко второй. Если такое состояние может создать проблемы для кода, который может выполняться между двумя операциями, должны быть использованы какие-либо формы блокировки.

Битовые операции

Тип **atomic_t** хорош для выполнения целой арифметики. Однако, он не работает так же хорошо при необходимости манипулировать атомарным образом отдельными битами. Взамен, для этого цели ядро содержит ряд функций, которые изменяют или проверяют один бит атомарно. Так как вся операция выполняется за один шаг, прерывания (или другой процессор) не смогут вмешаться.

Атомарные битовые операции очень быстрые, так как они выполняют операции с использованием одной машинной инструкции без запрета прерываний всегда, когда нижележащая платформа может это сделать. Функции зависят от архитектуры и объявлены в ***<asm/bitops.h>***. Они гарантированно атомарные даже на компьютерах с SMP и полезны для сохранения согласованности между процессорами.

К сожалению, типы данных в этих функциях зависят от архитектуры. Аргумент **nr** (описывающий биты для манипулирования) обычно определяется как **int**, но для нескольких архитектур является **unsigned long**. Адрес изменяемого, как правило, указатель на **unsigned long**, но на нескольких архитектурах вместо этого используется **void ***.

Доступными битовыми операциями являются:

void set_bit(nr, void *addr);

Установить бит номер **nr** в объекте данных, на который указывает **addr**.

void clear_bit(nr, void *addr);
Очищает указанный бит в данных типа **unsigned long**, которые находятся по **addr**. Иначе говоря, его семантика такая же, как у ***set_bit***.

void change_bit(nr, void *addr);
Переключает бит.

test_bit(nr, void *addr);
Эта функция является единственной битовой операцией, которой не требуется быть атомарной; она просто возвращает текущее значение бита.

int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
Поведение атомарно, как у перечисленных выше функций, за исключением того, что они также возвращают предыдущее значение бита.

Когда эти функции используются для доступа и изменения общего флага, вы не должны делать ничего, кроме их вызова; они выполняют свои операции атомарным образом. С другой стороны, использование битовых операций для управления переменной блокировки, которая контролирует доступ к общей переменной, это немного сложнее и заслуживает примера. Наиболее современный код не использует битовые операции таким способом, однако код, подобный следующему, ещё существует в ядре.

Сегмент кода, которому необходимо получить доступ к объекту с общими данными, пытается получить блокировку атомарно используя либо ***test_and_set_bit***, либо ***test_and_clear_bit***. Здесь показана обычная реализация; она предполагает, что блокировка находится в бите **nr** по адресу **addr**. Предполагается также, что бит равен 0, когда блокировка свободна или ненулевой, когда блокировка занята.

```
/* пробуем установить блокировку */
while (test_and_set_bit(nr, addr) != 0)
    wait_for_a_while( );

/* выполняем работу */

/* освобождаем блокировку и проверяем... */
if (test_and_clear_bit(nr, addr) == 0)
    something_went_wrong( ); /* уже свободна: ошибка */
```

Если вы посмотрите исходный код ядра, вы найдёте код, который работает подобно этому примеру. Однако, гораздо лучше в новом коде использовать спин-блокировки; спин-блокировки хорошо отлажены, они учитывают такие вопросы, как прерывания и вытеснение в ядре, а другим, читающим ваш код, не надо будет напрягаться, чтобы понять, что вы делаете.

Последовательные блокировки

Ядро версии 2.6 содержит несколько новых механизмов, которые призваны обеспечить быстрый, свободный от блокировок доступ к общему ресурсу. Последовательные блокировки работают в ситуациях, когда защищаемые ресурсы малы, просты и часто запрашиваемы, и

когда доступ для записи является редким, но должен быть быстрым. По существу, они работают разрешая читателям свободный доступ к ресурсу, но требуя этих читателей проверять наличие конфликта с писателями, и когда такой конфликт происходит, повторить их запрос. Последовательные блокировки, как правило, не могут быть использованы для защиты структур данных с участием указателей, потому что читатель может быть следующим указателем, который является недопустимым, пока писатель меняет структуру данных.

Последовательные блокировки (seqlock-и) определены в ***<linux/seqlock.h>***. Есть два обычных метода для инициализации такого примитива (который имеет тип **seqlock_t**):

```
seqlock_t lock1 = SEQLOCK_UNLOCKED;

seqlock_t lock2;
seqlock_init(&lock2);
```

Доступ на чтение работает получая (беззнаковое) целочисленное значение последовательности на входе в критическую секцию. На выходе это значение последовательности сравнивается с текущим значением; если есть несоответствие, чтение должно быть повторено. В результате, код читателя имеет вид вроде следующего:

```
unsigned int seq;

do {
    seq = read_seqbegin(&the_lock);
    /* делайте, что необходимо */
} while read_seqretry(&the_lock, seq);
```

Такая блокировка обычно используется для защиты каких-то простых вычислений, требующих много непротиворечивых величин. Если проверка в конце расчёта показывает, что произошла одновременная запись, результат может быть просто отброшен и перерасчитан.

Если ваш примитив последовательной блокировки должен быть доступен из обработчика прерываний, вы должны вместо этого использовать IRQ-безопасные версии:

```
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long
flags);
```

Писатели должны получить эксклюзивную блокировку, чтобы войти в критическую секцию, защищаемую примитивом последовательной блокировки. Чтобы это сделать, вызовите:

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

Блокировка записи реализована со спин-блокировкой, так что применяются все обычные ограничения.

Сделайте вызов:

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

для снятия блокировки. Так как для контроля записи используются спин-блокировки, имеются все обычные варианты:

```
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);

void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
```

Существует также ***write_tryseqlock***, которая возвращает ненулевое значение, если она смогла получить блокировку.

Прочитать-Скопировать-Обновить

Прочитать-Скопировать-Обновить (Read-copy-update, RCU) является расширенной схемой взаимного исключения, которая может принести высокую производительность в правильных условиях. Её использование в драйверов является редким, но не неизвестным, так что здесь стоит сделать краткий обзор. Те, кто заинтересовался полной информацией об алгоритме RCU, могут найти его в официальном документе, опубликованном его создателем (http://www.rdrop.com/users/paulmck/rclock/intro/rclock_intro.html).

RCU делает ряд ограничений на вид структуры данных, которую он может защищать. Она оптимизирована для ситуаций частого чтения и редкой записи. Защищаемые ресурсы должны быть доступны через указатели и все ссылки на эти ресурсы должны проводиться только атомарным кодом. Когда структура данных должна быть изменена, пишущий поток создаёт копию, изменяет копию, затем перенацеливает соответствующий указатель на новую версию, отсюда название этого алгоритма. Когда ядро уверено, что не осталось ссылок на старую версию, блокировка может быть освобождена.

В качестве примера реального использования RCU рассмотрим таблицы сетевой маршрутизации. Каждый исходящий пакет требует проверки таблиц маршрутизации для определения того, какой интерфейс должен быть использован. Проверка является быстрой и после того, как ядро нашло целевой интерфейс, оно больше не нуждается в таблице маршрутизации. RCU позволяет выполнять поиск маршрутов без блокировки, значительными увеличивая производительность. Starmode radio IP driver в ядре также использует RCU, чтобы отслеживать свой список устройств.

Код, использующий RCU, должен подключать ***<linux/rcupdate.h>***.

На читающей стороне код, использующий защищённые RCU структуры данных, должен окаймить свои ссылки вызовами ***rcu_read_lock*** и ***rcu_read_unlock***. В результате, RCU код, как правило, выглядит следующим образом:

```
struct my_stuff *stuff;

rcu_read_lock( );
stuff = find_the_stuff(args...);
do_something_with(stuff);
rcu_read_unlock( );
```

Вызов ***rcu_read_lock*** является быстрым, он отключает вытеснение в ядре, но ничего не ждёт. Код, который выполняется во время удержания "блокировки" чтения, должен быть атомарным. Нет ссылки на защищённый ресурс, которую можно было бы использовать после вызова ***rcu_read_unlock***. Коду, которому требуется изменить защищаемую структуру,

предстоит выполнить несколько шагов. Первая часть является простой; это создание новой структуры, копирование данные из старой, если необходимо, затем замена указателя, который виден коду чтения. На этом этапе с точки зрения читающей стороны изменение является полным; любой код, входящий в критическую секцию, видит новую версию данных.

Всё, что осталось - освободиться от старой версии. Проблемой, конечно, является то, что код, выполняющийся на других процессорах, может всё ещё иметь ссылку на старые данные, поэтому невозможно освободиться немедленно. Вместо этого код записи должен ждать, пока не узнает, что такая ссылка больше не существует. Поскольку весь код, удерживающий ссылки на эти структуры данных должен (по правилам) быть атомарным, мы знаем, что как только планировщик в системе выделит каждому процессору время по меньшей мере один раз, все ссылки должны уйти. Вот что делает RCU: он устанавливает обратный вызов, который ждёт, пока все процессоры получат время; затем этот обратный вызов выполняет работу по очистке.

Код, который изменяет защищённые RCU структуры данных, должен получить свой обратный вызов для очистки путём создания **struct rcu_head**, хотя инициализация этой структуры каким-либо способом не требуется. Часто эта структура просто внедрена в большой ресурс, который защищён RCU. После завершения изменения этого ресурса вызов должен выполнить:

```
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
```

Данная **func** вызывается, когда это безопасно для освобождения ресурсов; ей передаётся тот же самый **arg**, который был передан в ***call_rcu***. Как правило, **func** необходимо сделать только одно - вызвать ***kfree***.

Полный интерфейс RCU является более сложным, чем то, что мы здесь рассмотрели; он включает, например, вспомогательные функции для работы с защищёнными связными списками. Для полной картины смотрите соответствующие заголовочные файлы.

Краткая справка

Эта глава ввела большой набор символов для управления конкуренцией. Здесь приведены наиболее важные из них:

#include <asm/semaphore.h>
Подключает файл, который определяет семафоры и операции с ними.

DECLARE_MUTEX(name);
DECLARE_MUTEX_LOCKED(name);
Два макроса для объявления и инициализации семафора, используемого в режиме взаимного исключения.

void init_MUTEX(struct semaphore *sem);
void init_MUTEX_LOCKED(struct semaphore *sem);
Эти две функции могут быть использованы для инициализации семафора во время выполнения.

void down(struct semaphore *sem);
int down_interruptible(struct semaphore *sem);
int down_trylock(struct semaphore *sem);
void up(struct semaphore *sem);
Блокировка и разблокировка семафора. ***down*** в случае необходимости помещает вызывающий процесс в непрерываемый сон; ***down_interruptible***, наоборот, может быть

прервана сигналом. ***down_trylock*** не засыпает; вместо этого она сразу же возвращается, если семафор недоступен. Код, который блокирует семафор, в конечном итоге должен разблокировать его с помощью ***up***.

struct rw_semaphore;
init_rwsem(struct rw_semaphore *sem);

Версии чтения/записи семафоров и функция инициализации.

void down_read(struct rw_semaphore *sem);
int down_read_trylock(struct rw_semaphore *sem);
void up_read(struct rw_semaphore *sem);

Функции для получения и освобождения доступа на чтение для семафора чтения/записи .

void down_write(struct rw_semaphore *sem);
int down_write_trylock(struct rw_semaphore *sem);
void up_write(struct rw_semaphore *sem);
void downgrade_write(struct rw_semaphore *sem);

Функции для управления доступом на запись для семафора чтения/записи .

#include <linux/completion.h>
DECLARE_COMPLETION(name);
init_completion(struct completion *c);
INIT_COMPLETION(struct completion c);

Подключаемый файл с описанием механизма завершения Linux и обычные методы для инициализации завершений. **INIT_COMPLETION** должен использоваться только для переинициализации завершения, которое было использовано ранее.

void wait_for_completion(struct completion *c);

Ожидает события завершения, чтобы просигнализировать.

void complete(struct completion *c);
void complete_all(struct completion *c);

Сигнал события завершения. ***complete*** будит, самое большее, один ожидающий поток, в то время как ***complete_all*** будит всех ожидающий.

void complete_and_exit(struct completion *c, long retval);

Сигнализирует о событии завершения, делая вызов ***completion*** и вызов ***exit*** для текущего потока.

#include <linux/spinlock.h>
spinlock_t lock = SPIN_LOCK_UNLOCKED;
spin_lock_init(spinlock_t *lock);

Подключает файл, определяющий интерфейс спин-блокировки и два способа инициализации блокировок.

void spin_lock(spinlock_t *lock);
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);
void spin_lock_irq(spinlock_t *lock);
void spin_lock_bh(spinlock_t *lock);

Различные способы блокирования спин-блокировки и, возможно, запрета прерываний.

int spin_trylock(spinlock_t *lock);
int spin_trylock_bh(spinlock_t *lock);

Незацикливающиеся версии указанных выше функций; возвращают 0 в случае неудачи при получении блокировки, ненулевое иначе.

void spin_unlock(spinlock_t *lock);
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
void spin_unlock_irq(spinlock_t *lock);

```
void spin_unlock_bh(spinlock_t *lock);
```
Соответствующие способы освобождения спин-блокировок.
```
rwlock_t lock = RW_LOCK_UNLOCKED
rwlock_init(rwlock_t *lock);
```
Это два способа инициализации блокировок чтения/записи .
```
void read_lock(rwlock_t *lock);
void read_lock_irqsave(rwlock_t *lock, unsigned long flags);
void read_lock_irq(rwlock_t *lock);
void read_lock_bh(rwlock_t *lock);
```
Функции для получения доступа на чтение для блокировки чтения/записи.
```
void read_unlock(rwlock_t *lock);
void read_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void read_unlock_irq(rwlock_t *lock);
void read_unlock_bh(rwlock_t *lock);
```
Функции для освобождения доступа на чтения для спин-блокировки чтения/записи .
```
void write_lock(rwlock_t *lock);
void write_lock_irqsave(rwlock_t *lock, unsigned long flags);
void write_lock_irq(rwlock_t *lock);
void write_lock_bh(rwlock_t *lock);
```
Функции для получения доступа на запись для блокировки чтения/записи.
```
void write_unlock(rwlock_t *lock);
void write_unlock_irqrestore(rwlock_t *lock, unsigned long flags);
void write_unlock_irq(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```
Функции для освобождения доступа на запись для спин-блокировки чтения/записи .

```
#include <asm/atomic.h>
atomic_t v = ATOMIC_INIT(value);
void atomic_set(atomic_t *v, int i);
int atomic_read(atomic_t *v);
void atomic_add(int i, atomic_t *v);
void atomic_sub(int i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t *v);
int atomic_dec_return(atomic_t *v);
```
Целочисленные переменные с атомарным доступом. Переменные **atomic_t** должны быть доступны только через эти функции.

```
#include <asm/bitops.h>
void set_bit(nr, void *addr);
void clear_bit(nr, void *addr);
void change_bit(nr, void *addr);
test_bit(nr, void *addr);
```

int test_and_set_bit(nr, void *addr);
int test_and_clear_bit(nr, void *addr);
int test_and_change_bit(nr, void *addr);
Атомарный доступ к битовым величинам; они могут быть использованы для флагов или переменных блокировки. Использование этих функций позволяет избежать состояния гонок, связанного с одновременным доступом к битам.

#include <linux/seqlock.h>
seqlock_t lock = SEQLOCK_UNLOCKED;
seqlock_init(seqlock_t *lock);
Подключение файла, определяющего последовательные блокировки и два способа их инициализации.
unsigned int read_seqbegin(seqlock_t *lock);
unsigned int read_seqbegin_irqsave(seqlock_t *lock, unsigned long flags);
int read_seqretry(seqlock_t *lock, unsigned int seq);
int read_seqretry_irqrestore(seqlock_t *lock, unsigned int seq, unsigned long flags);
Функции для получения доступа на чтение к ресурсам, защищённым последовательной блокировкой.
void write_seqlock(seqlock_t *lock);
void write_seqlock_irqsave(seqlock_t *lock, unsigned long flags);
void write_seqlock_irq(seqlock_t *lock);
void write_seqlock_bh(seqlock_t *lock);
int write_tryseqlock(seqlock_t *lock);
Функции для получения доступа на запись к ресурсу, защищённому последовательной блокировкой.
void write_sequnlock(seqlock_t *lock);
void write_sequnlock_irqrestore(seqlock_t *lock, unsigned long flags);
void write_sequnlock_irq(seqlock_t *lock);
void write_sequnlock_bh(seqlock_t *lock);
Функции для освобождения доступа на запись к ресурсу, защищённому последовательной блокировкой.

#include <linux/rcupdate.h>
Подключает файл, необходимый для использования механизма чтения-копирования-обновления (RCU) .
void rcu_read_lock;
void rcu_read_unlock;
Макросы для получения атомарного доступа на чтение к ресурсу, защищённому RCU.
void call_rcu(struct rcu_head *head, void (*func)(void *arg), void *arg);
Организует обратный вызов для запуска безопасного освобождения защищённого RCU ресурса после того, как все процессоры получили от планировщика задач время для работы.

Глава 6, Расширенные операции символьного драйвера

В Главе 3 [39] мы построили завершённый драйвер устройства, пользователь может писать в него и читать из него. Но реальное устройство обычно предоставляет больше возможностей, чем синхронные ***read*** и ***write***. Теперь, когда мы оснащены средствами отладки на случай, если что-то пойдёт наперекосяк и твёрдым пониманием вопросов конкуренции, чтобы избегать проблем, мы можем смело идти вперед и создать более совершенный драйвер. В этой главе рассматриваются несколько концепций, которые вам необходимо понять, чтобы писать полнофункциональные драйверы символьных устройств. Мы начинаем с реализации системного вызова ***ioctl***, который является общим интерфейсом, используемым для управления устройством. Затем мы разберём различные способам синхронизации с пользовательским пространством; к концу этой главы вы будете иметь хорошую идею, как помещать процессы в сон (и будить их), реализовывать неблокирующей ввод/вывод, а также информировать пользовательское пространство, когда ваши устройства доступны для чтения или записи. Мы заканчиваем просмотром, как в рамках драйверов реализовать несколько различных политик доступа к устройству. Идеи, обсуждаемые здесь, демонстрируются несколькими модифицированными версиями драйвера ***scull***. Напомним ещё раз, всё осуществляется с использованием находящегося в памяти виртуального устройства, так что вы можете опробовать этот код самостоятельно без необходимости иметь какое-либо особенное аппаратное обеспечение. К настоящему времени вы, возможно, хотите запачкать ваши руки, работая с реальным оборудованием, но этого придётся подождать до Главы 9 [224].

ioctl

Большинству драйверов в дополнение к возможности чтения и записи устройства необходима возможность управления аппаратурой разными способами через драйвер устройства. Большинство устройств может выполнять операции за рамками простой передачи данных; пользовательское пространство часто должно иметь возможность запросить, например, блокировку устройством своих шторок, извлечение носителя информации, сообщить об ошибке информации, изменение скорости передачи, либо самоликвидацию. Эти операции обычно поддерживаются через метод ***ioctl*** (команда управления вводом-выводом), который реализует системный вызов с тем же названием.

В пользовательском пространстве системный вызов ***ioctl*** имеет следующий прототип:

```
int ioctl(int fd, unsigned long cmd, ...);
```

Прототип выделяется в списке системных вызовов Unix из-за точек, которые обычно отмечают функцию с переменным числом аргументов. Однако, в реальной системе системный вызов в действительности не может иметь переменное количество аргументов. Системные вызовы должны иметь чётко определённые прототипы, потому что пользовательские программы могут получать к ним доступ только через аппаратные "ворота". Таким образом, точки в прототипе представляют не переменное количество аргументов, а один дополнительный аргумент, традиционно определяемый как **char *argp**. Точки просто не допускают проверку типа во время компиляции. Фактический характер третьего аргумента зависит от используемой команды управления (второй аргумент). Некоторые команды не требуют никаких аргументов, некоторым требуется целые значения, а некоторые используют указатель на другие данные. Использование указателя является способом передачи в вызов ***ioctl*** произвольных данных; устройство затем сможет обменяться любым количеством данных с пользовательским пространством.

Неструктурированный характер вызова ***ioctl*** вызвал его падение в немилость среди разработчиков ядра. Каждая команда ***ioctl*** является, по сути, отдельным, обычно недокументированным системным вызовом, и нет никакой возможности для проверки этих вызовов каким-либо всеобъемлющим образом. Кроме того, трудно сделать, чтобы неструктурированные аргументы ***ioctl*** работали одинаково на всех системах; учитывая, например, 64-х разрядные системы с пользовательским процессом, запущенном в 32-х разрядном режиме. В результате, существует сильное давление, чтобы осуществлять разные операции контроля только какими-либо другими способами. Возможные альтернативы включают вложения команд в поток данных (мы обсудим этот подход далее в этой главе) или использование виртуальных файловых систем, или **sysfs**, или специфичные драйверные файловые системы. (Мы будем рассматривать **sysfs** в Главе 14 347.) Тем не менее, остается фактом, что ***ioctl*** часто является самым простым и наиболее прямым выбором для относящихся к устройству операций. Метод драйвера ***ioctl*** имеет прототип, который несколько отличается от версии пользовательского пространства:

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

Указатели **inode** и **filp** являются значениями, соответствующими файловому дескриптору **fd**, передаваемого приложением, и являются теми же параметрами, передаваемыми в метод ***open***. Аргумент **cmd** передаётся от пользователя без изменения, а необязательный аргумент **arg** передаётся в виде **unsigned long**, независимо от того, был ли он задан пользователем как целое или указатель. Если вызывающая программа не передаёт третий аргумент, значение **arg**, полученное операцией драйвера, не определено. Из-за отключённой проверки типа дополнительного аргумента компилятор не сможет предупредить вас, если в ***ioctl*** передаётся неправильный аргумент, и любая связанная с этим ошибка будет труднообнаруживаемой.

Как можно было себе представить, большинство реализаций ***ioctl*** содержат большой переключатель, который выбирает правильное поведение в зависимости от аргумента **cmd**. Разные команды имеют различные числовые значения, которые для упрощения кодирования обычно задаются символическими именами. Символическое имя присваивается через определение препроцессора. Заказные драйверы обычно объявляют такие символы в своих заголовочных файлах; для ***scull*** их объявляет ***scull.h***. Чтобы иметь доступ к этим символам, пользовательские программы должны, конечно, подключить этот заголовочный файл.

Выбор команд ioctl

Прежде чем писать код для ***ioctl***, необходимо выбрать цифры, которые соответствуют командам. Инстинктивно многие программисты выбирают небольшие числа, начиная с 0 или 1 и далее увеличивая значения. Есть, однако, веские причины не делать это таким образом. Номера команд ***ioctl*** должны быть уникальными в системе в целях предотвращения ошибок, вызванных правильной командой для неправильного устройства. Такое несоответствие не является маловероятным и программа могла бы обмануть сама себя, пытаясь изменить скорость передачи данных входного потока, не связанного с последовательным портом, такого, как FIFO или аудио устройства. Если каждый номер ***ioctl*** является уникальным, программа получает ошибку **EINVAL** раньше, чем успевает сделать что-то непреднамеренное.

Чтобы помочь программистам создавать уникальные коды команд ***ioctl***, эти коды были разделены на несколько битовых полей. Первые версии Linux использовали 16-ти разрядные числа: верхние восемь "магических" чисел связывались с устройством, а нижние восемь были последовательными номерами, уникальными для данного устройства. Это произошло потому, что Линус был "невежественен" (его собственные слова); лучшее разделение битовых полей было придумано лишь позднее. К сожалению, довольно много драйверов всё ещё используют старое соглашение. Они вынуждены: изменение кодов команд нарушило бы работу многих бинарных программ и это не то, что разработчики ядра готовы сделать.

Для выбора номеров ***ioctl*** для вашего драйвера в соответствии с соглашением ядра Linux вы должны сначала проверить ***include/asm/ioctl.h*** и ***Documentation/ioctl-number.txt***. Этот заголовок определяет битовые поля для использования: тип (системный номер), порядковый номер, направление передачи и размер аргумента. Файл ***ioctl-number.txt*** перечисляет системные номера, используемые в ядре (* Однако, поддержка этого файла была несколько ограниченной в последнее время.), поэтому вы сможете выбрать свой собственный номер в системе и избежать дублирования. В этом текстовом файле также перечислены причины, по которым должно быть использовано данное соглашение.

Утверждённый способ определения номеров команд ***ioctl*** использует четыре битовых области, которые имеют следующие значения. Новые символы, введённые в этом списке, определяются в ***<linux/ioctl.h>***.

type
Системный номер. Просто выберите одно число (после консультации с ***ioctl-number.txt***) и используйте его в драйвере. Это поле шириной восемь бит (**_IOC_TYPEBITS**).

number
Порядковый (последовательный) номер. Это поле шириной восемь бит (**_IOC_NRBITS**).

direction
Направление передачи данных, если данная команда предполагает передачу данных. Возможными значениями являются **_IOC_NONE** (без передачи данных), **_IOC_READ**, **_IOC_WRITE** и **_IOC_READ | _IOC_WRITE** (данные передаются в обоих направлениях). Передача данных с точки зрения приложения; **_IOC_READ** означает чтение из устройства, так что драйвер должен писать в пространство пользователя. Обратите внимание, что поле является битовой маской, **_IOC_READ** и **_IOC_WRITE** могут быть извлечены при помощи логической операции.

size
Размер предполагаемых пользовательских данных. Ширина этого поля зависит от архитектуры, но, как правило, 13 или 14 бит. Вы можете найти это значение для вашей

конкретной архитектуры в макросе **_IOC_SIZEBITS**. Не обязательно использовать поле **size** - ядро не проверяет его - но это хорошая идея. Правильное использование этого поля может помочь обнаружить ошибки программирования в пользовательском пространстве и позволит реализовать обратную совместимость, если вам когда-нибудь понадобится изменить размер соответствующего элемента данных. Однако, если вам требуются более крупные структуры данных, вы можете просто проигнорировать поле **size**. Как используется это поле, мы рассмотрим в ближайшее время.

Заголовочный файл ***<asm/ioctl.h>***, который подключается ***<linux/ioctl.h>***, определяет макрос, который поможет задать номера команд следующим образом: **_IO(type,nr)** (для команды, которая не имеет аргумента), **_IOR(type,nr,datatype)** (для чтения данных из драйвера), **_IOW(type,nr,datatype)** (для записи данных) и **_IOWR(type,nr,datatype)** (для двунаправленной передачи). Поля **type** и **fields**, передаваемые в качестве аргументов, и поле **size** получаются применением **sizeof** к аргументу **datatype**.

Заголовок определяет также макросы, которые могут быть использованы в вашем драйвере для декодирования номеров: **_IOC_DIR(nr)**, **_IOC_TYPE(nr)**, **_IOC_TYPE(nr)**, **_IOC_NR(nr)** и **_IOC_SIZE(nr)**. Мы не будем углубляться более подробно в эти макросы, так как заголовочный файл ясен, а ниже в этом разделе показан код примера.

Вот как в ***scull*** определяются некоторые команды ***ioctl***. В частности, эти команды устанавливают и получают настраиваемые параметры драйвера.

```
/* Используем 'k' как системный номер */
#define SCULL_IOC_MAGIC 'k'
/* Пожалуйста, используйте в вашем коде другое 8-ми битовое число */

#define SCULL_IOCRESET    _IO(SCULL_IOC_MAGIC, 0)

/*
* S означает "Set" ("Установить") через ptr,
* T означает "Tell" ("Сообщить") прямо с помощью значения аргумента
* G означает "Get" ("Получить"): ответ устанавливается через указатель
* Q означает "Query" ("Запрос"): ответом является возвращаемое значение
* X означает "eXchange" ("Обменять"): переключать G и S автоматически
* H означает "sHift" ("Переключить"): переключать T и Q автоматически
*/
#define SCULL_IOCSQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
#define SCULL_IOCSQSET    _IOW(SCULL_IOC_MAGIC, 2, int)
#define SCULL_IOCTQUANTUM _IO(SCULL_IOC_MAGIC, 3)
#define SCULL_IOCTQSET    _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOCGQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOCGQSET    _IOR(SCULL_IOC_MAGIC, 6, int)
#define SCULL_IOCQQUANTUM _IO(SCULL_IOC_MAGIC, 7)
#define SCULL_IOCQQSET    _IO(SCULL_IOC_MAGIC, 8)
#define SCULL_IOCXQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOCXQSET    _IOWR(SCULL_IOC_MAGIC,10, int)
#define SCULL_IOCHQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOCHQSET    _IO(SCULL_IOC_MAGIC, 12)

#define SCULL_IOC_MAXNR 14
```

Реальный файл исходника определяет несколько дополнительных команд, которые не были

здесь показаны.

Мы решили реализовать оба способа получения целочисленных аргументов: по указателю и явным значением (хотя по принятому соглашению ***ioctl*** должна обмениваться значениями по указателю). Аналогичным образом, оба пути используются для возвращения целого числа: по указателю или устанавливая возвращаемое значение. Это работает, пока возвращаемое значение является положительным целым числом; как вы уже знаете, положительное значение сохраняется при возвращении из любого системного вызова (как мы видели для ***read*** и ***write***), а отрицательное значение считается ошибкой и используется для установки **errno** в пользовательском пространстве. (* На самом деле, все реализации libc, использующиеся в настоящее время (включая uClibc), рассматривают как коды ошибок только значения диапазона от -4095 до -1. К сожалению, возможность возвращения больших отрицательных чисел, а не маленьких, не очень полезна.)

Операции "обменять" и "переключить" не являются особенно полезными для ***scull***. Мы реализовали "обмен", чтобы показать, как драйвер может объединить отдельные операции в одну атомарную, и "переключить" для пары к "сообщить" и "запрос". Есть случаи, когда необходимы атомарные операции проверить-и-установить, как эти, в частности, когда приложениям необходимо установить или освободить блокировки.

Точный порядковый номер команды не имеет определённого значения. Он используется только, чтобы отличить команды друг от друга. На самом деле, вы можете даже использовать тот же порядковый номер для команды чтения и записи, поскольку фактический номер ***ioctl*** отличается в битах "направления", но нет никакой причины, почему бы вы не захотели сделать это. Мы решили не использовать порядковые номера команд нигде, кроме декларации, поэтому мы не назначили им символические значения. Вот почему в данном ранее определении появляются явные номера. Приведённый пример показывает один из способов использования номеров команд, но вы вольны делать это по-другому.

За исключением небольшого числа предопределённых команд (будет обсуждено в ближайшее время), значение аргумента **cmd** в ***ioctl*** в настоящее время не используется ядром и весьма маловероятно, что это будет в будущем. Таким образом, можно, если вы почувствовали себя ленивым, избегать сложных показанных ранее деклараций и явно декларировать набор скалярных чисел. С другой стороны, если бы вы сделали это, вы бы вряд ли выиграли от использования битовых полей и вы бы столкнулись с трудностями, если бы когда-то представили свой код для включения в основное ядро. Заголовок ***<linux/kd.h>*** является примером этого старомодного подхода, использующего для определения команд ***ioctl*** 16-ти разрядные скалярные значения. Этот исходный файл полагался на скалярные значения, потому что он использовал конвенцию того времени, а не из-за лени. Изменение его сейчас вызвало бы неуместную несовместимость.

Возвращаемое значение

Реализация ***ioctl***, как правило, использует команду **switch**, основанную на номере команды. Но что должно быть в варианте **default**, когда номер команды не совпадает с допустимыми операциями? Вопрос спорный. Некоторые функции ядра возвращают **-EINVAL** (“Invalid argument”, "Недопустимый аргумент"), что имеет смысл, поскольку этот командный аргумент действительно не является допустимым. Стандарт POSIX, однако, утверждает, что если была выдана неуместная команда ***ioctl***, должно быть возвращено **-ENOTTY**. Данный код ошибки интерпретируется библиотекой Си как "несоответствующая команда ***ioctl*** для устройства", которая является обычно именно тем, что должен услышать программист. Хотя по-прежнему в

ответ на недействительную команду ***ioctl*** очень распространено возвращение **-EINVAL**.

Предопределённые команды

Хотя системный вызов ***ioctl*** наиболее часто используется для воздействия на устройства, несколько команд распознаются ядром. Обратите внимание, что эти команды при применении к вашему устройству декодируются ***до того***, как вызываются ваши собственные файловые операции. Таким образом, если вы выбираете тот же номер для одной из ваших команд ***ioctl***, вы никогда не увидите запрос для этой команды, а также приложение получит нечто неожиданное из-за конфликта между номерами ***ioctl***.

Предопределённые команды разделены на три группы:

- Те, которые могут быть выполнены на любом файле (обычный, устройство, FIFO или сокет);
- Выполняемые только на обычных файлах;
- Зависящие от типа файловой системы;

Команды последней группы выполняются реализацией главной файловой системы (так, например, работает команда ***chattr***). Авторам драйверов устройств интересна только первая группа команд, чьим системным номером является "T". Рассмотрение работы других групп остаётся читателю в качестве упражнения; ***ext2_ioctl*** является наиболее интересной функцией (и более лёгкой для понимания, чем можно было ожидать), поскольку она реализует флаг "append-only" ("только добавление") и флаг "immutable" ("неизменяемый").

Следующие команды ***ioctl*** предопределены для любого файла, включая специальные файлы устройств:

FIOCLEX
: Установить флаг "закрыть-при-выходе" (close-on-exec, File IOctl CLose on EXec). Установка этого флага вызывает закрытие дескриптора файла, когда вызывающий процесс выполняет новую программу.

FIONCLEX
: Очищает флаг "закрыть-при-выходе" (close-on-exec, File IOctl Not CLose on EXec). Команда восстанавливает общее поведение файла, отменяя то, что делает вышеприведённая **FIOCLEX**.

FIOASYNC
: Установить или сбросить асинхронные уведомления для файла (описывается далее в этой главе в разделе "Асинхронное сообщение" 160). Заметим, что ядро Linux до версии 2.2.4 неправильно использовало эту команду для модификации флага **O_SYNC**. Поскольку оба действия можно осуществить с помощью ***fcntl***, никто на самом деле не использует команду **FIOASYNC**, о которой сообщается здесь только для полноты.

FIOQSIZE
: Эта команда возвращает размер файла или каталога; однако, при применении к файлу устройства она возвращает ошибку **ENOTTY**.

FIONBIO
: "Файловая IOctl НеБлокирующего Ввода/Вывода" (“File IOctl Non-Blocking I/O”) (описана в

разделе "Блокирующие и неблокирующие операции" 143). Этот вызов изменяет флаг **O_NONBLOCK** в **filp->f_flags**. Третий аргумент системного вызова используется для обозначения, должен ли флаг быть установлен или очищен. (Мы будем рассматривать роль этого флага далее в этой главе.) Обратите внимание, что обычным способом изменить этот флаг является системный вызов ***fcntl***, использующий команду **F_SETFL**.

Последним пунктом в списке представлен новый системный вызов, ***fcntl***, который выглядит как ***ioctl***. Фактически, вызов ***fcntl*** очень похож на ***ioctl*** тем, что он получает аргумент команды и дополнительный (необязательный) аргумент. Он сохраняется отдельным от ***ioctl*** в основном по историческим причинам: когда разработчики Unix столкнулись с проблемой контроля операций ввода/вывода, они решили, чтобы файлы и устройства отличались. В то время единственными устройствами с реализаций ***ioctl*** были телетайпы, что объясняет, почему **-ENOTTY** является стандартным ответом на неправильную команду ***ioctl***. Всё изменилось, но ***fcntl*** остаётся отдельным системным вызовом.

Использование аргумента ioctl

Перед просмотром кода ***ioctl*** для драйвера ***scull*** необходимо сначала разобраться, как использовать дополнительный аргумент. Если это целое число, это несложно: его можно использовать напрямую. Однако, если он является указателем, об этом требуется позаботиться.

Когда указатель используется как ссылка в пространстве пользователя, мы должны гарантировать, что пользовательский адрес является действительным. Попытка доступа с непроверенным заданным пользователем указателем может привести к неправильному поведению, сообщению ядра Oops, повреждению системы, или проблемам с безопасностью. Обеспечение надлежащей проверки каждого используемого адреса пользовательского пространства и возвращение ошибки, если он является недействительным, является ответственностью драйвера.

В Главе 3 39 мы рассмотрели функции ***copy_from_user*** и ***copy_to_user***, которые могут быть использованы для безопасного перемещения данных в и из пространства пользователя. Эти функции могут быть использованы так же и в методах ***ioctl***, но вызовы ***ioctl*** часто связаны с небольшими объектами данных, которыми можно более эффективно манипулировать другими способами. Сначала проверка адреса (без передачи данных) осуществляется с помощью функции ***access_ok***, которая объявлена в ***<asm/uaccess.h>***:

```
int access_ok(int type, const void *addr, unsigned long size);
```

Первый аргумент должен быть либо **VERIFY_READ** или **VERIFY_WRITE**, в зависимости от того, какое действие будет выполняться: чтение или запись памяти пользовательского пространства. Аргумент **addr** содержит адрес в пользовательском пространстве, а **size** является счётчиком байтов. Если, например, ***ioctl*** надо прочесть целое число из пользовательского пространства, **size** является **sizeof(int)**. Если вам необходимы по данному адресу и чтение и запись, используйте **VERIFY_WRITE**, поскольку это расширенный вариант **VERIFY_READ**.

В отличие от большинства функций ядра, ***access_ok*** возвращает булево значение: 1 в случае успеха (доступ ОК) и 0 для ошибки (доступ не ОК). Если она возвращает ложь, то обычно драйвер должен вернуть вызывающему **-EFAULT**.

Необходимо обратить внимание на несколько интересных вещей относительно ***access_ok***. Во-первых, она не делает полную работу проверки доступа к памяти; она лишь проверяет, что эта память по ссылке находится в области памяти, к которой этот процесс мог бы разумно иметь доступ. В частности, ***access_ok*** гарантирует, что адрес не указывает на память области ядра. Во-вторых, большинству кода драйвера фактически нет необходимости вызывать ***access_ok***. Описанные ниже процедуры доступа к памяти позаботятся об этом за вас. Тем не менее, мы демонстрируем её использование, чтобы вы могли увидеть, как это делается.

Исходник ***scull*** проверяет битовые поля в номере ***ioctl***, чтобы проверить аргументы перед командой **switch**:

```
int err = 0, tmp;
int retval = 0;

/*
 * проверить тип и номер битовых полей и не декодировать
 * неверные команды: вернуть ENOTTY (неверный ioctl) перед access_ok( )
 */
if (_IOC_TYPE(cmd) != SCULL_IOC_MAGIC) return -ENOTTY;
if (_IOC_NR(cmd) > SCULL_IOC_MAXNR) return -ENOTTY;

/*
 * направление является битовой маской и VERIFY_WRITE отлавливает передачи R/
W
 * `направление' является ориентированным на пользователя, в то время как
 * access_ok является ориентированным на ядро, так что концепции "чтение" и
 * "запись" являются обратными
 */
if (_IOC_DIR(cmd) & _IOC_READ)
    err = !access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
else if (_IOC_DIR(cmd) & _IOC_WRITE)
    err = !access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
if (err) return -EFAULT;
```

После вызова ***access_ok*** драйвер может безопасно выполнять фактическую передачу. В дополнение к функциям ***copy_from_user*** и ***copy_to_user*** программист может использовать набор функций, которые оптимизированы для наиболее часто используемых размеров данных (один, два, четыре и восемь байт). Эти функции описаны в следующем списке и определены в ***<asm/uaccess.h>***:

put_user(datum, ptr)
__put_user(datum, ptr)
Эти макросы пишут данные в пользовательское пространство; они сравнительно быстрые и следует вызывать их вместо ***copy_to_user***, когда передаются одинарные значения. Макросы были написаны для передачи в ***put_user*** любого типа указателя, пока он является адресом пользовательского пространства. Размер передаваемых данных зависит от типа аргумента **ptr** и определяется во время компиляции с помощью директив компилятора **sizeof** и **typeof**. В результате, если **ptr** является указателем на **char**, передаётся один байт, и так далее для двух, четырёх и, возможно, восьми байт. ***put_user*** проверяет, чтобы убедиться, что этот процесс может писать по данному адресу в памяти. Он возвращает 0 в случае успеха, и **-EFAULT** при ошибке. ***__put_user*** выполняет меньше проверок (он не вызывает ***access_ok***), но всё ещё может не сработать, если указываемая память не доступна пользователю для записи. Таким образом, ***__put_user***

следует использовать только если область памяти уже была проверена ***access_ok***. Как правило, вы вызываете ***__put_user*** для выигрыша нескольких циклов, когда вы реализуете метод ***read***, или при копировании нескольких объектов и, таким образом, вызываете ***access_ok*** только один раз перед первой передачей данных, как показано выше для ***ioctl***.

get_user(local, ptr)
__get_user(local, ptr)
Эти макросы используются для получения одинарных данных из пространства пользователя. Они ведут себя как ***put_user*** и ***__put_user***, но передают данные в обратном направлении. Полученные значения хранятся в локальной переменной **local**; возвращаемое значение показывает, является ли операция успешной. Здесь так же следует использовать ***__get_user*** только если адрес уже был проверен ***access_ok***.

Если делается попытка использовать одну из перечисленных функций для передачи значения, которое не совпадает с заданными величинами, результатом является обычно странное сообщение от компилятора, такое, как “conversion to non-scalar type requested” ("запрошено обращение к не скалярному типу"). В таких случаях должны быть использованы ***copy_to_user*** или ***copy_from_user***.

Разрешения и запрещённые операции

Доступ к устройству управляется разрешениями на файл(ы) устройства и драйвер обычно не участвует в проверке разрешений. Однако, есть ситуации, когда любой пользователь получает права чтения/записи на устройство, но некоторые операции управления всё ещё должны быть запрещены. Например, не все пользователи ленточного накопителя должны иметь возможность установить размер блока по умолчанию и пользователю, которому был предоставлен доступ на чтение/запись дискового устройства, должно, вероятно, быть отказано в возможности его отформатировать. В подобных случаях драйвер должен выполнять дополнительные проверки, чтобы убедиться, что пользователь имеет право выполнять запрошенную операцию.

В Unix системах привилегированные операции были традиционно ограничены учётной записью суперпользователя. Это означало, что привилегия была вещью "всё или ничего" - суперпользователь может делать абсолютно всё, но все остальные пользователи сильно ограничены. Ядро Linux предоставляет более гибкую систему, названную ***capabilities (разрешения, мандаты)***. Система, базирующаяся на разрешениях, оставляет режим "всё или ничего" позади и разделяет привилегированные операции на отдельные подгруппы. Таким образом, каждый пользователь (или программа) могут быть уполномочены выполнять особые привилегированные операции, не предоставляя возможности выполнять другие, не связанные операции. Ядро использует ***разрешения*** исключительно для управления правами доступа и экспортирует два системных вызова, называемых ***capget*** и ***capset***, чтобы предоставить возможность управлять ею из пространства пользователя.

Полный набор вариантов разрешений можно найти в ***<linux/capability.h>***. Они являются единственными вариантами, известными системе; для авторов драйверов или системных администраторов нет возможности определить новые без изменения исходного кода ядра. Часть этих средств, которые могли бы представлять интерес для авторов драйверов, включает в себя следующее:

CAP_DAC_OVERRIDE
Возможность отменить ограничения доступа (контроль доступа к данным, data access

control или DAC) к файлам и каталогам.

CAP_NET_ADMIN
Возможность выполнять задачи администрирования сети, в том числе те, которые затрагивают сетевые интерфейсы.

CAP_SYS_MODULE
Возможность загрузки или удаления модулей ядра.

CAP_SYS_RAWIO
Возможность выполнять "сырые" операции ввода/вывода. Примеры включают доступ к портам устройств или прямое общение с устройствами USB.

CAP_SYS_ADMIN
Всеобъемлющее разрешение, которое обеспечивает доступ ко многим операциям по администрированию системы.

CAP_SYS_TTY_CONFIG
Возможность выполнять задачи конфигурации tty.

Перед выполнением привилегированных операций драйвер устройства должен проверить, имеет ли вызывающий процесс соответствующие разрешения; невыполнение этого может привести к выполнению пользовательским процессом несанкционированных операций с плохими последствиями для стабильности системы или безопасности. Проверки разрешений осуществляются функцией ***capable*** (определённой в ***<linux/sched.h>***):

```
int capable(int capability);
```

В примере драйвера ***scull*** любой пользователь имеет право запрашивать квант и размер квантового набора. Однако, только привилегированные пользователи могут изменять эти значения, так как неподходящие значения могут плохо повлиять на производительность системы. При необходимости, реализация ***ioctl*** в ***scull*** проверяет уровень привилегий пользователя следующим образом:

```
if (! capable (CAP_SYS_ADMIN))
    return -EPERM;
```

В отсутствие более характерного разрешения для этой задачи, для этой проверки был выбран вариант **CAP_SYS_ADMIN**.

Реализация команд ioctl

Реализация ***ioctl*** в ***scull*** только передаёт настраиваемые параметры устройства и оказывается простой:

```
switch(cmd) {
    case SCULL_IOCRESET:
    scull_quantum = SCULL_QUANTUM;
    scull_qset = SCULL_QSET;
    break;

case SCULL_IOCSQUANTUM: /* Установить: arg указывает на значение */
```

```
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_quantum, (int __user *)arg);
    break;

case SCULL_IOCTQUANTUM: /* Сообщить: arg является значением */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    scull_quantum = arg;
    break;

case SCULL_IOCGQUANTUM: /* Получить: arg является указателем на результат */
    retval = __put_user(scull_quantum, (int __user *)arg);
    break;

case SCULL_IOCQQUANTUM: /* Запрос: возвращает его (оно положительно) */
    return scull_quantum;

case SCULL_IOCXQUANTUM: /* Обменять: использует arg как указатель */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    retval = __get_user(scull_quantum, (int __user *)arg);
    if (retval == 0)
        retval = __put_user(tmp, (int __user *)arg);
    break;

case SCULL_IOCHQUANTUM: /* Переключить: как Tell + Query */
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    tmp = scull_quantum;
    scull_quantum = arg;
    return tmp;

default: /* избыточно, так как cmd была проверена на MAXNR */
    return -ENOTTY;
}
return retval;
```

scull также включает в себя шесть элементов, которые действуют на ***scull_qset***. Они идентичны таким же в ***scull_quantum*** и не стоят показа в распечатке.

Шесть способов передать и получить аргументы выглядят примерно следующим образом с точки зрения вызывающего (то есть из пространства пользователя):

```
int quantum;

ioctl(fd,SCULL_IOCSQUANTUM, &quantum); /* Установить по указателю */
ioctl(fd,SCULL_IOCTQUANTUM, quantum);  /* Установить по значению */

ioctl(fd,SCULL_IOCGQUANTUM, &quantum); /* Получить по указателю */
quantum = ioctl(fd,SCULL_IOCQQUANTUM); /* Получить как возвращаемое значение
*/

ioctl(fd,SCULL_IOCXQUANTUM, &quantum); /* Обменять по указателю */
```

```
quantum = ioctl(fd,SCULL_IOCHQUANTUM, quantum); /* Обменять по значению */
```

Конечно, нормальный драйвер не будет реализовывать такое сочетание режимов вызова. Мы сделали это здесь только чтобы продемонстрировать различные способы, как это можно делать. Однако, обычно обмен данными будет выполняться последовательно либо с помощью указателей, или по значению, и смешения этих двух методов можно было бы избежать.

Управление устройством без ioctl

Иногда управление устройством лучше реализовать записью в него управляющих последовательностей. Например, эта техника используется в консольном драйвере, где так называемые управляющие последовательности используются для перемещения курсора, изменения цвета по умолчанию, или выполняют другие задачи настройки. Пользой от реализации управления устройством таким образом является то, что пользователь может управлять устройством просто записывая данные, без необходимости использовать (или иногда писать) программы, предназначенные только для настройки устройства. Когда устройства могут управляться таким образом, программе, выдающей команды, часто даже не требуется работать на той же системе, где находится контролируемое устройство.

Например, программа ***setterm*** воздействует на настройку консоли (или другого терминала) печатая управляющие последовательности. Управляющая программа может жить на другом компьютере, потому что работу по конфигурации выполняет простое перенаправление потока данных. Это то, что происходит каждый раз, когда вы запускаете удалённый сеанс терминала: управляющие последовательности печатаются удаленно, но воздействуют на местный терминал; техника, однако, не ограничивается терминалами.

Недостатком контроля печатанием является то, что это добавляет устройству ограничения политик; например, это жизнеспособно только если вы уверены, что управляющая последовательность не может появиться в данных, записываемых в устройство во время нормальной работы. Это только отчасти верно для терминала. Хотя текстовый дисплей предназначен для отображения только ASCII символов, иногда управляющие символы могут проскользнуть в записываемые данные и могут, следовательно, повлиять на установки консоли. Это может произойти, например, когда вы выполняете команду ***cat*** для бинарного файла на экран; беспорядок в результате может не содержать ничего и в итоге вы часто имеете на вашей консоли неправильный шрифт.

Управление записью ***является***, определённо, способом, подходящим для тех устройств, которые не передают данные, а только реагируют на команды, таких, как автоматизированные устройства.

Например, драйвер, написанный для удовольствия одним из ваших авторов, передвигает камеру по двум осям. В этом драйвере "устройством" является просто пара старых шаговых двигателей, в которые в действительности нельзя писать или читать из них. Понятие "передача потока данных" в шаговый двигатель имеет мало или вообще не имеет смысла. В этом случае драйвер интерпретирует то, что было записано в виде ASCII команд и преобразует запросы в последовательности импульсов, которые управляют шаговыми двигателями. Отчасти, идея очень похожа на AT команды, которые вы посылаете в модем для установки связи, с главным отличием: последовательный порт, используемый для взаимодействия с модемом, должен так же передавать реальные данные. Преимущество прямого управления устройством в том, что можно использовать ***cat*** для перемещения камеры без написания и компиляции специального кода для выдачи вызовов ***ioctl***.

При написании командно-ориентированного драйвера нет никаких причин для реализации метода ***ioctl***. Дополнительные команды проще реализовать и использовать в интерпретаторе.

Иногда, впрочем, можно выбрать действовать наоборот: вместо включения интерпретатора в метод ***write*** и избегания ***ioctl***, можно выбрать исключение ***write*** вообще и использовать исключительно команды ***ioctl*** и сопроводить драйвер специальной утилитой командной строки для отправки этих команд драйверу. Этот подход перемещает сложности из пространства ядра в пространство пользователя, где с ними может быть легче иметь дело, и помогает сохранить драйвер небольшим, запрещая использование простых команды ***cat*** или ***echo***.

Блокирующий Ввод/Вывод

В Главе 3 39 мы рассмотрели, как осуществлять методы драйвера ***read*** и ***write***. В тот момент, однако, мы пропустили один важный вопрос: как должен отреагировать драйвер, если он не может сразу удовлетворить запрос? Вызов ***read*** может прийти, когда данные не доступны, но ожидаемы в будущем. Или процесс может попытаться вызвать ***write***, но устройство не готово принять данные, потому что ваш выходной буфер полон. Вызывающий процесс обычно не заботится о таких вопросах; программист просто ожидает вызвать ***read*** или ***write*** и получить возвращение вызова после выполнения необходимой работы. Таким образом, в подобных случаях драйвер должен (по умолчанию) ***блокировать*** процесс, помещая его в режим сна, на время выполнения запроса.

Этот раздел показывает, как поместить процесс в сон и позднее снова его разбудить. Как обычно, однако, мы сначала должны объяснить несколько концепций.

Знакомство с засыпанием

Что означает для процесса "спать"? Когда процесс погружается в сон, он помечается, как находящийся в особом состоянии и удаляется из очереди выполнения планировщика. До тех пор, пока не произойдёт что-то, чтобы изменить такое состояние, этот процесс не будет запланирован на любом процессоре и, следовательно, не будет работать. Спящий процесс убирается прочь из системы, ожидая каких-то будущих событий.

Драйверу устройства в Linux отправить процесс в сон легко. Есть, тем не менее, несколько правил, которые необходимо иметь в виду, чтобы код имел возможность заснуть безопасным образом.

Первое из этих правил: никогда не засыпать, когда вы работаете в атомарном контексте. Мы познакомились с атомарной операцией в Главе 5 101; атомарный контекст является просто состоянием, когда должны быть выполнены несколько шагов без какого-либо вида одновременного доступа. Это означает по отношению ко сну, что ваш драйвер не может спать удерживая спин-блокировку, последовательную блокировку или RCU блокировку. Также нельзя засыпать, если запрещены прерывания. Вполне допустимо заснуть, удерживая семафор, но вы должны очень внимательно смотреть на любой код, который это делает. Если код засыпает, держа семафор, любой другой поток, ожидающий этот семафор, также спит. Таким образом, любое засыпание при удержании семафоров должно быть коротким и вы должны убедить себя, что удерживая семафор вы не блокируете процесс, который в конечном итоге будет вас будить.

При засыпании следует помнить также ещё одну вещь: когда вы просыпаетесь, вы никогда не знаете, сколько времени ваш процесс мог не выполняться процессором или что могло измениться за это время. Вы также обычно не знаете, есть ли другой процесс, который заснул,

ожидая того же события; этот процесс может проснуться до вас и захватить какой-нибудь ресурс, который вы ожидали. Конечным результатом является то, что вы не можете делать предположений о состоянии системы после пробуждения, и вы должны проверять для гарантии, что условия, которых вы ждали, действительно верные.

Другим соответствующим моментом, конечно, является то, что ваш процесс не может заснуть, пока не убедится, что где-то кто-то другой разбудит его. Код, выполняющий пробуждение, должен быть способен найти ваш процесс, чтобы выполнить свою работу. Уверенность, что пробуждение произойдёт, является результатом продумывания вашего кода и точного знания для каждого засыпания, что ряд событий приведёт к тому, что сон закончится. Напротив, обеспечение возможности для вашего спящего процесса быть найденным осуществляется через структуру данных, названную ***очередью ожидания***. Очередь ожидания является тем, чем называется: список всех процессов, ожидающих определённого события.

В Linux очередь управляется с помощью "головы очереди ожидания", структуры типа **wait_queue_head_t**, которая определена в ***<linux/wait.h>***. Голова очереди ожидания может быть определена и проинициализирована статически:

```
DECLARE_WAIT_QUEUE_HEAD(name);
```

или динамически, как здесь:

```
wait_queue_head_t my_queue;
init_waitqueue_head(&my_queue);
```

Мы вернёмся к структуре очереди ожидания в ближайшее время, но мы знаем достаточно, чтобы сейчас бросить первый взгляд на засыпание и пробуждение.

Простое засыпание

Когда процесс засыпает, он делает это в надежде, что какие-то условия станут в будущем истинными. Как мы уже отмечали ранее, любой процесс, который спит, должен проверить, чтобы убедиться, что состояние, которое он ждал, действительно истинное, когда он проснётся снова. Простейший способом заснуть в ядре Linux является макрос, названный ***wait_event*** (в нескольких вариантах); он сочетает в себе обработку деталей засыпания с проверкой состояния ожидающего процесса. Формами ***wait_event*** являются:

```
wait_event(queue, condition)
wait_event_interruptible(queue, condition)
wait_event_timeout(queue, condition, timeout)
wait_event_interruptible_timeout(queue, condition, timeout)
```

Во всех вышеуказанных формах **queue** является головой очереди ожидания для использования. Обратите внимание, что она передаётся "по значению". **condition (условие)** является произвольным логическим выражением, которое оценивается макросом до и после сна; пока **condition** оценивается как истинное значение, процесс продолжает спать. Заметим, что **condition** может быть оценено произвольное число раз, поэтому это не должно иметь каких-либо побочных эффектов.

Если вы используете ***wait_event***, ваш процесс помещается в непрерываемый сон, который, как мы уже отмечали ранее, как правило, не то, что вы хотите. Предпочтительной

альтернативой является ***wait_event_interruptible***, который может быть прерван сигналом. Эта версия возвращает целое число, которое вы должны проверить; ненулевое значение означает, что ваш сон был прерван каким-то сигналом и ваш драйвер должен, вероятно, вернуть **-ERESTARTSYS**. Последние версии (***wait_event_timeout*** и ***wait_event_interruptible_timeout***) ожидают ограниченное время; после истечения этого временного периода (в пересчёте на тики (jiffies), которые мы будем обсуждать в Глава 7 174), макросы возвращаются со значением 0 независимо от оценки **condition**.

Вторая часть картины, конечно, пробуждение. Какой-то иной поток исполнения (другой процесс или, вероятно, обработчик прерывания) должен пробудить вас, так как ваш процесс, конечно, спит. Основная функция, которая будит спящий процесс, называется ***wake_up***. Она поставляется в нескольких формах (но мы сейчас рассмотрим только две из них):

```
void wake_up(wait_queue_head_t *queue);
void wake_up_interruptible(wait_queue_head_t *queue);
```

wake_up будит все процессы, ожидающие в данной очереди (хотя ситуация немного сложнее, как мы увидим позже). Другая форма (***wake_up_interruptible***) ограничивает себя процессами, находящимися в прерываемом сне. В общем, эти две на практике неотличимы (если вы используете прерываемый сон); принято использовать ***wake_up***, если вы используете ***wait_event*** и ***wake_up_interruptible***, если вы используете ***wait_event_interruptible***.

Теперь мы знаем достаточно, чтобы взглянуть на простой пример засыпания и пробуждения. В исходниках примера вы можете найти модуль под названием ***sleepy***. Он реализует устройство с простым поведением: любой процесс, который пытается читать из устройства, погружается в сон. Всякий раз, когда процесс пишет в устройство, все спящие процессы пробуждаются. Это поведение реализуется следующими методами ***read*** и ***write***:

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;

ssize_t sleepy_read (struct file *filp, char __user *buf, size_t count,
loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) going to sleep\n",
                        current->pid, current->comm);
    wait_event_interruptible(wq, flag != 0);
    flag = 0;
    printk(KERN_DEBUG "awoken %i (%s)\n", current->pid, current->comm);
    return 0; /* конец файла */
}

ssize_t sleepy_write (struct file *filp, const char __user *buf, size_t
count, loff_t *pos)
{
    printk(KERN_DEBUG "process %i (%s) awakening the readers...\n",
                        current->pid, current->comm);
    flag = 1;
    wake_up_interruptible(&wq);
    return count; /* успешно, избегаем повтора */
}
```

Обратите внимание на использование в этом примере переменной **flag**. Используя ***wait_event_interruptible*** для проверки условия, которое должно стать истинным, мы используем **flag**, чтобы создать это условие.

Интересно, что произойдёт, если ***два*** процесса ждут, когда вызывается ***sleepy_write***. Так как ***sleepy_read*** сбрасывает **flag** в 0, как только она просыпается, можно подумать, что второй проснувшийся процесс немедленно вернётся в сон. На однопроцессорной системе, то есть почти всегда, это и происходит. Но важно понять, почему вы не можете рассчитывать на такое поведение. Вызов ***wake_up_interruptible будет*** будить оба спящих процесса. Вполне возможно, что они оба заметят, что **flag** ненулевой перед перед тем, как кто-то получит возможность сбросить его. Для этого простого модуля это состояние гонок несущественно. В настоящем драйвере такая гонка может создать редкие сбои, которые трудно диагностировать. Для правильной работы необходимо, чтобы ровно один процесс видел ненулевое значение, оно должно быть проверено атомарным образом. В ближайшее время мы увидим, как такую ситуацию обрабатывает настоящий драйвер. Но сначала мы должны рассмотреть другую тему.

Блокирующие и неблокирующие операции

Прежде чем рассмотрим реализацию полнофункциональных методов ***read*** и ***write***, мы должны коснуться последнего момента, то есть решения, когда процесс помещать в сон. Есть случаи, когда реализация правильной семантики Unix требует, чтобы операция не блокировалась, даже если она не может быть выполнена полностью.

Есть также случаи, когда вызывающий процесс информирует вас, что он не хочет блокирования, независимо от того, сможет ввод/вывод что-то выполнить вообще. Явно неблокирующий ввод/вывод обозначается флагом **O_NONBLOCK** в **filp->f_flags**. Этот флаг определён в ***<linux/fcntl.h>***, который автоматически подключается ***<linux/fs.h>***. Флаг получил свое название "открыть без блокировки" (“open-nonblock”), поскольку он может быть указан во время открытия (и первоначально мог быть указан только там). Если вы просмотрите исходный код, то сможете найти несколько ссылок на флаг **O_NDELAY**; это альтернативное имя для **O_NONBLOCK**, принятое для обеспечения совместимости с кодом System V. Флаг очищен по умолчанию, так как нормальным поведением процесса при ожидании данных является только сон. В случае блокирующий операции, которая является такой по умолчанию, для соблюдения стандартной семантики должно быть реализовано следующее поведение:

- Если процесс вызывает ***read***, но никакие данные (пока) не доступны, процесс должен заблокироваться. Процесс пробуждается, как только приходят какие-то данные, и эти данные возвращаются вызывающему, даже если их меньше, чем запрошено аргументом метода **count**.
- Если процесс вызывает ***write*** и в буфере нет места, процесс должен заблокироваться и он должен быть в другой очереди ожидания, не той, которая используется для чтения. Когда какие-то данные будут записаны в аппаратное устройство и в выходном буфере появится свободное место, процесс пробуждается и вызов ***write*** успешен, хотя эти данные могут быть лишь частично записаны, если в буфере не достаточно места для запрошенного **count** байт.

Оба эти положения предполагают, что для ввода и вывода существуют буферы; на практике их имеет почти каждый драйвер устройства. Входной буфер необходим, чтобы избежать потери данных, которые прибывают, когда никто их не читает. В отличие от этого, данные не могут быть потеряны при ***записи***, потому что если системный вызов не принимает байты

данных, они остаются в буфере пользовательского пространства. Несмотря на это, выходной буфер почти всегда полезен для выжимания из аппаратуры более высокой производительности.

Прирост производительности при реализации выходного буфера в драйвере происходит в результате сокращения числа переключений контекста и переходов пользователь-ядро/ядро-пользователь. Без выходного буфера (предполагающего медленное устройство) каждым системным вызовом принимаются лишь один или несколько символов и хотя один процесс спит в ***write***, другой процесс работает (то самое переключение контекста). Когда первый процесс разбужен, он возобновляется (другое переключение контекста), ***write*** возвращается (переход ядро/пользователь) и процесс вновь получает системный вызов, чтобы записать больше данных (переход пользователь/ядро); вызов блокируется и цикл продолжается. Кроме того, выходной буфер позволяет драйверу принимать при каждом вызове ***write*** большие куски данных, соответственно увеличивая производительность. Если этот буфер достаточно большой, вызов ***write*** будет успешным с первой попытки - забуферированные данные будут переданы в устройство позже, без необходимости идти обратно в пространство пользователя за вторым или третьим вызовом ***write***. Выбор подходящего размера выходного буфера, очевидно, зависит от устройства.

Мы не используем входной буфер в ***scull***, так как данные уже доступны, когда вызвана ***read***. Аналогично, не используется выходной буфер, так как данные просто скопированы в область памяти, связанную с устройством. По сути, устройство представляет собой буфер, поэтому реализация дополнительных буферов была бы излишеством. Мы рассмотрим использование буферов в Главе 10 246.

Поведение ***read*** и ***write*** различно, если не указан **O_NONBLOCK**. В этом случае вызовы просто вернут **-EAGAIN** (“try it again”, "попробуйте снова"), если процесс вызывает ***read***, когда данные не доступны или если он вызывает ***write***, когда в буфере нет места.

Как и следовало ожидать, неблокирующие операции возвращаются немедленно, позволяя приложению собирать данные. Приложения должны быть осторожны при использовании функций ***stdio*** при работе с неблокирующими файлами, потому что их легко спутать с неблокирующим возвратом EOF. Они всегда должны проверять номер ошибки.

Естественно, **O_NONBLOCK** имеет смысл также и в методе ***open***. Это происходит, когда вызов может действительно заблокироваться на длительное время; например, при открытии (для доступа на чтение) FIFO, в который никто не пишет (пока), или при доступе к заблокированному файлу на диске. Обычно открытие устройство или успешно или неудачно, без необходимости ожидания внешних событий. Иногда, однако, открытие устройства требует долгой инициализации и вы можете выбрать поддержку **O_NONBLOCK** в вашем методе ***open*** возвращая сразу после начала процесса инициализации устройства **-EAGAIN**, если флаг установлен. Драйвер может также реализовать блокирующее открытие для поддержки политик доступа по аналогии с блокировками файла. Мы увидим одну из таких реализаций далее в этой главе в разделе "Блокирующее открытие как альтернатива EBUSY" 167.

Некоторые драйверы могут также реализовать специальную семантику для **O_NONBLOCK**; например, открытие ленточного устройства обычно блокируется, пока не будет вставлена лента. Если ленточный накопитель открыт с **O_NONBLOCK**, открытие удастся немедленно, независимо от того, присутствует носитель или нет.

Флаг неблокирования воздействует только файловые операции ***read***, ***write*** и ***open***.

Пример блокирующего ввода/вывода

Наконец, мы переходим к примеру реального метода драйвера, который реализует блокирующий ввод/вывод. Этот пример взят из драйвера ***scullpipe***; это особая форма ***scull***, которая реализует канало-подобное устройство.

В драйвере процесс, заблокированный в вызове ***read***, пробуждается, когда поступают данные; обычно аппаратура выдаёт прерывание для сигнализации о таком событии и драйвер будит ожидающие процессы при обработке прерывания. Драйвер ***scullpipe*** работает по-другому, так что он может работать не требуя какого-либо особенного оборудования или обработки прерывания. Мы решили использовать другой процесс для генерации данных и пробуждения читающего процесса; аналогично, читающие процессы используются для пробуждения пишущих процессов, которые ждут, когда пространство буфера станет доступным.

Драйвер устройства использует структуру устройства, которая содержит две очереди ожидания и буфер. Размер буфера настраивается обычным способом (во время компиляции, во время загрузки, или во время работы).

```
struct scull_pipe {
    wait_queue_head_t inq, outq;       /* очереди чтения и записи */
    char *buffer, *end;                /* начало и конец буфера */
    int buffersize;                    /* используется при работе с
указателем */
    char *rp, *wp;                     /* откуда читать, куда писать */
    int nreaders, nwriters;            /* число открытий для чтения/записи */
    struct fasync_struct *async_queue; /* асинхронные читатели */
    struct semaphore sem;              /* семафор взаимного исключения */
    struct cdev cdev;                  /* структура символьного устройства */
};
```

Реализация ***read*** управляет и блокирующим и неблокирующим вводом и выглядит следующим образом:

```
static ssize_t scull_p_read (struct file *filp, char __user *buf, size_t
count, loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;

    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    while (dev->rp == dev->wp) { /* читать нечего */
        up(&dev->sem); /* освободить блокировку */
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" reading: going to sleep\n", current->comm);
        if (wait_event_interruptible(dev->inq, (dev->rp != dev->wp)))
            return -ERESTARTSYS; /* сигнал: передать уровню файловой системы
для обработки */
        /* иначе цикл, но сначала перезапросим блокировку */
        if (down_interruptible(&dev->sem))
```

```
            return -ERESTARTSYS;
    }
    /* ok, данные есть, вернуть что-нибудь */
    if (dev->wp > dev->rp)
        count = min(count, (size_t)(dev->wp - dev->rp));
    else /* указатель записи возвращён в начало, вернуть данные в dev->end */
        count = min(count, (size_t)(dev->end - dev->rp));
    if (copy_to_user(buf, dev->rp, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->rp += count;
    if (dev->rp == dev->end)
        dev->rp = dev->buffer; /* вернуться к началу */
    up (&dev->sem);

    /* и наконец, разбудить всех писателей и вернуться */
    wake_up_interruptible(&dev->outq);
    PDEBUG("\"%s\" did read %li bytes\n",current->comm, (long)count);
    return count;
}
```

Как вы можете видеть, мы оставили в коде некоторые строки с **PDEBUG**. При компиляции драйвера вы можете разрешить сообщения, чтобы было проще проследить взаимодействие различных процессов.

Давайте внимательно посмотрим, как ***scull_p_read*** обрабатывает ожидание данных. Цикл ***while*** проверяет буфер, удерживая семафор устройства. Если мы узнаём, что там есть данные, мы можем вернуть их пользователю сразу же, без сна, поэтому всё тело цикла пропускается. Вместо этого, если буфер пуст, мы должны заснуть. Однако, прежде чем мы сможем это сделать, мы должны освободить семафор устройства; если бы мы заснули, удерживая его, писатели никогда бы не имели возможность разбудить нас. После того, как семафор был освобождён, мы делаем быструю проверку, чтобы увидеть, не запросил ли пользователь неблокирующий ввод/вывод и вернуться, если это так. В противном случае, настало время вызова ***wait_event_interruptible***.

Как только мы получили вызов, что-то разбудило нас, но мы не знаем, что. Одна возможность заключается в том, что процесс получил сигнал. Оператор **if**, который содержит вызов ***wait_event_interruptible***, проверяет этот случай. Эта проверка обеспечивает правильную и ожидаемую реакцию на сигналы, которые могли бы быть ответственными за пробуждение этого процесса (так как мы были в прерываемом сне). Если получен сигнал и он не был заблокирован процессом, правильное поведение - дать верхним слоям ядра обработать это событие. Для этого драйвер возвращает вызывающему **-ERESTARTSYS**; это значение используется внутри слоем виртуальной файловой системы (VFS), который либо перезапускает системный вызов, либо возвращает **-EINTR** в пространство пользователя. Мы используем такой же тип проверки при обработке сигнала в каждой реализации ***read*** и ***write***.

Однако, даже при отсутствии сигнала мы пока не знаем точно, есть ли данные, чтобы их забрать. Кто-то другой мог так же ожидать данные и он мог выиграть гонку и получить данные первым. Поэтому мы должны снова получить семафор устройства; только после этого мы сможем снова проверить буфер чтения (в цикле **while**) и действительно узнать, что мы можем вернуть данные в буфере пользователю. Конечным результатом всего этого кода является то, что когда мы выходим из цикла **while**, мы знаем, что семафор удерживается и буфер содержит

данные, которые мы можем использовать.

Просто для полноты позвольте нам заметить, что ***scull_p_read*** может заснуть в другом месте, после того, как мы получили семафор устройства: это вызов ***copy_to_user***. Если ***scull*** засыпает при копировании данных от ядра к пользовательскому пространству, он спит удерживая семафор устройства. Удержание семафора в данном случае является оправданным, поскольку он не вызывает взаимоблокировку системы (мы знаем, что ядро будет выполнять копирование в пространство пользователя и разбудит нас, не пытаясь в процессе заблокировать тот же семафор) и важно то, что массив памяти устройства не меняется, пока драйвер спит.

Подробности засыпания

Многие драйверы в состоянии удовлетворить свои требования к засыпанию функциями, которые мы рассматривали до сих пор. Однако, есть ситуации, когда требуется более глубокое понимание, как работает механизм очереди ожидания в Linux. Комплексное блокирование или требования производительности могут заставить драйвер использовать низкоуровневые функции для выполнения сна. В этом разделе мы рассмотрим более низкий уровень, чтобы получить понимание того, что происходит на самом деле, когда процесс засыпает.

Как процесс засыпает

Если вы загляните в ***<linux/wait.h>***, то увидите, что структура данных типа **wait_queue_head_t** довольно проста; она содержит спин-блокировку и связный список. Этот список является очередью ожидания, объекты которой объявлены с типом **wait_queue_t**. Эта структура содержит информацию о спящем процессе и как он хотел бы проснуться.

Первым шагом в процессе помещения процесса в сон обычно является создание и инициализация структуры **wait_queue_t**, с последующим дополнением её к соответствующей очереди ожидания. Когда всё на месте, тот, кто должен пробуждать, сможет найти нужные процессы.

Следующим шагом будет установить состояние процесса, чтобы пометить его как спящий. Есть несколько состояний задачи, определённых в ***<linux/sched.h>***. **TASK_RUNNING** означает, что процесс способен работать, хотя не обязательно выполнение процессором в какой-то определённый момент. Есть два состояния, которые показывают, что процесс спит: **TASK_INTERRUPTIBLE** и **TASK_UNINTERRUPTIBLE**; они соответствуют, конечно, двум типам сна. Другие состояния обычно не представляют интерес для авторов драйверов.

В ядре версии 2.6 коду драйвера обычно нет необходимости напрямую управлять состоянием процессом. Однако, если вам необходимо сделать это, используется вызов:

```
void set_current_state(int new_state);
```

В старом коде вместо этого вы часто встретите что-то наподобие такого:

```
current->state = TASK_INTERRUPTIBLE;
```

Но изменения непосредственно **current** в такой форме не рекомендуются; при изменении структуры данных такой код легко ломается. Однако, вышеприведённый код показывает, что изменение текущего состояния процесса само по себе не помещает его в режим сна. Изменив

текущее состояние, вы изменили способ рассмотрения процесса планировщиком, но вы ещё не отдали процессор.

Отказ от процессора является окончательным шагом, но сначала необходимо выполнить другое: вы должны сначала проверить условие, которое вы ожидаете, засыпая. Невыполнение этой проверки создаёт условия для состояния гонок; что произойдёт, если условие выполнилось, пока вы были заняты выше в процессе и какой-то другой поток только что пытался разбудить вас? Вы могли пропустить этот сигнал для пробуждения вообще и спать дольше, чем предполагалось. Следовательно, ниже в коде, который засыпает, вы обычно увидите что-то похожее на:

```
if (!condition)
    schedule( );
```

Проверяя наше состояние ***после*** установки состояния процесса, мы застрахованы от всех возможных последовательностей событий. Если условие, которое мы ожидаем, произошло перед установкой состояния процесса, мы узнаём об этом в этой проверке и не засыпаем. Если пробуждение происходит позже, этот процесс сделан работоспособным независимо от того, было ли выполнено засыпание.

Вызов ***schedule*** это, конечно, способ вызова планировщика и передача процессора. Всякий раз, когда вы вызываете эту функцию, вы предлагаете ядру рассмотреть, какой процесс должен быть запущен и передать управление этому процессу, если это необходимо. Таким образом, никогда не известно, когда ***schedule*** вернёт управление вашему коду.

После проверки **if** и возможного вызова (и возвращения из) ***schedule***, предстоит сделать некоторую очистку. Поскольку код больше не намерен спать, он должен сбросить состояние задачи в **TASK_RUNNING**. Если код только что вернулся из ***schedule***, этот шаг является ненужным; эта функция не вернётся, пока этот процесс не перейдёт в работающее состояние. Но если вызов ***schedule*** был пропущен, потому что больше не было необходимости спать, состояние процесса будет некорректным. Необходимо также удалить этот процесс из очереди ожидания, или он может быть разбужен более одного раза.

Ручное управление засыпанием

В предыдущих версиях ядра Linux нетривиальное засыпание требовало от программиста обрабатывать все вышеперечисленные шаги вручную. Это был утомительный процесс с привлечением значительного количества подверженного ошибкам шаблонного кода. Программисты могут всё ещё кодировать ручное управление засыпанием таким образом, если они этого хотят; ***<linux/sched.h>*** содержит все необходимые определения, а ядра изобилует примерами. Однако, существует более простой способ.

Первым шагом является создание и инициализация объекта очереди ожидания. Это, как правило, делается таким макросом:

```
DEFINE_WAIT(my_wait);
```

здесь **name** является именем объекта переменной очереди ожидания. Вы также можете делать это в два этапа:

```
wait_queue_t my_wait;
init_wait(&my_wait);
```

Но обычно легче поставить строчку с **DEFINE_WAIT** в начало цикла, реализующего ваш сон.

Следующим шагом будет добавить ваш объект очереди ожидания в очередь и установить состояние процесса. Обе эти задачи решаются этой функцией:

```
void prepare_to_wait(wait_queue_head_t *queue,
                     wait_queue_t *wait,
                     int state);
```

Здесь, **queue** и **wait** являются головой очереди ожидания и очередью ожидания процесса соответственно. **state** является новым состоянием процесса; оно должна быть либо **TASK_INTERRUPTIBLE** (для прерываемых состояний сна, которые, как правило, то, что вы хотите) или **TASK_UNINTERRUPTIBLE** (для непрерываемых состояний сна).

После вызова ***prepare_to_wait*** этот процесс может вызвать ***schedule*** - после проверки, чтобы быть уверенным, что ждать всё ещё необходимо. После возвращения из ***schedule*** наступает время очистки. Эта задача тоже обрабатывается специальной функцией:

```
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
```

После этого ваш код может проверить своё состояние и посмотреть, нет ли необходимости подождать ещё.

Давно настало время для примера. Раньше мы рассматривали метод ***read*** в ***scullpipe***, который использует ***wait_event***. Метод ***write*** в этом же драйвере вместо этого выполняет свои ожидания с ***prepare_to_wait*** и ***finish_wait***. Обычно вы не будете таким образом смешивать методы в рамках одного драйвера, но мы сделали это для того, чтобы показать оба способа обработки засыпания.

Во-первых, давайте для полноты посмотрим на сам метод записи:

```
/* Как много свободного места? */
static int spacefree(struct scull_pipe *dev)
{
    if (dev->rp = = dev->wp)
        return dev->buffersize - 1;
    return ((dev->rp + dev->buffersize - dev->wp) % dev->buffersize) - 1;
}

static ssize_t scull_p_write(struct file *filp, const char __user *buf,
size_t count, loff_t *f_pos)
{
    struct scull_pipe *dev = filp->private_data;
    int result;
    if (down_interruptible(&dev->sem))
        return -ERESTARTSYS;

    /* Убедимся, что для записи есть место */
    result = scull_getwritespace(dev, filp);
    if (result)
        return result; /* scull_getwritespace вызвал up(&dev->sem) */
```

```
    /* ok, место есть, примем что-то */
    count = min(count, (size_t)spacefree(dev));
    if (dev->wp >= dev->rp)
        count = min(count, (size_t)(dev->end - dev->wp)); /* к концу буфера
*/
    else /* the write pointer has wrapped, fill up to rp-1 */
        count = min(count, (size_t)(dev->rp - dev->wp - 1));
    PDEBUG("Going to accept %li bytes to %p from %p\n", (long)count, dev->wp,
buf);
    if (copy_from_user(dev->wp, buf, count)) {
        up (&dev->sem);
        return -EFAULT;
    }
    dev->wp += count;
    if (dev->wp == dev->end)
        dev->wp = dev->buffer; /* вернуть указатель в начало */
    up(&dev->sem);

    /* наконец, пробудить любого читателя */
    wake_up_interruptible(&dev->inq); /* блокированного в read( ) и select( )
*/

    /* и послать сигнал асинхронным читателям, объясняется позже в Главе 6 */
    if (dev->async_queue)
        kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
    PDEBUG("\"%s\" did write %li bytes\n",current->comm, (long)count);
    return count;
}
```

Этот код выглядит похожим на метод ***read*** за исключением того, что мы поместили код, который засыпает, в отдельную функцию, названную ***scull_getwritespace***. Его задача заключается в том, чтобы убедиться, что в буфере есть место для новых данных, или спать в случае необходимости, пока пространство не освободится. После того, как пространство появилось, ***scull_p_write*** может просто скопировать данные пользователя туда, настроить указатели и пробудить любые процессы, которые могут находится в ожидании чтения данных.

Код, который фактически обрабатывает сон:

```
/* Ожидание пространства для записи; вызывающий должен удерживать семафор
устройства.
 * В случае ошибки семафор будет освобождён перед возвращением. */
static int scull_getwritespace(struct scull_pipe *dev, struct file *filp)
{
    while (spacefree(dev) == 0) { /* полон */
        DEFINE_WAIT(wait);

        up(&dev->sem);
        if (filp->f_flags & O_NONBLOCK)
            return -EAGAIN;
        PDEBUG("\"%s\" writing: going to sleep\n",current->comm);
        prepare_to_wait(&dev->outq, &wait, TASK_INTERRUPTIBLE);
        if (spacefree(dev) == 0)
            schedule( );
        finish_wait(&dev->outq, &wait);
```

```
        if (signal_pending(current))
            return -ERESTARTSYS; /* сигнал: передать на уровень файловой
системы для обработки */
        if (down_interruptible(&dev->sem))
            return -ERESTARTSYS;
    }
    return 0;
}
```

Отметим ещё раз содержание цикла **while**. При наличии свободного места сон не требуется, эта функция просто возвращается. В противном случае, она должна отказаться от семафора устройства и ждать. Код использует ***DEFINE_WAIT*** для подготовки объекта очереди ожидания и ***prepare_to_wait*** для подготовки к фактическому засыпанию. Затем идёт обязательная проверка буфера; мы должны обработать случай, когда в буфере появляется доступное пространство после того, как мы вошли в цикл **while** (и отказались от семафора), но прежде чем поместим себя в очередь ожидания. Без такой проверки, если бы читающие процессы смогли бы полностью освободить буфер в это время, мы могли бы не только пропустить сигнал пробуждения, а даже когда-нибудь заснуть навечно. Убедившись, что мы должны заснуть, мы можем вызвать ***schedule***.

Стоит снова посмотреть на этот случай: что случится, если пробуждения произойдёт между проверкой **if** и вызовом ***schedule***? В таком случае всё хорошо. Сигнал пробуждения сбрасывает состояние процесса в **TASK_RUNNING** и ***schedule*** возвращается, хотя не обязательно сразу. Пока проверка происходит после того, как процесс поместил себя в очередь ожидания и изменил своё состояние, всё будет работать.

Для завершения мы вызываем ***finish_wait***. Вызов ***signal_pending*** говорит нам, были ли мы разбужены сигналом; если так, то мы должны вернуться к пользователю и дать ему возможность попробовать позже. В противном случае, мы повторно получаем семафор и снова как обычно проверяем наличие свободного пространства.

Эксклюзивные ожидания

Мы видели, что когда процесс вызывает для очереди ожидания ***wake_up***, все процессы, ожидающие в этой очереди, становятся работающими. Во многих случаях это правильное поведение. Однако, в других, возможно, необходимо знать заранее, что только одному из разбуженных процессов удастся получить желаемый ресурс, а остальные будут просто вынуждены продолжать спать. Однако, каждый из этих процессов, чтобы получить процессор, борется за ресурсы (и любые управляющие блокировки) и возвращается спать явным образом. Если количество процессов в ожидании очереди большое, такое поведение "громадного стада" может значительно ухудшить производительность системы.

В ответ на реальные проблемы огромного множества, разработчики ядра добавили в ядро опцию "эксклюзивное ожидание". Эксклюзивное ожидание работает очень похоже на нормальное засыпание с двумя важными отличиями:

- Если объект очереди ожидания имеет установленный флаг **WQ_FLAG_EXCLUSIVE**, он добавляется в конец очереди ожидания. Без этого флага, наоборот, добавляется в начало.
- Когда для очереди ожидания вызвана ***wake_up***, она останавливается после пробуждения первого процессе, который имеет установленный флаг **WQ_FLAG_EXCLUSIVE**.

Конечным результатом является то, что процессы, находящиеся в эксклюзивном ожидании пробуждаются по одному за раз, упорядоченным образом и не создают огромного стада. Однако, ядро всё же пробуждает одновременно всех ожидающих неэксклюзивно.

Об использовании эксклюзивного ожидания в драйвере стоит задуматься, если встретились два условия: вы ожидаете значительной борьбы за ресурс и пробуждения одного процесса достаточно, чтобы полностью употребить ресурс, когда он станет доступен. Эксклюзивные ожидания хорошо работают, например, на веб-сервере Apache; когда приходит новое соединение, только один из (часто многих) процессов Apache в системе должен пробудиться для его обслуживания. Мы, однако, не использовали эксклюзивные ожидания в драйвере ***scullpipe***; редко, чтобы читатели боролись за данные (или писатели в пространство буфера) и мы не можем знать, что один из читателей после пробуждения употребит все имеющиеся данные.

Помещение процесса в прерываемое ожидание является просто вопросом вызова ***prepare_to_wait_exclusive***:

```
void prepare_to_wait_exclusive(wait_queue_head_t *queue,
                               wait_queue_t *wait,
                               int state);
```

Этот вызов при использовании вместо ***prepare_to_wait*** устанавливает "эксклюзивный" флаг в объекте очереди ожидания и добавляет процесс в конец очереди ожидания. Обратите внимание, что нет способ выполнения эксклюзивного ожидания с помощью ***wait_event*** и её вариантов.

Детали пробуждения

Объяснение, которое мы представили для пробуждения процесса проще, чем то, что на самом деле происходит внутри ядра. Фактическое поведение является результатом контроля функцией объекта очереди ожидания пробуждаемого процесса. Функция по умолчанию для пробуждения (* Она имеет образное имя *default_wake_function*.) устанавливает процесс в работающее состояние и, возможно, выполняет переключение контекста для этого процесса, если он имеет более высокий приоритет. Драйверы устройств никогда не должны заменять её другой функцией пробуждения; если вы уверены, что ваш случай является исключением, смотрите ***<linux/wait.h>*** для информации о том, как это делать.

Мы ещё не видели все варианты ***wake_up***. Большинству авторов драйвера другие не требуются, но для полноты здесь полный набор:

wake_up(wait_queue_head_t *queue);
wake_up_interruptible(wait_queue_head_t *queue);
wake_up пробуждает каждый процесс в очереди, которая не находится в эксклюзивном ожидании, и только одного ожидающего эксклюзивно, если он существует. ***wake_up_interruptible*** делает то же самое, за исключением того, что она пропускает процессы в непрерываемом сне. Эти функции, прежде чем вернуться, могут пробудить один или более процессов для планировщика (хотя это не произойдет, если они вызываются из атомарного контекста).

wake_up_nr(wait_queue_head_t *queue, int nr);
wake_up_interruptible_nr(wait_queue_head_t *queue, int nr);
Эти функции выполняются аналогично ***wake_up*** за исключением того, что они могут

разбудить до **nr** ожидающих эксклюзивно вместо одного. Обратите внимание, что передача 0 интерпретируется как запрос на пробуждение всех ожидающих эксклюзивно, а не ни одного из них.

wake_up_all(wait_queue_head_t *queue);
wake_up_interruptible_all(wait_queue_head_t *queue);
Эта форма ***wake_up*** пробуждает все процессы, независимо от того, выполняют ли эксклюзивное ожидание или нет (хотя прерываемая форма всё же пропускает процессы, выполняющие непрерываемое ожидания).

wake_up_interruptible_sync(wait_queue_head_t *queue);
Как правило, процесс, который пробудил, может вытеснить текущий процесс и быть запланирован в процессор до возвращения ***wake_up***. Иными словами, вызов ***wake_up*** может не быть атомарным. Если процесс, вызывающий ***wake_up***, работает в атомарном контексте (например, держит спин-блокировку или является обработчиком прерываний), это перепланирование не произойдёт. Обычно, эта защита является адекватной. Если, однако, вам необходимо указать это явно, чтобы не потерять процессор в это время, вы можете использовать "синхронный" вариант ***wake_up_interruptible***. Эта функция является наиболее часто используемой, когда вызывающий так или иначе собирается перепланировать работу, и более эффективно в первую очередь просто закончить небольшую оставшуюся работу.

Если всё перечисленное выше на первый взгляд не совсем ясно, не беспокойтесь. Очень малому числу драйверов необходимо вызывать что-то, кроме ***wake_up_interruptible***.

Древняя история: sleep_on

Если вы проводите какое-то время, копаясь в исходных текстах ядра, вы, скорее всего, встречали две функции, которыми мы до сих пор пренебрегли для обсуждения:

```
void sleep_on(wait_queue_head_t *queue);
void interruptible_sleep_on(wait_queue_head_t *queue);
```

Как и следовало ожидать, эти функции безоговорочного помещают текущий процесс в сон в данную очередь. Однако, эти функции являются сильно устаревшими и вы никогда не должны использовать их. Проблема очевидна, если вы об этом подумаете: ***sleep_on*** не предлагает никакого способа защититься от состояния гонок. Всегда существует окно между тем, когда ваш код решит, что должен заснуть, и когда действительно заснёт в ***sleep_on***. Пробуждение, которое приходит в течение этого окна, пропускается. По этой причине код, который вызывает ***sleep_on***, никогда не бывает полностью безопасным.

В текущих планах удаление из ядра вызова ***sleep_on*** и его вариантов (есть несколько форм со временем ожидания, которые мы не показали) в не слишком отдалённом будущем.

Тестирование драйвера Scullpipe

Мы видели, как реализует блокирующий ввод/вывод драйвер ***scullpipe***. Если вы захотите попробовать его, исходник этого драйвера может быть найден в другой книге примеров. Блокирующий ввод/вывод в действии можно увидеть, открыв два окна. В первом можно запустить такую команду, как ***cat /dev/scullpipe***. Если затем в другом окне скопировать любой файл в ***/dev/scullpipe***, вы должны увидеть содержимое файла, которое появится в первом окне.

Тестирование неблокирующей деятельности сложнее, поскольку обычные программы, доступные оболочке, не выполняют неблокирующие операции. Каталог исходников ***misc-progs*** содержит следующую простую программу, названную ***nbtest***, для тестирования неблокирующих операций. Всё, что она делает, это копирование своего входа на свой выход, используя неблокирующий ввод/вывод и задержку между попытками. Время задержки задаётся в командной строке и по умолчанию равно одной секундой.

```
int main(int argc, char **argv)
{
    int delay = 1, n, m = 0;

    if (argc > 1)
        delay = atoi(argv[1]);
    fcntl(0, F_SETFL, fcntl(0,F_GETFL) | O_NONBLOCK); /* stdin */
    fcntl(1, F_SETFL, fcntl(1,F_GETFL) | O_NONBLOCK); /* stdout */

    while (1) {
        n = read(0, buffer, 4096);
        if (n >= 0)
            m = write(1, buffer, n);
        if ((n < 0 || m < 0) && (errno != EAGAIN))
            break;
        sleep(delay);
    }
    perror(n < 0 ? "stdin" : "stdout");
    exit(1);
}
```

Если запустить эту программу под процессом утилиты трассировки, такой как ***strace***, можно увидеть успешность или неудачу каждой операции, в зависимости от того, имеются ли данные при попытке выполнения операций.

poll и select

Приложения, использующие неблокирующий ввод/вывод, часто также используют системные вызовы ***poll***, ***select*** и ***epoll***. ***poll***, ***select*** и ***epoll*** в сущности имеют одинаковую функциональность: каждый позволяет процессу определить, может ли он читать или записывать данные в один или более открытых файлов без блокировки. Эти вызовы могут также блокировать процесс, пока любой из заданного набора файловых дескрипторов не станет доступным для чтения или записи. Поэтому они часто используются приложениями, которые должны использовать много входных и выходных потоков, не застревая на каком-то одном из них. Такая же функциональность предлагается множеством функций, потому что две были реализованы в Unix почти в одно и то же время двумя разными группами: ***select (выбор)*** был введён в BSD Unix, тогда как ***poll (опрос)*** был решением System V. Вызов ***epoll*** (* На самом деле *epoll* представляет собой набор из трёх вызовов, которые вместе могут быть использованы для достижения функциональности последовательного опроса. Однако, для наших целей мы можем думать о нём как об одном вызове.) был добавлен в версии 2.5.45 в качестве одного из способов сделать функцию опроса масштабируемой к тысячам файловых дескрипторов.

Поддержка любого из этих вызовов требует поддержки со стороны драйвера устройства. Эта поддержка (для всех трёх вызовов) обеспечивается с помощью метода драйвера ***poll (опрос)***. Этот метод имеет следующий прототип:

```
unsigned int (*poll) (struct file *filp, poll_table *wait);
```

Метод драйвера вызывается всякий раз, когда программа пользовательского пространства выполняет системный вызов ***poll***, ***select*** или ***epoll*** с участием дескриптора файла, связанного с драйвером. Метод устройства отвечает за эти два действия:

1. Вызвать ***poll_wait*** для одной или более очередей ожидания, что может свидетельствовать об изменении в статусе опроса. Если нет файловых дескрипторов, доступных в настоящее время для ввода/вывода, ядро заставит процесс ждать в очереди ожидания для всех файловых дескрипторов, переданных системному вызову.
2. Возврат битовой маски, описывающей операции (если таковые имеются), которые могут быть немедленно выполнены без блокировки.

Обе эти операции, как правило, просты и очень похожи от одного драйвера к другому. Предполагается, однако, что только драйвер может предоставить эту информацию и, следовательно, они должны быть реализованы индивидуально каждым драйвером.

Структура **poll_table**, второй аргумент метода ***poll***, используется в ядре для реализации вызовов ***poll***, ***select*** и ***epoll***; она объявлена в ***<linux/poll.h>***, которая должна быть подключена в исходник драйвера. Авторам драйверов не требуется знать о её внутренностях и они должны использовать её как непрозрачный объект; она передаётся в метод драйвера так, чтобы драйвер мог загружать её для каждой очереди ожидания, которая могла бы пробудить этот процесс и изменить статус операции ***poll***. Драйвер добавляет очередь ожидания к структуре ***poll_table*** вызывая функцию ***poll_wait***:

```
void poll_wait (struct file *filp, wait_queue_head_t *wait_queue, poll_table
*wait);
```

Вторая задача выполняется методом ***poll*** возвращением битовой маски описания, какие операции могут быть завершены немедленно; это тоже просто. Например, если в устройстве имеются данные, ***чтение*** выполнилось бы без сна; методу ***poll*** следует показать такое положение дел. Для обозначения возможных операций используются несколько флагов (определяемых через ***<linux/poll.h>***):

POLLIN
Этот бит должен быть установлен, если устройство может быть прочитано без блокировки.

POLLRDNORM
Этот бит должен быть установлен, если "нормальные" данные доступны для чтения. Читаемое устройство возвращает **(POLLIN | POLLRDNORM)**.

POLLRDBAND
Этот бит показывает, что для чтения из устройства доступны данные вне логического канала связи (из дополнительного вспомогательного канала). В настоящее время используется только в одном месте в ядре Linux (код DECnet) и, как правило, не применим к драйверам устройств.

POLLPRI
Высокоприоритетные данные (вспомогательного канала) могут быть прочитаны без блокировки. Этот бит заставляет ***select*** сообщить, что на файле произошло условие

исключения, потому что ***select*** сообщает о дополнительных данных как о состоянии исключения.

POLLHUP
Когда процесс, читающий это устройство, видит конец файла, драйвер должен установить **POLLHUP** (зависание). Процесс, вызывающий ***select*** говорит, что устройство это читаемо, как это диктуется функциональностью ***select***.

POLLERR
В устройстве произошла ошибка. При вызове ***poll*** об устройстве сообщается как о читаемом и записываемом, а ***read*** и ***write*** возвращают код ошибки без блокирования.

POLLOUT
Этот бит устанавливается в возвращаемом значении, если в это устройство можно записать без блокировки.

POLLWRNORM
Этот бит имеет такое же значение, как **POLLOUT** и иногда он действительно является тем же номером. Записываемое устройство возвращает **(POLLOUT | POLLWRNORM)**.

POLLWRBAND
Как и **POLLRDBAND**, этот бит означает, что данные с ненулевым приоритетом могут быть записаны в устройство. Только реализация ***poll*** для датаграммы (пакет данных + адресная информация) использует этот бит, так как датаграммы могут передавать данные в дополнительном канале.

Стоит повторить, что **POLLRDBAND** и **POLLWRBAND** имеют смысл только для файловых дескрипторов, связанных с сокетами: драйверы устройств обычно не будут использовать эти флаги.

Описание ***poll*** занимает много места для того, что является относительно простым для использования на практике. Рассмотрим реализацию метода ***poll*** в ***scullpipe***:

```
static unsigned int scull_p_poll(struct file *filp, poll_table *wait)
{
    struct scull_pipe *dev = filp->private_data;
    unsigned int mask = 0;

    /*
     * Этот буфер является круговым; он считается полным
     * если "wp" находится прямо позади "rp" и пустым, если
     * они равны.
     */
    down(&dev->sem);
    poll_wait(filp, &dev->inq, wait);
    poll_wait(filp, &dev->outq, wait);
    if (dev->rp != dev->wp)
        mask |= POLLIN | POLLRDNORM; /* читаемо */
    if (spacefree(dev))
        mask |= POLLOUT | POLLWRNORM; /* записываемо */
    up(&dev->sem);
    return mask;
}
```

Этот код просто добавляет две очереди ожидания ***scullpipe*** к **poll_table**, затем устанавливает соответствующие битовые маски в зависимости от того, могут ли данные быть считаны или записаны.

Показанный код ***poll*** опускает поддержку "конца файла", потому что ***scullpipe*** не поддерживает условие "конец файла". Для большинства реальных устройств метод ***poll*** должен вернуть **POLLHUP**, если данных больше нет (или не будет). Если вызывающий использовал системный вызов ***select***, об этом файле сообщается как о читаемом. Независимо от того, использовался ***poll*** или ***select***, приложение знает, что оно может вызвать ***чтение*** не ожидая вечно и метод ***read*** вернётся, просигнализировав 0 о конце файла.

С реальными FIFO, например, читатель видит конец файла, когда все писатели закрывают этот файл, тогда как читатель ***scullpipe*** никогда не видит конца файла. Поведение отличается, потому что FIFO предназначен, чтобы быть каналом связи между двумя процессами, а ***scullpipe*** является мусорной корзиной, куда каждый может поместить данные, пока есть хотя бы один читатель. Кроме того, нет смысла заново реализовывать то, что уже доступно в ядре, так что мы выбрали в нашем примере другое поведение для реализации.

Реализация "конца файла" так же, как в FIFO, означало бы сделать проверку **dev->nwriters** в ***read*** и в ***poll*** и сообщать о "конце файла" (как описано выше), если нет процесса, имеющего устройство открытым для записи. Однако, к сожалению, при такой реализации, если читатель открыл устройство ***scullpipe*** перед писателем, он будет видеть "конец файла", не имея шанса дождаться данных. Лучшим способом решить эту проблему будет осуществлять блокировку в ***open***, как это делают реальные FIFO; эта задача остаётся как упражнение для читателя.

Взаимодействие с read и write

Целью вызовов ***poll*** и ***select*** является определить заранее, будет ли операция ввода/вывода блокирована. В этом отношении они дополняют ***read*** и ***write***. Что более важно, ***poll*** и ***select*** полезны, потому что они позволяют приложению одновременно ожидать несколько потоков данных, хотя мы и не используем эту функцию в примерах ***scull***. Чтобы сделать работу корректной, необходима правильная реализация трёх вызовов: хотя о следующих правилах уже более или менее говорилось, мы просуммировали их здесь.

Чтение данных из устройства

- Если есть данные во входном буфере, вызов ***read*** должен вернуться немедленно, без каких-либо заметных задержек, даже если доступно меньше данных, чем запрошено приложением, и драйвер уверен, что оставшиеся данные прибудут в ближайшее время. Вы всегда можете вернуть меньше данных, чем запрошено, по крайней мере один байт, если это удобно по любой причине (мы делали это в ***scull***). В этом случае ***poll*** должен вернуть **POLLIN | POLLRDNORM**.
- В случае отсутствия данных во входном буфере, по умолчанию ***read*** должен блокироваться, пока нет по крайней мере одного байта. С другой стороны, если установлен **O_NONBLOCK**, ***read*** сразу же возвращается со значением **-EAGAIN** (хотя в некоторых старых версиях System V в данном случае возвращается 0). В этих случаях ***poll*** должен сообщить, что устройство не доступно для чтения, пока не будет получен по крайней мере один байт. Как только в буфере оказываются некоторые данные, мы откатываемся к предыдущему случаю.

- Если мы в конце файла, ***read*** должна немедленно вернуться с возвращаемым значением 0, независимо от **O_NONBLOCK**. ***poll*** в этом случае должен сообщить **POLLHUP**.

Запись в устройство

- Если есть место в выходном буфере, ***write*** должен вернуться без задержки. Он может принять меньше данных, чем запросил вызов, но он должен принять как минимум один байт. В этом случае ***poll*** сообщает, что устройство доступно для записи, возвращая **POLLOUT | POLLWRNORM**.
- Если выходной буфер заполнен, по умолчанию ***write*** блокируется, пока не освобождается некоторое пространство. Если установлен **O_NONBLOCK**, ***write*** немедленно возвращается со значением **-EAGAIN** (старые System V возвращали 0). В этих случаях ***poll*** должен сообщить, что файл не доступен для записи. Если, с другой стороны, устройство не может принять какие-либо дополнительные данные, ***write*** возвращает **-ENOSPC** (“No space left on device”, "Нет места на устройстве") независимо от установки **O_NONBLOCK**.
- Никогда не делайте ожидания в вызове ***write*** для ожидания передачи данных, прежде чем вернуться, даже если **O_NONBLOCK** очищен. Многие приложения используют ***select***, чтобы узнать, будет ли блокирован ***write***. Если об устройстве сообщается как о доступном для записи, вызов не должен блокироваться. Если программа, использующая устройство, стремится к тому, что данные из очереди в выходном буфере передавались на самом деле, такой драйвер должен предоставить метод ***fsync***. Например, съёмное устройство должно иметь точку входа ***fsync***.

Хотя это хороший набор общих правил, следует также признать, что каждое устройство является уникальным, и что иногда правила должны быть слегка изменены. Например, устройства, ориентированные на запись (такие как ленточные накопители), не могут выполнять частичные записи.

Сброс на диск в процессе вывода

Мы видели, как метод ***write*** сам по себе не учитывает всех потребностей передачи данных. Функция ***fsync***, вызываемая системным вызовом с тем же именем, заполняет этот пробел. Прототипом метода является

```
int (*fsync) (struct file *file, struct dentry *dentry, int datasync);
```

Если какое-либо приложение всегда должно быть уверено, что данные были отправлены в устройство, метод ***fsync*** должен выполняться независимо от того, установлен ли **O_NONBLOCK**. Вызов ***fsync*** должен вернуться только тогда, когда данные в устройство были полностью переданы (то есть, выходной буфер пуст), даже если это занимает некоторое время. Аргумент **datasync** используется, чтобы различать системные вызовы ***fsync*** и ***fdatasync***; как таковой, он представляет интерес только для кода файловой системы и может быть проигнорирован драйверами.

Метод ***fsync*** не имеет необычных особенностей. Вызов не критичен по времени, так что в каждом драйвере устройства можно реализовать его на вкус автора. В большинстве случаев символьные драйвера просто имеют указатель **NULL** в их **fops**. Блочные устройства, с другой стороны, всегда реализуют метод общего назначения ***block_fsync***, который, в свою очередь, сбрасывает все блоки устройства, ожидая завершения ввода/вывода.

Нижележащая структура данных

Фактическая реализация системных вызовов ***poll*** и ***select*** является достаточно простой для тех, кто заинтересовался, как это работает; ***epoll*** является немного более сложным, но построен на том же механизме. Всякий раз, когда пользовательское приложение вызывает ***poll***, ***select*** или ***epoll_ctl*** (* Это функция, которая создаёт внутреннюю структуру данных для будущих вызовов *epoll_wait*.), ядро вызывает метод ***poll*** всех файлов, на которые ссылается системный вызов, передавая каждому из них ту же **poll_table**. Структура **poll_table** является только обёрткой вокруг функции, которая создаёт реальную структуру данных. Для ***poll*** и ***select*** эта структура является связным списком страниц памяти, содержащих структуры **poll_table_entry**. Каждая **poll_table_entry** содержит структуру **file** и указатели **wait_queue_head_t** передаются в ***poll_wait*** наряду со связанным объектом очереди ожидания. Вызов ***poll_wait*** иногда также добавляет этот процесс к данной очереди ожидания. В целом вся структура должна обслуживаться ядром так, чтобы этот процесс мог быть удалён из всех этих очередей до возврата ***poll*** или ***select***.

Если ни один из опрошенных драйверов не показывает, что ввод/вывод может происходить без блокировки, вызов ***poll*** просто спит, пока одна из (может быть многих) очередей ожидания его не разбудит.

Интересным в реализации ***poll*** является то, что метод драйвера ***poll*** может быть вызван с указателем **NULL** в качестве аргумента **poll_table**. Эта ситуация может произойти по нескольким причинам. Если приложение, вызывающее ***poll***, передало значение времени ожидания 0 (что свидетельствует, что не должно быть ожидания), нет никаких причин, чтобы собирать очередь ожидания, и система просто ничего не делает. Указатель **poll_table** также сразу устанавливается в **NULL** после любого опроса драйвера, показывающего, что ввод/вывод возможен. Так как с этих пор ядро знает, что ожидания не будет, оно не строит список очередей ожидания.

После завершения вызова ***poll*** структура **poll_table** освобождается и все объекты очереди ожидания, ранее добавленные к таблице опроса (если таковая имеется), будут удалены из таблицы и их очередей ожидания.

Мы постарались показать на Рисунке 6-1 структуры данных, участвующие в опросе; этот рисунок является упрощённым представлением реальной структуры данных, поскольку он игнорирует многостраничный характер таблицы опроса и игнорирует файловый указатель, который является частью каждой **poll_table_entry**. Читателю, заинтересованному в фактической реализации, настоятельно рекомендуется заглянуть в ***<linux/poll.h>*** и ***fs/select.c***.

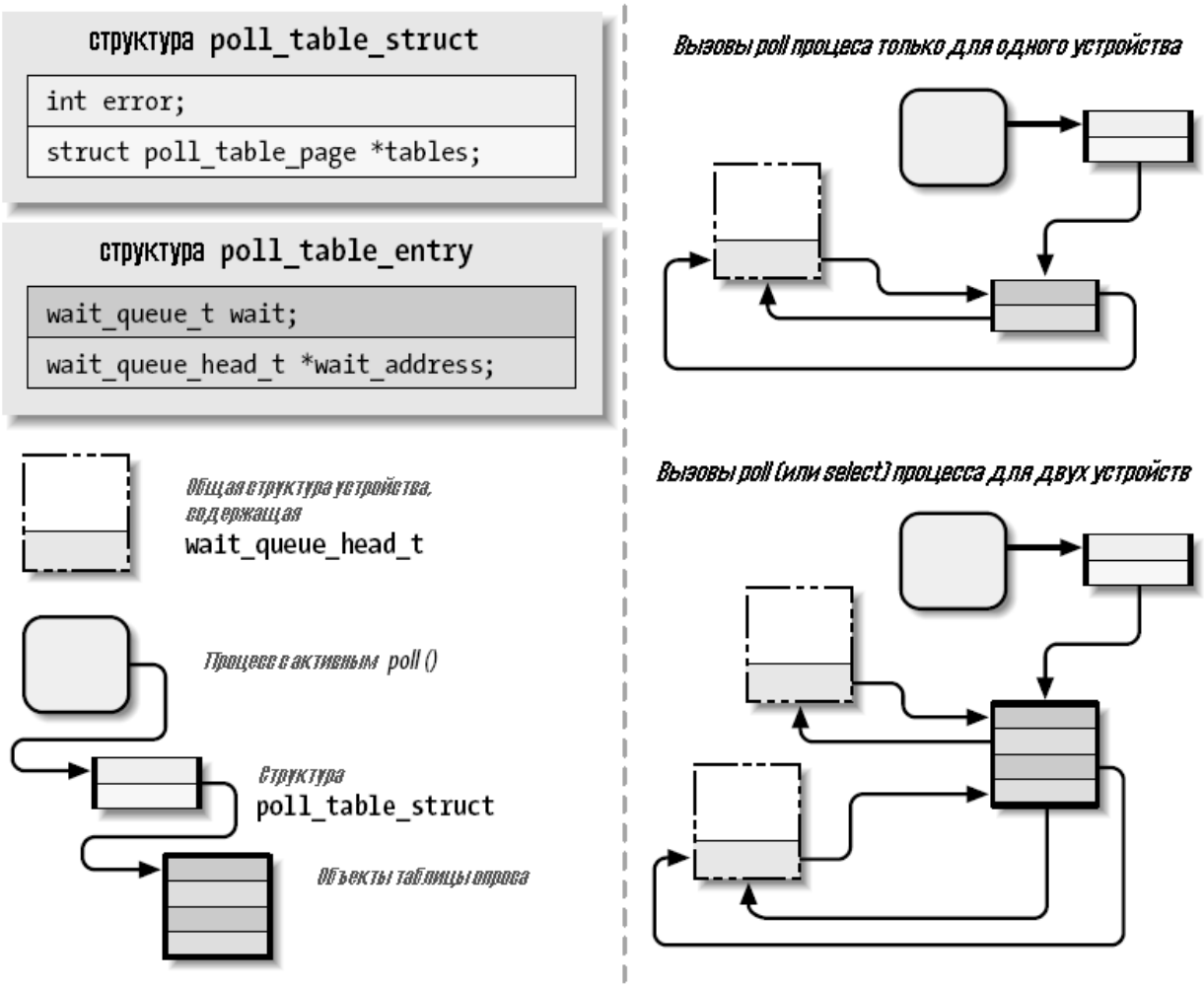


Рисунок 6-1. Структура данных, скрывающаяся за poll

На данный момент можно понять мотив для нового системного вызова ***epoll***. В типичном случае вызов ***poll*** или ***select*** включает в себя лишь небольшое количество файловых дескрипторов, так что расходы на создание структуры данных невелики. Однако, существуют приложения, которые работают с тысячами файловых дескрипторов. При этом создание и удаление этой структуры данных между каждой операцией ввода/вывода становится непомерно дорогим. Семейство системного вызова ***epoll*** позволяет приложениям такого вида создать внутреннюю структуру данных ядра только один раз и использовать её много раз.

Асинхронное сообщение

Хотя сочетание блокирующих и неблокирующих операций и метода ***select*** большую часть времени является достаточным для запросов к устройству, в некоторых ситуациях управление по методикам, которые мы видели до сих пор, не эффективно.

Давайте представим себе процесс, который выполняет продолжительный вычислительный цикл с низким приоритетом, но нуждается в обработке входящих данных так быстро, как это возможно. Если этот процесс реагирует на новые наблюдения, получаемые от какого-то периферийного устройства сбора данных, хотелось бы знать сразу, когда появляются новые данные. Это приложение может быть написано так, чтобы регулярно вызывать ***poll*** для проверки данных, однако, во многих ситуациях есть лучший путь. Разрешая асинхронное

уведомление, это приложение может получить сигнал всякий раз, когда данные становятся доступными и не должно заниматься опросом.

Пользовательским программам необходимо выполнить два шага, чтобы разрешить асинхронное уведомление от входного файла. Во-первых, они определяют процесс как "владельца" этого файла. Когда процесс вызывается командой **F_SETOWN**, используя системный вызов ***fcntl***, **process ID** владельца процесса сохраняется для последующего использования в **filp->f_owner**. Этот шаг необходим для ядра, чтобы знать, кого уведомлять. Для того, чтобы действительно разрешить асинхронное уведомление, пользовательским программам необходимо установить в устройстве флаг **FASYNC** с помощью команды **F_SETFL** ***fcntl***.

После выполнения этих двух вызовов входной файл может запросить доставку сигнала **SIGIO** при получении новой информации. Этот сигнал передаётся в процесс (или группу процессов, если значение отрицательное), хранящийся в **filp->f_owner**.

Например, следующие строки кода пользовательской программы разрешают асинхронное уведомление текущего процесса для входного файла **stdin**:

```
signal(SIGIO, &input_handler); /* пустышка; лучше использовать sigaction( )
*/
fcntl(STDIN_FILENO, F_SETOWN, getpid( ));
oflags = fcntl(STDIN_FILENO, F_GETFL);
fcntl(STDIN_FILENO, F_SETFL, oflags | FASYNC);
```

Программа в исходниках, названная ***asynctest***, является простой программой, которая, как показано, считывает **stdin**. Она может быть использована для тестирования асинхронных возможностей ***scullpipe***. Программа похожа на ***cat***, но не прекращается при достижении конца файла; она реагирует только на ввод, но не на отсутствие ввода.

Однако, следует отметить, что не все устройства поддерживают асинхронное уведомление, и вы может принять решение не предлагать его. Приложения обычно предполагают, что асинхронные возможности доступны только для сокетов и терминалов.

Осталась одна проблема с уведомлением ввода. Когда процесс получает **SIGIO**, он не знает, какой входной файл предлагает новый ввод. Если более чем один файл имеет возможность асинхронно уведомить об этом процесс, ожидающий ввод, приложение должно по-прежнему прибегать к ***poll*** или ***select***, чтобы выяснить, что произошло.

С точки зрения драйвера

Более актуальной темой для нас является то, как асинхронную сигнализацию может реализовать драйвер устройства. В следующем списке подробности последовательности операций с точки зрения ядра:

1. При вызове **F_SETOWN** ничего не происходит, кроме того, что **filp->f_owner** присваивается значение.
2. Когда выполняется **F_SETFL**, чтобы включить **FASYNC**, вызывается метод драйвера ***fasync***. Этот метод вызывается всякий раз, когда в **filp->f_flags** меняется значение **FASYNC** для уведомления драйвера об изменении, чтобы он мог реагировать должным образом. При открытии файла флаг по умолчанию очищается. Мы будем рассматривать

стандартную реализацию метода драйвера далее в этом разделе.
3. При поступлении данных все процессы, зарегистрированные для асинхронного уведомления, должны отправить сигнал **SIGIO**.

Реализация первого шага тривиальна - со стороны драйвера ничего делать не надо, другие шаги включают в себя поддержание динамической структуры данных, чтобы отслеживать разных асинхронных читателей; их может быть несколько. Эта динамическая структура данных, тем не менее, не зависит от определённого устройства и ядро предлагает подходящую реализацию общего назначения, чтобы вам не пришлось переписывать тот же самый код в каждом драйвере.

Общая реализация, предлагаемая Linux, основана на одной структуре данных и двух функциях (которые называются вторым и третьим шагами, описанными ранее). Заголовком, который декларирует соответствующий материал, является ***<linux/fs.h>*** (здесь ничего нового) и структура данных, названная **struct fasync_struct**. Как и с очередью ожидания, нам необходимо вставить указатель на структуру в зависящую от устройства структуру данных.

Две функции, которые вызывает драйвер, соответствуют следующим прототипам:

```
int fasync_helper(int fd, struct file *filp, int mode, struct fasync_struct
**fa);
void kill_fasync(struct fasync_struct **fa, int sig, int band);
```

fasync_helper вызывается, чтобы добавить или удалить записи из списка заинтересованных процессов, когда для открытого файла изменяется флаг **FASYNC**. Все эти аргументы, за исключением последнего, предоставляются методом ***fasync*** и могут быть переданы напрямую через него. ***kill_fasync*** используется, чтобы при поступлении данных просигнализировать заинтересованным процессам. Её аргументами являются сигнал для отправки (обычно **SIGIO**) и диапазон, который почти всегда **POLL_IN** (* POLL_IN является символом, используемым в коде асинхронного уведомления; это эквивалентно POLLIN | POLLRDNORM.) (но это может быть использовано для передачи "срочно" или о наличии данных во вспомогательном канале в сетевом коде). Вот как реализуется метод ***fasync*** в ***scullpipe***:

```
static int scull_p_fasync(int fd, struct file *filp, int mode)
{
    struct scull_pipe *dev = filp->private_data;

    return fasync_helper(fd, filp, mode, &dev->async_queue);
}
```

Понятно, что вся работа выполняется ***fasync_helper***. Однако, было бы невозможно реализовать функциональность без метода в драйвере, так как вспомогательной функции требуется доступ к правильному указателю на структуру **fasync_struct *** (здесь **&dev->async_queue**) и только драйвер может предоставить такую информацию.

Затем при поступлении данных, до посылки сигнала асинхронным читателям, должна быть выполнена следующая команда. Поскольку новые данные для читателя ***scullpipe*** порождаются процессом, делающим запись, эта команда выполняется в методе ***write*** драйвера ***scullpipe***.

```
if (dev->async_queue)
    kill_fasync(&dev->async_queue, SIGIO, POLL_IN);
```

Заметим, что некоторые устройства также реализуют асинхронное уведомление, чтобы сообщить, когда устройство может быть доступно для записи; конечно, в этом случае ***kill_fasync*** должна вызываться с режимом **POLL_OUT**.

Может показаться, что мы всё доделали, но всё ещё отсутствует одна вещь. Мы должны вызвать наш метод ***fasync***, когда файл закрыт для удаления этого файла из списка активных асинхронных читателей. Хотя этот вызов требуется, только если **filp->f_flags** имеет установленный **FASYNC**, вызов функции в любом случае не повредит и является обычной реализацией. Следующие строчки, например, являются частью метода ***release*** в ***scullpipe***:

```
/* удалить этот filp из асинхронно уведомляемых filp-ов */
scull_p_fasync(-1, filp, 0);
```

Структура данных, лежащая в основе асинхронного уведомления, почти идентична структуре **wait_queue**, потому что обе эти ситуации связаны с ожиданием события. Разница в том, что вместо структуры **task_struct** используется структура **file**. Затем, чтобы послать сигнал процессу, для получения **f_owner** используется структура **file** из очереди.

Произвольный доступ в устройстве

Одной из последних вещей, которые мы должны обсудить в этой главе, является метод ***llseek***, который полезен (для некоторых устройств) и легко реализуем.

Реализация llseek

Метод ***llseek*** реализует системные вызовы ***lseek*** и ***llseek***. Мы уже заявили, что если метод ***llseek*** отсутствует в операциях устройства, реализация по умолчанию в ядре выполняет доступ, изменяя **filp->f_pos**, то есть текущую позицию чтения/записи в файле. Пожалуйста, обратите внимание, что для того, чтобы системный вызов ***lseek*** работал правильно, методы ***read*** и ***write*** должны поддерживать это, используя и обновляя объект смещения, который они получают в качестве аргумента.

Вам может потребоваться предоставить собственный метод ***llseek***, если операции доступа соответствуют физической работе с устройством. Простой пример можно увидеть в драйвере ***scull***:

```
loff_t scull_llseek(struct file *filp, loff_t off, int whence)
{
    struct scull_dev *dev = filp->private_data;
    loff_t newpos;

    switch(whence) {
    case 0: /* SEEK_SET */
        newpos = off;
        break;

    case 1: /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;

    case 2: /* SEEK_END */
        newpos = dev->size + off;
```

```
        break;

    default: /* не может произойти */
        return -EINVAL;
    }

    if (newpos < 0) return -EINVAL;
    filp->f_pos = newpos;
    return newpos;
}
```

Единственной зависимой от устройства операцией здесь является получение от устройства длины файла. В ***scull*** методы ***read*** и ***write*** работают как необходимо, как показано в Главе 3 39.

Хотя только что показанная реализация имеет смысл для ***scull***, который обрабатывает хорошо определённую область данных, большинство устройств предлагают поток данных, а не область данных (просто подумайте о последовательных портах или клавиатуре), и произвольный доступ к таким устройствам не имеет смысла. Если это относится и к вашему устройству, вы не можете просто отказаться от объявления операции ***llseek***, так как метод по умолчанию разрешает произвольный доступ. Наоборот, вы должны сообщить ядру, что устройство не поддерживает ***llseek*** вызовом ***nonseekable_open*** в вашем методе ***open***:

```
int nonseekable_open(struct inode *inode; struct file *filp);
```

Этот вызов отмечает данный **filp** как не поддерживающий произвольный доступ; для такого файла ядро никогда не позволяет успешно выполнить вызов ***lseek***. Пометив файл таким образом, вы можете быть уверены, что не будут предприниматься попытки позиционироваться в файле через системные вызовы ***pread*** и ***pwrite***.

Для завершения следует также установить в вашей структуре **file_operations** вместо метода ***llseek*** специальную вспомогательную функцию ***no_llseek***, которая определена в ***<linux/fs.h>***.

Контроль доступа к файлу устройства

Предложение контроля доступа иногда жизненно важно для надежности узла устройства. Не только неавторизованным пользователям не должно разрешаться использовать устройство (ограничение обеспечивается битами разрешения файловой системы), но иногда только одному авторизованному пользователю должно быть разрешено открыть устройство в одно и то же время.

Проблема аналогична использованию терминалов. В этом случае процесс ***авторизации (login)*** изменяет владельца узла устройства, когда пользователь регистрируется в системе, с тем, чтобы запретить другим пользователям вмешиваться или подсматривать поток данных терминала. Однако, непрактично использовать привилегированную программу для изменения прав собственности на устройство каждый раз, когда оно открывается, только чтобы предоставить к нему уникальный доступ.

До сих пор не был показан ни один из кодов, реализующий любой контроль за доступом к битам разрешения файловой системы. Если системный вызов ***open*** перенаправляет запрос к драйверу, ***open*** успешен. Познакомимся теперь с несколькими техниками для реализации некоторых дополнительных проверок.

Каждое устройство, показанное в этом разделе, имеет такое же поведение как простое устройство ***scull*** (то есть, использует постоянную область памяти), но отличается от ***scull*** контролем доступа, который реализован в операциях ***open*** и ***release***.

Однократно-открываемые устройства

Способ решения в лоб обеспечивает контроль доступа разрешая открытие устройства только одному процессу в одно и то же время (однократное открытие). Этой техники лучше избегать, поскольку она препятствует изобретательности пользователя. Пользователю может понадобиться запустить различные процессы на одном устройстве, один читающий информацию о статусе, а другой для записи данных. В некоторых случаях пользователи могут многое сделать, запуская несколько простых программ через скрипт оболочки, пока они могут получать конкурентный доступ к устройству. Иными словами, реализация поведения однократного открытия сводится к созданию политики, которая может встать на пути того, что захотят сделать ваши пользователи.

Разрешение только одному процесс открывать устройство имеет нежелательные свойства, но это также простейший контроль доступа для реализации в драйвере устройства, так что это показано здесь. Исходный код взят из устройства, названного ***scullsingle***.

Устройство ***scullsingle*** содержит переменную **atomic_t**, названную **scull_s_available**; переменная инициализируется значением единицы, указывающей, что устройство действительно доступно. Вызов ***open*** уменьшает и проверяет **scull_s_available** и отказывает в доступе, если кто-то другой уже открыл устройство:

```
static atomic_t scull_s_available = ATOMIC_INIT(1);

static int scull_s_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev = &scull_s_device; /* информация устройства */

    if (! atomic_dec_and_test (&scull_s_available)) {
        atomic_inc(&scull_s_available);
        return -EBUSY; /* уже открыто */
    }

    /* затем, всё остальное скопировано из простого устройства scull */
    if ( (filp->f_flags & O_ACCMODE) == O_WRONLY)
        scull_trim(dev);
    filp->private_data = dev;
    return 0; /* успешно */
}
```

Вызов ***release***, с другой стороны, отмечает, что устройство больше не занято:

```
static int scull_s_release(struct inode *inode, struct file *filp)
{
    atomic_inc(&scull_s_available); /* освободить устройство */
    return 0;
}
```

Как правило, мы рекомендуем вам поместить флаг открытия **scull_s_available** в структуру

устройства (здесь, **Scull_Dev**), потому что концептуально это относится к устройству. Драйвер ***scull***, однако, использует отдельную переменную для хранения флага, чтобы она могла использоваться той же структурой устройства и методами, как в простом устройстве ***scull***, и минимизации дублирования кода.

Ограничение доступа: один пользователей в один момент времени

Следующий шагом после однократно открываемого устройства является предоставление одному пользователю открыть устройство в нескольких процессах, но позволить только одному пользователю иметь устройство открытым. Это решение позволяет легко проверять устройство, так как пользователь может читать и писать несколькими процессами одновременно, но предполагает, что пользователь берёт определённую ответственность за поддержание целостности данных во время нескольких доступов. Это достигается добавлением проверок в метод ***open***; такие проверки проводятся ***после*** обычной проверки разрешений и могут лишь сделать доступ более ограниченным, чем определённый битами разрешения для владельца и группы. Это та же политика доступа, которая используется для терминалов, но она не прибегает к внешней программе проверки привилегий.

Такие политики доступа немного сложнее осуществить, чем политики однократного открытия. В этом случае необходимы два объекта: счётчик открытий и uid "владельца" устройства. Ещё раз, самое лучшее место для таких объектов - в структуре устройства; наши пример использует вместо этого глобальные переменные, по причине, объяснённой ранее для ***scullsingle***. Имя этого устройства ***sculluid***.

Вызов ***open*** предоставляет доступ на первое открытие, но запоминает владельца устройства. Это означает, что пользователь может открыть устройство несколько раз, что позволяет взаимодействующим процессам работать одновременно на этом устройстве. В то же время, ни один другой пользователь не может открыть его, что позволяет избежать постороннего вмешательства. Так как эта версия функции практически идентична предыдущей, ниже приводится только соответствующая часть:

```
spin_lock(&scull_u_lock);
if (scull_u_count &&
   (scull_u_owner != current->uid) && /* разрешить пользователю */
   (scull_u_owner != current->euid) && /* разрешить, чтобы ни делал su */
   !capable(CAP_DAC_OVERRIDE)) { /* также разрешить root */
    spin_unlock(&scull_u_lock);
    return -EBUSY; /* -EPERM бы смутил пользователя */
}

if (scull_u_count == 0)
    scull_u_owner = current->uid; /* забрать его */

scull_u_count++;
spin_unlock(&scull_u_lock);
```

Обратите внимание, что код ***sculluid*** имеет две переменные (**scull_u_owner** и **scull_u_count**), которые контролируют доступ к устройству и которые могут быть доступны одновременно нескольким процессам. Чтобы сделать эти переменные безопасными, мы контролируем доступ к ним спин-блокировкой (**scull_u_lock**). Без такой блокировки два (или более процессов) могли бы проверить **scull_u_count** в один момент времени и оба могли бы

заключить, что они имеют право получить устройство в собственность. Спин-блокировка используется здесь потому, что блокировка удерживается очень короткое время и драйвер никак не может заснуть, удерживая блокировку.

Мы решили вернуть **-EBUSY**, а не **-EPERM** даже если код выполняет проверку разрешения, чтобы указать пользователю, которому отказано в доступе, правильное направление. Реакция “Permission denied” ("Доступ запрещен"), как правило, для проверки режима и владельца ***/dev*** файла, а “Device busy” ("Устройство занято") правильно предлагает, чтобы пользователь проверил, не используется ли устройство другим процессом.

Этот код также проверяет, обладает ли процесс, пытающийся открыть, способностью переопределить права доступа к файлу; если это так, то открытие разрешено, даже если открывающий процесс не является владельцем устройства. В этом случае разрешение **CAP_DAC_OVERRIDE** хорошо вписывается в задачу .

Метод ***release*** выглядит следующим образом:

```
static int scull_u_release(struct inode *inode, struct file *filp)
{
    spin_lock(&scull_u_lock);
    scull_u_count--; /* ничего другого */
    spin_unlock(&scull_u_lock);
    return 0;
}
```

И вновь мы должны получить блокировку до изменения счётчика, чтобы нам не соревноваться с другим процессом.

Блокирующее открытие как альтернатива EBUSY

Когда устройство не является доступным, возвращение ошибки обычно является наиболее разумным подходом, но есть ситуации, в которых пользователь предпочёл бы подождать устройство.

Например, если канал передачи данных используется как для передачи отчётов на регулярной, плановой основе (с использованием ***crontab***), так и для случайного использования в соответствии с потребностями людей, гораздо лучше для запланированной операции быть немного задержанной, а не ошибочной только потому, что канал в настоящее время занят.

Программист должен выбрать один из вариантов при разработке драйвера устройства и правильный ответ зависит от конкретной решаемой задачи.

Альтернативой EBUSY, как вы уже могли догадаться, является реализация блокирующего ***open***. Устройство ***scullwuid*** представляет собой версию ***sculluid***, которая ожидает устройство в ***open*** вместо возвращения -EBUSY. Она отличается от ***sculluid*** только следующей частью операции ***open***:

```
spin_lock(&scull_w_lock);
while (! scull_w_available( )) {
    spin_unlock(&scull_w_lock);
    if (filp->f_flags & O_NONBLOCK) return -EAGAIN;
    if (wait_event_interruptible (scull_w_wait, scull_w_available( )))
```

```
        return -ERESTARTSYS; /* вернуть слою файловой системы для обработки
*/
    spin_lock(&scull_w_lock);
}
if (scull_w_count == 0)
    scull_w_owner = current->uid; /* забрать его */
scull_w_count++;
spin_unlock(&scull_w_lock);
```

Реализация снова основана на очереди ожидания. Если устройство не является в настоящее время доступным, процесс, пытающийся открыть файл, помещается в очередь ожидания, пока владеющий процесс закрывает устройство.

Затем за пробуждение любого ожидающего процесса отвечает метод ***release***:

```
static int scull_w_release(struct inode *inode, struct file *filp)
{
    int temp;

    spin_lock(&scull_w_lock);
    scull_w_count--;
    temp = scull_w_count;
    spin_unlock(&scull_w_lock);
    if (temp == 0)
        wake_up_interruptible_sync(&scull_w_wait); /* пробудить другие uid-ы
*/
    return 0;
}
```

Вот пример того, где вызов ***wake_up_interruptible_sync*** имеет смысл. Когда мы выполняем пробуждение, мы просто собираемся вернуться в пользовательское пространство, которое является местом работы планировщика задач системы. Вместо того, чтобы потенциально иметь возможность переключения планировщика задач, когда мы выполняем пробуждение, лучше просто вызвать "синхронную" версию и закончить свою работу.

Проблемой при реализации блокирующего открытия является то, что это действительно неприятно для интерактивного пользователя, который должен гадать, что идёт не так. Интерактивный пользователь обычно вызывает стандартные команды, такие как ***cp*** и ***tar*** и не может просто добавить **O_NONBLOCK** в вызов ***open***. Тот, кто делает резервную копию с помощью ленточного накопителя в другой комнате предпочёл бы получить простое сообщение "устройство или ресурс заняты" вместо того, чтобы гадать, почему жёсткий диск так тих сегодня, в то время как ***tar*** должна сканировать его. Такую проблему (необходимость в различных, несовместимых политиках для одного устройства) часто лучше всего решать реализацией отдельного узла устройства для каждой политики доступа. Пример такой практики может быть найден в драйвере ленточного накопителя Linux, который предусматривает несколько файлов устройств для одного устройства. Разные файлы устройства, например, для записи диска с или без сжатия, или для автоматической перемотки ленты при закрытии устройства.

Клонирование устройства при открытии

Другой техникой управления контролем доступа является создание разных частных копий устройства, в зависимости от процесса, открывающего его.

Очевидно, что это возможно только, если устройство не связано с аппаратным объектом; ***scull*** является примером такого "программного" устройства. Внутренности ***/dev/tty*** используют аналогичную технику для того, чтобы дать своему процессу другой "вид", который представлен входной точкой в ***/dev***. Когда копии устройства создаются программным драйвером, мы называем их виртуальными устройствами - как виртуальные консоли используют одно физическое устройство терминала.

Хотя этот вид контроля доступа необходим редко, его реализация может разъяснить, показывая, как легко код ядра может изменить перспективу приложения из окружающего мира (то есть компьютера).

Виртуальные устройства в пределах пакета ***scull*** реализует узел устройства ***/dev/scullpriv***. Реализация ***scullpriv*** использует номер процесса, контролирующего терминал, качестве ключа для доступа к виртуальному устройству. Тем не менее, вы можете легко изменить исходник для использования для ключа любого целого значения; каждый выбор ведёт к другой политике. Например, использование uid приводит к разным виртуальным устройством для каждого пользователя, при использовании pid ключ создаёт новое устройство для каждого получающего к нему доступ процесса.

Решение об использовании управляющего терминала призвано позволить лёгкое тестирование устройство с помощью перенаправления ввода/вывода: устройство является общим для всех команд, работающих на том же виртуальном терминале, и сохраняется отдельным для видения командами, работающими на другом терминале.

Метод ***open*** выглядит подобно следующему коду. Он должен искать правильное виртуальное устройство и, возможно, создать новое. Заключительная часть функции не показывается, потому что скопирована из обычного ***scull***, который мы уже видели.

```
/* зависящая от клона структура данных, включающая поле key */

struct scull_listitem {
    struct scull_dev device;
    dev_t key;
    struct list_head list;
};

/* список устройств и блокировка для его защиты */
static LIST_HEAD(scull_c_list);
static spinlock_t scull_c_lock = SPIN_LOCK_UNLOCKED;

/* Поиск устройства или создание нового, если его нет */
static struct scull_dev *scull_c_lookfor_device(dev_t key)
{
    struct scull_listitem *lptr;

    list_for_each_entry(lptr, &scull_c_list, list) {
        if (lptr->key == key)
            return &(lptr->device);
    }

    /* не найдено */
    lptr = kmalloc(sizeof(struct scull_listitem), GFP_KERNEL);
```

```
    if (!lptr)
        return NULL;

    /* инициализация устройства */
    memset(lptr, 0, sizeof(struct scull_listitem));
    lptr->key = key;
    scull_trim(&(lptr->device)); /* проинициализировать его */
    init_MUTEX(&(lptr->device.sem));

    /* поместить его в список */
    list_add(&lptr->list, &scull_c_list);
    return &(lptr->device);
}

static int scull_c_open(struct inode *inode, struct file *filp)
{
    struct scull_dev *dev;
    dev_t key;

    if (!current->signal->tty) {
        PDEBUG("Process \"%s\" has no ctl tty\n", current->comm);
        return -EINVAL;
    }
    key = tty_devnum(current->signal->tty);

    /* поискать устройство scullc в списке */
    spin_lock(&scull_c_lock);
    dev = scull_c_lookfor_device(key);
    spin_unlock(&scull_c_lock);

    if (!dev)
        return -ENOMEM;

    /* затем, всё остальное скопировано из обычного устройства scull */
```

Метод ***release*** не делает ничего особенного. Это, как правило, освобождение устройства при последнем закрытии, но мы решили не поддерживать счётчик открытия, чтобы упростить тестирование драйвера. Если бы устройство было освобождено при последнем закрытии, вы не смогли бы прочитать те же данные после записи в устройство, если бы фоновый процесс не держал его открытым. Пример драйвера предлагает более простой подход хранения данных, так что вы найдёте его там в следующем ***open***. Устройства освобождается при вызове ***scull_cleanup***.

Этот код использует общий механизм связных списков Linux, предпочитая не реализовывать те же возможности с нуля. Списки Linux обсуждаются в Главе 11 275.

Вот реализация ***release*** для ***/dev/scullpriv***, которая закрывает обсуждение методов устройства.

```
static int scull_c_release(struct inode *inode, struct file *filp)
{
    /*
     * ничего не делать, потому что устройство является стойким.
```

```
     * 'настоящее' клонированное устройство должно быть освобождено при
последнем закрытии.
     */
    return 0;
}
```

Краткая справка

Эта глава представляет следующие символы и заголовочные файлы:

#include <linux/ioctl.h>
Объявляет все макросы, которые используются для определения команд ***ioctl***. В настоящее время он подключается с помощью ***<linux/fs.h>***.

_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS
Число бит, доступное для различных битовых полей команд ***ioctl***. Есть также ещё четыре макроса, которые определяют **MASK**и и четыре, определяющие **SHIFT**ы, но они в основном для внутреннего пользования. **_IOC_SIZEBITS** является важным значением для проверки, так как оно меняется в зависимости от архитектуры.

_IOC_NONE
_IOC_READ
_IOC_WRITE
Возможные значения битового поля "направление". "Чтение" и "Запись" являются разными битами и могут быть сложены командой OR (ИЛИ) для указания чтения/записи. Значения базируются на 0.

_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(type,nr,size)
_IOWR(type,nr,size)
Макросы, используемые для создания команд ***ioctl***.

_IOC_DIR(nr)
_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)
Макросы, которые используются для декодирования команд. В частности, **_IOC_TYPE (nr)** является комбинацией по ИЛИ для **_IOC_READ** и **_IOC_WRITE**.

#include <asm/uaccess.h>
int access_ok(int type, const void *addr, unsigned long size);
Проверяет, что указатель в пользовательском пространстве является верным. ***access_ok*** возвращает ненулевое значение, если доступ будет разрешён.

VERIFY_READ
VERIFY_WRITE
Возможные значения аргумента **type** в ***access_ok***. **VERIFY_WRITE** является надстройкой **VERIFY_READ**.

#include <asm/uaccess.h>
int put_user(datum,ptr);

int get_user(local,ptr);
int __put_user(datum,ptr);
int __get_user(local,ptr);
Макросы, которые используются для хранения и получения данных в или из пространства пользователя. Передаваемое количество байт зависит от **sizeof(*ptr)**. Обычные версии сначала вызывают ***access_ok***, в то время как специальные версии (***__put_user*** и ***__get_user***) предполагают, что ***access_ok*** уже вызывалась

#include <linux/capability.h>
Определяет различные символы **CAP_**, описывающие какие возможности доступа может иметь процесс пользовательского пространства.
int capable(int capability);
Возвращает ненулевое значение, если этот процесс имеет данное разрешение.

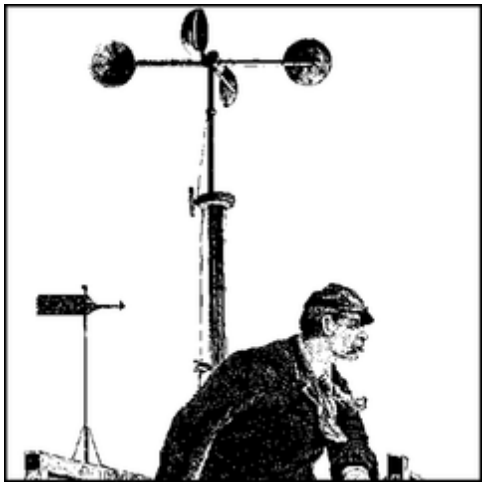
#include <linux/wait.h>
typedef struct { /* ... */ } wait_queue_head_t;
void init_waitqueue_head(wait_queue_head_t *queue);
DECLARE_WAIT_QUEUE_HEAD(queue);
Определяет тип для очереди ожидания Linux. ***wait_queue_head_t*** должен быть явно проинициализирован либо ***init_waitqueue_head*** во время выполнения, либо **DECLARE_WAIT_QUEUE_HEAD** во время компиляции.
void wait_event(wait_queue_head_t q, int condition);
int wait_event_interruptible(wait_queue_head_t q, int condition);
int wait_event_timeout(wait_queue_head_t q, int condition, int time);
int wait_event_interruptible_timeout(wait_queue_head_t q, int condition, int time);
Помещает процесс в сон в данной очереди, пока данное **condition** (условие) оценивается как истинное значение.
void wake_up(struct wait_queue *q);
void wake_up_interruptible(struct wait_queue *q);
void wake_up_nr(struct wait_queue *q, int nr);
void wake_up_interruptible_nr(struct wait_queue *q, int nr);
void wake_up_all(struct wait_queue *q);
void wake_up_interruptible_all(struct wait_queue *q);
void wake_up_interruptible_sync(struct wait_queue *q);
Пробуждает процессы, которые являются спящими в очереди **q**. Форма ***_interruptible*** пробуждает только прерываемые процессы. Как правило, пробуждается только один ожидающий эксклюзивно, но это поведение можно изменить с помощью форм ***_nr*** или ***_all***. Версия ***_sync*** не переключает процессор перед возвратом.

#include <linux/sched.h>
set_current_state(int state);
Устанавливает состояние выполнения для текущего процесса. **TASK_RUNNING** означает, что он готов к запуску, а состояниями сна являются **TASK_INTERRUPTIBLE** и **TASK_UNINTERRUPTIBLE**.
void schedule(void);
Выбирает работающий процесс из очереди выполнения. Выбранный процесс может быть **current (текущим)** или любым другим.
typedef struct { /* ... */ } wait_queue_t;
init_waitqueue_entry(wait_queue_t *entry, struct task_struct *task);
Тип ***wait_queue_t*** используется для помещения процесса в очередь ожидания.

void prepare_to_wait(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void prepare_to_wait_exclusive(wait_queue_head_t *queue, wait_queue_t *wait, int state);
void finish_wait(wait_queue_head_t *queue, wait_queue_t *wait);
Вспомогательные функции, которые могут быть использованы в коде для ручного управления сном.
void sleep_on(wiat_queue_head_t *queue);
void interruptible_sleep_on(wiat_queue_head_t *queue);
Устаревшие и осуждаемые для использования функции, которые безоговорочно помещают текущий процесс в сон.

#include <linux/poll.h>
void poll_wait(struct file *filp, wait_queue_head_t *q, poll_table *p)
Помещает текущий процесс в очередь ожидания сразу, без планировщика. Предназначена для использования методом ***poll*** драйверами устройств.
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);
"Помощник" для реализации метода устройства ***fasync***. Аргумент **mode** является тем же значением, которое передаётся в метод, а **fa** указывает на зависимую от устройства **fasync_struct ***.
void kill_fasync(struct fasync_struct *fa, int sig, int band);
Если драйвер поддерживает асинхронное уведомление, эта функция может быть использована для отправки сигнала процессам, зарегистрированным в **fa**.
int nonseekable_open(struct inode *inode, struct file *filp);
loff_t no_llseek(struct file *file, loff_t offset, int whence);
nonseekable_open должна быть вызвана в методе ***open*** любого устройства, которое не поддерживает произвольный доступ. Такие устройства должны также использовать ***no_llseek*** как свой метод ***llseek***.

Глава 7, Время, задержки и отложенная работа



На данный момент мы знакомы с основами написания полнофункционального символьного модуля. Однако, драйверам реального мира необходимо делать больше, чем осуществлять операции, которые управляют устройством; они должны решать такие вопросы, как синхронизация, управление памятью, доступ к оборудованию и многое другое. К счастью, для облегчения задачи написания драйвера ядро экспортирует много средств. В следующих нескольких главах мы опишем некоторые ресурсы ядра, которые можно использовать. В этой главе описывается решение проблем синхронизации. Работа со временем включает в себя следующие задачи, в порядке усложнения:

- Измерение временных интервалов и сравнение времени;
- Получение текущего времени;
- Задержка операции на указанное время;
- Планирование асинхронных функций для выполнения в более позднее время;

Измерение временных промежутков

Ядро следит за течением времени с помощью таймера прерываний. Прерывания подробно описаны в Главе 10 246.

Прерывания таймера генерируются через постоянные интервалы системным аппаратным таймером; этот интервал программируется во время загрузки ядром в соответствии со значением **HZ**, которое является архитектурно-зависимой величиной, определённой в ***<linux/param.h>*** или файле подплатформы, подключаемом им. Значения по умолчанию в распространяемых исходных текстах ядра имеют диапазон от 50 до 1200 тиков в секунду на реальном оборудовании, снижаясь до 24 в программных эмуляторах. Большинство платформ работают на 100 или 1000 прерываний в секунду; значением по умолчанию для популярных ПК x86 является 1000, хотя в предыдущих версиях (вплоть до 2.4) оно было 100. По общему правилу, даже если вы знаете значение **HZ**, никогда не следует рассчитывать на определённое значение при программировании.

Те, кто хотят систему с другой частотой прерываний, могут изменить значение **HZ**. Если вы изменяете **HZ** в заголовочном файле, вам необходимо перекомпилировать ядро и все модули с новым значением. Вы можете захотеть увеличить **HZ** для получения более высокого

разрешения в асинхронных задачах, если вы готовы платить накладные расходы от дополнительных прерываний таймера для достижения ваших целей. Действительно, повышение **HZ** до 1000 было довольно распространено для промышленных систем x86, использующих ядро версии 2.4 или 2.2. Однако, для текущих версий лучшим подходом к прерыванию таймера является сохранение значения по умолчанию для **HZ**, в силу нашего полного доверия разработчикам ядра, которые, несомненно, выбрали лучшее значение. Кроме того, некоторые внутренние расчёты в настоящее время осуществляются только для **HZ** в диапазоне от 12 до 1535 (смотрите ***<linux/timex.h>*** и RFC-1589).

Значение внутреннего счётчика ядра увеличивается каждый раз, когда происходит прерывание от таймера. Счётчик инициализируется 0 при загрузке системы, поэтому он представляет собой число тиков системных часов после последней загрузки. Счётчик является 64-х разрядной переменной (даже на 32-х разрядных архитектурах) и называется **jiffies_64**. Однако, авторы драйверов обычно используют переменную **jiffies** типа **unsigned long**, которая то же самое, что и **jiffies_64** или её младшие биты. Использование **jiffies**, как правило, предпочтительнее, поскольку это быстрее, и доступ к 64-х разрядному значению **jiffies_64** не обязательно является атомарным на всех архитектурах.

В дополнение к управляемому ядром механизму тиков с низкой разрешающей способностью, некоторые процессорные платформы имеют счётчик высокого разрешения, который программное обеспечение может читать. Хотя фактическое использование несколько различается на разных платформах, иногда это очень мощный инструмент.

Использование счётчика тиков

Счётчик и специальные функции для его чтения живут в ***<linux/jiffies.h>***, хотя вы обычно будете просто подключать ***<linux/sched.h>***, который автоматически подключает ***jiffies.h***. Излишне говорить, что ***jiffies*** и ***jiffies_64*** должны рассматриваться как только читаемые.

Всякий раз, когда ваш код должен запомнить текущее значение **jiffies**, он может просто обратится к переменной **unsigned long**, которая объявлена как **volatile** (нестабильная), чтобы компилятор не оптимизировал чтения памяти. Вам необходимо прочитать текущий счётчик, когда вашему коду необходимо рассчитать будущий момент времени, как показано в следующем примере:

```
#include <linux/jiffies.h>
unsigned long j, stamp_1, stamp_half, stamp_n;

j = jiffies; /* читаем текущее значение */
stamp_1 = j + HZ; /* позже на 1 секунду */
stamp_half = j + HZ/2; /* пол-секунды */
stamp_n = j + n * HZ / 1000; /* n миллисекунд */
```

Этот код не имеет никаких проблем с переполнением **jiffies** до тех пор, пока различные значения сравниваются правильным способом. Хотя на 32-х разрядных платформах счётчик переполняется только один раз в 50 дней при **HZ** равном 1000, ваш код должен быть подготовлен к встрече этого события. Для сравнения вашего закэшированного значения (например, вышеприведённого **stamp_1**) и текущего значения, вы должны использовать один из следующих макросов:

```
#include <linux/jiffies.h>
int time_after(unsigned long a, unsigned long b);
```

```
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
```

Первый возвращает истину, когда ***a***, как копия **jiffies**, представляет собой время после ***b***, второй возвращает истину, когда время ***a*** перед временем ***b***, а последние два сравнивают как "позже или равно" и "до или равно". Код работает преобразуя значения в **signed long**, вычитая их и проверяя результат. Если вам необходимо узнать разницу между двумя значениями **jiffies** безопасным способом, можно использовать тот же прием: diff = (long)t2 - (long)t1;.

Можно конвертировать разницу значений **jiffies** в миллисекунды простым способом:

```
msec = diff * 1000 / HZ;
```

Иногда, однако, необходимо обмениваться представлением времени с программами пользовательского пространства, которые, как правило, предоставляют значения времени структурами **timeval** и **timespec**. Эти две структуры предоставляют точное значение времени двумя числами: секунды и микросекунды используются в старой и популярной структуре **timeval**, а в новой структуре **timespec** используются секунды и наносекунды. Ядро экспортирует четыре вспомогательные функции для преобразования значений времени в **jiffies** в и из этих структур:

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

unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffies, struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies, struct timeval *value);
```

Доступ к 64-х разрядному счётчику тиков не так прост, как доступ к **jiffies**. В то время, как на 64-х разрядных архитектурах эти две переменные являются фактически одной, доступ к значению для 32-х разрядных процессоров не атомарен. Это означает, что вы можете прочитать неправильное значение, если обе половинки переменной обновляются, пока вы читаете их. Вряд ли вам когда-нибудь понадобится прочитать 64-х разрядный счётчик, но в этом случае вы будете рады узнать, что ядро экспортирует специальную вспомогательную функцию, которая делает для вас правильное блокирование:

```
#include <linux/jiffies.h>
u64 get_jiffies_64(void);
```

В приведённом выше прототипе используется тип **u64**. Это один из типов, определённых в ***<linux/types.h>***, он обсуждается в Главе 11 275 и представляет собой беззнаковый 64-х разрядный тип.

Если вам интересно, как 32-х разрядные платформы обновляют в одно и то же время 32-х разрядный и 64-х разрядный счётчик, почитайте скрипт компоновщика для вашей платформы (найдите файл, имя которого соответствует **vmlinux*.lds***). Там символ **jiffies** определён для доступа к младшему слову 64-х разрядного значения, в зависимости от платформы используется прямой или обратный порядок битов (little-endian или big-endian). Собственно, тот же приём используется для 64-х разрядных платформ, так что переменные **unsigned long** и **u64** доступны по одному адресу.

Наконец, отметим, что фактическая тактовая частота почти полностью скрыта от пользовательского пространства. Макрос **HZ** всегда преобразуется в 100, когда программы пользовательского пространства подключают ***param.h***, и каждый счётчик, передаваемый в пользовательское пространство, преобразуется соответственно. Этот применимо к ***clock(3)***, ***times(2)*** и любой соответствующей функции. Единственным доказательством для пользователя значения **HZ** является то, насколько быстро происходят прерывания таймера, это показывается в ***/proc/interrupts***. Например, вы можете получить **HZ** путём деления этого счётчика на время работы системы, сообщаемое ***/proc/uptime***.

Процессорно-зависимые регистры

Если необходимо измерять очень короткие промежутки времени или требуется исключительно высокая точность в цифрах, можно прибегнуть к платформо-зависимым ресурсам, выбрав точность взамен переносимости.

В современных процессорах острому спросу на эмпирические цифры производительности мешает собственная непредсказуемость времени выполнения в большинстве типов процессоров из-за кэш памяти, "планирования" инструкций и предсказания переходов. В ответ производители процессоров предоставили способ расчёта тактов, как простой и надёжный способ измерения временных интервалов. Таким образом, большинство современных процессоров имеет регистр счётчика, который постоянно увеличивается один раз в каждом такте. В настоящее время этот счётчик времени является единственным надёжным способом для выполнения задач, требующих хронометража с высоким разрешением.

Детали отличаются от платформы к платформе: регистр может быть или не быть читаемым из пространства пользователя, он может или не может быть доступен для записи и он может быть 64 или 32 бита. В последнем случае, вы должны быть готовы к обработке переполнения так же, как мы это делали со счётчиком тиков. Регистр может даже не существовать для вашей платформы, или он может быть реализован во внешнем устройстве разработчиком аппаратуры, если процессор не имеет этой функции и вы имеете дело с компьютером специального назначения.

Независимо от возможности обнуления этого регистра, мы настоятельно не рекомендуем его обнулять, даже когда аппаратура позволяет это. Вы, в конце концов, можете не быть в любой момент времени единственным пользователем счётчика; на некоторых платформах, поддерживающих SMP, например, ядро зависит от такого счётчика для синхронизации между процессорами. Поскольку вы всегда можете измерить разницу между значениями, пока эта разница не превышает время переполнения, вы можете выполнить работу, не претендуя на исключительное право собственности регистром, изменяя его текущее значение.

Наиболее известным регистром счётчика является TSC (timestamp counter, счётчик временных меток), введённый в x86 процессоры, начиная с Pentium и с тех пор присутствует во всех конструкциях процессора, включая платформу x86_64. Это 64-х разрядный регистр, который считает тактовые циклы процессора; он может быть прочитан и из пространства ядра и из пользовательского пространства.

После подключения ***<asm/msr.h>*** (заголовок для x86, имя которого означает “machine-specific registers”, "машинно-зависимые регистры"), вы можете использовать один из этих макросов:

```
rdtsc(low32,high32);
rdtscl(low32);
```

```
rdtscll(var64);
```

Первый макрос атомарно читает 64-х разрядное значение в две 32-х разрядные переменные; следующий макрос (“read low half”, "чтение младшей половины") читает младшую половину регистра в 32-х разрядную переменную, отбрасывая старшую половину; последний читает 64-х разрядное значение в переменную **long long**, отсюда и имя. Все эти макросы хранят значения в своих аргументах.

Чтение младшей половины счётчика достаточно для наиболее распространённых применений TSC. Процессор 1 ГГц переполняет его только каждые 4.2 секунды, так что вам не придётся иметь дело с многорегистровыми переменными, если промежуток времени, который вы измеряете, гарантированно занимает меньше времени. Однако, частоты процессоров растут с течением времени, так же как и увеличиваются требования к измерению времени, скорее всего, в будущем всё чаще будет необходимо читать 64-х разрядный счётчик.

В качестве примера, следующие строки непосредственно измеряют время выполнения инструкции используя только младшую половину регистра:

```
unsigned long ini, end;
rdtscl(ini); rdtscl(end);
printk("time lapse: %li\n", end - ini);
```

Некоторые другие платформы предлагают аналогичную функциональную возможность и заголовки ядра предлагают архитектурно-независимую функцию, которую можно использовать вместо ***rdtsc***. Она называется ***get_cycles***, определена в ***<asm/timex.h>*** (подключаемого с помощью ***<linux/timex.h>***). Её прототипом является:

```
#include <linux/timex.h>
cycles_t get_cycles(void);
```

Эта функция определена для каждой платформы и она всегда возвращает 0 на платформах, которые не имеют регистра счётчика циклов. Тип **cycles_t** является соответствующим беззнаковым типом для хранения считанного значения.

Несмотря на наличие архитектурно-независимой функции, мы хотели бы получить благоприятную возможность показать пример встраивания ассемблерного кода. С этой целью мы реализуем функцию ***rdtscl*** для процессоров MIPS, которая работает так же, как и аналогичная для x86.

Мы основываем этот пример на MIPS, потому что большинство процессоров MIPS имеют 32-х разрядный счётчик как регистр 9 их внутреннего "сопроцессора 0". Для доступа к этому регистру, читаемого только из пространства ядра, вы можете определить следующий макрос, который выполняет ассемблерную инструкцию “move from coprocessor 0” ("переместить от сопроцессора 0"): (* завершающая инструкция nop необходима для защиты, чтобы компилятор не обратился к целевому регистру в инструкции сразу же после *mfc0*. Этот тип блокировки является типичным для процессоров RISC и компилятор всё ещё может наметить полезные инструкции в слотах задержки. В этом случае мы используем nop, потому что встраиваемый ассемблер является чёрным ящиком для компилятора и может быть выполнен без оптимизации.)

```
#define rdtscl(dest) \
__asm__ __volatile__("mfc0 %0,$9; nop" : "=r" (dest))
```

Вместе с этим макросом MIPS процессор может выполнять тот же код, показанный ранее для x86.

Во встраиваемом ассемблере ***gcc*** распределение регистров общего назначения остаётся компилятору. Макрос просто указывает использовать 0% для размещения "аргумента 0", который позднее указан как "любой регистр (r), используемый в качестве выходного (=)". Этот макрос также заявляет, что выходной регистр должен соответствовать выражению **dest** языка Си. Синтаксис для встраиваемого ассемблера является очень мощным, но довольно сложным, особенно для архитектур, имеющих ограничения на то, что может делать каждый регистр (в частности, семейство x86). Синтаксис описан в документации ***gcc***, обычно предоставляемой в дереве документации ***info***.

Короткий фрагмент кода Си, показанный в этом разделе, был запущен на x86-процессоре класса K7 и MIPS VR4181 (с помощью только что описанного макроса). Первый сообщил о промежутке времени в 11 тактов, а второй только о 2 тактах. Небольшая цифра была ожидаема, поскольку обычно RISC процессоры выполняют одну инструкцию за такт.

Существует ещё одна вещь, которую стоит знать о счётчиках временных меток: они не обязательно синхронизированы между процессорами в многопроцессорных системах. Чтобы быть уверенными в получении согласованного значения, вы должны запретить прерывание для кода, который запрашивает счётчик.

Определение текущего времени

Код ядра всегда может получить представление о текущем время, глядя на значение **jiffies**. Обычно, тот факт, что значение предоставляет только время, прошедшее с последней загрузки, не влияет на драйвер, потому что его жизнь ограничена временем работы системы. Как было показано, драйверы могут использовать текущее значение **jiffies** для расчёта интервалов времени между событиями (например, чтобы отличить двойной клик от одинарного в драйвере устройства ввода или вычисления времени ожидания). Одним словом, значения **jiffies** почти всегда достаточно, когда вам требуется измерять временные интервалы. Если требуются очень точные измерения для короткого промежутка времени, на помощь приходят процессорно-зависимые регистры (хотя они приносят серьёзные вопросы переносимости).

Вполне вероятно, что драйвер никогда не должен знать времени стенных часов, выраженного в месяцах, днях и часах; эта информация, как правило, необходима только пользовательским программам, таким, как ***cron*** и ***syslogd***. Работу с реальным временем обычно лучше оставить пользовательскому пространству, где библиотека Си предлагает лучшую поддержку, кроме того, такой код часто слишком связан с политикой, чтобы принадлежать ядру. Однако, существует функция ядра, которая превращает время часов в значение **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);
```

Повторим: непосредственная работа со временем стенных часов в драйвере часто признак того, что реализована политика, и должна быть поставлена под сомнение.

Хотя вы не будете иметь дело с человеко-читаемым представлением времени, иногда

необходимо иметь дело с абсолютным временем даже в пространстве ядра. Для этого цели ***<linux/time.h>*** экспортирует функцию ***do_gettimeofday***. При вызове она заполняет структуру **timeval** по указателю, который используется в системном вызове ***gettimeofday***, знакомыми значениями секунд и микросекунд. Прототипом ***do_gettimeofday*** является:

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

Исходник утверждает, что ***do_gettimeofday*** имеет "разрешение около микросекунды", поскольку она запрашивает аппаратные средства синхронизации, какая часть текущего тика уже прошла. Однако, точность варьируется от одной архитектуры к другой, поскольку она зависит от реального используемого аппаратного механизма. Например, некоторые ***m68knommu*** процессоры, системы ***Sun3*** и других системы ***m68k*** не могут предложить больше, чем разрешение тика. Системы ***Pentium***, с другой стороны, предлагают очень быстрые и точные измерения под-тиков через чтение счётчика тактов, описанное ранее в этой главе.

Текущее время также доступно (хотя и с точностью до тика) от переменной **xtime**, переменной из структуры **timespec**. Прямое использование этой переменной не рекомендуется, поскольку трудно получить атомарный доступ к обоим полям. Таким образом, ядро предоставляет вспомогательную функцию ***current_kernel_time***:

```
#include <linux/time.h>
struct timespec current_kernel_time(void);
```

Код для извлечения текущего времени разными способами можно получить из модуля ***jit*** (“just in time”, "точно в срок") в файлах исходных тектов на FTP сайте O'Reilly. ***jit*** создаёт файл, названный ***/proc/currentime***, который возвращает при чтении следующие значения в ASCII:

- Текущие значения **jiffies** и **jiffies_64**, как шестнадцатеричные числа;
- Текущее время, возвращаемое ***do_gettimeofday***;
- **timespec**, возвращаемое ***current_kernel_time***;

Мы решили использовать динамический файл ***/proc*** для сохранения шаблонного кода минимальным - не стоит создавать целое устройство только для возвращения небольшой текстовой информации.

Этот файл возвращает строки текста непрерывно, пока модуль загружен; каждый системный вызов ***read*** собирает и возвращает один набор данных, организованный для лучшей читаемости в две строки. Всякий раз, когда вы читаете несколько наборов данных менее чем за тик таймера, вы увидите разница между ***do_gettimeofday***, которая опрашивает оборудование, а другие значения обновляются только по тикам таймера.

```
phon% head -8 /proc/currentime
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630126
                              1062370899.629161488
0x00bdbc1f 0x0000000100bdbc1f 1062370899.630150
                              1062370899.629161488
0x00bdbc20 0x0000000100bdbc20 1062370899.630208
                              1062370899.630161336
0x00bdbc20 0x0000000100bdbc20 1062370899.630233
                              1062370899.630161336
```

На вышеприведённом снимке экрана обратите внимание на две интересные вещи. Во-первых, значение ***current_kernel_time***, хотя и выражено в наносекундах, имеет точность только до тика; ***do_gettimeofday*** постоянно сообщает позднее время, но не позднее чем следующий тик таймера. Во-вторых, 64-х разрядный счётчик тиков имеет установленным наименее значащий бит старшего 32-х битного слова. Это происходит потому, что значение по умолчанию для **INITIAL_JIFFIES**, используемое во время загрузки для инициализации счётчика, вызывает переполнение младшего слова через несколько минут после времени загрузки, чтобы помочь обнаружить проблемы, связанные с этим переполнением. Этот начальное смещение в счётчике не имеет никакого эффекта, поскольку **jiffies** не имеет отношения ко времени настенных часов. В ***/proc/uptime***, когда ядро получает из счётчика время работы, перед преобразованием начальное смещение удаляется.

Отложенный запуск

Драйверам устройств часто необходимо отложить выполнение определённого фрагмента кода на какой-то период времени, обычно, чтобы позволить аппаратуре выполнить какую-то задачу. В этом разделе мы рассмотрим различные методы получения задержек. Обстоятельства каждой ситуации определяют, какую технику использовать лучше всего; пройдёмся по всем и укажем преимущества и недостатки каждой из них.

Важно рассмотреть, как требуемая задержка сравнивается с тактовыми тиками, учитывая разницу **HZ** на различных платформах. Задержки, которые гарантировано больше, чем такт тика, и не страдают от его грубой точности, могут воспользоваться системными часами. Очень короткие задержки обычно должны быть реализованы программными циклами. Между этими двумя случаями лежит переходная зона. В этой главе мы используем фразу "длинная" задержка для обозначения многотиковой задержки, которая может быть на некоторых платформах так же мала, как и несколько миллисекунд, но всё же длинной по мнению процессора и ядра.

В следующих разделах рассказывается о различных задержки, используя довольно долгий путь от различных интуитивных, но неуместных решений, к правильному решению. Мы выбрали этот путь, потому что он позволяет провести более углублённое обсуждение вопросов, связанных со временем в ядре. Если вам не терпится найти правильный код, просто просмотрите этот раздел.

Длинные задержки

Иногда драйверу необходимо отложить исполнение на относительно длительные периоды - больше, чем один тактовый тик. Есть несколько способов выполнить такого рода задержки; мы начнём с самой простой техники, затем перейдём к более совершенным техникам.

Ожидание в состоянии занятости

Если вы хотите отложить исполнение на много тактовых тиков, допуская некоторый люфт в значении, самой простой (но не рекомендуемой) реализацией является цикл, который мониторит счётчик тиков. Реализация ожидания в состоянии занятости (busy-waiting) обычно выглядит как следующий код, где **j1** является значением **jiffies** по истечении задержки:

```
while (time_before(jiffies, j1))
cpu_relax( );
```

Вызов ***cpu_relax*** является архитектурно-зависимым способом сообщить, что вы ничего не делаете с процессором на данный момент. Во многих системах он совсем ничего не делает; на симметричных многопоточных ("hyperthreaded") системах он может переключить ядро на другой поток. В любом случае, когда возможно, этот подход определённо следует избегать. Мы показываем его здесь, потому что иногда вы можете запустить этот код, чтобы лучше понять внутренности другого кода.

Давайте посмотрим, как работает этот код. Цикл гарантированно рабочий, потому что **jiffies** объявлена заголовками ядра как **volatile** (нестабильная) и, таким образом, извлекается из памяти каждый раз, когда какой-либо код Си обращается к ней. Хотя технически правильный (в этом он работает как задумано), этот цикл ожидания серьёзно снижает производительность системы. Если вы не настроили ядро для вытесняющих операций, цикл полностью блокирует процессор на продолжительность задержки; планировщик никогда не вытеснит процесс, который работает в пространстве ядра, и компьютер выглядит совершенно мёртвым до достижения время **j1**. Проблема становится менее серьёзной, если вы работаете на ядре с вытеснением, потому что пока код удерживает блокировку, какое-то время процессора может быть использовано для других целей. Однако, ожидание готовности всё же дорого и на системах с вытеснением.

Ещё хуже, если случится, что прерывания запрещены при входе в цикл, **jiffies** не будет обновляться и условие **while** останется верным навсегда. Запуск вытесняющего ядра также не поможет и вы будете вынуждены нажать большую красную кнопку.

Эта реализация кода задержки, как и последующих, доступна в модуле ***jit***. Файлы ***/proc/jit****, создаваемые модулем, создают задержку в целую секунду каждый раз, когда вы читаете строку текста и строки гарантировано будут 22 байта каждая. Если вы хотите протестировать код ожидания, вы можете читать ***/proc/jitbusy***, который имеет цикл ожидания в одну секунду перед возвращением каждой строки.

Обязательно читайте, самое большее, одну строку (или несколько строк) за раз из ***/proc/jitbusy***. Упрощённый механизм ядра для регистрации файлов ***/proc*** вызывает метод ***read*** снова и снова для заполнения буфера данных по запросу пользователя. Следовательно, такая команда, как ***cat /proc/jitbusy***, если она читает 4 Кб за раз, "подвесит" компьютер на 186 секунд.

Предлагаемой командой для чтения ***/proc/jitbusy*** является ***dd bs=22 < /proc/jitbusy***, которая так же опционально определяет и количество блоков. Каждая 22-х байтовая строка, возвращаемая файлом, отображает младшие 32 бита значения счётчика тиков до и после задержки. Это пример запуска на ничем не загруженном компьютере:

```
phon% dd bs=22 count=5 < /proc/jitbusy
   1686518    1687518
   1687519    1688519
   1688520    1689520
   1689520    1690520
   1690521    1691521
```

Всё выглядит хорошо: задержки ровно одну секунду (1000 тиков) и следующий системный вызов ***read*** начинается сразу после окончания предыдущего. Но давайте посмотрим, что происходит в системе с большим числом интенсивно работающих процессов (и не вытесняющем ядре):

```
phon% dd bs=22 count=5 < /proc/jitbusy
   1911226    1912226
   1913323    1914323
   1919529    1920529
   1925632    1926632
   1931835    1932835
```

Здесь, каждый системный вызов ***read*** задерживается ровно на одну секунду, но ядру может потребоваться более 5-ти секунд, перед передачей управления процессу ***dd***, чтобы он мог сделать следующий системный вызов. Это ожидаемо в многозадачной системе; процессорное время является общим для всех запущенных процессов, а интенсивно занимающий процессор процесс имеет динамически уменьшаемый приоритет. (Обсуждение политик планирования выходит за рамки этой книги.)

Показанный выше тест под нагрузкой был выполнен во время работы примера программы ***load50***. Эта программа разветвляет множество процессов, которые ничего не делают, но выполняют это интенсивно занимая процессор. Эта программа является частью примеров файлов, сопровождающих эту книгу и по умолчанию разветвляет 50 процессов, хотя число может быть задано в командной строке. В этой главе и в других частях книги, такие тесты с загруженной системой были выполнены с работающим ***load50*** на иначе простаивающем компьютере.

Если вы повторите команду во время работы вытесняющего ядра, вы не найдёте заметной разницы при незанятом процессоре и поведением под нагрузкой:

```
phon% dd bs=22 count=5 < /proc/jitbusy
   14940680  14942777
   14942778  14945430
   14945431  14948491
   14948492  14951960
   14951961  14955840
```

Здесь не существует значительной задержки между окончанием системного вызова и началом следующего, но отдельные задержки гораздо больше, чем одна секунда: до 3.8 секунды в показанном примере и со временем возрастают. Эти значения демонстрируют, что этот процесс был прерван во время задержки переключением на другой процесс. Промежуток между системными вызовами является не только результатом переключения для этого процесса, поэтому здесь невозможно увидеть никакой специальной задержки. (возможно, речь идёт о том, что на фоне такой задержки от планировщика процессов невозможно заметить собственно заданную задержку)

Уступание процессора

Как мы видели, ожидание готовности создаёт большую нагрузку на систему в целом; мы хотели бы найти лучшую технику. Первым изменением, которое приходит на ум, является явное освобождение процессора, когда мы в нём не заинтересованы. Это достигается вызовом функции ***schedule***, объявленной в ***<linux/sched.h>***:

```
while (time_before(jiffies, j1)) {
    schedule( );
}
```

Этот цикл может быть протестирован чтением ***/proc/jitsched***, так же, как выше читается ***/***

proc/jitbusy. Однако, он всё ещё не является оптимальным. Текущий процесс ничего не делает, он освобождает процессор, но остаётся в очереди выполнения. Если это единственный исполняемый процесс, он на самом деле работает (вызывает планировщик, который выбирает тот же самый процесс, который вызывает планировщик, который ...). Иными словами, загрузка машины (среднее количество запущенных процессов) является по крайней мере единицей, а задача простоя (idle) (процесс с номером 0, называемый также по исторической причинам ***swapper***) никогда не работает. Хотя этот вопрос может показаться неуместным, работа задачи простоя, когда компьютер не используется, снимает нагрузку на процессор, снижая его температуру, и увеличивает срок его службы, а также срок работы батарей, если компьютер является вашим ноутбуком. Кроме того, поскольку процесс фактически выполняется во время задержки, он несёт ответственность за всё время, которое потребляет.

Поведение ***/proc/jitsched*** фактически аналогично работе ***/proc/jitbusy*** с вытесняющим ядром. Это пример работы на незагруженной системе:

```
phon% dd bs=22 count=5 < /proc/jitsched
   1760205    1761207
   1761209    1762211
   1762212    1763212
   1763213    1764213
   1764214    1765217
```

Интересно отметить, что каждый ***read*** иногда заканчивается ожиданием несколько больших тактовых тиков, чем запрашивалось. Эта проблема становится всё сильнее и сильнее, когда система становится занятой, и драйвер может в конечном итоге ожидать больше, чем предполагалось. После того, как процесс освободил процессор с помощью ***schedule***, нет никаких гарантий, что этот процесс получит процессор обратно в ближайшее время. Поэтому, таким образом, вызов ***schedule*** является небезопасным решением для потребностей драйвера, помимо того, что будет плохим для вычислительной системы в целом. Если протестировать ***jitsched*** время работы ***load50***, можно увидеть, что задержка, связанная с каждой строкой, будет увеличена на несколько секунд, потому что когда истекает время ожидания, процессор используют другие процессы.

Время ожидания

Условно-оптимальные циклы задержки, показанные до сих пор, работают, наблюдая за счётчиком тиков, ничего никому не сообщая. Но самым лучшим способом реализовать задержку, как вы можете себе представить, обычно является попросить ядро сделать это за вас. Существуют два способа создания основанных на тиках ожиданий (timeouts), в зависимости от того, ждёт ваш драйвер другого события или нет.

Если ваш драйвер использует очередь ожидания, чтобы дождаться какого-то другого события, но вы также хотите быть уверены, что она работает в течение определённого периода времени, можно использовать ***wait_event_timeout*** или ***wait_event_interruptible_timeout***:

```
#include <linux/wait.h>
long wait_event_timeout(wait_queue_head_t q, condition, long timeout);
long wait_event_interruptible_timeout(wait_queue_head_t q, condition, long
timeout);
```

Эти функции засыпают в данной очереди ожидания, но они возвращаются после истечения времени ожидания (в пересчёте на тики). Таким образом, они реализуют ограниченный сон, который не длится вечно. Обратите внимание, что время ожидания представляет собой число тиков ожидания, а не абсолютное значение времени. Значение представлено знаковым числом, потому что иногда это результат вычитания, хотя функции жалуются через оператор ***printk***, если установленное время ожидания отрицательно. Если время ожидания истекает, функции возвращают 0; если процесс разбужен другим событием, он возвращает оставшуюся задержку выраженную в тиках. Возвращаемое значение не может быть отрицательным, даже если задержка больше, чем ожидалось из-за загрузки системы. (верно для wait_event_timeout, прерываемая версия вернёт -ERESTARTSYS)

Файл ***/proc/jitqueue*** показывает задержку на основе ***wait_event_interruptible_timeout***, хотя модуль не ждёт никакого события и использует 0 в качестве условия:

```
wait_queue_head_t wait;
init_waitqueue_head (&wait);
wait_event_interruptible_timeout(wait, 0, delay);
```

Наблюдаемое поведение при чтении ***/proc/jitqueue*** почти оптимальное даже при нагрузке:

```
phon% dd bs=22 count=5 < /proc/jitqueue
   2027024    2028024
   2028025    2029025
   2029026    2030026
   2030027    2031027
   2031028    2032028
```

Поскольку процесс чтения (***dd*** в вышеприведённом примере) не находится в рабочей очереди, пока ждёт истечения времени простоя, вы не видите разницы в поведении кода независимо от того, работает вытесняющее ядро или не вытесняющее.

wait_event_timeout и ***wait_event_interruptible_timeout*** были разработаны имея в виду аппаратный драйвер, когда исполнение может быть возобновлено любым из двух способов: либо кто-то вызовет ***wake_up*** в очереди ожидания, или истечёт время ожидания. Это не применимо к ***jitqueue***, так как для очереди ожидания никто не вызывает ***wake_up*** (в конце концов, другой код даже не знает об этом), поэтому процесс всегда просыпается по истечении времени задержки. Для удовлетворения этой самой ситуации, где вы хотите, чтобы задержка исполнения не ожидала каких-либо особых событий, ядро предлагает функцию ***schedule_timeout***, так что вы можете избежать объявления и использования лишнего заголовка очереди ожидания:

```
#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);
```

Здесь ***timeout*** - число тиков для задержки. Возвращается значение 0, если функция вернулась перед истечением данного времени ожидания (в ответ на сигнал). ***schedule_timeout*** требует, чтобы вызывающий сначала устанавливал текущее состояние процесса, поэтому типичный вызов выглядит следующим образом:

```
set_current_state(TASK_INTERRUPTIBLE);
schedule_timeout (delay);
```

Предыдущие строки (из ***/proc/jitschedto***) заставляют процесс спать, пока заданное время

не прошло. Так как ***wait_event_interruptible_timeout*** опирается внутри на ***schedule_timeout***, мы не будем надоедать показом возвращаемых ***jitschedto*** чисел, потому что они такие же, как и для ***jitqueue***. Опять же, следует отметить, что может получиться дополнительное временной интервал между истечением времени ожидания и началом фактического выполнения процесса.

В только что показанном примере первая строка вызывает ***set_current_state***, чтобы настроить всё так, что планировщик не будет запускать текущий процесс снова, пока программа не поместит его обратно в состояние **TASK_RUNNING**. Для обеспечения непрерываемой задержки используйте взамен **TASK_UNINTERRUPTIBLE**. Если вы забыли изменить состояние текущего процесса, вызов ***schedule_timeout*** ведёт себя как вызов ***schedule*** (то есть ведёт себя как ***jitsched***), устанавливая таймер, который не используется.

Если вы хотите поиграть с четырьмя ***jit*** файлами в разных системных ситуациях или на разных ядрах, или попробовать другой способ задерживать выполнение, вам может потребоваться настроить величину задержки при загрузке модуля, устанавливая параметр модуля ***delay***.

Короткие задержки

Когда драйверу устройства необходимо в своём оборудовании иметь дело с задержками, используемые задержки, как правило, несколько десятков микросекунд самое большее. В этом случае полагаться на тактовые тики, определённо, не тот путь.

Функции ядра ***ndelay***, ***udelay*** и ***mdelay*** хорошо обслуживают короткие задержки, задерживая исполнение на указанное число наносекунд, микросекунд или миллисекундах соответственно. (** u* в *udelay* представляет греческую букву мю и используется как *микро*.) Их прототипы:

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

Фактическая реализация функций находится в ***<asm/delay.h>***, являясь архитектурно-зависимой, и иногда построена на внешней функции. Каждая архитектура реализует ***udelay***, но другие функции могут быть не определены; если их нет, ***<linux/delay.h>*** предлагает по умолчанию версию, основанную на ***udelay***. Во всех случаях задержка достигает, по крайней мере, желаемого значения, но может быть больше; на деле, в настоящее время платформы не достигают точности до наносекунд, хотя некоторые из них предлагают субмикросекундную точность. Задержка более чем запрошенное значение, как правило, не проблема, так как небольшие задержки в драйвере обычно необходимы для ожидания оборудования и требуются ожидания по крайней мере заданного промежутка времени.

Реализация ***udelay*** (и, возможно, ***ndelay*** тоже) использует программный цикл на основе расчёта быстродействия процессора во время загрузки, используя целочисленную переменную **loops_per_jiffy**. Однако, если вы хотите посмотреть реальный код, необходимо учитывать, что реализация для x86 весьма сложна из-за разных исходных текстов, базирующихся на типе процессора, выполняющего код.

Чтобы избежать переполнения целого числа в расчётах цикла, ***udelay*** и ***ndelay*** ограничивают передаваемое им значение сверху. Если ваш модуль не может загрузиться и выводит неопределённый символ (unresolved symbol), ***__bad_udelay***, это значит, вы вызвали ***udelay*** со слишком большим аргументом. Однако, следует отметить, что во время компиляции

проверка может быть выполнена только на постоянные значения, и что это реализовано не на всех платформах. Как правило, если вы пытаетесь получить задержку на тысячи наносекунд, вы должны использовать ***udelay*** вместо ***ndelay***; аналогично, задержки масштаба миллисекунд должны выполняться ***mdelay***, а не одной из более точных функций.

Важно помнить, что эти три функции задержки являются ожидающими в состоянии занятости (busy-waiting); другие задачи не могут быть запущены в течение этого времени цикла. Таким образом, они повторяют, хотя и в другом масштабе, поведение ***jitbusy***. Таким образом, эти функции должны быть использованы только когда нет другой альтернативы.

Существует и другой способ получения миллисекундных (и более) задержек, которые не выполняют ожидание в состоянии занятости. Файл ***<linux/delay.h>*** декларирует следующие функции:

```
void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);
```

Первые две функции помещают вызывающий процесс в сон на заданное число **millisecs**. Вызов ***msleep*** является непрерываемым; вы можете быть уверены, что процесс спит по крайней мере заданное число миллисекунд. Если ваш драйвер сидит в очереди ожидания и вы хотите пробуждения для прерывания сна, используйте ***msleep_interruptible***. Возвращаемое значение ***msleep_interruptible*** обычно 0; однако, если этот процесс проснулся раньше, возвращаемое значение является числом миллисекунд, оставшихся от первоначально запрошенного периода сна. Вызов ***ssleep*** помещает процесс в непрерываемый сон на заданное число секунд.

В общем, если вы можете мириться с задержками больше, чем запросили, вы должны использовать ***schedule_timeout***, ***msleep*** или ***ssleep***.

Таймеры ядра

Если необходимо запланировать действия на позднее время без блокирования текущего процесса до наступления момента времени, инструментом для вас являются таймеры ядра. Такие таймеры используются для планирования выполнения функции в определённое время в будущем, основываясь на тактовых тиках, и могут использоваться для различных задач; например, опрос устройства путём проверки его состояния через регулярные промежутки времени, когда оборудование не может генерировать прерывания. Другим типичным использованием таймеров ядра является отключение двигателя дисковода или завершение другой длительной операции выключения. В таких случаях задержка возвращения из ***close*** создала бы ненужные (и неожиданные) трудности для прикладной программы. Наконец, само ядро использует таймеры в ряде ситуаций, включая реализацию ***schedule_timeout***.

Таймер ядра является структурой данных, которая инструктирует ядро для выполнения заданных пользователем функции с заданным пользователем аргументом в заданное пользователем время. Реализация находится в ***<linux/timer.h>*** и ***kernel/timer.c*** и подробно описана в разделе "Реализация таймеров ядра" 191.

Функции, запланированные для запуска, почти наверняка ***не*** работают, пока выполняется процесс, который их регистрирует. Вместо этого, они запускаются асинхронно. До сих пор всё, что мы сделали в наших примерах драйверов - это работа в контексте процесса, выполняющего системные вызовы. Однако, когда запущен таймер, процесс, который

запланировал его, может спать, выполняться на другом процессоре, или, вполне возможно, вообще завершиться.

Это асинхронное выполнение напоминает то, что происходит, когда происходит аппаратное прерывание (которое подробно рассматривается в Главе 10 246). В самом деле, таймеры ядра работают как результат "программного прерывания". При запуске в атомарном контексте этого рода на ваш код налагается ряд ограничений. Функции таймера должны быть атомарными всеми способами, которые мы обсуждали в разделе "Спин-блокировки и контекст атомарности" 112 в Главе 5 101, но есть некоторые дополнительные вопросы, вызванные отсутствием контекста процесса. Теперь вы введём эти ограничения; они будут рассматриваться снова в нескольких местах в последующих главах. Повторение делается потому, что правила для атомарных контекстов должны усердно соблюдаться, или система окажется в тяжёлом положении. Некоторые действия требуют для выполнения контекст процесса. Когда вы находитесь за пределами контекста процесса (то есть в контексте прерывания), вы должны соблюдать следующие правила:

- Не разрешён доступ к пользовательскому пространству. Из-за отсутствия контекста процесса, нет пути к пользовательскому пространству, связанному с любым определённым процессом.
- Указатель **current** не имеет смысла в атомарном режиме и не может быть использован, так как соответствующий код не имеет связи с процессом, который был прерван.
- Не может быть выполнено засыпание или переключение. Атомарный код не может вызвать ***schedule*** или какую-то из форм ***wait_event*** и не может вызвать любые другие функции, которые могли бы заснуть. Например, вызов ***kmalloc(..., GFP_KERNEL)*** идёт против правил. Семафоры также не должны быть использованы, поскольку они могут спать.

Код ядра может понять, работает ли он в контексте прерывания, вызовом функции ***in_interrupt( )***, которая не имеет параметров и возвращает ненулевое значение, если процессор в настоящее время работает в контексте прерывания, аппаратного или программного. Функцией, связанной с ***in_interrupt()*** является ***in_atomic()***. Она возвращает ненулевое значение, когда переключение не допускается; это включает в себя аппаратный и программный контексты прерывания, а также любое время, когда удерживается спин-блокировка. В последнем случае, **current** может быть действительным, но доступ к пользовательскому пространству запрещён, поскольку это может привести к переключению. Всякий раз, когда вы используете ***in_interrupt()***, следует всерьёз рассмотреть вопрос, не является ли ***in_atomic()*** тем, что вы действительно имеете в виду. Обе функции объявлены в ***<asm/hardirq.h>***.

Ещё одной важной особенностью таймеров ядра является то, что задача может перерегистрировать себя для запуска снова в более позднее время. Это возможно, потому что каждая структура **timer_list** не связана со списком активных таймеров перед запуском и, следовательно, может быть немедленно перекомпонована где угодно. Хотя переключение на одну и ту же задачу снова и снова может показаться бессмысленной операцией, иногда это бывает полезно. Например, это может быть использовано для реализации опроса устройств.

Кроме того, стоит знать, что в многопроцессорных системах, таймерная функция (функция, запускаемая таймером) выполняется тем же процессором, который её зарегистрировал для достижения лучшего расположения кэша, когда это возможно. Поэтому таймер, который перерегистрирует себя, всегда запускается на том же процессоре.

Важной особенностью таймеров о которой, однако, не следует забывать является то, что они являются потенциальным источником состояний гонок даже на однопроцессорных системах. Это прямой результате их асинхронности с другим кодом. Таким образом, любые структуры данных, которые используются таймерной функцией, должны быть защищены от одновременного доступа либо использованием атомарных типов (обсуждаемых в разделе "Атомарные переменные" 119 в Главе 5 101) или использованием спин-блокировок (обсуждаемых в Главе 5 101).

API таймера

Ядро предоставляет драйверам ряд функций для декларации, регистрации и удаления таймеров ядра. Ниже приводится выдержка, показывающая основные стандартные блоки:

```
#include <linux/timer.h>
struct timer_list {
    /* ... */
    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);
int del_timer(struct timer_list * timer);
```

Структура данных включает в себя больше полей, чем показано, но эти три предназначены для доступа снаружи кодом таймера. Поле **expires** представляет значение **jiffies**, которое таймер ожидает для запуска; в это время функция **function** вызывается с **data** в качестве аргумента. Если необходимо передать много объектов в аргументе, можно собрать их в единую структуру данных и передать указатель, приведя к **unsigned long**, это безопасная практика на всех поддерживаемых архитектурах и довольно распространена в управлении памятью (как описывается в Главе 15 395). Значение **expires** не является типом **jiffies_64**, поскольку не ожидается, что таймер сработает очень далеко в будущем и на 32-х разрядных платформах 64-х разрядные операции медленны.

Структуры должны быть проинициализированы перед использованием. Этот шаг гарантирует, что все поля правильно настроены, в том числе те, которые не видимы для вызывающего. Инициализация может быть осуществлена вызовом ***init_timer*** или присвоением **TIMER_INITIALIZER** статической структуре, в соответствии с вашими потребностями. После инициализации, перед вызовом ***add_timer***, можно изменить три открытых поля. Чтобы отключить зарегистрированный таймер до его истечения, вызовите ***del_timer***. Модуль ***jit*** включает файл примера, ***/proc/jitimer*** (для "только по таймеру"), который возвращает строку заголовка и шесть строк данных. Строки данных показывают текущее окружение, где выполняется код; первая создаётся файловой операцией ***read***, остальные - по таймеру. Следующий вывод был записан во время компиляции ядра:

```
phon% cat /proc/jitimer
  time    delta inirq  pid   cpu command
 33565837   0     0    1269   0   cat
 33565847  10     1    1271   0   sh
 33565857  10     1    1273   0   cpp0
```

```
 33565867  10     1    1273   0   cpp0
 33565877  10     1    1274   0   cc1
 33565887  10     1    1274   0   cc1
```

В этом выводе, поле **time** является значением **jiffies**, когда код запускается, **delta** является изменением **jiffies** относительно предыдущей строки, **inirq** - это логическое значение, возвращаемое ***in_interrupt***, **pid** и **command** относятся к текущему процессу и **cpu** является номером используемого процессора (всегда 0 на однопроцессорных системах).

Если вы прочитаете ***/proc/jitimer*** при выгрузке системы, вы обнаружите, что контекстом таймера является процесс 0, задача простоя (idle), которая называется “swapper” ("планировщик своппинга") в основном по историческим причинам.

Таймер используется для генерации данных ***/proc/jitimer*** по умолчанию каждые 10 тиков, но при загрузке модуля можно изменить значение, установив параметр **tdelay** (timer delay, задержка таймера).

Следующий отрывок кода показывает часть ***jit***, связанную с таймером ***jitimer***. Когда процесс пытается прочитать наш файл, мы устанавливаем таймер следующим образом:

```
unsigned long j = jiffies;

/* заполняем данные для нашей таймерной функции */
data->prevjiffies = j;
data->buf = buf2;
data->loops = JIT_ASYNC_LOOPS;

/* регистрируем таймер */
data->timer.data = (unsigned long)data;
data->timer.function = jit_timer_fn;
data->timer.expires = j + tdelay; /* параметр */
add_timer(&data->timer);

/* ждём заполнения буфера */
wait_event_interruptible(data->wait, !data->loops);
```

Фактическая функция, вызываемая по таймеру, выглядит следующим образом:

```
void jit_timer_fn(unsigned long arg)
{
    struct jit_data *data = (struct jit_data *)arg;
    unsigned long j = jiffies;
    data->buf += sprintf(data->buf, "%9li %3li %i %6i %i %s\n",
                         j, j - data->prevjiffies, in_interrupt( ) ? 1 :
0,
                         current->pid, smp_processor_id( ), current-
>comm);

    if (--data->loops) {
        data->timer.expires += tdelay;
        data->prevjiffies = j;
        add_timer(&data->timer);
    } else {
        wake_up_interruptible(&data->wait);
```

```
        }
}
```

API таймера включает в себя несколько больше функций, чем те, которые введены выше. Следующий набор завершает список предложения ядра:

int mod_timer(struct timer_list *timer, unsigned long expires);
Обновление времени истечения срока таймера, общая задача, для которой используется таймер ожидания (опять же, типичный пример - таймер выключения мотора дисковода). ***mod_timer*** может быть вызвана при неактивных таймерах, также, как обычно используется ***add_timer***.

int del_timer_sync(struct timer_list *timer);
Работает как ***del_timer***, но также гарантирует, что когда она вернётся, таймерная функция не запущена на каком-то процессоре. ***del_timer_sync*** используется, чтобы избежать состояний состязания на многопроцессорных системах, и аналогична ***del_timer*** на однопроцессорных ядрах. Этой функции в большинстве ситуаций следует отдавать предпочтение перед ***del_timer***. Эта функция может заснуть, если вызывается из неатомарного контекста, но находится в активном ожидании в других ситуациях. Будьте очень осторожны вызывая ***del_timer_sync*** при удержании блокировок; если таймерная функция попытается получить ту же блокировку, система может заблокироваться. Если эта таймерная функция перерегистрирует себя, вызывающий должен сначала убедиться, что эта перерегистрация не произошла; обычно это достигается установкой флага “shutting down” ("выключить"), который проверяется таймерной функцией.

int timer_pending(const struct timer_list * timer);
Возвращает истину или ложь, чтобы показать, будет ли таймер в настоящее время запланирован для запуска, чтением одной из скрытых полей структуры.

Реализация таймеров ядра

Хотя вам не требуется знать, как реализованы таймеры ядра для их использования, реализация их интересна и стоит посмотреть на их внутренности.

Реализация счётчиков была разработана, чтобы соответствовать следующим требованиям и предположениям:

- Управление таймером должно быть как можно более лёгким.
- Конструкция должна позволять увеличивать количество активных таймеров.
- Большинство таймеров истекают в течение нескольких секунд или, самое большее, минут, в то время как таймеры с длительными задержками довольно редки.
- Таймер должен работать на том же процессоре, который его зарегистрировал.

Решение, разработанное разработчиками ядра, основано на копии структуры данных для каждого процессора. Структура **timer_list** включает в себя указатель на такую структуру данных в своём поле **base**. Если **base** является **NULL**, таймер не запланирован для запуска; в противном случае, указатель говорит, какая структура данных (и, следовательно, какой процессор) запускает его. Копии объектов данных для каждого процессора описываются в разделе "Копии переменных для процессора" 217 в Главе 8 203.

Всякий раз, когда код ядра регистрирует таймер (через ***add_timer*** или ***mod_timer***),

операции в конечном итоге выполняются ***internal_add_timer*** (в ***kernel/timer.c***), которая, в свою очередь, добавляет новый таймер в двусвязный список таймеров в рамках "каскадной таблицы", связанной с текущим процессором.

Каскадные таблицы работают так: если таймер истекает в следующих тиках от 0 до 255, он добавляется в один из 256 списков, посвященных таймерам малого диапазона, используя самые младшие биты поля **expires**. Если он истекает дальше в будущем (но до 16384 тиков), он добавляется в один из 64 списков на основе битов 9-14 поля **expires**. Для таймеров истекающих ещё позже, тот же приём используется для битов 15-20, 21-26 и 27-31. Таймеры с полем времени окончания ещё дальше в будущем (что может случиться только на 64-х разрядных платформах) делятся на задержки со значением 0xffffffff и таймеры с **expires** в прошлом планируются для запуска в следующем тике таймера. (Таймер, который уже истёк, иногда может быть зарегистрирован в ситуациях высокой нагрузки, особенно если вы работаете с вытесняющим ядром.)

После запуска ***__run_timers***, он запускает все отложенные таймеры на текущий тик таймера. Если **jiffies** в настоящее время является кратной 256, функция также заново делит один из списков таймеров следующего уровня на 256 списков короткого диапазона, возможно каскадируя один или нескольких других уровней также в соответствии с битовым представлением **jiffies**.

Это подход, хотя и чрезвычайно сложный на первый взгляд, выполняется очень хорошо как с несколькими таймерами, так и с большим их числом. Время, необходимое для управления каждым активным таймером, не зависит от количества уже зарегистрированных таймеров и ограничено несколькими логическими операциями над двоичным представлением поля **expires**. Накладным расходом, связанным с этой реализацией, является память для 512 заголовков листов (256 краткосрочных списков и 4 группы из 64 списков более длительных диапазонов), то есть 4 Кб памяти.

Функция ***__run_timers***, как показано ***/proc/jitimer***, будет запущена в атомарном контексте. В добавок к уже описанным ограничениям, это приносит интересную особенность: таймер истекает только в заданное время, даже если вы не работаете на вытесняющем ядре и процессор занят в пространстве ядра. Вы можете видеть, что происходит, когда вы читаете ***/proc/jitbusy*** в фоновом режиме и ***/proc/jitimer*** с высоким приоритетом. Хотя система кажется прочно заблокированной системным вызовом с активным ожиданием, таймеры ядра всё же работают нормально.

Однако, имейте в виду, что таймер ядра далёк от совершенства, он страдает от дрожания и других артефактов, вносимых аппаратными прерываниями, а также другими таймерами и другими асинхронными задачами. Хотя таймер, связанный с простым цифровым вводом/выводом, может быть достаточен для таких простых задач, как запуск шагового двигателя или другой любительской электроники, это обычно не подходит для производственных систем в промышленных условиях. Для выполнения таких задач вам скорее всего придётся прибегнуть к расширению ядра для реального времени.

Микрозадачи

Другим объектом ядра, связанным с вопросом времени, является механизм ***микрозадачи*** (***tasklet, тасклет***). Он в основном используется в управлении прерываниями (мы увидим его снова в Главе 10 246).

Микрозадачи в некотором смысле напоминают таймеры ядра. Они всегда выполняются во

время прерывания, они всегда работают на том же процессоре, который их запланировал, и они получают **unsigned long** аргумент. Однако, в отличие от таймеров ядра, невозможно запросить выполнения функции в заданное время. Запланировав микрозадачу, вы просто попросите, чтобы она выполнилась позже, во время, выбранное ядром. Такое поведение особенно полезно для обработчиков прерываний, когда аппаратное прерывание должно обслуживаться как можно быстрее, но большинство управлений данными можно смело отложить на более позднее время. На самом деле микрозадача, как и таймер ядра, выполняется (в атомарном режиме) в контексте "программного прерывания", механизма ядра, который выполняет асинхронные задачи при разрешённых аппаратных прерываниях.

Микрозадача существует как структура данных, которая должна быть проинициализирована перед использованием. Инициализация может быть выполнена путём вызова специальной функции или объявлением структуры с помощью определённых макросов:

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

struct tasklet_struct {
    /* ... */
    void (*func)(unsigned long);
    unsigned long data;
};

void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long),
unsigned long data);
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
```

Микрозадачи предлагают ряд интересных особенностей:

- Микрозадачу можно отключить и повторно включить позже; она не будет выполнена, пока она не включена столько раз, сколько была отключена.
- Так же как таймеры, микрозадача может перерегистрировать себя.
- Микрозадача может быть запланирована для выполнения в нормальном или высоком приоритете. Последняя группа всегда выполняется первой.
- Микрозадачи могут быть запущены сразу же, если система не находится под большой нагрузкой, но не позднее, чем при следующем тике таймера.
- Микрозадачи могут конкурировать с другими микрозадачами, но строго последовательны в отношении самих себя - та же микрозадача никогда не работает одновременно более чем на одном процессоре. Кроме того, как уже отмечалось, микрозадача всегда работает на том же процессоре, который её запланировал.

Модуль ***jit*** включает два файла, ***/proc/jitasklet*** и ***/proc/jitasklethi***, которые возвращают те же данные, что и ***/proc/jitimer***, описанный в разделе "Таймеры ядра" 187. Когда вы читаете один из этих файлов, вы получаете обратно заголовок и шесть строк данных. Первая строка данных описывает контекст вызывающего процесса, а другие строки описывают контекст последовательных запусков процедуры микрозадачи. Это пример работы во время компиляции ядра:

```
phon% cat /proc/jitasklet
   time   delta  inirq    pid   cpu command
 6076139     0      0      4370   0   cat
 6076140     1      1      4368   0   cc1
```

```
6076141     1     1      4368   0   cc1
6076141     0     1         2   0   ksoftirqd/0
6076141     0     1         2   0   ksoftirqd/0
6076141     0     1         2   0   ksoftirqd/0
```

Как подтверждается приведенными выше данными, микрозадача выполняется при следующем тике таймера до тех пор, пока процессор занят выполнением процесса, но она запускается сразу, когда процессор простаивает. Ядро предоставляет набор потоков ядра ***ksoftirqd***, один на процессор, чтобы просто запускать обработчики "программных прерываний", такие, как функция ***tasklet_action***. Таким образом последние три запуска микрозадачи проходили в контексте потока ядра ***ksoftirqd***, связанного с CPU **0**. Реализация ***jitasklethi*** использует высокоприоритетную микрозадачу, объясняемую в последующем списке функций.

Фактический код в ***jit***, который реализует ***/proc/jitasklet*** и ***/proc/jitasklethi***, почти идентичен коду, который реализует ***/proc/jitimer***, но вместо таймера он использует вызовы микрозадачи. Ниже приводится список деталей интерфейса ядра для микрозадач после того, как структура микрозадачи была проинициализирована:

void tasklet_disable(struct tasklet_struct *t);
Эта функция отключает данную микрозадачу. Микрозадача по-прежнему может быть запланирована с помощью ***tasklet_schedule***, но её выполнение отложено, пока микрозадача снова не будет включена. Если микрозадача в настоящее время работает, эта функция активно ждёт завершения микрозадачи; таким образом, после вызова ***tasklet_disable*** вы можете быть уверены, что микрозадача не работает нигде в системе.

void tasklet_disable_nosync(struct tasklet_struct *t);
Отключает микрозадачу, но не дожидается завершения никакой запущенной функции. Когда возвращается, микрозадача является отключённой и не будет планироваться в будущем, пока снова не будет включена, но всё ещё может работать на другом процессоре, когда функция возвращается.

void tasklet_enable(struct tasklet_struct *t);
Включает микрозадачу, которая была ранее отключена. Если микрозадача уже запланирована, она будет запущена в ближайшее время. Вызов ***tasklet_enable*** должен соответствовать каждому вызову ***tasklet_disable***, так как ядро отслеживает "счётчик отключений" для каждой микрозадачи.

void tasklet_schedule(struct tasklet_struct *t);
Планирует микрозадачу на исполнение. Если микрозадача запланирована снова прежде, чем у неё появился шанс заработать, она запустится только один раз. Однако, если она запланирована ***во время*** работы, она запустится снова после своего завершения; это гарантирует, что события, происходящие во время обработки других событий, получат должное внимание. Такое поведение также позволяет микрозадаче перепланировать себя.

void tasklet_hi_schedule(struct tasklet_struct *t);
Планирует микрозадачу для выполнения с более высоким приоритетом. Когда обработчик программного прерывания работает, он выполняет высокоприоритетные микрозадачи перед другими задачами программных прерываний, в том числе "нормальных" микрозадач. В идеале только задания с требованиями малой задержки (такие, как заполнение звукового буфера) должны использовать эту функцию, чтобы

избежать дополнительных задержек, вводимых другими обработчиками программных прерываний. На самом деле */proc/jitasklethi* не показывает видимое человеку отличие от */proc/jitasklet*.

void tasklet_kill(struct tasklet_struct *t);
Эта функция гарантирует, что микрозадачу не планируется запустить снова; её обычно вызывают, когда устройство закрывается или модуль удаляется. Если микрозадача запланирована на запуск, функция ждёт её выполнения. Если микрозадача перепланирует сама себя, вы должны запретить ей перепланировать себя перед вызовом *tasklet_kill*, как и в случае *del_timer_sync*.

Микрозадачи реализованы в *kernel/softirq.c*. Два списка микрозадач (нормальный и высокоприоритетный) объявлены как структуры данных, имеющие копии для каждого процессора, используя такой же механизм родства процессоров, как и для таймеров ядра. Структура данных, используемая для управления микрозадачами, является простым связным списком, потому что микрозадачи не имеют ни одного из требований сортировки таймеров ядра.

Очереди задач

Очереди задач (workqueue) поверхностно похожи на микрозадачи; они позволяют коду ядра запросить, какая функция будет вызвана в будущем. Есть, однако, некоторые существенные различия между ними, в том числе:

- Микрозадачи работают в контексте программного прерывания, в результате чего весь код микрозадачи должен быть атомарным. Вместо этого функции очереди задач выполняются в контексте специального процесса ядра; в результате чего они имеют больше гибкости. В частности, функции очереди задач могут спать.
- Микрозадачи всегда работают на процессоре, который их изначально запланировал. Очереди задач по умолчанию работают таким же образом.
- Код ядра может дать запрос, чтобы отложить выполнение функций очереди задач на заданный интервал.

Ключевым различием между ними двумя является то, что микрозадачи выполняются быстро, в течение короткого периода времени и в атомарном режиме, а функции очереди задач могут иметь более высокие задержки, но не должны быть атомарными. Каждый механизм имеет ситуации, когда он уместен.

Очереди задач имеют тип **struct workqueue_struct**, которая определена в ***<linux/workqueue.h>***. Очередь задач должна быть явно создана перед использованием, используя одну из двух следующих функций:

```
struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
```

Каждая очередь задач имеет один или более специализированных процессов ("потоки ядра"), которые запускают функции, помещённые в очередь. Если вы используете *create_workqueue*, вы получите очередь задач, которая имеет специальный поток для каждого процессора в системе. Во многих случаях все эти потоки являются просто излишними; если одного рабочего потока будет достаточно, вместо этого создайте очередь задач с помощью *create_singlethread_workqueue*.

Чтобы поместить задачу в очередь задач, необходимо заполнить структуру **work_struct**. Этот можно сделать во время компиляции следующим образом:

```
DECLARE_WORK(name, void (*function)(void *), void *data);
```

Где **name** является именем структуры, которая должна быть объявлена, **function** является функцией, которая будет вызываться из очереди задач, и **data** является значением для передачи в эту функцию. Если необходимо создать структуру **work_struct** во время выполнения, используйте следующие два макроса:

```
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);
```

INIT_WORK делает более серьёзную работу по инициализации структуры; вы должны использовать его в первый раз при создании структуры. **PREPARE_WORK** делает почти такую же работу, но он не инициализирует указатели, используемые для подключения в очередь задач структуры **work_struct**. Если есть любая возможность, что в настоящее время структура может быть помещена в очередь задач и вы должны изменить эту структуру, используйте **PREPARE_WORK** вместо **INIT_WORK**.

Для помещения работы в очередь задач существуют две функции:

```
int queue_work(struct workqueue_struct *queue, struct work_struct *work);
int queue_delayed_work(struct workqueue_struct *queue, struct work_struct
*work, unsigned long delay);
```

Любая из них добавляет **work** к данной очереди. Однако, если используется ***queue_delayed_work***, фактическая работа не выполняется, пока не пройдёт по крайней мере **delay** тиков. Возвращаемое значение этих функций является ненулевым, если **work** была успешно добавлена в очередь; нулевой результат означает, что эта структура **work_struct** уже ожидает в очереди и не была добавлена во второй раз.

В некоторое время в будущем функция **work** будет вызвана с заданным значением **data**. Эта функция будет работать в контексте рабочего потока, поэтому он может заснуть в случае необходимости, хотя вы должны знать, как этот сон может повлиять на любые другие задачи, находящиеся в той же очереди задач. Однако, такая функция не может получить доступ в пользовательское пространство. Так как она работает внутри в потоке ядра, просто нет доступа в пользовательское пространства.

Если вам необходимо отменить ожидающую запись в очереди задач, можно вызвать:

```
int cancel_delayed_work(struct work_struct *work);
```

Возвращаемое значение отлично от нуля, если запись была отменена ещё до начала исполнения. Ядро гарантирует, что выполнение данной записи не будет начато после вызова ***cancel_delayed_work***. Однако, если ***cancel_delayed_work*** возвращает 0, запись уже может работать на другом процессоре и может всё ещё быть запущена после вызова ***cancel_delayed_work***. Чтобы иметь абсолютную уверенность, что функция **work** не работает нигде в системе после того, как ***cancel_delayed_work*** вернула 0, вы должны затем сделать вызов:

```
void flush_workqueue(struct workqueue_struct *queue);
```

После возвращения ***flush_workqueue***, никакая из функций, помещённых перед этим для вызова в очередь, больше нигде в системе не работает.

Когда вы закончите с очередью задач, можно избавиться от неё:

```
void destroy_workqueue(struct workqueue_struct *queue);
```

Общая очередь

Драйвер устройства во многих случаях не нуждается в своей собственной очереди задач. Если вы только изредка помещаете задачи в очередь, может оказаться более эффективным просто использовать общую очередь задач по умолчанию, предоставляемую ядром. Однако, при использовании этой очереди, вы должны знать, что будете делить её с другими. Среди прочего, это означает, что вы не должны монополизировать очередь на длительные периоды времени (не использовать длительные засыпания) и вашим задачам может потребоваться больше времени для получения ими процессора.

Модуль ***jiq*** (“just in queue”, "только в очередь") экспортирует два файла, которые демонстрируют использование общей очереди задач. Они используют одну структуру **work_struct**, которая создана таким образом:

```
static struct work_struct jiq_work;

    /* это строка в jiq_init( ) */
    INIT_WORK(&jiq_work, jiq_print_wq, &jiq_data);
```

Когда процесс читает ***/proc/jiqwq***, модуль начинает серию перемещений по общей очереди задач без каких-либо задержек. Функция использует это:

```
int schedule_work(struct work_struct *work);
```

Обратите внимание, что при работе с общей очередью используется другая функция; в качестве аргумента требуется только структура **work_struct**. Фактический код в ***jiq*** выглядит следующим образом:

```
prepare_to_wait(&jiq_wait, &wait, TASK_INTERRUPTIBLE);
schedule_work(&jiq_work);
schedule( );
finish_wait(&jiq_wait, &wait);
```

Реальная рабочая функция выводит строку так же, как делает модуль ***jit***, затем в случае необходимости повторно помещает структуру **work_struct** в очередь задач. Вот ***jiq_print_wq*** полностью:

```
static void jiq_print_wq(void *ptr)
{
    struct clientdata *data = (struct clientdata *) ptr;

    if (! jiq_print (ptr))
        return;

    if (data->delay)
```

```
        schedule_delayed_work(&jiq_work, data->delay);
    else
        schedule_work(&jiq_work);
}
```

Если пользователь читает устройство с задержкой (***/proc/jiqwqdelay***), рабочая функция повторно помещает себя в режиме задержки с помощью ***schedule_delayed_work***:

```
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
```

Если вы посмотрите на вывод из этих двух устройств, он выглядит примерно так:

```
% cat /proc/jiqwq
    time   delta preempt  pid cpu command
  1113043      0       0    7   1 events/1
  1113043      0       0    7   1 events/1
  1113043      0       0    7   1 events/1
  1113043      0       0    7   1 events/1
  1113043      0       0    7   1 events/1
% cat /proc/jiqwqdelay
    time   delta preempt  pid cpu command
  1122066      1       0    6   0 events/0
  1122067      1       0    6   0 events/0
  1122068      1       0    6   0 events/0
  1122069      1       0    6   0 events/0
  1122070      1       0    6   0 events/0
```

Когда читается ***/proc/jiqwq***, между печатью каждой строки нет очевидной задержки. Когда вместо этого читается ***/proc/jiqwqdelay***, есть задержка ровно на один тик между каждой строкой. В любом случае мы видим одинаковое печатаемое имя процесса; это имя потока ядра, который выполняет общую очередь задач. Номер процессора напечатан после косой черты; никогда не известно, какой процессор будет работать при чтении ***/proc*** файла, но рабочая функция в последующий период всегда будет работать на том же процессоре.

Если необходимо отменить работающую запись, помещённую в общую очередь, можно использовать ***cancel_delayed_work***, как описано выше. Однако, очистка (flushing) общей очереди задач требует отдельной функции:

```
void flush_scheduled_work(void);
```

Поскольку не известно, кто ещё может использовать эту очередь, никогда не известно, сколько времени потребуется для возвращения ***flush_scheduled_work***.

Краткая справка

Эта глава представляет следующие символы.

Сохранение времени

#include <linux/param.h>

HZ

Символ **HZ** определяет число тактовых тиков, генерируемых за секунду.

#include <linux/jiffies.h>
volatile unsigned long jiffies
u64 jiffies_64
Переменная **jiffies_64** увеличивается один раз за каждой тактовый тик; таким образом, она увеличивается **HZ** раз в секунду. Код ядра чаще всего использует **jiffies**, которая то же самое, что и **jiffies_64** на 64-х разрядных платформах и является его младшей половиной на 32-х разрядных платформах.
int time_after(unsigned long a, unsigned long b);
int time_before(unsigned long a, unsigned long b);
int time_after_eq(unsigned long a, unsigned long b);
int time_before_eq(unsigned long a, unsigned long b);
Эти логические выражения сравнивают тики безопасным способом, без проблем в случае переполнения счётчика и без необходимости доступа к **jiffies_64**.
u64 get_jiffies_64(void);
Получает **jiffies_64** без состояния гонок.

#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);
Преобразует представление времени между тиками и другими представлениями.

#include <asm/msr.h>
rdtsc(low32,high32);
rdtscl(low32);
rdtscll(var64);
Макрос для x86 для чтения тактового счетчика. Они читают его как две 32-х разрядные половины, читая только младшую половину или читая всё это в переменную **long long**.

#include <linux/timex.h>
cycles_t get_cycles(void);
Возвращает тактовый счётчик независимым от платформы образом. Если процессор не предлагает функцию тактового счётчика, возвращается 0.

#include <linux/time.h>
unsigned long mktime(year, mon, day, h, m, s);
Возвращает количество секунд с начала эпохи, основанной на шести аргументах типа **unsigned int**.
void do_gettimeofday(struct timeval *tv);
Возвращает текущее время, как и микросекунд секунд с начала эпохи, с лучшим разрешением, которое может предоставить оборудование. На большинстве платформ разрешающая способность одна микросекунда или лучше, хотя в некоторых платформах предлагается только разрешение с точностью до тиков.
struct timespec current_kernel_time(void);
Возвращает текущее время с разрешением в один тик.

Задержки

#include <linux/wait.h>

long wait_event_interruptible_timeout(wait_queue_head_t *q, condition, signed long timeout);

Помещает текущий процесс в сон в очередь ожидания, задавая значение **timeout** в тиках. Для непрерываемого сна используйте ***schedule_timeout*** (смотрите ниже).

#include <linux/sched.h>
signed long schedule_timeout(signed long timeout);

Вызывает планировщик убедившись, что текущий процесс разбужен по окончании времени ожидания. Вызывающий должен сначала вызвать ***set_current_state*** для установки своего состояния как прерываемый или непрерываемый сон.

#include <linux/delay.h>
void ndelay(unsigned long nsecs);
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);

Вводит задержки на целое число наносекунд, микросекунд и миллисекундах. Задержка достигает по крайней мере желаемого значения, но она может быть и больше. Аргумент каждой функции не должна превышать предела, специфичного для каждой платформы (обычно, несколько тысяч).

void msleep(unsigned int millisecs);
unsigned long msleep_interruptible(unsigned int millisecs);
void ssleep(unsigned int seconds);

Помещает процесс в сон на заданное число миллисекунд (или секунд, в случае ***ssleep***).

Таймеры ядра

#include <asm/hardirq.h>
int in_interrupt(void);
int in_atomic(void);

Возвращает булево значение, сообщающее, выполняется ли вызывающий код в контексте прерывания или в атомарном контексте. Контекст прерывания является внешним по отношению к контексту процесса во время обработки и аппаратного и программного прерывания. В атомарном контексте вы не можете с удержанием спин-блокировки планировать ни контекст прерывания ни контекст процесса.

#include <linux/timer.h>
void init_timer(struct timer_list * timer);
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);

Эта функция и статическая декларация по таймерной структуры являются двумя способами инициализации структуры данных **timer_list**.

void add_timer(struct timer_list * timer);

Регистрирует таймерную структуру для запуска на текущем процессор.

int mod_timer(struct timer_list *timer, unsigned long expires);

Изменяет время окончания уже запланированной таймерной структуры. Она может также выступать в качестве альтернативы ***add_timer***.

int timer_pending(struct timer_list * timer);

Макрос, который возвращает логическое значение для определения, зарегистрирована ли уже для запуска таймерная структура.

void del_timer(struct timer_list * timer);
void del_timer_sync(struct timer_list * timer);

Удаление таймера из списка активных таймеров. Последняя функция гарантирует, что таймер не запущен на другом процессоре.

Микрозадачи

#include <linux/interrupt.h>
DECLARE_TASKLET(name, func, data);
DECLARE_TASKLET_DISABLED(name, func, data);
void tasklet_init(struct tasklet_struct *t, void (*func)(unsigned long), unsigned long data);

Первые два макроса декларируют структуру микрозадачи, а функция ***tasklet_init*** инициализирует структуру микрозадачи, которая была получена путём создания во время работы или другими способами. Второй из макросов DECLARE помечает микрозадачу как отключённую.

void tasklet_disable(struct tasklet_struct *t);
void tasklet_disable_nosync(struct tasklet_struct *t);
void tasklet_enable(struct tasklet_struct *t);

Отключает и снова включает микрозадачу. Каждое ***отключение*** должно сопровождаться ***включением*** (вы можете отключить микрозадачу даже если она уже отключена). Функция ***tasklet_disable*** ожидает завершения микрозадачи, если она работает на другом процессоре. Версия ***nosync*** не делает этого дополнительного шага.

void tasklet_schedule(struct tasklet_struct *t);
void tasklet_hi_schedule(struct tasklet_struct *t);

Планирует микрозадачу для запуска либо как "нормальную" микрозадачу, либо как высокоприоритетную. При запуске программных прерываний высокоприоритетные микрозадачи рассматриваются первыми, в то время как "нормальные" микрозадачи запускаются позже.

void tasklet_kill(struct tasklet_struct *t);

Удаляет микрозадачу из списка активных, если она запланирована для запуска. Как и ***tasklet_disable***, эта функция может блокироваться на многопроцессорных системах, ожидая завершения микрозадачи, если она в настоящее время работает на другом процессоре.

Очереди задач

#include <linux/workqueue.h>
struct workqueue_struct;
struct work_struct;

Структуры, являющиеся очередью задач и рабочей записью, соответственно.

struct workqueue_struct *create_workqueue(const char *name);
struct workqueue_struct *create_singlethread_workqueue(const char *name);
void destroy_workqueue(struct workqueue_struct *queue);

Функции для создания и уничтожения очередей задач. Вызов ***create_workqueue*** создаёт очередь с рабочим потоком на каждом процессоре в системе; наоборот, ***create_singlethread_workqueue*** создаёт очередь задач с одним рабочим процессом.

DECLARE_WORK(name, void (*function)(void *), void *data);
INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);
PREPARE_WORK(struct work_struct *work, void (*function)(void *), void *data);

Макросы, которые объявляют и инициализируют записи очереди задач.

int queue_work(struct workqueue_struct *queue, struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *queue, struct work_struct *work, unsigned long delay);

Функции, которые последовательно работают для запуска из очереди задач.

int cancel_delayed_work(struct work_struct *work);
void flush_workqueue(struct workqueue_struct *queue);

Используйте ***cancel_delayed_work*** для удаления записи из очереди задач; ***flush_workqueue*** гарантирует, что никакая из записей в очереди задач не работает где-либо в системе.

int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);

Функции для работы с общей очередью задач.

Глава 8, Выделение памяти

До сих пор для выделения и освобождения памяти мы использовали ***kmalloc*** и ***kfree***. Однако, ядро Linux предлагает богатый набор примитивов выделения памяти. В этом главе мы рассмотрим другие способы использования памяти в драйверах устройств и как оптимизировать ресурсы памяти вашей системы. Мы не углубляемся в фактическое управление памятью на различных архитектурах. Модули не участвуют в вопросах сегментации, управления страницами и тому подобных, так как ядро предлагает драйверам унифицированный интерфейс управления памятью. Кроме того, мы не будем описывать внутренние детали управления памятью в этой главе, а перенесём это в Главу 15 395.

Как работает kmalloc

Механизм выделения памяти ***kmalloc*** является мощным инструментом и легко изучаемый из-за его сходства с ***malloc***. Функция быстра (если не блокируется) и не очищает память, которую получает; выделенная область по-прежнему сохраняет предыдущее содержимое. (* Среди прочего, это означает, что вы должны явно очищать любую память, которая может быть передана в пользовательское пространство или записана на устройство; в противном случае вы рискуете раскрыть информацию, которая должна быть сохранена в тайне.) Выделенная область является непрерывной в физической памяти. В нескольких следующих разделах мы подробно поговорим о ***kmalloc***, поэтому вы сможете сравнить её с методами выделения памяти, которые мы обсудим позже.

Аргумент flags

Запомните, что прототип для ***kmalloc*** это:

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

void *kmalloc(size_t size, int flags);
```

Первым аргументом ***kmalloc*** является размер блока, который будет выделен. Второй аргумент, флаги выделения, гораздо более интересен, так как он контролирует поведение ***kmalloc*** рядом способов.

Наиболее часто используемый флаг, **GFP_KERNEL**, означает, что выделение (внутренне

выполняемое в конечном счёте вызовом ***__get_free_pages***, которая является источником префикса **GFP_**) производится от имени процесса, запущенного в пространстве ядра. Иными словами, это означает, что вызывающая функция выполняет системный вызов от имени процесса. Использование **GFP_KERNEL** означает, что ***kmalloc*** может поместить текущий процесс в сон для ожидания страницы при вызове в ситуациях недостатка памяти. Следовательно, функция, которая выделяет память с использованием **GFP_KERNEL**, должна быть повторно входимой и не может выполняться в атомарном контексте. Хотя текущий процесс спит, ядро принимает надлежащие меры, чтобы найти свободную память либо сбрасывая буфера на диск, либо выгружая из памяти пользовательский процесс.

GFP_KERNEL не всегда правильный флаг выделения для использования; иногда ***kmalloc*** вызывается вне контекста процесса. Данный тип вызова может случиться, например, в обработчиках прерывания, микрозадачах и таймерах ядра. В этом случае процесс **current** не должен быть помещён в сон и драйвер должен использовать взамен флаг **GFP_ATOMIC**. Ядро обычно старается сохранить несколько свободных страниц для выполнения атомарного выделения памяти. Когда используется **GFP_ATOMIC**, ***kmalloc*** может использовать даже последнюю свободную страницу. Однако, если этой последней страницы не существует, выделение не удаётся.

Вместо или в дополнение к **GFP_KERNEL** и **GFP_ATOMIC** могут быть использованы другие флаги, хотя эти два охватывают большую часть потребностей драйверов устройств. Все флаги определены в ***<linux/gfp.h>*** и некоторые флаги имеют префикс двойного подчеркивания, например, **__GFP_DMA**. Кроме того, есть символы, обозначающие часто используемые сочетания флагов; они не имеют префикса и иногда называются приоритетами выделения. К последним относятся:

GFP_ATOMIC
: Используется для выделения памяти в обработчиках прерываний и другом коде вне контекста процесса. Никогда не засыпает.

GFP_KERNEL
: Нормальное выделение памяти ядра. Может заснуть.

GFP_USER
: Используется для выделения памяти для страниц пространства пользователя; может заснуть.

GFP_HIGHUSER
: Как и GFP_USER, но выделяет из верхней области памяти, если таковая имеется. Верхняя область памяти описана в следующем подразделе.

GFP_NOIO
GFP_NOFS
: Эти флаги функционируют как **GFP_KERNEL**, но они добавляют ограничения на то, что ядро может сделать, чтобы удовлетворить эту просьбу. Выделение с **GFP_NOFS** не разрешает выполнять любые вызовы файловой системы, а **GFP_NOIO** запрещает инициирование любого ввода/вывода для всего. Они используются главным образом в коде файловой системы и виртуальной памяти, где операции выделения может быть разрешено заснуть, но рекурсивные вызовы файловой системы будет плохой идеей.

Флаги выделения, перечисленные выше, могут быть расширены за счёт операции ИЛИ с любым из следующих флагов, которые изменяют осуществление выделения:

__GFP_DMA
: Этот флаг запрашивает, чтобы выделение произошло в DMA-совместимой зоне памяти

(DMA, direct memory access, прямой доступ к памяти, ПДП). Точное значение зависит от платформы и это объясняется в следующем разделе.

__GFP_HIGHMEM
Этот флаг показывает, что выделяемая память может быть расположена в верхней области памяти.

__GFP_COLD
Как правило, при распределитель памяти пытается вернуть страницы "горячего кэша" - страницы, которые могут быть найдены в кэше процессора. Вместо этого, флаг запрашивает "холодную" страницу, которая не была когда-то использована. Это полезно для выделения страниц для чтения при DMA, где присутствие в кэше процессора не является полезным. Смотрите раздел "Прямой доступ к памяти" 423 в Главе 15 395 для полного описания как выделить буферы DMA.

__GFP_NOWARN
Этот редко используемый флаг предотвращает вывод предупреждений ядра (с помощью ***printk***), когда выделение не может быть выполнено.

__GFP_HIGH
Этот флаг помечает высокоприоритетный запрос, который имеет право израсходовать даже последние страницы памяти, сохраняемые ядром для чрезвычайных ситуаций.

__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
Эти флаги изменяют поведение распределителя, когда он испытывает трудности выполнения распределения. **__GFP_REPEAT** означает "попробуй немного подольше", повторяя попытку, но выделение по-прежнему может не состояться. Флаг **__GFP_NOFAIL** приказывает распределителю никогда не закончиться неудачно; он работает так долго, как необходимо для удовлетворения этого запроса. Использование **__GFP_NOFAIL** настоятельно очень не рекомендуется; вероятно, никогда не будет достаточного основания для использования его в драйвере устройства. Наконец, **__GFP_NORETRY** приказывает распределителю отказаться сразу же, если запрашиваемая память не доступна.

Зоны памяти

Оба **__GFP_DMA** и **__GFP_HIGHMEM** имеют платформо-зависимую роль, хотя их использование поддерживается на всех платформах.

Ядро Linux знает как минимум о трёх ***зонах памяти***: DMA-совместимая память, обычная память и верхняя область памяти. Хотя выделение обычно происходит в ***обычной*** зоне, установка одного из только что упомянутых битов потребует, чтобы память была выделена из другой зоны. Идея состоит в том, что каждая компьютерная платформа, которая должна знать о специальных диапазонах памяти (вместо того, чтобы рассматривать всё ОЗУ равнозначно), будет подпадать под эту абстракцию.

DMA-совместимая память является памятью, которая живёт в диапазоне привилегированных адресов, где периферия может выполнять DMA доступ. На наиболее разумных платформах в этой зоне живёт вся память. На x86 зона DMA использует первые 16

Мб ОЗУ, где устаревшие ISA устройства могут выполнять DMA; PCI устройства не имеют такого ограничения.

Верхняя область памяти является механизмом, используемым для разрешения доступа к (относительно) большому количеству памяти на 32-х разрядных платформах. Эта память не может быть доступна непосредственно из ядра без предварительного создания специального отображения и, как правило, с ней труднее работать. Однако, если ваш драйвер использует большие объёмы памяти, он будет лучше работать на больших системах, если сможет использовать верхнюю область памяти. Смотрите раздел "Верхняя и нижняя область памяти" 398 в Главе 15 395 для подробного описания того, как работает верхняя область памяти и как её использовать. Всякий раз, когда по запросу выделения памяти выделяется новая страница, ядро строит список зон, которые могут быть использованы для поиска. Если указан **__GFP_DMA**, поиск происходит только в зоне DMA: если в нижних адресах нет доступной памяти, выделение не удаётся. Если не установлен специальный флаг, исследуются как обычная, так и DMA память; если установлен **__GFP_HIGHMEM**, для поиска свободных страниц используются все три зоны. (Заметьте, однако, что ***kmalloc*** не может выделить верхнюю область памяти.)

Ситуация является более сложной на системах с неоднородным доступом к памяти (nonuniform memory access, NUMA). Как правило, распределитель пытается найти память, локальную для процессора, выполняющего выделение, хотя существуют способы изменения этого поведения.

Механизм, ответственный за зоны памяти реализован в ***mm/page_alloc.c***, а инициализация зоны находится в платформо-зависимых файлах, как правило, в ***mm/init.c*** дерева ***arch***. Мы вернёмся к этим вопросам в Главе 15 395.

Аргумент size

Ядро управляет физической памятью системы, которая доступна только кусками размером со страницу. В результате, ***kmalloc*** выглядит сильно отличающейся от обычной реализации ***malloc*** пользовательского пространства. Простая, ориентированная на динамическое распределение техника быстро столкнулась бы с трудностями; ей будет трудно работать на границах страниц. Таким образом, ядро использует специальную ориентированную на страницы технику выделения, чтобы получить лучшее использование оперативной памяти системы.

Linux обрабатывает выделение памяти создавая набор пулов объектов памяти фиксированных размеров. Запросы выделения обрабатываются походом в пул, который содержит достаточно большие объекты и передаёт запрашивающему обратно весь кусок памяти. Схема управления памятью достаточно сложна и её подробности обычно не интересны для авторов драйверов устройств.

Однако, тем, что разработчики драйверов должны иметь в виду, является то, что ядро может выделять лишь некоторые предопределённые массивы с фиксированным количеством байт. Если вы запросите произвольное количество памяти, вы, вероятно, получите немного больше, чем просили, до двух раз больше. Кроме того, программисты должны помнить, что наименьшее выделение, которое может сделать ***kmalloc*** так же велико, как 32 или 64 байта, в зависимости от размера страницы используемого архитектурой системы.

Существует верхний предел размера кусков памяти, которые могут быть выделены ***kmalloc***. Этот предел зависит от архитектуры и параметров конфигурации ядра. Если ваш код должен

быть полностью переносимым, он не может рассчитывать на возможность выделять ничего больше 128 Кб. Однако, если вам требуется больше, чем несколько килобайт, есть лучшие способы, чем получение памяти через ***kmalloc***, которые мы опишем позже в этой главе.

Заготовленные кэши (Lookaside Caches)

Драйвер устройства часто заканчивает тем, что выделяет память для многих объектов одинакового размера, снова и снова. Учитывая, что ядро уже поддерживает набор пулов памяти объектов, имеющих один размер, почему бы не добавить некоторые специальные пулы для объектов большого размера? В самом деле, ядро действительно имеет средство для создания пула такого сорта, который часто называют ***lookaside cache (подготовленный заранее, заготовленный кэш)***. Драйверы устройств, как правило, не обращаются с памятью так, чтобы оправдать использование заготовленного кэша, но могут быть исключения, USB и SCSI драйверы в Linux версии 2.6 используют кэши.

Менеджер кэша в ядре Linux иногда называют "распределитель кусков" ("slab allocator"). Поэтому, его функции и типы объявлены в ***<linux/slab.h>***. Распределитель кусков реализует кэши, которые имеют тип **kmem_cache_t**; они создаются с помощью вызова ***kmem_cache_create***:

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

Функция создаёт новый объект кэша, который может содержать любое количество областей памяти одинакового размера, задаваемого аргументом **size**. Аргумент **name** ассоциируется с кэшом и функциями как служебная информация, используемая для отслеживания проблем; Как правило, он устанавливается как имя типа структуры, которая кэшируется. Кэш хранит указатель на имя, а не копирует его, поэтому драйвер должен передать указатель на имя в статической памяти (обычно, такое имя является только строкой из букв). Имя не может содержать пробелы.

offset является смещением первого объекта на странице; он может быть использован для обеспечения особого выравнивания для выделенных объектов, но вы, скорее всего, будет использовать 0 для запроса значения по умолчанию. **flags** контролирует, как осуществляется выделение и является битовой маской из следующих флагов:

SLAB_NO_REAP
: Установка этого флага защищает кэш от уменьшения, когда система ищет память. Установка этого флага обычно является плохой идеей; важно избегать действий, излишне ограничивающих свободу распределителя памяти.

SLAB_HWCACHE_ALIGN
: Этот флаг требует, чтобы каждый объект данных был выровнен по строке кэша (наименьшей единице информации, которую можно записать в кэш); фактическое выравнивание зависит от топологии кэша данной платформы. Эта опция может быть хорошим выбором, если ваш кэш содержит элементы, к которым часто идёт обращение на многопроцессорных машинах. Однако, заполнение, необходимое для достижения

выравнивания линий кэша, может в конечном итоге потратить значительные объемы памяти.

SLAB_CACHE_DMA
Этот флаг требует, чтобы каждый объект данных располагался в зоне памяти DMA.

Существует также набор флагов, которые могут быть использованы во время отладки выделения кэша; для подробностей смотрите ***mm/slab.c***. Однако, обычно эти флаги устанавливаются глобально через опции конфигурации ядра на системах, используемых для разработки.

Аргументы **constructor** и **destructor** для этой функции являются необязательными функциями (но деструктора не может быть без конструктора); первая может быть использован для инициализации вновь созданных объектов, а вторая может быть использована для "очистки" объектов до передачи их памяти обратно в систему в целом.

Конструкторы и деструкторы могут быть полезны, но есть несколько ограничений, которые следует иметь в виду. Конструктор вызывается, когда выделяется память для набора объектов; так как память может содержать несколько объектов, конструктор может быть вызван несколько раз. Вы не можете предполагать, что конструктор будет вызываться как непосредственный результат выделения памяти объекту. Аналогичным образом, деструкторы могут быть вызваны неизвестно когда в будущем, а не сразу же после того, как объект был освобождён. Конструкторам и деструкторам может или не может быть позволено спать, в зависимости от того, передают ли они флаг **SLAB_CTOR_ATOMIC** (где **CTOR** - сокращение для ***конструктор***).

Для удобства программист может использовать одну и ту же функцию для конструктора и деструктора; распределитель кусков всегда передаёт флаг **SLAB_CTOR_CONSTRUCTOR**, когда вызываемый является конструктором.

После того, как кэш объектов создан, вы можете выделять объекты из него, вызывая ***kmem_cache_alloc***:

```
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);
```

Здесь, аргумент **cache** является кэшем, созданным ранее; **flags** - те же, что вы бы передали ***kmalloc*** и учитываются, если ***kmem_cache_alloc*** требуется выйти и выделить себе больше памяти.

Чтобы освободить объект, используйте ***kmem_cache_free***:

```
void kmem_cache_free(kmem_cache_t *cache, const void *obj);
```

Когда код драйвера закончил работать с кэшем, как правило, при выгрузке модулей, следует освободить свой кэш следующим образом:

```
int kmem_cache_destroy(kmem_cache_t *cache);
```

Операция уничтожения успешна, только если все объекты, полученные из кэша, были возвращены в него. Таким образом, модуль должен проверить статус, возвращённый ***kmem_cache_destroy***; ошибка указывает на какой-то вид утечки памяти в модуле (так как некоторые объекты были потеряны).

Дополнительной пользой от использования заготовленных кэшей является то, что ядро ведёт статистику использования кэша. Эти данные могут быть получены из ***/proc/slabinfo***.

scull, основанный на кешах кусков: scullc

Пришло время для примера. ***scullc*** является урезанной версией модуля ***scull***, реализующей только пустое устройство - постоянную область памяти. В отличие от ***scull***, который использует ***kmalloc***, ***scullc*** использует кэш-память. Размер кванта может быть изменён в время компиляции и во время загрузки, но не во время работы - это потребует создания новой кэш-памяти и мы не хотим иметь дело с этими ненужными деталями.

scullc является полноценным примером, который может быть использован для опробования распределителя кусков. Он отличается от ***scull*** лишь несколькими строками кода. Во-первых, мы должны задекларировать наш собственной кэш кусков:

```
/* декларируем наш указатель кэша: используем его для всех устройств */
kmem_cache_t *scullc_cache;
```

Создание кэша кусков обрабатывается (во время загрузки модуля) следующим образом:

```
/* scullc_init: создаём кэш для нашего кванта */
scullc_cache = kmem_cache_create("scullc", scullc_quantum,
        0, SLAB_HWCACHE_ALIGN, NULL, NULL); /* конструктора/деструктора нет
*/
if (!scullc_cache) {
    scullc_cleanup( );
    return -ENOMEM;
}
```

Вот как выделяется память квантов:

```
/* Выделить квант используя кэш-память */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = kmem_cache_alloc(scullc_cache, GFP_KERNEL);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, scullc_quantum);
}
```

А это строки освобождения памяти:

```
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        kmem_cache_free(scullc_cache, dptr->data[i]);
```

Наконец, во время выгрузки модуля мы должны вернуть кэш-память системе:

```
/* scullc_cleanup: освободить кэш нашего кванта */
if (scullc_cache)
    kmem_cache_destroy(scullc_cache);
```

Основным отличием при переходе от ***scull*** к ***scullc*** является незначительное улучшение

скорости и лучшее использование памяти. Поскольку кванты выделяются из пула памяти фрагментов точно необходимого размера, их размещение в памяти такое плотное, как это возможно, в отличие от квантов ***scull***, которые создают непредсказуемую фрагментацию памяти.

Пулы памяти

Есть места в ядре, где выделению памяти не может быть разрешено быть неудачным. В качестве способа обеспечения выделения памяти в таких ситуациях разработчики ядра создали абстракцию, известную как ***пул памяти*** (или "mempool"). Пул памяти на самом деле это просто форма заготовленного кэша, который старается всегда держать у себя список свободной памяти для использования в чрезвычайных ситуациях.

Пул памяти имеет тип **mempool_t** (определённый в ***<linux/mempool.h>***); вы можете создать его с помощью ***mempool_create***:

```
mempool_t *mempool_create(int min_nr,
                          mempool_alloc_t *alloc_fn,
                          mempool_free_t *free_fn,
                          void *pool_data);
```

Аргумент **min_nr** является минимальным числом выделенных объектов, которые пул должен всегда сохранять вокруг. Фактическое выделение и освобождение объектов обрабатывается ***alloc_fn*** и ***free_fn***, которые имеют такие прототипы:

```
typedef void *(mempool_alloc_t)(int gfp_mask, void *pool_data);
typedef void (mempool_free_t)(void *element, void *pool_data);
```

Последний параметр ***mempool_create*** (**pool_data**) передаётся в ***alloc_fn*** и ***free_fn***.

При необходимости вы можете написать специализированные функции для обработки выделения памяти для пулов памяти. Однако, обычно вы просто хотите дать обработчику распределителю кусков ядра выполнить за вас такую задачу. Существуют две функции (***mempool_alloc_slab*** и ***mempool_free_slab***), которые выполняют соответствующие согласования между прототипами выделения пула памяти и ***kmem_cache_alloc*** и ***kmem_cache_free***. Таким образом, код, который создаёт пулы памяти, часто выглядит следующим образом:

```
cache = kmem_cache_create(. . .);
pool = mempool_create(MY_POOL_MINIMUM,
                      mempool_alloc_slab, mempool_free_slab,
                      cache);
```

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

```
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
```

После создания пула памяти функция выделения будет вызвана достаточное число раз для создания пула предопределённых объектов. После этого вызовы ***mempool_alloc*** пытаются обзавестись дополнительными объектами от функции выделения; когда такое выделение оканчивается неудачей, возвращается один из предопределённых объектов (если таковые

сохранились). Когда объект освобождён ***mempool_free***, он сохраняется в пуле, если количество предопределённых объектов в настоящее время ниже минимального; в противном случае он будет возвращён в систему.

Размер пула памяти может быть изменён с помощью:

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

В случае успеха этот вызов изменяет размеры пула, чтобы иметь по крайней мере ***new_min_nr*** объектов.

Если пул памяти вам больше не нужен, верните его системе:

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

Вы должны вернуть все выделенные объекты перед уничтожением пула памяти, или ядро в результате выдаст ошибку Oops.

Если вы предполагаете использование в своём драйвере пула памяти, пожалуйста, имейте в виду: пулы памяти выделяют участок памяти, который находится в списке, бездействует и недоступен для любого реального использования. Пулы памяти легко съедают большие объёмы памяти. Вместо этого, почти в каждом случае предпочтительной альтернативой является работа без пула памяти и обработка возможных неудач выделения памяти. Если есть способ в вашем драйвере отреагировать на ошибку выделения таким образом, чтобы не поставить под угрозу целостность системы, делайте это таким способом. Использование пулов памяти в коде драйвера должны быть редким.

get_free_page и друзья

Если модулю необходимо выделять большие блоки памяти, как правило, лучше использовать странично-ориентированные техники. Запрос целых страниц также имеет и другие преимущества, которые представлены в Главе 15 395.

Чтобы выделить страницы, доступны следующие функции:

get_zeroed_page(unsigned int flags);
Возвращает указатель на новую страницу и заполняет страницу нулями.

__get_free_page(unsigned int flags);
Подобно ***get_zeroed_page***, но страницу не очищает.

__get_free_pages(unsigned int flags, unsigned int order);
Выделяет память и возвращает указатель на первый байт области памяти, которая потенциально размером с несколько (физически непрерывных) страниц, но не обнуляет область.

Аргумент **flags** работает так же, как с ***kmalloc***; как правило, используется **GFP_KERNEL** или **GFP_ATOMIC**, возможно, с добавлением флага **__GFP_DMA** (для памяти, которая может быть использована для ISA-операций прямого доступа к памяти) или **__GFP_HIGHMEM**, когда может быть использована верхняя область памяти. (* Хотя на самом деле для выделения страниц верхней области памяти в действительности должна быть использована *alloc_pages* (описываемая в ближайшее время) по причинам, которые мы не можем объяснить до Главы 15

395.) **order** (порядок) является результатом логарифма с основанием двойки числа запрошенных или освобождаемых страниц (то есть, log2N). Например, **order** равен 0, если вы хотите одну страницу и 3, если вы запрашиваете восемь страниц. Если **order** является слишком большим (нет такой непрерывной области такого размера), выделение страниц не удаётся. Функция ***get_order***, которая принимает целочисленный аргумент, может быть использована для получения **order** из размера (который должен быть степенью двойки) на данной платформе. Максимально допустимое значение для **order** составляет 10 или 11 (соответствующее 1024 или 2048 страниц) в зависимости от архитектуры. Как бы то ни было, шансы успешного выделения при порядке 10 иначе, чем на только что загруженной системе с большим количеством памяти, малы.

Если вам интересно, ***/proc/buddyinfo*** расскажет вам, как много блоков каждого порядка доступно для каждой зоны памяти в системе.

Когда программа заканчивает работать со страницами, она может освободить их одной из следующих функций. Первая функция является макросом, который использует вторую:

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

При попытке освободить другое число страниц от того, что вы выделили, карта памяти становится повреждённой и система позднее попадёт в беду.

Стоит подчеркнуть, что ***__get_free_pages*** и другие функции могут быть вызваны в любой время и подчиняются тем же правилам, которые мы видели для ***kmalloc***. В определённых обстоятельствах функции могут не выделить память, особенно когда используется **GFP_ATOMIC**. Поэтому программа, вызывающая эти функции выделения, должна быть готова обработать ошибку выделения.

Хотя ***kmalloc(GFP_KERNEL)*** иногда заканчивается неудачей, когда нет доступной памяти, ядро делает всё возможное, чтобы выполнить запросы на выделение памяти. Таким образом, можно легко затруднить реагирование системы, запрашивая слишком много памяти. Например, вы можете свалить компьютер, поместив слишком много данных в устройство ***scull***; система начинает сканирование в попытке выгрузить как можно больше для того, чтобы выполнить запрос ***kmalloc***. Так как каждый ресурс поглощается растущим устройством, компьютер скоро оказывается непригодным для использования; в этот момент вы больше не можете даже запустить новый процесс, чтобы попытаться решить эту проблему. Мы не будем решать этот вопрос в ***scull***, так как это просто пример модуля, а не реальный инструмент для размещения в многопользовательской системе. Как программист, вы должны быть, тем не менее, осторожными, поскольку модуль является привилегированным кодом и может открыть новые дыры в безопасности системы (наиболее вероятной, как только что говорилось, является дыра отказа службы).

scull, использующий целые страницы: scullp

В целях проверки выделения страниц по-настоящему, вместе с другим примером кода мы выпустили модуль ***scullp***. Это сокращённый ***scull***, как и ***scullc***, с которым мы познакомились ранее. Память кванта, выделяемая ***scullp***, является целыми страницами или наборами страниц: значение для ***scullp_order*** по умолчанию 0, но может быть изменено во время компиляции или загрузки. Следующие строки показывают, как он выделяет память:

```
/* Здесь выделяется память для одного кванта */
```

```
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = (void *)__get_free_pages(GFP_KERNEL, dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

Код для освобождения памяти в ***scullp*** выглядит следующим образом:

```
/* This code frees a whole quantum-set */
for (i = 0; i < qset; i++)
    if (dptr->data[i])
        free_pages((unsigned long)(dptr->data[i]), dptr->order);
```

На уровне пользователя, разница воспринимается в первую очередь как улучшение скорости и лучшее использование памяти, потому что нет внутренней фрагментации памяти. Мы запускали тесты, копируя 4 Мб из ***scull0*** к ***scull1***, а затем из ***scullp0*** к ***scullp1***; результаты показали некоторое улучшение в использовании пространства ядра процессора.

Увеличение производительности не слишком большое, потому что ***kmalloc*** создана быть быстрой. Основным преимуществом выделения страниц на самом деле является не скорость, а более эффективное использование памяти. Выделение по страницам не создаёт отходов памяти, в то время как при использовании ***kmalloc*** теряется непредсказуемый объём памяти, из-за зернистости распределения. Но самое большое преимущество функции ***__get_free_page*** то, что полученная страницы полностью ваша, и теоретически можно собрать страницы в линейной области соответствующими настройками таблиц страниц. Например, можно разрешить пользовательскому процессу сделать ***mmap*** области памяти, полученной как целые несвязанные страницы. Мы обсуждаем такую операцию в Главе 15 395, где мы покажем, как ***scullp*** предлагает связывать память, то, что ***scull*** предложить не может.

Интерфейс alloc_pages

Для полноты картины мы вводим ещё один интерфейс для выделения памяти, хотя мы ещё не будем готовы его использовать, пока не закончим Главу 15 395. В настоящее время достаточно сказать, что структура **page** является внутренней структурой ядра, которая описывает страницу памяти. Как мы увидим, есть много мест в ядре, где необходимо работать со структурами страниц; они особенно полезны в любой ситуации, где вы могли бы иметь дело с верхней областью памяти, которая не имеет постоянного адреса в пространстве ядра.

Настоящим ядром распределителя страниц Linux является функция, названная ***alloc_pages_node***:

```
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int
order);
```

Эта функция также имеет два варианта (которые просто макросы); эти версии то, что вы, скорее всего, будете использовать:

```
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
```

Основная функция, ***alloc_pages_node***, принимает три аргумента. **nid** является ID узла NUMA (* Компьютеры и многопроцессорные системы с NUMA (неоднородным доступом к

памяти), где память является "локальной" для определённых групп процессоров ("узлов"). Доступ к локальной памяти происходит быстрее, чем доступ к не локальной памяти. В таких системах выделение памяти на правильном узле является важным. Однако, авторы драйверов обычно не должны беспокоиться о проблемах NUMA.), чья память должна быть выделена, **flags** является обычными **GFP_** флагами выделения и **order** определяет размер выделяемой памяти. Возвращаемое значение является указателем на первую из (возможно, нескольких) структур страниц, описывающих выделенную память, или, как обычно, **NULL** в случае неудачи. ***alloc_pages*** упрощает ситуацию путём выделения памяти на текущем узле NUMA (она вызывает ***alloc_pages_node*** с возвращением значения из ***numa_node_id*** как параметр **nid**). И, конечно, ***alloc_page*** опускает параметр **order** и выделяет одну страницу.

Чтобы освободить страницы, выделенные таким образом, вы должны использовать одно из следующих действий:

```
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
```

Если у вас есть особые знания о том, вероятно ли, что содержимое одной страницы постоянно находится в кэше процессора или нет, вы должны проинформировать ядро с помощью ***free_hot_page*** (для страниц, постоянно находящихся в кэше) или ***free_cold_page***. Эта информация помогает распределителю памяти оптимизировать использование памяти в системе.

vmalloc и друзья

Следующей функцией распределения памяти, которую мы показываем вам, является ***vmalloc***, которая выделяет непрерывную область памяти в ***виртуальном*** адресном пространстве. Хотя страницы не являются последовательными в физической памяти (каждая страница получена отдельным вызовом ***alloc_page***), ядро видит их как непрерывный диапазон адресов. ***vmalloc*** возвращает **0** (**NULL** адрес), если происходит ошибка, в противном случае, она возвращает указатель на линейную область памяти как минимум заданного размера.

Мы описываем здесь ***vmalloc***, потому что это один из основных механизмов распределения памяти Linux. Однако, следует отметить, что использование ***vmalloc*** в большинство ситуаций не рекомендуется. Память, полученная от ***vmalloc***, работает чуть менее эффективно и на некоторых архитектурах размер адресного пространства, отведённый для ***vmalloc***, относительно мал. Код, который использует ***vmalloc***, может получить холодный приём, если направляется для включения в ядро. Если возможно, следует работать непосредственно с отдельными страницами вместо того, чтобы сгладить проблемы с помощью ***vmalloc***.

Тем не менее, давайте посмотрим, как работает ***vmalloc***. Прототипами функции и её родственников (***ioremap***, которая является не строго функцией выделения, обсуждаемой ниже в этом разделе) являются:

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

void *vmalloc(unsigned long size);
void vfree(void * addr);
void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void * addr);
```

Стоит подчеркнуть, что адреса памяти, возвращаемые ***kmalloc*** и ***_get_free_pages***, также являются виртуальными адресами. Их фактическим значением всё ещё манипулирует MMU (memory management unit (блок управления памятью), как правило, часть центрального процессора), прежде чем для них будут использованы адреса физической памяти. (* На самом деле некоторые архитектуры определяют диапазоны "виртуальных" адресов как зарезервированные при адресации физической памяти. Когда это происходит, ядро Linux использует эту особенность, и ядро и *__get_free_pages* используют адреса, лежащие в одном из этих диапазонов памяти. Разница невидима для драйверов устройств и другого кода, который не связан непосредственно с подсистемой управления памяти ядра.) ***vmalloc*** не отличается в том, как она использует аппаратные средства, а скорее в том, как ядро выполняет задачу выделения.

(Виртуальный) диапазон адресов, используемый ***kmalloc*** и ***__get_free_pages*** имеет связь один к одному с физической памятью, возможно, сдвинутый на постоянную величину **PAGE_OFFSET**; эти функции не требуют изменения таблицы страниц для этого диапазона адресов. Диапазон адресов, используемый ***vmalloc*** и ***ioremap***, с другой стороны, является полностью синтетическим и каждое выделение памяти строит (виртуальную) область памяти соответствующей настройкой таблиц страниц.

Эта разница может быть понята сравнением указателей, возвращаемых функциями выделения. На некоторых платформах (например, x86), адреса, возвращаемые ***vmalloc***, только вне адресов, которые использует ***kmalloc***. На других платформах (например, MIPS, IA-64 и x86_64), они принадлежат совершенно другому адресному диапазону. Доступные для ***vmalloc*** адреса находятся в диапазоне от **VMALLOC_START** до **VMALLOC_END**. Оба символа определены в ***<asm/pgtable.h>***.

Адреса, выделенные ***vmalloc***, не могут быть использованы вне микропроцессора, потому что они имеют смысл только поверх MMU процессора. Когда драйверу требуется реальный физический адрес (например, адрес DMA, используемый периферийным оборудованием для управления системной шиной), вы не сможете легко использовать ***vmalloc***. Правильным временем для вызова ***vmalloc*** является то, когда вы выделяете память для большого последовательного буфера, который существует лишь в программе. Важно отметить, что ***vmalloc*** имеет большие накладные расходы, чем ***__get_free_pages***, потому что она должна получить память и построить таблицы страниц. Таким образом, не имеет смысла вызывать ***vmalloc*** для выделения только одной страницы.

Примером функции в ядре, которая использует ***vmalloc***, является системный вызов ***create_module***, использующий ***vmalloc*** для получения пространства для создаваемого модуля. Код и данные модуля позднее копируются в выделенное пространство, используя ***copy_from_user***. Таким образом, модуль выглядит загруженным в непрерывную память. Вы можете проверить, просмотрев в ***/proc/kallsyms***, что символы ядра, экспортируемые модулями, лежат в диапазоне памяти отличном от символов, экспортируемых самим ядром.

Память, выделенная ***vmalloc***, освобождается ***vfree***, таким же образом, как ***kfree*** освобождает память, выделенную ***kmalloc***.

Как и ***vmalloc***, ***ioremap*** строит новые таблицы страниц; однако, в отличие от ***vmalloc***, она на самом деле не выделяет какую-то память. Возвращаемым значением ***ioremap*** является специальный виртуальный адрес, который может быть использован для доступа к указанному физическому диапазону адресов; виртуальный адрес, полученный впоследствии, освобождается вызовом ***iounmap***.

ioremap является наиболее полезной для связи (физического) адреса PCI буфера с (виртуальным) пространством ядра. Например, она может быть использована для доступа в кадровому буферу видео PCI устройства; такие буферы, как правило, связаны с верхним диапазоном адресов, для которого ядро строит таблицы страниц во время загрузки. Вопросы PCI объяснены более подробно в Главе 12 288.

Стоит отметить, что ради переносимости вы не должны прямо обращаться к адресам, возвращаемым ***ioremap***, как если бы они были указателями на память. Наоборот, следует всегда использовать ***readb*** и другие функции ввода/вывода, представленные в Главе 9 224. Это требование применяется, поскольку некоторые платформы, такие как Alpha, не в состоянии непосредственно связать области памяти PCI с адресным пространством процессора из-за различий между спецификациями передачи данных PCI и процессоров Alpha.

Обе ***ioremap*** и ***vmalloc*** являются странично ориентированными (они работают, изменяя таблицы страниц), следовательно, перевыделенный или выделенный размер округляется до ближайшей границы страницы. ***ioremap*** имитирует невыровненное связывание "округлением вниз" адреса для переназначения и возвращением смещения в первой переназначенной странице. Один небольшим недостатком ***vmalloc*** является то, что она не может быть использована в атомарном контексте, потому что внутри она использует ***kmalloc (GFP_KERNEL)*** для запроса места для хранения таблиц страниц и поэтому может заснуть. Это не должно быть проблемой - если использование ***__get_free_page*** недостаточно хорошо для обработчика прерывания, дизайн программы нуждается в некоторой очистке.

scull использующий виртуальные адреса: scullv

Пример кода, использующего ***vmalloc***, приводится в модуле ***scullv***. Как и ***scullp***, этот модуль является урезанной версией ***scull***, которая использует другую функции выделения памяти для получения пространство для хранения данных устройства.

Модуль выделяет память по 16 страниц за раз. Для достижения лучшей производительности распределение производится большими блоками, чем в ***scullp***, и чтобы показать то, что делается слишком долго с другими методами распределения чтобы быть нецелесообразным. Выделение более одной страницы с помощью ***__get_free_pages*** склонно к ошибке и даже если это удастся, может быть медленным. Как мы видели ранее, ***vmalloc*** быстрее, чем другие функции при выделении нескольких страниц, но несколько медленнее при получении одной страницы из-за накладных расходов при построении таблицы страниц. ***scullv*** разработан подобно ***scullp***. **order** определяет "порядок" каждого выделения памяти и по умолчанию равен 4. Единственным различием между ***scullv*** и ***scullp*** является управление выделением памяти. Для получения новой памяти эти строки используют ***vmalloc***:

```
/* Выделяем квант используя виртуальные адреса */
if (!dptr->data[s_pos]) {
    dptr->data[s_pos] = (void *)vmalloc(PAGE_SIZE << dptr->order);
    if (!dptr->data[s_pos])
        goto nomem;
    memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);
}
```

а эти строки освобождают память:

```
/* Освободить набор квантов */
for (i = 0; i < qset; i++)
```

```
    if (dptr->data[i])
        vfree(dptr->data[i]);
```

Если вы скомпилируете оба модулей с включённой отладкой, вы сможете увидеть распределение их данных, читая файлы, которые они создают в */proc*. Это снимок был сделан на системе x86_64:

```
salma% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1535135
 item at 000001001847da58, qset at 000001001db4c000
       0:1001db56000
       1:1003d1c7000

salma% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
 item at 000001001847da58, qset at 0000010013dea000
       0:ffffff0001177000
       1:ffffff0001188000
```

Следующий вывод, вместо этого, пришёл из системы x86:

```
rudo% cat /tmp/bigfile > /dev/scullp0; head -5 /proc/scullpmem

Device 0: qset 500, order 0, sz 1535135
 item at ccf80e00, qset at cf7b9800
       0:ccc58000
       1:cccdd000

rudo% cat /tmp/bigfile > /dev/scullv0; head -5 /proc/scullvmem

Device 0: qset 500, order 4, sz 1535135
 item at cfab4800, qset at cf8e4000
       0:d087a000
       1:d08d2000
```

Значения показывают два различных поведения. На x86_64 физические адреса и виртуальные адреса связаны с совершенно разными диапазонами адресов (0x100 и 0xffffff00), в то время как на x86 компьютерах ***vmalloc*** возвращает виртуальные адреса просто выше связанных с использованием физической памяти.

Копии переменных для каждого процессора

Per-CPU variables (По-Процессорные переменные) являются интересной особенностью ядра версии 2.6. Когда вы создаёте по-процессорную переменную, каждый процессор в системе получает собственную копию этой переменной. Желание сделать это может показаться странным, но оно имеет свои преимущества. Доступ к по-процессорным переменным не требует (почти) блокирования, потому что каждый процессор работает со своей собственной копией. По-процессорные переменные могут также оставаться в своих процессорных кэшах, что приводит к существенно более высокой производительности для часто обновляемых параметров.

Хороший пример использования по-процессорной переменной можно найти в сетевой

подсистеме. Ядро поддерживает бесконечные счётчики, отслеживая, как много пакетов каждого типа было получено; эти счетчики могут быть обновлены тысячи раз в секунду. Вместо того, чтобы иметь дело с вопросами кэширования и блокировки, сетевые разработчики поместили счётчики статистики в по-процессорные переменные. Обновления происходят теперь без блокировки и быстро. В редких случаях, в которых пользовательское пространство делает запрос, чтобы увидеть значения счётчиков, это просто вопрос сложения версий каждого процессора и возвращения общей суммы.

Декларация по-процессорных переменных может быть найдена в ***<linux/percpu.h>***. Чтобы создать по-процессорную переменную во время компиляции, используйте этот макрос:

```
DEFINE_PER_CPU(type, name);
```

Если переменная (которая будет называться **name**) является массивом, включается информация о типе массива. Таким образом, по-процессорный массив из трёх чисел был бы создан так:

```
DEFINE_PER_CPU(int[3], my_percpu_array);
```

По-процессорными переменными можно манипулировать почти без явного блокирования. Помните, что ядро версии 2.6 является вытесняющим; это бы не следует делать в середине критической секции, которая модифицирует по-процессорную переменную. Также будет не хорошо, если ваш процесс будет перемещён в другой процессор в середине доступа к по-процессорной переменной. По этой причине вы должны явно использовать макрос ***get_cpu_var*** для непосредственного доступа к данной переменной текущего процессора и вызвать ***put_cpu_var*** по окончании. Вызов ***get_cpu_var*** возвращает величину для прямого доступа к текущей процессорной версии переменной и отключает вытеснение. Поскольку возвращается сама переменная, она может быть напрямую использоваться для присваивания или управления. Например, один счётчик в сетевом коде увеличивается этими двумя командами:

```
get_cpu_var(sockets_in_use)++;
put_cpu_var(sockets_in_use);
```

Вы можете получить доступ другой процессорной копии переменной с помощью:

```
per_cpu(variable, int cpu_id);
```

Если вы пишете код, который включает в себя доступ к значению каждой по-процессорной переменной, вам, конечно, необходимо реализовать схему блокировки, чтобы сделать доступ безопасным.

Динамическое создание по-процессорной переменной также возможно. Эти переменные могут быть созданы так:

```
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
```

В большинстве случаев ***alloc_percpu*** делает свою работу, вы можете вызвать ***__alloc_percpu*** в случаях, когда требуется особое выравнивание. В любом случае, по-процессорная переменная может быть возвращена системе с помощью ***free_percpu***. Доступ к динамически созданной по-процессорной переменной осуществляется через ***per_cpu_ptr***:

```
per_cpu_ptr(void *per_cpu_var, int cpu_id);
```

Этот макрос возвращает указатель на версию ***per_cpu_var***, соответствующей данному **cpu_id**. Если вы просто читаете другую процессорную версию переменной, вы можете разыменовать указатель и что-то сделать с ней. Однако, если вы манипулируете текущей процессорной версией, то вероятно, сначала надо убедиться, что вы не можете быть передвинуты из этого процессора. Если весь доступ к по-процессорной переменной происходит с удержанием спин-блокировки, всё хорошо. Однако, обычно для блокировки вытеснения при работе с переменной вам необходимо использовать ***get_cpu***. Таким образом, код, использующий динамические по-процессорные переменные, как правило, выглядит следующим образом:

```
int cpu;

cpu = get_cpu( )
ptr = per_cpu_ptr(per_cpu_var, cpu);
/* работаем с ptr */
put_cpu( );
```

При использовании по-процессорных переменных во время компиляции, макросы ***get_cpu_var*** и ***put_cpu_var*** заботятся об этих деталях. Динамические по-процессорные переменные требуют более явной защиты.

По-процессорные переменные могут быть экспортированы в модули, но вы должны использовать специальную версию макроса:

```
EXPORT_PER_CPU_SYMBOL(per_cpu_var);
EXPORT_PER_CPU_SYMBOL_GPL(per_cpu_var);
```

Для доступа к таким переменным внутри модуля, объявите её так:

```
DECLARE_PER_CPU(type, name);
```

Использование **DECLARE_PER_CPU** (вместо **DEFINE_PER_CPU**) сообщает компилятору, что создаётся внешняя ссылка.

Если вы хотите использовать по-процессорные переменные для создания простого целочисленного счётчика, взгляните на заготовленную реализацию в ***<linux/percpu_counter.h>***. Наконец, отметим, что некоторые архитектуры имеют ограниченный объём адресного пространства, доступного для по-процессорных переменных. Если вы создаёте по-процессорные переменные, вы должны попытаться сохранить их число небольшим.

Получение больших буферов

Как мы уже отмечали в предыдущих разделах, выделение больших непрерывных буферов памяти может привести к неудачам. Системная память фрагментируется с течением времени и есть вероятность, что действительно большая область памяти просто не будет доступна. Поскольку, как правило, есть способы выполнения работы без огромных буферов, разработчики ядра не дают высокого приоритета для выполнения работы большого выделения памяти. Прежде чем пытаться получить большую область памяти, вам следует всерьёз задуматься об альтернативах. Самым лучшим способом выполнения больших операций ввода/вывода являются операции разборки/сборки, которые мы обсуждаем в разделе

"Преобразования разборки/сборки" 432 в Главе 15 395.

Получение выделенного буфера во время загрузки

Если вам действительно необходим огромный буфер физически непрерывной памяти, часто лучший подход - выделить запрашиваемую память во время загрузки. Выделение памяти во время загрузки является только способом получить последовательные страницы памяти в обход ограничений, вводимых ***__get_free_pages*** на размер буфера, как с точки зрения максимально допустимого размера, так и ограниченного выбора размеров. Выделение памяти при загрузке является "грязной" техникой, потому что она обходит все политики управления памятью, резервируя частный пул памяти. Эта техника является негибкой и безвкусной, но она также наименее подвержена неудаче. Нет необходимости говорить, что модуль не может выделять память во время загрузки; только драйверы, непосредственно связанные с ядром, могут сделать это.

Заметной проблемой с выделением памяти при загрузке является то, что она не является реальным вариантом для обычного пользователя, поскольку этот механизм доступен только для кода, связанным с образом ядра. Драйвер устройства, использующий этот тип выделения памяти, может быть установлен или заменён только пересборкой ядра и перезагрузкой компьютера.

При загрузке ядра он получает доступ ко всей физической памяти, доступной в системы. Затем он инициализирует каждую из её подсистем, вызывая функции инициализации подсистемы, разрешая коду инициализации выделить буфер памяти для частного использования, сокращая объём оперативной памяти, остающийся для обычных системных операций.

Выделение памяти во время загрузки выполняется вызовом одной из следующих функций:

```
#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 целые страницы (если они заканчиваются ***_pages***) или не странично-выровненные области памяти. Выделенная память может быть верхней областью памяти, если не используется одна из ***_low*** версий. При выделении этого буфера для драйвера устройства, вы, вероятно, захотите использовать его для операций DMA, а это не всегда возможно с верхней областью памяти; таким образом, вы, вероятно, захотите использовать один из ***_low*** вариантов.

Освобождение памяти, выделенной при загрузке, редко; вы почти наверняка будете не в состоянии получить её позже, если этого захотите. Однако, существует интерфейс для освобождения этой память:

```
void free_bootmem(unsigned long addr, unsigned long size);
```

Обратите внимание, что неполные страницы, освобождаемые таким образом, не возвращаются к системе, но если вы используете эту технику, вы, вероятно, выделяется довольно много целых страниц с самого начала.

Если вы должны использовать выделение памяти во время загрузки, необходимо связать свой драйвер непосредственно с ядром. Просмотрите файлы в исходных текстах ядра в ***Documentation/kbuild*** для получения дополнительной информации о том, как это должно быть сделано.

Краткая справка

Функциями и символами, связанными с выделением памяти, являются:

#include <linux/slab.h>
void *kmalloc(size_t size, int flags);
void kfree(void *obj);
Наиболее часто используемый интерфейс для выделения памяти.

#include <linux/mm.h>
GFP_USER
GFP_KERNEL
GFP_NOFS
GFP_NOIO
GFP_ATOMIC
Флаги, которые контролируют, как выполняется распределение памяти, начиная с наименее для всего ограничительного. **GFP_USER** и **GFP_KERNEL**, по порядку, позволяют текущему процессу быть помещённым в сон для удовлетворения запроса. **GFP_NOFS** и **GFP_NOIO** запрещают операции с файловой системой и все операции ввода/вывода соответственно, а выделение памяти с **GFP_ATOMIC** совсем не может спать.
__GFP_DMA
__GFP_HIGHMEM
__GFP_COLD
__GFP_NOWARN
__GFP_HIGH
__GFP_REPEAT
__GFP_NOFAIL
__GFP_NORETRY
Эти флаги изменяют поведение ядра при выделении памяти.

#include <linux/malloc.h>
kmem_cache_t *kmem_cache_create(char *name, size_t size, size_t offset,
unsigned long flags, constructor( ), destructor( ));
int kmem_cache_destroy(kmem_cache_t *cache);
Создают и уничтожают кусковый кэш. Кэш может быть использован для выделения нескольких объектов одинакового размера.
SLAB_NO_REAP
SLAB_HWCACHE_ALIGN
SLAB_CACHE_DMA
Флаги, которые могут быть указаны при создании кэша.
SLAB_CTOR_ATOMIC
SLAB_CTOR_CONSTRUCTOR
Флаги, которые распределитель может передать в функции конструктора и деструктора.
void *kmem_cache_alloc(kmem_cache_t *cache, int flags);

void kmem_cache_free(kmem_cache_t *cache, const void *obj);
Выделяет и освобождает память для одного объекта из кэша.

/proc/slabinfo
Виртуальный файл, содержащий статистические данные об использовании кускового кэша.

#include <linux/mempool.h>
mempool_t *mempool_create(int min_nr, mempool_alloc_t *alloc_fn,
mempool_free_t *free_fn, void *data);
void mempool_destroy(mempool_t *pool);
Функции для создания пулов памяти, которые пытаются избежать неудач при выделении памяти, сохраняя "чрезвычайный список" выделенных объектов.
void *mempool_alloc(mempool_t *pool, int gfp_mask);
void mempool_free(void *element, mempool_t *pool);
Функции для выделения объектов из (и их возвращения в) пулов памяти.
unsigned long get_zeroed_page(int flags);
unsigned long __get_free_page(int flags);
unsigned long __get_free_pages(int flags, unsigned long order);
Странично-ориентированные функции выделения памяти. ***get_zeroed_page*** возвращает единственную, обнулённую страницу. Все другие варианты вызова не инициализируют содержимое возвращаемой страниц(ы).
int get_order(unsigned long size);
Возвращает порядок (order) выделения, связанный с размером в текущей платформы в соответствии с **PAGE_SIZE**. Аргумент должен быть степенью двойки и возвращаемое значение по крайней мере 0.
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
Функции, которые используются для освобождения памяти при странично-ориентированный выделениях.
struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);
struct page *alloc_pages(unsigned int flags, unsigned int order);
struct page *alloc_page(unsigned int flags);
Все варианты самого низкоуровнего распределителя страниц в ядре Linux.
void __free_page(struct page *page);
void __free_pages(struct page *page, unsigned int order);
void free_hot_page(struct page *page);
void free_cold_page(struct page *page);
Различные способы освобождения страниц, выделенных одной из форм ***alloc_page***.

#include <linux/vmalloc.h>
void * vmalloc(unsigned long size);
void vfree(void * addr);

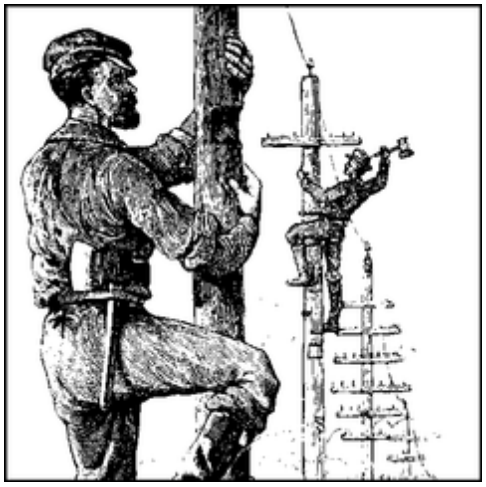
#include <asm/io.h>
void * ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);
Функции, которые выделяют или освобождают непрерывное виртуальное адресное пространство. ***ioremap*** адресует физическую память с помощью виртуальных адресов, а ***vmalloc*** выделяет свободные страницы. Области, связанные с ***ioremap***,

освобождаются ***iounmap***, а страницы, полученные ***vmalloc***, освобождаются ***vfree***.

#include <linux/percpu.h>
DEFINE_PER_CPU(type, name);
DECLARE_PER_CPU(type, name);
Макросы, которые определяют и декларируют по-процессорные переменные.
per_cpu(variable, int cpu_id)
get_cpu_var(variable)
put_cpu_var(variable)
Макросы, которые обеспечивают доступ к статически объявленным по-процессорным переменным.
void *alloc_percpu(type);
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(void *variable);
Функции, которые выполняют выделение и освобождение во время работы с по-процессорными переменными.
int get_cpu( );
void put_cpu( );
per_cpu_ptr(void *variable, int cpu_id)
get_cpu получает ссылку (на переменную) для текущего процессора (следовательно, предотвращает вытеснение и переход к другому процессору) и возвращает идентификатор процессора; ***put_cpu*** возвращает эту ссылку (то есть переменную в процессор). Чтобы получить доступ к динамически выделенной по-процессорной переменной, используйте ***per_cpu_ptr*** с идентификатором процессора, версия (переменной) которого должна быть доступна. Манипуляции текущей версией по-процессорной переменной должны, вероятно, быть окружены вызовами ***get_cpu*** и ***put_cpu***.

#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);
void free_bootmem(unsigned long addr, unsigned long size);
Функции (которые могут быть использованы только драйверами непосредственно встроенными в ядро), которые осуществляют выделение и освобождение памяти во время начальной загрузки системы.

Глава 9, Взаимодействие с аппаратными средствами



Хотя игра со ***scull*** и подобными игрушками является хорошим знакомством с интерфейсом программного обеспечения драйвера Linux, реализация ***реального*** устройства требует оборудования. Драйвер является абстрактным слоем между программными концепциями и схемой аппаратуры; как таковой, он должен разговаривать с ними обоими. До сих пор мы рассматривали внутренности программных концепций; эта глава завершает картину, показывая, как драйвер может доступаться к портам ввода/вывода и памяти ввода/вывода, одновременно оставаясь переносимым между платформами Linux.

Эта глава продолжает традицию оставаться независимыми от специфического оборудования, насколько это возможно. Вместе с тем, где требуются конкретные примеры, мы используем простые цифровые порты ввода/вывода (такие, как стандартный параллельный порт ПК), чтобы показать, как работают инструкции ввода/вывода и обычный кадровый буфер видео памяти, чтобы показать ввод/вывод, связанный с памятью.

Мы выбрали простой цифровой ввод/вывод, поскольку это самая простая форма порта ввода/вывода. Кроме того, параллельный порт реализует сырой ввод/вывод и доступен в большинстве компьютеров: биты данных, записываемые в устройство, появляются на выходных контактах и уровни напряжения на входных контактах прямо доступны процессору. На практике вы должны подключить к порту светодиоды или принтер, чтобы реально увидеть результаты операций цифрового ввода/вывода, но базовое оборудования предельно просто в использовании.

Порты ввода/вывода и память ввода/вывода

Каждое периферийное устройство управляется записью и чтением его регистров. В большинстве случаев устройство имеет несколько регистров и они доступны по последовательным адресам либо в адресном пространстве памяти, либо в адресном пространстве ввода/вывода.

На аппаратном уровне нет концептуальной разницы между областями памяти и областями ввода/вывода: обе они доступны установкой электрических сигналов на адресной шине и шине управления (то есть сигналами ***чтения*** и ***записи***) (* Не все компьютерные платформы используют сигнал *чтения* и *записи*; некоторые имеют другие средства для адресации внешних цепей. Однако, на программном уровне разница не имеет значения и для упрощения

обсуждения мы будем предполагать, что все имеют *чтение* и *запись*.) и чтением или записью на шине данных.

Хотя некоторые производители процессоров реализовали в своих чипах единое адресное пространство, другие решили, что периферийные устройства отличаются от памяти и, следовательно, заслуживают отдельного адресного пространства. Некоторые процессоры (прежде всего семейство x86) имеют раздельные электрические линии ***чтения*** и ***записи*** для портов ввода/вывода и специальные инструкции процессора для доступа к портам.

Так как периферийные устройства изготовлены с учётом периферийных шин и на персональном компьютере смоделированы самые популярные шины ввода/вывода, даже процессоры, которые не имеют отдельного адресного пространства для портов ввода/вывода, должны подделывать чтение и запись портов ввода/вывода при доступе к некоторым периферийным устройствам, обычно, с помощью внешних чипсетов (наборов микросхем) или дополнительных схем в ядре процессора. Последнее решение распространено в крошечных процессорах, предназначенных для встроенного использования.

По той же причине Linux реализует концепцию портов ввода/вывода на всех компьютерных платформах, где работает, даже на платформах, где процессор реализует единое адресное пространство. Осуществление доступа к порту иногда зависит от особенностей реализации и модели данного компьютера (потому что разные модели используют разные чипсеты для связывания шинных транзакций с адресным пространством памяти).

Даже если периферийная шина имеет отдельное адресное пространство для портов ввода/вывода, не все устройства связывают свои регистры с портами ввода/вывода. В то время, как использование портов ввода/вывода распространено для периферийных плат ISA, большинство устройств PCI связывают регистры с областью адресов в памяти. Такой подход с вводом/выводом в память является наиболее предпочтительным, поскольку он не требует использования инструкций процессора специального назначения; процессорные ядра выполняют доступ к памяти намного более эффективно и при обращении к памяти компилятор имеет гораздо больше свободы в распределении регистров и выборе режима адресации.

Регистры ввода/вывода и обычная память

Несмотря на сильное сходство между аппаратными регистрами и памятью, программист, адресующий регистры ввода/вывода, должен быть осторожен, чтобы избежать обмана оптимизацией в процессоре (или компиляторе), которые могут изменить ожидаемое поведение ввода/вывода.

Основным различием между регистрами ввода/вывода и ОЗУ является то, что операции ввода/вывода имеют побочные эффекты, а операции с памятью не имеют никакого: единственным эффектом записи в память является сохранение значения по адресу, а чтение из памяти возвращает последнее записанное туда значение. Так как скорость доступа к памяти является столь важной для производительности процессора, случай "без-побочных-эффектов" был оптимизирован несколькими способами: значения кэшируются и инструкции чтения/записи переупорядочиваются.

Компилятор может кэшировать значения данных в регистрах процессора без записи их в память и даже если он сохраняет их, обе операции записи и чтения могут выполняться в кэш-памяти не достигая физической памяти. Реорганизация может случиться как на уровне компилятора, так и на аппаратном уровне: часто последовательность инструкций может быть выполнена быстрее, если она выполняется в порядке, отличном от того, который появляется в

тексте программы, например, для предотвращения блокировки в конвейере RISC. На процессорах CISC операции, которые занимают значительное количество времени, могут быть выполнены одновременно с другими более быстрыми.

Эти оптимизации являются прозрачными и благоприятными, когда применяются к обычной памяти (по крайней мере на однопроцессорных системах), однако они могут быть фатальными для правильных операций ввода/вывода, потому что они сталкиваются с этими "побочными эффектами", что является основной причиной, почему драйверы адресуют регистры ввода/вывода. Процессор не может прогнозировать ситуацию, в которой некоторые другие процессы (работающие на отдельном процессоре, или нечто происходящее внутри контроллера ввода/вывода), зависят от порядка доступа к памяти. Компилятор или процессор может просто пытаться перехитрить вас и переупорядочить операции, которые вы запросили; результатом могут быть странные ошибки, которые очень трудно выявить. Таким образом, драйвер должен обеспечить, чтобы при доступе к регистрам кэширование не производилось и не имело место переупорядочивание чтения или записи.

Проблема с аппаратным кэшированием является самой простой из стоящих: нижележащее оборудование уже настроено (автоматически или кодом инициализации Linux) для отключения любого аппаратного кэширования при доступе к областям ввода/вывода (независимо от того, являются ли они областями памяти или портов).

Решением для (избегания) оптимизации компилятором и аппаратного переупорядочивания является размещение барьера памяти между операциями, которые должны быть видимыми для аппаратуры (или другим процессором) в определённом порядке. Linux предоставляет четыре макроса для покрытия всех возможных потребностей упорядочивания:

#include <linux/kernel.h>
void barrier(void)

Эта функция говорит компилятору вставить барьер памяти, но не влияет на оборудование. Скомпилированный код сохраняет в память все значения, которые в настоящее время модифицированы и постоянно находятся в регистрах процессора, и перечитывает их позже, когда они необходимы. Вызов ***barrier*** препятствует оптимизациям компилятора пересечь этот барьер, но оставляет свободу оборудованию делать своё собственное переупорядочивание.

#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);

Эти функции вставляют аппаратные барьеры памяти в поток скомпилированных инструкций; их фактическая реализация зависит от платформы. ***rmb*** (барьер чтения памяти) гарантирует, что любые чтения, находящиеся до барьера, завершатся до выполнения любого последующего чтения. ***wmb*** гарантирует порядок в операциях записи, а инструкция ***mb*** гарантирует оба порядка. Каждая из этих функций является надстройкой над ***barrier***.
read_barrier_depends является особой более слабой формой барьера чтения. Если ***rmb*** препятствует переупорядочиванию всех чтений через барьер, ***read_barrier_depends*** блокирует только переупорядочивание операций чтения, которые зависят от данных других операций чтения. Различие тонкое и оно не существует на всех архитектурах. Если вы не понимаете точно, что происходит, и у вас нет оснований полагать, что полный барьер чтения чрезмерно уменьшает производительность, вам, вероятно, следует

придерживаться использования ***rmb***.

void smp_rmb(void);

void smp_read_barrier_depends(void);

void smp_wmb(void);

void smp_mb(void);

Эти версии барьерных макросов вставляют аппаратные барьеры только тогда, когда ядро скомпилировано для многопроцессорных систем; в противном случае, все они становятся простым вызовом ***barrier***.

Типичное использование барьеров памяти в драйвере устройства может иметь такую форму:

```
writel(dev->registers.addr, io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb( );
writel(dev->registers.control, DEV_GO);
```

В этом случае важно убедиться, что все регистры устройства, контролирующие отдельные операции, были надлежащим образом установлены перед командами начала работы. Барьер памяти обеспечивает завершение команд записи в необходимом порядке.

Так как барьеры памяти влияют на производительность, они должны использоваться только там, где они действительно необходимы. Разные виды барьеров могут также иметь разные характеристики производительности, поэтому стоит использовать по возможности наиболее подходящий тип. Например, на архитектуре x86 ***wmb( )*** в настоящее время ничего не делает, поскольку за пределами процессора записи не переупорядочиваются. Чтение является переупорядочиваемым, поэтому ***mb( )*** работает медленнее, чем ***wmb( )***.

Стоит отметить, что большинство других примитивов ядра, имеющих дело с синхронизацией, такие, как операции спи-блокировки и **atomic_t**, также функционируют как барьеры памяти. Также следует отметить, что некоторые периферийные шины (такие, как шины PCI) имеют собственные проблемы кэширования; мы обсудим их, когда мы доберёмся до них в последующих главах.

Некоторые архитектуры позволяют эффективно комбинировать присваивание и барьер памяти. Ядро предоставляет несколько макросов, которые выполняют эту комбинацию; в стандартном случае они определяются следующим образом:

```
#define set_mb(var, value) do {var = value; mb( );} while 0
#define set_wmb(var, value) do {var = value; wmb( );} while 0
#define set_rmb(var, value) do {var = value; rmb( );} while 0
```

Где уместно, ***<asm/system.h>*** определяет эти макросы для использования архитектурно-зависимых инструкций, которые выполняют задачу более быстро. Обратите внимание, что ***set_rmb*** определена лишь на небольшом числе архитектур. (Использование конструкции **do...while** является стандартной идиомой языка Си, которая заставляет раскрытый макрос во всех контекстах работать как обычный оператор Си.)

Использование портов ввода/вывода

Порты ввода/вывода являются средством, с помощью которого драйверы общаются со многими устройствами, по крайней мере, часть времени. Настоящий раздел посвящён различным функциям, доступным для использования портов ввода/вывода; мы также коснёмся некоторых проблем совместимости.

Назначение портов ввода/вывода

Как и следовало ожидать, вы не должны пойти и начать простукивать порты ввода-вывода, не убедившись сначала, что вы имеете эксклюзивный доступ к этим портам. Ядро предоставляет интерфейс регистрации, который позволяет вашему драйверу затребовать те порты, которые ему требуются. Основой функцией этого интерфейса является ***request_region***:

```
#include <linux/ioport.h>
struct resource *request_region(unsigned long first, unsigned long n, const
char *name);
```

Эта функция сообщает ядру, что вы хотели бы использовать **n** портов, начиная с **first**. Параметр **name** должен быть именем вашего устройства. Возвращаемое значение не **NULL**, если выделение успешно. Если от ***request_region*** вы получаете обратно **NULL**, вы не сможете использовать желаемые порты.

Все назначенные порты показываются в ***proc/ioports***. Если вы не в состоянии получить необходимый набор портов, то оно является местом для поиска, чтобы узнать, кто получил их первым.

Когда вы закончите работу с набором портов ввода/вывода (возможно, при выгрузке модулей), они должна быть возвращены системе:

```
void release_region(unsigned long start, unsigned long n);
```

Существует также функция, которая позволяет вашему драйверу проверить и увидеть, доступен ли данный набор портов ввода/вывода:

```
int check_region(unsigned long first, unsigned long n);
```

Здесь возвращаемое значение является отрицательным кодом ошибки, если данные порты будут не доступны. Эта функция не рекомендуется к использованию, поскольку возвращаемое ей значение не даёт никаких гарантий, будет ли выделение успешным; проверка и затем выделение не являются атомарной операцией. Мы приводим её здесь потому, что некоторые драйверы продолжают её использовать, но вы должны всегда использовать ***request_region***, которая выполняет необходимые блокировки для того, чтобы распределение осуществлялось безопасным, атомарным образом.

Управление портами ввода/вывода

После того, как драйвер запросил диапазон портов ввода/вывода, необходимых ему для своей деятельности, он должен читать и/или писать в эти порты. В этом в большинстве аппаратуры имеются различия между 8-ми разрядными, 16-ти разрядными и 32-х разрядными портами. Обычно вы не можете смешивать их, как обычно это делается с системой доступа к памяти. (* Иногда порты ввода/вывода организованы как память, и вы можете (к примеру)

связать две 8-ми разрядных операции записи в одну 16-ти разрядную операцию. Это относится, например, к платам видео ПК. Но, как правило, вы не можете рассчитывать на эту особенность.)

Следовательно, программы на Си должны вызывать разные функции для доступа к портам разной размерности. Как предположено в предыдущем разделе, компьютерные архитектуры, которые поддерживают только связанные с памятью регистры ввода/вывода, изображают порт ввода/вывода переназначением адресов портов на адреса памяти и ядро скрывает детали от драйвера, чтобы облегчить переносимость. Заголовки ядра Linux (в частности, архитектурно-зависимый заголовок ***<asm/io.h>***) определяют для доступа к портам ввода/вывода следующие встраиваемые (inline) функции:

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
Чтение и запись байтовых портов (шириной восемь бит). Для одних платформ аргумент **port** определяется как **unsigned long** и **unsigned short** для других. Тип, возвращаемый ***inb***, также различен между архитектурами.

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
Эти функции осуществляют доступ к 16-ти разрядным портам (шириной в одно слово); они не доступны при компиляции для платформы S390, которая поддерживает только байтовый ввод/вывод.

unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
Эти функции осуществляют доступ к 32-х разрядным портам. **longword** объявляется либо как **unsigned long**, или как **unsigned int**, в зависимости от платформы. Как и ввод/вывод по словам, "длинный" ввод/вывод не доступен на S390.

Теперь, когда мы используем **unsigned** без дальнейших уточнений типа, мы имеем в виду зависящее от архитектуры определение, чьё точное определение не является актуальным. Функции почти всегда переносимы, потому что компилятор автоматически приводит значения при присваивании и то, что они **unsigned** помогает предотвратить предупреждения во время компиляции. Информация не будет потеряна при таких привидениях до тех пор, пока программист назначает разумные значения, чтобы избежать переполнения. В этой главе мы придерживаемся такого соглашения "неполной типизации".

Обратите внимание, что для портов ввода/вывода нет 64-х разрядных операций. Даже на 64-х разрядных архитектурах адресное пространство портов использует (максимум) 32-х разрядные данные.

Доступ к портам ввода/вывода из пользовательского пространства

Только что описанные функции в первую очередь предназначены для использования драйверами устройств, но они также могут быть использованы из пользовательского пространства, по крайней мере на компьютерах класса ПК. Библиотека GNU Си определяет их в ***<sys/io.h>***. В целях использования в коде пользовательского пространства для ***inb*** и друзей должны применяться следующие условия:

- Программа должна быть скомпилирована с опцией ***-O***, чтобы заставить раскрыть встраиваемые (inline) функции.
- Для получения разрешения на выполнение операций ввода/вывода на портах должны быть использованы системные вызовы ***ioperm*** или ***iopl***. ***ioperm*** получает разрешение на отдельные порты, а ***iopl*** получает разрешение на всё пространство ввода/вывода. Обе эти функции являются особенностями x86.
- Для вызова ***ioperm*** или ***iopl*** программа должна запускаться с правами суперпользователя. (* Технически, он должен иметь разрешение CAP_SYS_RAWIO, но на большинстве существующих систем это то же самое, как запуск с правами суперпользователя.) Либо один из её предков уже должен иметь доступ к порту, запускаясь с правами суперпользователя.

Если данная платформа не имеет системных вызовов ***ioperm*** и ***iopl***, пространство пользователя всё же может получить доступ к портам ввода/вывода с помощью файла устройства ***/dev/port***. Однако, следует отметить, что содержание файла очень зависит от платформы и вряд ли полезно где-то, кроме ПК.

Исходники примеров ***misc-progs/inp.c*** и ***misc-progs/outp.c*** являются минимальным инструментом для чтения и записи портов из командной строки в пространстве пользователя. Ожидается, что они установлены под разными именами (например, ***inb***, ***inw*** и ***inl*** манипулируют байтами, пословными, или "длинными" портами в зависимости от того, какое имя вызвано пользователем). Под x86 они используют ***ioperm*** или ***iopl***, на других платформах - ***/dev/port***.

Программы могут выполнять setuid суперпользователя, если вы хотите жить опасно и играть с вашим оборудованием без получения явных привилегий. Однако, пожалуйста, не устанавливайте ими setuid на промышленной системе; это являются дырой безопасности по дизайну.

Строковые операции

В дополнение к однократным операциям ввода и вывода некоторые процессоры реализуют специальные инструкции для передачи последовательности байтов, слов или двойных слов в и из одного порта ввода/вывода такой же размерности. Это так называемые ***string instructions (строковые операции)*** и они выполняют задачу быстрее, чем это может сделать цикл языка Си. Следующие макросы реализуют такую концепцию строчного ввода/вывода либо с помощью одной машинной команды, либо выполняя плотный цикл, если целевой процессор не имеет инструкции для выполнения строчного ввода/вывода. При компиляции для платформы S390 макрос не определён совсем. Это не должно быть проблемой переносимости, поскольку эта платформа обычно не разделяет драйверы устройств с другими платформами, так как её периферийные шины являются отличными от других.

Прототипами для строковых функций являются:

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
Чтение и запись **count** байтов, начиная с адреса памяти **addr**. Данные читаются из или записываются в единственный порт **port**.

void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);

Чтение или запись 16-ти разрядных значений в один 16-ти разрядный порт.

void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
Чтение или запись 32-х разрядных значений в один 32-х разрядный порт.

При использовании строковых функций следует иметь в виду одну вещь: они передвигают прямой поток байтов в или из порта. Когда порт и данная система имеют разные правила порядка байтов, результат может удивить. Чтение порта с ***inw*** переставляет байты, если необходимо, чтобы считанное значение соответствовало порядку байт платформы. Строковые функции, наоборот, не выполняют такой перестановки.

Пауза ввода/вывода

Некоторые платформы, прежде всего i386, могут иметь проблемы, когда процессор пытается передать данные слишком быстро в или из шины. Проблемы могут возникнуть, когда процессор разогнан по отношению к периферийной шине (подумайте здесь об ISA) и могут появляться, когда плата устройства является слишком медленной. Решением является вставить небольшую задержку после каждой инструкции ввода/вывода, если за ней следует другая такая же инструкция. На x86 пауза достигается выполнением инструкции **outb** в порт 0x80 (обычно, но не всегда неиспользуемый), или активным ожиданием. Смотрите для подробностей файл ***io.h*** в подкаталоге ***asm*** вашей платформы.

Если ваше устройство пропускает некоторые данные, или если вы опасаетесь, что можете пропустить некоторые, вместо обычных можно воспользоваться функциями с паузой. Функции с паузой точно такие же, как перечисленные выше, но их имена заканчиваются на ***_p***; они называются ***inb_p***, ***outb_p*** и так далее. Эти функции определены для большинства поддерживаемых архитектур, хотя они часто преобразуются в тот же код, как и ввод/вывод без паузы, поскольку нет необходимости для дополнительной паузы, если архитектура работает с достаточно современной периферийной шиной.

Зависимости от платформы

Инструкции ввода/вывода по своей природе очень зависимы от процессора. Поскольку они работают с подробностями того, как процессор обрабатывает перемещения данных туда и оттуда, очень трудно скрыть различия между системами. Как следствие, значительная часть исходного кода, связанная с портом ввода/вывода, зависит от платформы.

Вы можете увидеть одну из несовместимостей, типизацию данных, посмотрев обратно на список функций, где аргументы вводятся по разному в зависимости от архитектурных различий между платформами. Например, порт является **unsigned short** на x86 (где процессор поддерживает пространство ввода-вывода в 64 Кб), но **unsigned long** на других платформах, порты которых являются только специальными местами в том же адресном пространстве памяти.

Другие зависимости от платформы возникают от основных структурных различий в процессорах и, следовательно, неизбежны. Мы не будем вдаваться в подробности различий, потому что мы предполагаем, что вы не будете писать драйверы для определённой системы без понимания базового оборудования. Вместо этого, здесь представлен обзор возможных архитектур, поддерживаемых ядра:

IA-32 (x86)
x86_64

Эта архитектура поддерживает все функции, описанные в этой главе. Номера портов имеет тип **unsigned short**.

IA-64 (Itanium)

Поддерживаются все функции; порты являются **unsigned long** (и отображены на память). Строковые функции являются реализованными на Си.

Alpha

Поддерживаются все функции и порты являются отображёнными на память. Реализация портов ввода/вывода является различной в разных платформах Alpha, в зависимости от используемого чипсета. Строковые функции реализованы на Си и определены в ***arch/alpha/lib/io.c***. Порты являются ***unsigned long***.

ARM

Порты являются отображёнными на память и поддерживаются все функции; строковые функции реализованы на Си. Порты имеют тип **unsigned int**.

Cris

Эта архитектура не поддерживает абстракции порта ввода/вывода даже в режиме эмуляции; различные операции с портами определены как вообще ничего не делающие.

M68k
M68k-nommu

Порты являются отображёнными на память. Поддерживаются строковые функции и типом порта является **unsigned char ***.

MIPS
MIPS64

Порт MIPS поддерживает все функции. Строковые операции реализованы жёстким ассемблерным циклом, потому что процессор не имеет строкового ввода/вывода на машинном уровне. Порты являются отображёнными в память; они являются **unsigned long**.

PA-RISC

Поддерживаются все функции; порты являются **int** на PCI-базирующихся системах и **unsigned short** на системах EISA, за исключением строковых операций, которые используют номера портов **unsigned long**.

PowerPC
PowerPC64

Поддерживаются все функции; порты имеют тип **unsigned char *** на 32-х разрядных системах и **unsigned long** на 64-х разрядных системах.

S390

По аналогии с M68K, заголовок для этой платформы поддерживает только порты ввода/вывода шириной с байт без строковых операций. Порты являются указателями ***char*** и отображены на память.

Super-H

Порты являются **unsigned int** (отображены на память) и поддерживаются все функции.

SPARC
SPARC64

Вновь, пространство ввода/вывода является отображённым на память. Версии функций портов определены для работы с портами типа **unsigned long**.

Любопытный читатель может извлечь больше информации из файлов ***io.h***, которые иногда определяют несколько архитектурно-зависимых функций в дополнение к тем, что мы описываем в этой главе. Предупреждаем однако, что некоторые из этих файлов достаточно сложны для чтения. Интересно отметить, что нет процессоров вне семейства x86, имеющих другое адресное пространство для портов, хотя некоторые из поддерживаемых семейств снабжены слотами с ISA и/или PCI (и обе шины реализуют отдельные адресные пространства для ввода/вывода и памяти).

Кроме того, некоторые процессоры (особенно ранние Alpha) не имели инструкций, которые передвигали один или два байта за один раз. (* Однобайтовый ввод/вывод не так важен, как можно себе представить, потому что это редкая операция. Для чтения/записи одного байта в любом адресном пространстве необходимо реализовать путь для данных, соединяющий младшие биты регистрового набора шины данных с любой позицией байта во внешней шине данных. Эти пути данных требуют дополнительных логических шлюзов для получения пути для каждой передачи данных. Удаление загрузок и сохранений размером с байт может принести пользу общей производительности системы.) Таким образом, их периферийные чипсеты имитировали 8-ми разрядные и 16-ти разрядные доступы ввод/вывода связывая их со специальным диапазоном адресов в адресном пространстве памяти. Таким образом, инструкции ***inb*** и ***inw***, которые обращаются к одному порту, реализованы двумя чтениями 32-х разрядной памяти работающими с разными адресами. К счастью, всё это скрыто от автора драйверов устройств внутри макросов, описанных в этом разделе, но мы считаем, что эту интересную особенность стоит принять к сведению. Если вы захотите поисследовать дальше, поищите примеры в ***include/asm-alpha/core_lca.h***.

Какие операции ввода/вывода выполняются на каждой платформе хорошо описано в руководстве программиста для каждой платформы; как правило, эти руководства доступны для скачивания в Интернете как PDF-файлы.

Пример порта ввода/вывода

Мы используем код примера, чтобы показать как порт ввода/вывода внутри драйвера устройства взаимодействует с цифровыми портами ввода/вывода общего назначения; такие порты имеются в большинстве компьютерных систем.

Цифровой порт ввода/вывода в самом общем воплощении является адресом байта для ввода/вывода, связанным либо с памятью, либо с портом. Когда вы записываете значение по адресу вывода, электрический сигнал, видимый на выходных контактах, изменяется в соответствии с отдельными записанными битами. Когда вы читаете значения из входного адреса, текущий логический уровень, видимый на входных контактах, возвращается в виде

отдельных битовых значений.

Фактическая реализация и программный интерфейс таких портов ввода/вывода изменяется от системы к системе. Большую часть времени контакты ввода/вывода управляются двумя адресами ввода/вывода: один, который позволяет выбрать, какие контакты используются для ввода данных и какие для вывода, и второй, по которому можно действительно читать или писать логические уровни. Иногда, однако, всё ещё проще и биты в оборудовании либо входные, либо выходные (но этом случае они больше не называются "вводом/выводом общего (универсального) назначения"); параллельный порт, имеющийся на всех персональных компьютерах является одним из таких портов ввода/вывода не столь общего назначения. В любом случае, контакты ввода/вывода пригодны для использования кодом примера, который мы представим в ближайшее время.

Обзор параллельного порта

Поскольку мы ожидаем, что большинство читателей используют платформу x86 в виде так называемого "персонального компьютера", мы считаем, что стоит объяснить, каким образом разработан параллельный порт ПК. Параллельный порт является тем периферийным интерфейсом, который выбран для запуска кода примера цифрового ввода/вывода на персональном компьютере. Хотя большинству читателей спецификации параллельного порта, вероятно, доступны, для вашего удобства мы просуммируем их здесь.

Параллельный интерфейс в минимальной конфигурации (мы рассматриваем режимы ECP и EPP) состоит из трех 8-ми разрядных портов. Стандарт ПК располагает порты ввода/вывода для первого параллельного интерфейса с **0x378**, а второго с **0x278**. Первый порт является двунаправленным регистром данных; он непосредственно связан с контактами 2 - 9 на физическом разъёме. Второй порт является доступным только для чтения регистром статуса; когда параллельный порт используется для принтера, этот регистр сообщает ряда аспектов состояния принтера, таких как "подключён", "нет бумаги", или "занят". Третий порт является только выходным регистром управления, который, среди прочего, контролирует, разрешены ли прерывания.

Уровни сигналов, используемые в параллельных соединениях, являются стандартными уровнями транзисторно-транзисторной логики (ТТЛ): 0 и 5 вольт, с логическим порогом около 1.2 вольта. Вы можете рассчитывать на порты по крайней мере соответствующие требованиям стандарту тока нагрузки для TTL LS, хотя наиболее современные параллельные порты делаются лучше для обоих значений токов и напряжений.

Параллельный разъём не изолирован от внутренней схемы компьютера, что было бы полезно, если вы хотите соединить логические формирователи прямо к порту. Но надо быть осторожным, чтобы сделать подключение правильно; схема параллельного порта легко повреждается, когда вы играете с собственной схемой, пока вы не добавите оптоизоляторы к вашей схеме. Вы можете выбрать использование подключаемых параллельных портов, если боитесь, что можете повредить материнскую плату.

Назначение битов показано на Рисунке 9-1. Вы можете получить доступ к 12 выходным битам и 5 входным битам, некоторые из которых являются логически инвертированными по пути своего сигнала. Одним из битов, не связанным с сигнальным контактом, является бит 4 (0x10) порта 2, который разрешает прерывания от параллельного порта. Мы используем этот бит в рамках реализации нами обработчика прерываний в Главе 10 273.

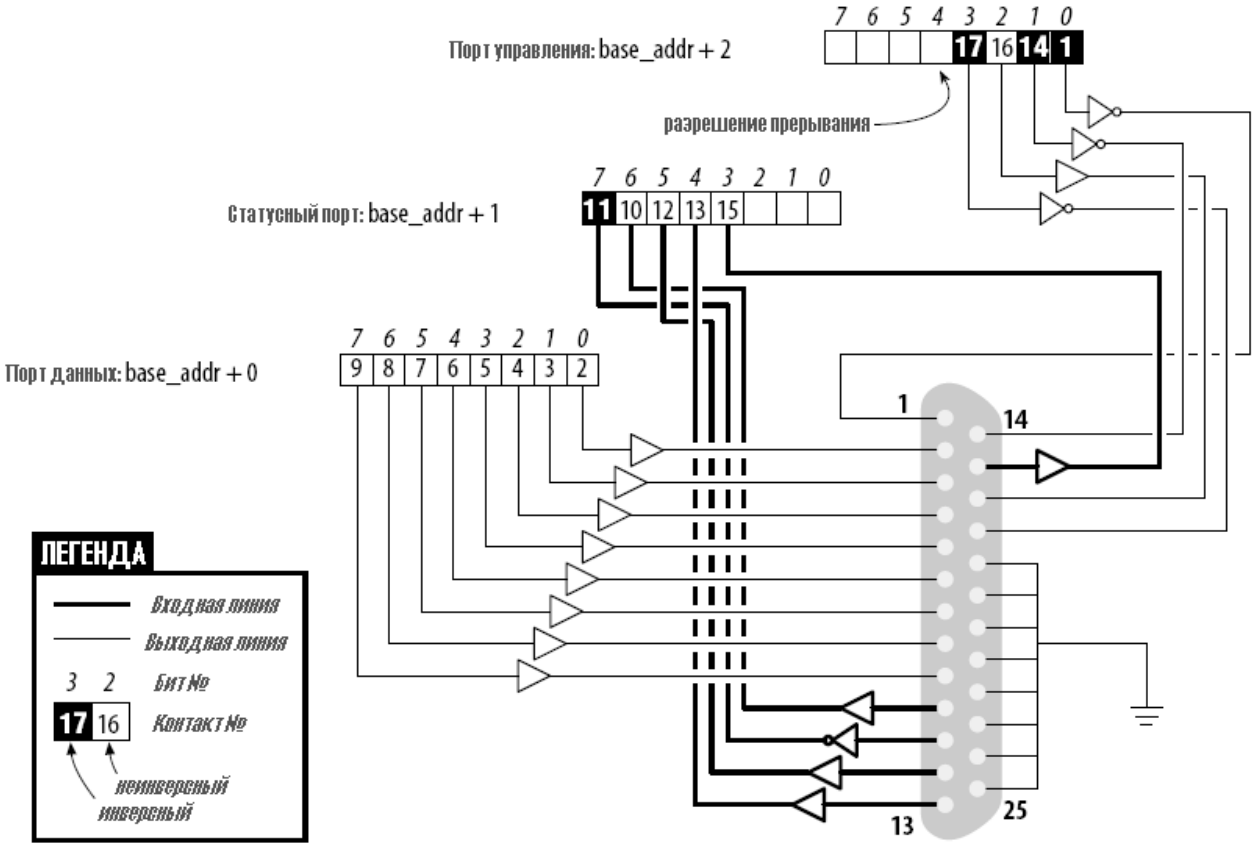


Рисунок 9-1. Назначение выводов параллельного порта

Пример драйвера

Драйвер, который мы представляем, называется ***short*** (Simple Hardware Operations and Raw Tests, Простые Аппаратные Операции и Сырые Тесты). Всё это делается через чтение и запись нескольких 8-ми разрядных портов, начиная с выбранного во время загрузки. По умолчанию он использует диапазон портов, определённых для параллельного интерфейса ПК. Каждый узел устройства (с уникальным младшим номером) обращается к своему порту. Драйвер ***short*** не делает ничего полезного; для внешнего использования он просто представляет одну инструкцию, воздействующую на порт. Если вы не работали с портом ввода/вывода, вы можете использовать ***short***, чтобы познакомиться с ним; вы можете измерить время, которое требуется для передачи данных через порт или поиграть в другие игры.

Для работы ***short*** на вашей системе он должен иметь свободный доступ к нижележащему аппаратному устройству (по умолчанию, параллельному интерфейсу); поэтому никакой другой драйвер не сможет получить его. Большинство современных дистрибутивов устанавливают драйверы параллельного порта как модули, загружаемые только когда это необходимо, так что обычно нет борьбы за адреса ввода/вывода. Однако, если вы получаете ошибку "невозможно получить адрес ввода/вывода" от ***short*** (в консоли или в файле системного журнала), какой-то другой драйвер, наверное, уже забрал порт. Быстрый просмотр ***/proc/ioports*** обычно сообщает, какой драйвер мешает. То же предостережение относится и к другим устройствам ввода/вывода, если вы не используете параллельный интерфейс. С этого момента мы просто ссылаемся на "параллельный интерфейс" для упрощения обсуждения. Однако, вы можете

установить базовый параметр модуля во время загрузки для перенаправления ***short*** на другие устройства ввода/вывода. Эта функция позволяет коду примера работать на любой платформе Linux, где у вас есть доступ к цифровому интерфейсу ввода/вывода, который доступен через ***outb*** и ***inb*** (даже при том, что реальное оборудование является отображённым на память на всех платформах, кроме x86). Позже, в разделе "Использование памяти ввода/вывода" 237, мы покажем также, как ***short*** может быть использован универсальным связанным с памятью цифровым вводом/выводом.

Чтобы посмотреть, что происходит на параллельном разъёме и если у вас есть небольшая наклонность для работы с оборудованием, вы можете припаять на выходные контакты несколько светодиодов. Каждый светодиод должен быть подключен последовательно с 1 кОм резистором, подключенным к земляному контакту (если, конечно, ваши светодиоды не имеют встроенного резистора). Если вы подключите выходной контакт ко входному, вы будете генерировать собственный входной сигнал для чтения из входных портов.

Обратите внимание, что вы не можете просто подключить принтер к параллельному порту и посмотреть данные, передаваемые в ***short***. Этот драйвер реализует простой доступ к портам ввода/вывода и не выполняет установку связи, необходимую принтеру для работы с данными. В следующей главе мы покажем образец драйвера (названного ***shortprint***), который способен управлять параллельными принтерами; однако, такой драйвер использует прерывания, поэтому мы ещё не можем добраться до него.

Если вы собираетесь смотреть параллельные данные, припаяв светодиоды к разъёму D типа, мы рекомендуем вам не использовать контакты 9 и 10, потому что мы соединим их вместе, чтобы позднее запускать код примера, показанный в Главе 10 246.

По отношению к ***short***, ***/dev/short0*** пишет и читает из 8-ми разрядного порта, расположенного по адресу ввода/вывода **base** (0x378, если не изменено во время загрузки). ***/dev/short1*** пишет в 8-ми разрядный порт, расположенный по **base** + 1, и так далее до **base** + 7.

Фактические операции вывода выполняются ***/dev/short0***, основанного на плотном цикле, использующем ***outb***. Используется инструкция барьера памяти, чтобы гарантировать, что операция вывода действительно имеет место и не оптимизирована снаружи:

```
while (count--) {
    outb(*(ptr++), port);
    wmb( );
}
```

Чтобы зажечь ваши светодиоды, вы можете выполнить следующую команду:

```
echo -n "any string" > /dev/short0
```

Каждый светодиод мониторит один бит выходного порта. Помните, что только последний записанный символ остаётся стоящим на выходных контактах достаточно долго, чтобы быть воспринятым вашими глазами. По этой причине мы рекомендуем вам предотвратить автоматическую вставку завершающей команды новой строки, передав в ***echo*** опцию ***-n***.

Чтение выполняется аналогичной функцией, построенной вокруг ***inb*** вместо ***outb***. Для чтения "значимых" значений из параллельного порта вы должны иметь какое-то оборудование, подключенное к входным выводам разъёма для генерации сигналов. Если нет сигнала, читается бесконечный поток одинаковых байтов. Если вы решили читать выходной порт, вы,

скорее всего, получите обратно последнее значение, записанное в порт (это относится и к параллельному интерфейсу и к большинству других схем цифрового ввода/вывода общего назначения). Таким образом, те, кто не склонен выключать свои паяльники, смогут прочитать текущее выходное значение порта 0x378, выполнив такую команду:

```
dd if=/dev/short0 bs=1 count=1 | od -t x1
```

Чтобы продемонстрировать использование всех инструкций ввода/вывода, есть три варианта каждого устройства ***short***: ***/dev/short0*** выполняет только что показанный цикл, ***/dev/short0p*** использует ***outb_p*** и ***inb_p*** вместо "быстрых" функций и ***/dev/short0s*** использует строковые инструкции. Есть восемь таких устройств, от ***short0*** до ***short7***. Хотя интерфейс параллельного имеет только три порта, может потребоваться больше, если для запуска ваших тестов используется другое устройство ввода/вывода.

Драйвер ***short*** выполняет абсолютный минимум управления оборудованием, но достаточен для показа использования инструкций ввода/вывода порта. Заинтересованные читатели могут захотеть посмотреть исходники модулей ***parport*** и ***parport_pc***, чтобы увидеть, насколько сложным это устройство может получиться в реальной жизни, чтобы поддерживать на параллельном порту целый ряд устройств (принтеры, ленточные накопители для резервного копирования, сетевые интерфейсы).

Использование памяти ввода/вывода

Несмотря на популярность портов вводы/вывода в мире x86, основным механизмом, используемым для обмена данными с устройствами, являются связанные с памятью регистры и память устройства. Оба называются памятью ввода/вывода, так как разница между регистрами и памятью для программного обеспечения не видна.

Память ввода/вывода является просто областью, похожей на оперативную память, что делает устройство доступным процессору по шине. Эта память может быть использована для разных целей, таких, как удержание видео данные или сетевых пакетов, а также реализации регистров оборудования, которые ведут себя так же, как порты ввода/вывода (то есть, они имеют побочные эффекты, связанные с чтением и записью в них).

Способ доступа к памяти ввода/вывода зависит от компьютерной архитектуры, шины и используемого устройства, хотя принципы везде одинаковы. Обсуждение в этой главе касается в основном памяти ISA и PCI, а также пытается передать общую информацию. Хотя доступ к памяти PCI приводится здесь, тщательное обсуждение PCI откладывается до Главы 12 288.

В зависимости от компьютерной платформы и используемой шины, память ввода/вывода может или не может быть доступна через таблицы страниц. Когда доступ происходит через таблицы страниц, ядро должно сначала принять меры, чтобы физический адрес был виден из вашего драйвера и это обычно означает, что вы должны вызвать ***ioremap***, прежде чем делать какой-либо ввод/вывод. Если нет необходимых таблиц страниц, память ввода/вывода выглядит точно так же, как порты ввода/вывода, и вы можете просто читать и писать в них с помощью собственных интерфейсных функций.

Независимо от того, требуется или нет для доступа к памяти ввода/вывода ***ioremap***, непосредственное использование указателей на память ввода/вывода не рекомендуется. Хотя даже если (как описывалось в разделе "Порты ввода/вывода и память ввода/вывода" 224) память ввода/вывода адресуется на аппаратном уровне как обычное ОЗУ, дополнительная

предосторожность, изложенная в разделе "Регистры ввода/вывода и обычная память" 225, предлагает избегать обычных указателей. Интерфейсные функции, используемые для доступа к памяти ввода/вывода, безопасны на всех платформах и являются сразу оптимизированными, когда операцию может выполнить прямое разыменование указателя.

Поэтому, хотя разыменование указателя работает (сейчас) на платформах x86, неудачное использование надлежащих макросов препятствует переносимости и читаемости драйвера.

Получение памяти ввода/вывода и отображение

Области памяти ввода/вывода должны быть выделены до начала использования. Интерфейсом для получения областей памяти (определённым в ***<linux/ioport.h>***) является:

```
struct resource *request_mem_region(unsigned long start, unsigned long len,
char *name);
```

Эта функция выделяет область памяти **len** байт, начиная со **start**. Если всё идёт хорошо, возвращается не **NULL** указатель; в противном случае возвращается значение **NULL**. Вся выделенная для ввода/вывода память перечислена в ***/proc/iomem***.

Когда больше не нужны, области памяти должны освобождаться:

```
void release_mem_region(unsigned long start, unsigned long len);
```

Существует также старая функция для проверки на наличие области памяти ввода/вывода:

```
int check_mem_region(unsigned long start, unsigned long len);
```

Но, как и ***check_region***, эта функция является небезопасной и её следует избегать.

Выделение памяти ввода/вывода не только необходимый шаг перед тем, как к этой памяти можно выполнять доступ. Необходимо также убедиться, что эта память ввода/вывода стала доступна для ядра. Получение памяти ввода/вывода не только вопрос разыменования указателя; на многих системах память ввода/вывода совсем не доступна напрямую таким образом. Так что отображение должно быть выполнено первым. Это роль функции ***ioremap***, представленной в разделе "vmalloc и друзья" 214 в Главе 8 221. Функция предназначена специально для присвоения виртуальных адресов для областей памяти ввода/вывода.

После выполнения ***ioremap*** (и ***iounmap***), драйвер устройства может получить доступ к любому адресу памяти ввода/вывода, независимо от того, связан ли он напрямую с виртуальным адресным пространством. Однако, следует помнить, что адреса, возвращаемые ***ioremap***, не должны быть разыменованы напрямую; вместо этого должны быть использованы функции доступа, предоставляемые ядром. Прежде чем мы углубимся в эти функции, мы пересмотрим получше прототипы ***ioremap*** и познакомимся с несколькими деталями, которые мы пропустили в предыдущей главе.

Функции вызываются в соответствии со следующим определением:

```
#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void * addr);
```

Прежде всего, обратите внимание на новую функцию ***ioremap_nocache***. Мы не давали её в Главе 8 203, поскольку её смысл, безусловно, связан с работой оборудования. Цитируем один из заголовков ядра: "она полезна, если некоторые регистры управления находятся в таких областях, где объединение записи или кэширование чтения не желательны". Действительно, реализация функции идентична ***ioremap*** на большинстве компьютерных платформ: в ситуациях, когда вся память ввода/вывода уже видна через некэшируемые адреса, нет оснований для реализации отдельной некэширующей версии ***ioremap***.

Доступ к памяти ввода/вывода

На некоторых платформах вы можете осуществить использование возвращаемого значения ***ioremap*** как указателя. Такое использование не является переносимым и разработчики ядра ведут работу по устранению любого такого использования всё больше и больше. Правильным способом доступа к памяти ввода/вывода является набор функций (определённых с помощью ***<asm/io.h>***), предусмотренных для этой цели.

Для чтения из памяти ввода/вывода воспользуйтесь одной из следующих:

```
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);
```

Здесь, **addr** должен быть адресом, полученным от ***ioremap*** (возможно, с целочисленным смещением); возвращаемое значение является тем, что было прочитано из данной памяти ввода/вывода.

Существует аналогичный набор функций для записи в память ввода/вывода:

```
void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);
```

Если вы должны читать или писать ряд значений по данному адресу памяти ввода/вывода, можно использовать версии функций с повторением:

```
void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);
```

Функции **ioread** читают **count** значений из данного **addr** в заданный **buf**, а функции **iowrite** записывают **count** значений из данного **buf** по заданному **addr**. Обратите внимание, что **count** выражается в размере записываемых данных; ***ioread32_rep*** считывает **count** 32-х разрядных значений, начиная с **buf**.

Все вышеописанные функции выполняют ввод/вывод по заданному **addr**. Вместо этого, если требуется работать с блоком памяти ввода/вывода, можно использовать одну из следующих:

```
void memset_io(void *addr, u8 value, unsigned int count);
void memcpy_fromio(void *dest, void *source, unsigned int count);
void memcpy_toio(void *dest, void *source, unsigned int count);
```

Эти функции ведут себя подобно их аналогам библиотеки Си.

Если вы почитаете исходный код ядра, вы увидите при использовании памяти ввода/вывода много вызовов старого набора функций. Эти функции по-прежнему работают, но их применение в новом коде не рекомендуется. Среди прочего, они являются менее безопасными, поскольку они не выполняют некоторые виды проверки типа. Тем не менее, мы представляем их здесь:

```
unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
```

Эти макросы используются для получения 8-ми разрядного, 16-ти разрядного и 32-х разрядного значений данных из памяти ввода/вывода.

```
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
```

Как и предыдущие функции, эти функции (макросы) используются для записи 8-ми разрядных, 16-ти разрядных и 32-х разрядных элементов данных.

Некоторые 64-х разрядные платформы также предлагают ***readq*** и ***writeq*** для операций памяти с учетверённым словом (8 байт) на шине PCI. Спецификация учетверённого слова исторически осталась со времени, когда все реальные процессоры имели 16-ти разрядные слова. На самом деле, имя L, используемое для 32-х разрядных значений, также стало неправильным, но переименование всего перепутало бы всё даже больше.

Порты как память ввода/вывода

Некоторое оборудование имеет интересную особенность: некоторые версии используют порты ввода/вывода, в то время как другие используют память ввода/вывода. Регистры, экспортируемые в процессор, являются в любом случае одинаковыми, но метод доступа различается. Как способ сделать жизнь проще для драйверов, имеющих дело с оборудованием такого рода, и как способ минимизации очевидных различий между портом ввода/вывода и доступом к памяти, ядро версии 2.6 предоставляет функцию, названную ***ioport_map***:

```
void *ioport_map(unsigned long port, unsigned int count);
```

Эта функция пересвязывает **count** портов ввода/вывода и делает их видимыми как память ввода/вывода. С этого момента драйвер может использовать ***ioread8*** и друзей с возвращёнными адресами и забыть, что он использует для всего порты ввода/вывода.

Когда больше не требуется, это отображение должно быть отменено:

```
void ioport_unmap(void *addr);
```

Эти функции делают порты ввода/вывода выглядящими подобно памяти. Заметьте, однако,

что порты ввода/вывода всё ещё должны быть выделены ***request_region***, прежде чем они могут быть переназначены этим способом.

Повторное использование short для памяти ввода/ввода

Модуль примера ***short***, представленный ранее для доступа к портам ввода/вывода, может быть также использован для доступа к памяти ввода/вывода. С этой целью вы должны сказать ему во время загрузки использовать память ввода/вывода; также вы должны изменить базовый адрес, чтобы он указывал на вашу область ввода/вывода.

Например, вот как мы использовали ***short*** для зажигания отладочных светодиодов на отладочной плате MIPS:

```
mips.root# ./short_load use_mem=1 base=0xb7ffffc0
mips.root# echo -n 7 > /dev/short0
```

Использование ***short*** для памяти ввода/вывода такое же, как и для портов ввода/вывода.

Следующий фрагмент показывает цикл, используемый ***short*** во время записи по адресам памяти:

```
while (count--) {
    iowrite8(*ptr++, address);
    wmb( );
}
```

Обратите внимание на использование здесь барьера памяти. Поскольку ***iowrite8***, вероятно, превращается на многих архитектурах в прямое присваивание, необходим барьер памяти, чтобы обеспечить запись в ожидаемом порядке.

short использует ***inb*** и ***outb***, чтобы показать, как это делается. Однако, простым упражнением для читателя было бы изменить ***short*** для переназначения портов ввода/вывода с помощью ***ioport_map*** и значительного упрощения остального кода.

ISA память ниже 1 Мб

Одной из самых известных областей памяти ввода/вывода является область ISA, имеющаяся на персональных компьютерах. Это область памяти между 640 Кб (0xA0000) и 1 Мб (0x100000). Таким образом, она находится прямо в середине обычной оперативной памяти системы. Это позиционирование может показаться немного странным; это артефакт решения, принятого в начале 1980-х, когда 640 Кб памяти казалось большим, чем кто-нибудь когда-нибудь сможет использовать.

Этот диапазон памяти относится к не-прямо-отображаемому классу памяти. (* На самом деле это не совсем верно. Диапазон памяти настолько мал и настолько часто используем, что во время загрузки системы для доступа к этим адресам ядро строит таблицы страниц. Тем не менее, виртуальный адрес, используемый для доступа к ним, не такой же, как физический адрес, и, таким образом, *ioremap* необходима в любом случае.) Вы можете читать/писать несколько байт в этот диапазон памяти, используя модуль ***short***, как объяснялось ранее, то есть устанавливая во время загрузки **use_mem**.

Хотя ISA память ввода/вывода существует только на компьютерах класса x86, мы думаем,

что она стоит нескольких слов и примера драйвера на ней.

Мы не собираемся обсуждать в этой главе PCI память, поскольку она является чистейшим видом памяти ввода/вывода: узнав физический адрес, вы можете просто переназначить его и получить доступ. "Проблемой" с PCI памятью ввода/вывода является то, что она не поддаётся рабочему примеру для этой главы, потому что мы не можем знать заранее физической адрес вашей PCI памяти для отображения, или какого-то другого безопасного доступа к этим областям. Мы выбрали для описания диапазон ISA памяти, потому что он и менее ясен и больше подходит для выполнения кода примера.

Чтобы продемонстрировать доступа к ISA памяти, мы используем ещё один небольшой глупый модуль (часть исходников примеров). По сути, он называется ***silly*** (глупым), как сокращение от Simple Tool for Unloading and Printing ISA Data (простого инструмента для Разгрузка и печати данных ISA), или нечто подобного.

Модуль дополняет функциональность ***short*** предоставлением доступа ко всему 384 Кб пространству памяти и показывая все различные функции ввода/вывода. Он содержит четыре узла устройств, которые выполняют те же задачи, используя различные функции передачи данных. Устройства ***silly*** действуют как окно над памятью ввода/вывода способом, похожем на ***/dev/mem***. Вы может читать и записывать данные, а также выполнять произвольный доступ с помощью ***lseek*** для случайного адреса памяти ввода/вывода.

Так как ***silly*** предоставляет доступ к ISA памяти, он должен начать с отображения физических адресов ISA на виртуальные адреса ядра. Когда-то давно в ядре Linux можно было просто присвоить указатель на интересующий ISA адрес, затем разыменовать его напрямую. Однако, в современном мире мы должны работать с системой виртуальной памяти и сначала переназначить диапазон памяти. Как объяснялось ранее для ***short***, это переназначение осуществляется ***ioremap***:

```
#define ISA_BASE 0xA0000
#define ISA_MAX 0x100000 /* для доступа к обычной памяти */

    /* эта строка появляется в silly_init */
    io_base = ioremap(ISA_BASE, ISA_MAX - ISA_BASE);
```

ioremap возвращает указатель, который может быть использован с ***ioread8*** и другими функциями, объяснёнными в разделе "Доступ к памяти ввода/вывода" 239.

Давайте посмотрим назад на наш модуль примера, чтобы увидеть, каким образом могут быть использованы эти функции. ***/dev/sillyb***, имеющий младший номер 0, адресует память ввода/вывода через ***ioread8*** и ***iowrite8***.

Следующий код показывает реализацию для ***read***, которая делает диапазон адресов 0xA0000 - 0xFFFFF доступным как виртуальный файл в диапазоне 0 - 0x5FFFF. Функция ***read*** структурирована как оператор переключения (switch) между различным режимами доступа; здесь показан случай ***sillyb***:

```
case M_8:
    while (count) {
        *ptr = ioread8(add);
        add++;
        count--;
        ptr++;
```

```
    }
    break;
```

Следующими двумя устройствами являются */dev/sillyw* (младший номер 1) и */dev/sillyl* (младший номер 2). Они работают подобно */dev/sillyb*, только они используют 16-ти разрядные и 32-х разрядные функции. Вот реализация *write* для *sillyl*, снова часть переключателя:

```
case M_32:
    while (count >= 4) {
        iowrite32(*(u32 *)ptr, add);
        add += 4;
        count -= 4;
        ptr += 4;
    }
    break;
```

Последним устройством является */dev/sillycp* (младший номер 3), которое для выполнения той же задачи использует функции *memcpy_*io*. Вот суть его реализации *read*:

```
case M_memcpy:
    memcpy_fromio(ptr, add, count);
    break;
```

Поскольку для обеспечения доступа к области памяти ISA была использована *ioremap*, *silly* должен вызвать *iounmap* при выгрузке модуля:

```
iounmap(io_base);
```

isa_readb и друзья

Взгляд на исходные коды ядра поднимет ещё один набор процедур с такими именами, как *isa_readb*. В самом деле, каждая из только что описанных функций имеет *isa_* эквивалент. Эти функции обеспечивают доступ к памяти ISA без необходимости отдельного шага *ioremap*. Однако, разработчики ядра говорят, что эти функции предназначены быть временными вспомогательными средствами портирования драйверов и что они могут уйти в будущем. Таким образом, необходимо избегать их использования.

Краткая справка

Эта глава представляет следующие символы, связанные с управлением оборудованием:

#include <linux/kernel.h>
void barrier(void)
Этот "программный" барьер памяти просит через эту инструкцию компилятор считать всю память нестабильной.

#include <asm/system.h>
void rmb(void);
void read_barrier_depends(void);
void wmb(void);
void mb(void);
Аппаратные барьеры памяти. Они просят через эту инструкцию процессор (и компилятор) выгрузить все чтения, записи памяти, или и то и другое.

#include <asm/io.h>
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

Функции, которые используются для чтения и записи портов ввода/вывода. Они также могут быть вызваны программами пространства пользователя при условии, что они имеют правильные привилегии для доступа к портам.

unsigned inb_p(unsigned port);
...

Если после операции ввода/вывода необходима небольшая задержка, вы можете использовать шесть аналогичных предыдущему набору функций с паузой; эти функции с паузой имеют имена, оканчивающиеся на ***_p***.

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);

"Строковые функции" оптимизированы для передачи данных из входного порта в область памяти, или наоборот. Такие передачи осуществляются чтением или записью в тот же порт **count** раз.

#include <linux/ioport.h>
struct resource *request_region(unsigned long start, unsigned long len, char *name);
void release_region(unsigned long start, unsigned long len);
int check_region(unsigned long start, unsigned long len);

Распределители ресурсов для портов ввода/вывода. (Не рекомендуемая) функция ***check*** возвращает 0 при успехе и меньше 0 в случае ошибки.

struct resource *request_mem_region(unsigned long start, unsigned long len, char *name);
void release_mem_region(unsigned long start, unsigned long len);
int check_mem_region(unsigned long start, unsigned long len);

Функции, которые занимаются распределением ресурсов для областей памяти.

#include <asm/io.h>
void *ioremap(unsigned long phys_addr, unsigned long size);
void *ioremap_nocache(unsigned long phys_addr, unsigned long size);
void iounmap(void *virt_addr);

ioremap переназначает физический диапазон адресов в виртуальное адресное пространство процессора, делая его доступным для ядра. ***iounmap*** освобождает назначения, когда они больше не требуются.

#include <asm/io.h>
unsigned int ioread8(void *addr);
unsigned int ioread16(void *addr);
unsigned int ioread32(void *addr);

void iowrite8(u8 value, void *addr);
void iowrite16(u16 value, void *addr);
void iowrite32(u32 value, void *addr);

Функции доступа, которые используются для работы с памятью ввода/вывода.

void ioread8_rep(void *addr, void *buf, unsigned long count);
void ioread16_rep(void *addr, void *buf, unsigned long count);
void ioread32_rep(void *addr, void *buf, unsigned long count);
void iowrite8_rep(void *addr, const void *buf, unsigned long count);
void iowrite16_rep(void *addr, const void *buf, unsigned long count);
void iowrite32_rep(void *addr, const void *buf, unsigned long count);

"Повторяющие" версии примитивов ввода/вывода для памяти.

unsigned readb(address);
unsigned readw(address);
unsigned readl(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);

Старые, небезопасные для типа данных функции для доступа к памяти ввода/вывода.

memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);

Функции, работающие с блоками памяти ввода/вывода.

void *ioport_map(unsigned long port, unsigned int count);
void ioport_unmap(void *addr);

Автор драйвера, который хочет рассматривать порты ввода/вывода, как будто они память ввода/вывода, может передать эти порты в ***ioport_map***. Отображение, когда больше не требуется, должно завершаться (с помощью ***ioport_unmap***).

Глава 10, Обработка прерываний

Хотя некоторыми устройствами можно управлять не используя ничего, кроме их областей ввода/вывода, большинство реальных устройств немного более сложные, чем эти. Устройствам приходится иметь дело с внешним миром, который часто включает в себя такие вещи, как вращающийся диск, движущиеся ленты, соединения с отдалёнными местами и так далее. Многое требуется выполнять в сроки, которые отличаются от таковых и намного медленнее, чем в процессоре. Так как почти всегда нежелательно, чтобы процессор ждал внешних событий, должен быть способ для устройства дать процессору знать, когда что-то произошло.

Этот способ, конечно, прерывания. ***Прерывание*** - это просто сигнал, который оборудование может послать, когда хочет внимания процессора. Linux обрабатывает прерывания во многом так же, как обрабатывает сигналы в пространстве пользователя. По большей части драйверу необходимо только зарегистрировать обработчик для прерываний своего устройства и обработать их должным образом при их получении. Естественно, что под простой картинкой есть некоторые сложности; в частности, обработчики прерываний несколько ограничены в действиях, которые они могут выполнять, как результат того, как они выполняются.

Трудно продемонстрировать использование прерываний без реального аппаратного устройства для их генерации. Таким образом, пример кода, используемый в этой главе, работает с параллельным портом. Такие порты начинают становиться дефицитом на современном оборудовании, но при везении большинство людей всё ещё в состоянии получить их поддержку на системе с имеющимся портом. Мы будем работать с модулем ***short*** из предыдущей главы; с некоторыми небольшими добавлениями он сможет генерировать и обрабатывать прерывания от параллельного порта. Название модуля, ***short***, на самом деле означает ***short int (короткое прерывание)*** (это похоже на Си, не правда ли?), чтобы напомнить нам, что он обрабатывает прерывания.

Однако, прежде чем мы углубимся в эту тему, пришло время для одного предостережения. Обработчики прерывания по своей природе работают одновременно с другим кодом. Таким образом, они неизбежно поднимают вопросы конкуренции и борьбы за структуры данных и оборудование. Если вы поддались соблазну пропустить обсуждение в Главе 5 124, мы понимаем. Но мы также рекомендуем вам вернуться назад и прочитать её теперь. Твёрдое понимание методов контроля конкуренции имеет жизненно важное значение при работе с

прерываниями.

Подготовка параллельного порта

Хотя параллельный интерфейс прост, он может вызвать прерывание. Эта возможность используется принтером для уведомления драйвера ***lp***, что он готов принять следующий символ в буфере.

Как и большинство устройств, параллельный порт на самом деле не генерирует прерывания, если ему поручили это делать; стандарт параллельного порта говорит, что установка бита 4 порта 2 (0x37a, 0x27a, или иного) разрешает уведомление прерываниями. для установки этого бита ***short*** выполняет простой вызов ***outb*** в момент инициализации модуля.

После разрешения прерываний параллельный интерфейс генерирует прерывание, когда электрический сигнала на выводе 10 (так называемый бит ACK, acknowledge, подтверждение) изменяется от низкого до высокого. Простейшим способом заставить интерфейс генерировать прерывания (за исключением подключения принтера к порту) является соединение контактов 9 и 10 на разъёме параллельного порта. Короткий кусок провода, вставленный в соответствующие отверстия в разъёме параллельного порта с обратной стороны системного блока, создаёт такое соединение. Назначение выводов параллельного порта показано на Рисунке 9-1.

Контакт 9 является самым старшим значащим битом байта данных параллельного порта. Если вы запишете бинарные данные в ***/dev/short0***, вы сгенерируете несколько прерываний. Однако, запись текста ASCII в порт совсем не будет генерировать прерывания, поскольку набор символов ASCII не имеет записей с установленным старшим битом.

Если вы предпочитаете избегать соединять контакты вместе, но у вас под рукой есть принтер, вы можете запустить пример обработчика прерывания используя реальный принтер, как показано ниже. Однако, следует отметить, что представляемые нами исследовательские функции зависят от наличия перемычки между выводами 9 и 10, и она необходима для экспериментирования с помощью нашего кода.

Установка обработчика прерывания

Если вы действительно хотите "видеть" сгенерированные прерывания, подключения к аппаратному устройству не достаточно; в системе должен быть настроен программный обработчик. Если ядру Linux не было сказано ожидать вашего прерывания, оно просто получит и проигнорирует его.

Линии прерываний являются ценным и часто ограниченным ресурсом, особенно когда есть только 15 или 16 из них. Ядро ведёт реестр линий прерываний, похожий на реестр портов ввода/вывода. Как ожидается, модуль запрашивает канал прерывания (или IRQ, для запроса прерывания), прежде чем использовать его и освобождает его, когда заканчивает работу. Как мы увидим, во многих ситуациях также ожидается, что модули будут способны делить линии прерывания с другими драйверами. Следующие функции, объявленные в ***<linux/interrupt.h>***, реализуют интерфейс регистрации прерывания:

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long flags,
```

```
                const char *dev_name,
                void *dev_id);

void free_irq(unsigned int irq, void *dev_id);
```

Значение, возвращаемое ***request_irq*** запрашивающей функции, либо 0, что означает успешность, либо отрицательный код ошибки, как обычно. Не редки случаи, когда функция возвращает **-EBUSY**, чтобы просигнализировать, что другой драйвер уже использует запрошенную линию прерывания. Аргументами функции являются:

unsigned int irq
Номер запрашиваемого прерывания.

irqreturn_t (*handler)(int, void *, struct pt_regs *)
Указатель на установленную функцию-обработчик. Мы обсудим аргументы этой функции и её возвращаемое значение далее в этой главе.

unsigned long flags
Как и следовало ожидать, битовая маска опций (описываемая далее), связанная с управлением прерыванием.

const char *dev_name
Строка, передаваемая в ***request_irq***, используется в ***/proc/interrupts***, чтобы показать владельца прерывания (смотрите следующий раздел).

void *dev_id
Указатель, используемый для общих линий прерываний. Это уникальный идентификатор, который используется, когда линия прерывания освобождена, и может также быть использован драйвером для указания на свою собственную область данных (с целью определения, какое устройство создало прерывание). Если прерывание не является общим, **dev_id** может быть установлен в **NULL**, но так или иначе, хорошая идея использовать этот элемент для указания на структуру устройства. Мы увидим практическое применение **dev_id** в разделе "Реализация обработчика" 257.

Битами, которые могут быть установлены в **flags**, являются:

SA_INTERRUPT
Если установлен, указывает на "быстрый" обработчик прерывания. Быстрые обработчики выполняются при запрещённых прерываниях на текущем процессоре (тема рассматривается в разделе "Быстрые и медленные обработчики" 256).

SA_SHIRQ
Этот бит сигнализирует, что прерывание может быть разделено между устройствами. Концепция общего использования изложена в разделе "Разделяемые прерывания" 266.

SA_SAMPLE_RANDOM
Этот бит показывает, что сгенерированные прерывания могут внести свой вклад в пул энтропии, используемый ***/dev/random*** и ***dev/urandom***. Эти устройства возвращают при чтении действительно случайные числа и предназначены для оказания помощи прикладному программному обеспечению выбирать безопасные ключи для шифрования. Такие случайные числа извлекаются из пула энтропии, в который вносят вклад

различные случайные события. Если ваше устройство генерирует прерывания в действительно случайные моменты времени, вам следует установить этот флаг. Если, с другой стороны, ваши прерывания предсказуемые (например, вертикальная развертка от захвата кадров), флаг не стоит устанавливать - это не будет способствовать каким-либо образом энтропии системы. Устройствам, которые могут быть под влиянием атакующих, не следует устанавливать этот флаг; например, сетевые драйверы могут быть зависимыми от предсказуемых периодичных пакетов извне и не должны вносить свой вклад в пул энтропии. Для дополнительной информации смотрите комментарии в ***drivers/char/random.c***.

Обработчик прерывания может быть установлен либо при инициализации драйвера, либо при первом открытии устройства. Хотя установка обработчика прерывания в функции инициализации модуля может звучать как хорошая идея, часто это не так, особенно если устройство не разделяет прерывания. Из-за ограниченного числа линий прерывания вы не хотите их растрачивать. Вы можете легко выключить больше устройств на вашем компьютер, чем существует прерываний. Если модуль запрашивает прерывание при инициализации, он запрещает любым другим драйверам использовать прерывание, даже если удерживающее его устройство никогда не используется. С другой стороны, запрос прерывания при открытии устройства позволяет некоторое совместное использование ресурсов.

Можно, например, запускать захват кадров на том же прерывании, как и модем до тех пор, пока вы не пользуетесь двумя устройствами одновременно. Весьма распространённая практика для пользователей - загрузить модуль для специального устройства при загрузке системы, даже если устройство используется редко. Приспособление сбора данных может использовать те же прерывания, как второй последовательный порт. Хотя не слишком трудно избежать подключения к поставщику услуг Интернета (ISP) во время сбора данных, необходимость выгрузить модуль, чтобы использовать модем, очень неприятна.

Правильным местом для вызова ***request_irq*** является первое открытие устройства ***перед*** поручением оборудованию генерировать прерывания. Местом для вызова ***free_irq*** является закрытие устройства последний раз, ***после*** указания оборудованию больше не прерывать процессор. Недостатком этого метода является то, что вам нужно сохранять счётчик открытий каждого устройства, чтобы знать, когда прерывания могут быть отключены.

Несмотря на это обсуждение, ***short*** запрашивает свою линию прерывания во время загрузки. Этот было сделано так, чтобы вы могли запускать тестовые программы без запуска дополнительных процессов, держащих устройство открытым. ***short***, таким образом, запрашивает прерывание внутри своей функции инициализации (***short_init***), а не делает это в ***short_open***, как бы сделал реальный драйвер устройства.

Прерыванием, запрошенным следующим кодом, является ***short_irq***. Реальное получение переменной (то есть определение, какое прерывание использовать) показано позже, поскольку это не относится к данному обсуждению. ***short_base*** является базовым адресом ввода/вывода используемого параллельного интерфейса; для разрешения подтверждающих прерываний делается запись в регистр 2.

```
if (short_irq >= 0) {
    result = request_irq(short_irq, short_interrupt, SA_INTERRUPT, "short",
NULL);
    if (result) {
        printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
        short_irq = -1;
```

```
    }
    else { /* фактическое разрешение -- подразумевается, что это *является*
параллельным портом */
        outb(0x10,short_base+2);
    }
}
```

Код показывает, что устанавливаемый обработчик является быстрым обработчиком (**SA_INTERRUPT**), не поддерживает совместное использование прерывания (**SA_SHIRQ** отсутствует) и не способствует энтропии системы (**SA_SAMPLE_RANDOM** тоже отсутствует). Следующий затем вызов ***outb*** разрешает подтверждающие прерывания для параллельного порта.

Не знаем насколько это важно, для архитектур i386 и x86_64 определена функция для запроса наличия линии прерывания:

```
int can_request_irq(unsigned int irq, unsigned long flags);
```

Эта функция возвращает ненулевое значение, если попытка получить данное прерывание успешна. Однако, следует отметить, что между вызовами ***can_request_irq*** и ***request_irq*** всё может в любой момент измениться.

Интерфейс /proc

Всякий раз, когда аппаратное прерывание достигает процессора, увеличивается внутренний счётчик, предоставляя возможность проверить, работает ли устройство так, как ожидалось.

Полученные прерывания показываются в ***/proc/interrupts***. Следующий снимок был получен на двухпроцессорной системе Pentium:

```
root@montalcino:/bike/corbet/write/ldd3/src/short# cat /proc/interrupts
           CPU0       CPU1
  0:     4848108         34    IO-APIC-edge  timer
  2:           0          0  XT-PIC cascade
  8:           3          1    IO-APIC-edge  rtc
 10:        4335          1   IO-APIC-level  aic7xxx
 11:        8903          0   IO-APIC-level  uhci_hcd
 12:          49          1    IO-APIC-edge  i8042
NMI:           0          0
LOC:     4848187    4848186
ERR:           0
MIS:           0
```

Первая колонка - это число прерываний. Вы можете видеть, что судя по отсутствующим прерываниям, этот файл показывает только прерывания, соответствующие установленным обработчикам. Например, первый последовательный порт (который использует прерывание номер 4) не показан, это свидетельствует, что модем не используется. В самом деле, даже если модем был использован ранее, но не использовался во время снимка, он не будет отображаться в файле; последовательные порты хорошо себя ведут и отключают свои обработчики прерываний при закрытом устройстве.

Распечатка ***/proc/interrupts*** показывает, как много прерываний было доставлено каждому процессору в системе. Как можно видеть из вывода, ядро Linux обычно обрабатывает

прерывания на первом процессоре, как способ улучшения использования кэша. (* Хотя некоторые крупные системы явно используют схемы балансирования прерываний для распределения нагрузки по обработке прерываний в системе). Последние два столбца предоставляют информацию о том программируемом контроллере прерываний, который обрабатывает прерывание (и о котором автору драйвера беспокоиться нет необходимости) и имя (имена) устройств(а), которые имеют зарегистрированные обработчики прерывания (указанные в аргументе **dev_name** для ***request_irq***).

Дерево ***/proc*** содержит другой относящийся к прерываниям файл, ***/proc/stat***; иногда вы будете находить один файл более полезными, иногда вы предпочтёте другой. ***/proc/stat*** ведёт запись некоторой низкоуровневой статистики о системной активности, включая (но не ограничиваясь ими) число прерываний, полученных после загрузки системы. Каждая строка ***stat*** начинается с текстовой строки, которая является ключом к строке; метка **intr** - то, что мы ищем. Следующий (усечённый) снимок был сделан вскоре после предыдущего:

```
intr 5167833 5154006 2 0 2 4907 0 2 68 4 0 4406 9291 50 0 0
```

Первое число - это общая сумма всех прерываний, а каждое из остальных представляет одну линию прерывания, начиная с прерывания 0. Все счётчики суммируются по всем процессорам в системе. Это снимок показывает, что прерывание номер 4 было использовано 4907 раз, хотя обработчик и не установлен. Если тестируемый вами драйвер получает и освобождает прерывания в каждом цикле открытия и закрытия, вы можете найти ***/proc/stat*** более полезным, чем ***/proc/interrupts***.

Ещё одно различие между этими двумя файлами в том, что ***interrupts*** является архитектурно-независимым (за исключением, быть может, нескольких строчек в конце), а ***stat*** является; число полей зависит от оборудования, на котором работает ядро. Количество доступных прерываний варьируется от небольшого, как 15 на SPARC, до значительного, как 256 на IA-64 и нескольких других системах. Интересно отметить, что число прерываний, определённых на x86, в настоящее время 224, а не 16, как могло ожидаться; это, как описано в ***include/asm-i386/irq.h***, зависит от использования Linux архитектурного предела, а не предела определённого реализацией (такого, как 16 источников прерываний на старомодном контроллере прерываний ПК).

Следующий снимок ***/proc/interrupts*** сделан на системе IA-64. Как вы можете видеть, помимо другой аппаратной маршрутизации обычных источников прерываний, вывод очень похож на показанный ранее из 32-х разрядной системы.

```
           CPU0       CPU1
 27:        1705      34141   IO-SAPIC-level  qla1280
 40:           0          0    SAPIC perfmon
 43:         913       6960   IO-SAPIC-level  eth0
 47:       26722        146   IO-SAPIC-level  usb-uhci
 64:           3          6    IO-SAPIC-edge  ide0
 80:           4          2    IO-SAPIC-edge  keyboard
 89:           0          0    IO-SAPIC-edge  PS/2 Mouse
239:     5606341    5606052            SAPIC  timer
254:       67575      52815            SAPIC  IPI
NMI:           0          0
ERR:           0
```

Автоопределение номера прерывания

Одной из наиболее сложных проблем для драйвера во время инициализации может быть как определить, какая линия прерывания будет использоваться устройством. Драйверу необходима информация, чтобы правильно установить обработчик. Хотя программист даже может потребовать от пользователя указать номер прерывания во время загрузки, это плохая практика, поскольку в большинстве случаев пользователь не знает номера, либо потому, что он не настроил перемычки, либо потому, что устройство их не имеет. Большинство пользователей хотят, чтобы их оборудование "просто работало" и не интересуются такими вопросами, как номера прерывания. Так что автоопределение номера прерывания является основным требованием для удобства использования драйвера.

Иногда автоопределение зависит от знаний, которые некоторые устройства имеют по умолчанию, которые редко, если когда-либо вообще, изменяются. В этом случае драйвер может считать, что применяются значения по умолчанию. Именно так по умолчанию ведёт себя с параллельным портом ***short***. Реализация проста, как показано в ***short***:

```
if (short_irq < 0) /* not yet specified: force the default on */
    switch(short_base) {
        case 0x378: short_irq = 7; break;
        case 0x278: short_irq = 2; break;
        case 0x3bc: short_irq = 5; break;
    }
```

Код присваивает номер прерывания в соответствии с выбранным базовым адресом ввода/вывода, позволяя пользователю переопределить значение по умолчанию во время загрузки с чем-то вроде:

```
insmod ./short.ko irq=x
```

short_base умолчанию 0x378, так что **short_irq** по умолчанию 7.

Некоторые устройства являются более передовыми в области проектирования и просто "сообщают", какое прерывание они собираются использовать. В этом случае драйвер получает номер прерывания, читая байт состояния одного из портов ввода/вывода устройства или конфигурационного пространства PCI. Когда целевое устройство обладает способностью сказать драйверу, какое прерывание собирается использовать, автоопределение номера прерывания просто означает зондирование прибора без дополнительной работы, требуемой для проверки прерывания. К счастью, большинство современного оборудования работает таким образом, например, стандарт PCI решает данную проблему, требуя периферийные устройства заявлять, какую линию (линии) прерывания они собираются использовать. Стандарт PCI обсуждается в Главе 12 288.

К сожалению, не все устройства дружественны к программисту и автоопределение может потребовать некоторых проверок. Техника очень проста: драйвер просит устройство генерировать прерывания и наблюдает, что происходит. Если всё идёт хорошо, активируется только одна линия прерывания.

Хотя зондирование просто в теории, фактическая реализация может быть неясной. Мы рассматриваем два варианта выполнения задачи: вызывая определённые в ядре вспомогательные функции и реализацию нашей собственной версии.

Проверка с помощью ядра

Ядро Linux предоставляет низкоуровневые средства проверки номера прерывания. Это работает только для неразделяемых прерываний, но большинство оборудования, способного работать в режиме разделяемого прерывания, в любом случае предлагает лучшие способы нахождения настроенного номера прерывания. Средство состоит из двух функций, объявленных в ***<linux/interrupt.h>*** (который также описывает механизм зондирования):

unsigned long probe_irq_on(void);
Эта функция возвращает битовую маску неназначенных прерываний. Драйвер должен сохранить возвращённую битовую маску и передать её позже в ***probe_irq_off***. После этого вызова драйвер должен обеспечить, чтобы его устройство сгенерировало по крайней мере одно прерывание.

int probe_irq_off(unsigned long);
После запроса прерывания устройством драйвер вызывает эту функцию, передавая в качестве аргумента битовую маску, которую перед этим вернула ***probe_irq_on***. ***probe_irq_off*** возвращает номер прерывания, которое было вызвано после "probe_on". Если прерывание не произошло, возвращается 0 (следовательно, IRQ 0 не может быть прозондировано, но в любом случае нет пользовательских устройств, которые могли бы использовать его на любой из поддерживаемых архитектур). Если произошло более чем одно прерывание (неоднозначное обнаружение), ***probe_irq_off*** возвращает отрицательное значение.

Программист должен быть осторожным и разрешить прерывания на устройстве после вызова ***probe_irq_on*** и отключить их перед вызовом ***probe_irq_off***. Кроме того, необходимо помнить, что служба ожидает прерывания в вашем устройстве после ***probe_irq_off***.

Модуль ***short*** демонстрирует использование такой проверки. Если вы загружаете модуль с **probe=1**, чтобы обнаружить вашу линию прерывания, выполняется следующий код, при условии, что контакты 9 и 10 разъёма параллельного порта соединены вместе:

```
int count = 0;
do {
    unsigned long mask;

    mask = probe_irq_on( );
    outb_p(0x10,short_base+2); /* разрешение подтверждений */
    outb_p(0x00,short_base);   /* очистить бит */
    outb_p(0xFF,short_base);   /* установить бит: прерывание! */
    outb_p(0x00,short_base+2); /* запретить подтверждение */
    udelay(5); /* подождать некоторое время */
    short_irq = probe_irq_off(mask);

    if (short_irq == 0) { /* ничего нет? */
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq = -1;
    }
    /*
     * Если более чем одна линия была активирована, результат
     * отрицателен. Следует обслужить это прерывание (не требуется для lpt
порта)
```

```
     * и продолжить цикл. Посмотреть максимум пять раз, затем закончить
     */
} while (short_irq < 0 && count++ < 5);
if (short_irq < 0)
     printk("short: probe failed %i times, giving up\n", count);
```

Обратите внимание на использование ***udelay*** перед вызовом ***probe_irq_off***. В зависимости от скорости вашего процессора, возможно, придётся подождать в течение короткого периода, чтобы дать время прерыванию произойти на самом деле.

Проверка может быть длительной задачей. Хотя для ***short*** это и не так, проверка устройства захвата кадров, например, требует задержку, по меньшей мере, 20 мс (которая велика для процессора), а другие устройства могут занять ещё больше времени. Поэтому лучше всего проверить линию прерывания только один раз, при инициализации модуля, независимо от того, устанавливаете ли вы обработчик при открытии устройства (как следует) или в функции инициализации (что не рекомендуется).

Интересно отметить, что на некоторых платформах (PowerPC, M68k, большинство реализаций MIPS и обе версии SPARC) зондирование является ненужным и, следовательно, предыдущие функции просто пустые заполнители, иногда называемые "бесполезной чепухой ISA". На других платформах зондирование осуществляется только для устройств ISA. В любом случае большинство архитектур определяют эти функции (даже если они пустые) для облегчения переносимости существующих драйверов устройств.

Самостоятельное тестирование

Зондирование также может быть реализовано без особых проблем самостоятельно в самом драйвере. Драйверы, которые должны осуществлять своё собственное зондирование, редки, но просмотр их работы даёт некоторое понимание данного процесса. Для этого модуль ***short*** выполняет самостоятельное обнаружение линии прерывания, если он загружен с **probe=2**.

Механизм аналогичен описанному выше: разрешить все неиспользуемые прерывания, затем подождать и посмотреть, что происходит. Тем не менее, мы можем использовать наши знания об устройстве. Часто устройство может быть настроено на использование одного номера прерывания из набора из трёх или четырёх; проверка только этих прерываний позволяет определить то единственно правильное, не проверяя все возможные прерывания.

Реализация ***short*** предполагает, что 3, 5, 7 и 9 являются единственно возможными значениями прерываний. Эти цифры на самом деле являются теми значениями, которые позволяют вам выбрать некоторые параллельные устройства.

Следующий код зондирует, проверяя все "возможные" прерывания и глядя на то, что происходит. **trials** - массив списка прерываний для проверки и имеет **0** в качестве маркера конца; массив **tried** используется для отслеживания обработчиков, которые действительно были зарегистрированы этим драйвером.

```
int trials[ ] = {3, 5, 7, 9, 0};
int tried[ ] = {0, 0, 0, 0, 0};
int i, count = 0;

/*
 * устанавливаем обработчик зондирования на все возможные линии. Запоминаем
```

```
 * результат (0 - успешно, или -EBUSY), чтобы освободить только
 * запрошенные
 */
for (i = 0; trials[i]; i++)
    tried[i] = request_irq(trials[i], short_probing, SA_INTERRUPT, "short
probe", NULL);

do {
    short_irq = 0; /* ещё ничего нет */
    outb_p(0x10,short_base+2); /* разрешить */
    outb_p(0x00,short_base);
    outb_p(0xFF,short_base); /* переключить бит */
    outb_p(0x00,short_base+2); /* запретить */
    udelay(5); /* подождать какое-то время */

    /* это значение было установлено обработчиком */
    if (short_irq == 0) { /* ничего нет? */
        printk(KERN_INFO "short: no irq reported by probe\n");
    }
    /*
     * Если более, чем одна линия была активирована, результат
     * отрицательный. Нам следует обработать прерывание (но для lpt порта
     * это не требуется) и повторить цикл. Делаем это максимум 5 раз
     */
} while (short_irq <=0 && count++ < 5);

/* конец цикла, выгрузить обработчик */
for (i = 0; trials[i]; i++)
    if (tried[i] == 0)
        free_irq(trials[i], NULL);

if (short_irq < 0)
    printk("short: probe failed %i times, giving up\n", count);
```

Можно не знать заранее "возможные" значения прерываний. В таком случае, необходимо исследовать все свободные прерывания, а не ограничиваться несколькими **trials[ ]**. Для зондирования всех прерываний необходимо проверить прерывания с **0** по **NR_IRQS-1**, где **NR_IRQS** определён в ***<asm/irq.h>*** и зависит от платформы.

Теперь мы пропустили только свой обработчик зондирования. Ролью обработчика является обновление **short_irq** в соответствии с фактически полученным прерыванием. Значение **0** в **short_irq** означает "ещё ничего", а отрицательное значение означает "неоднозначно". Эти значения выбраны для соответствия с ***probe_irq_off*** и чтобы позволить тому же коду в ***short.c*** вызывать оба вида зондирования.

```
irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
    if (short_irq = = 0) short_irq = irq; /* найдено */
    if (short_irq != irq) short_irq = -irq; /* неоднозначно */
    return IRQ_HANDLED;
}
```

Аргументы для обработчика описаны ниже. Знания, что **irq** является обрабатываемым прерыванием должно быть достаточным для понимания только что показанной функции.

Быстрые и медленные обработчики

Старые версии ядра Linux предпринимали большие усилия, чтобы различать "быстрые" и "медленные" прерывания. Быстрыми прерываниями были те, которые могли быть обработаны очень быстро, в то время как обработка медленных прерываний длилась значительно дольше. Медленные прерывания могли быть достаточно загружающими процессор и было целесообразно снова разрешать прерывания во время их обработки. В противном случае, задачи, требующие быстрого внимания, могли быть отложены на слишком долгий срок.

В современных ядрах большинство различий между быстрыми и медленными прерывания исчезли. Остаётся только одно: быстрые прерывания (те, которые были запрошены флагом **SA_INTERRUPT**) выполняются с запретом всех других прерываний на текущем процессоре. Обратите внимание, что другие процессоры могут всё ещё обрабатывать прерывания, хотя вы никогда не увидите двух процессоров, обрабатывающих одно и то же прерывание в одно и то же время. Итак, прерывание какого типа должен использовать ваш драйвер? В современных системах **SA_INTERRUPT** предназначен для использования только в нескольких особых ситуациях, таких как таймерные прерывания. Если у вас нет веских оснований для работы вашего обработчика прерываний при отключенных других прерываниях, не следует использовать **SA_INTERRUPT**.

Это описание должно удовлетворить большинство читателей, хотя кто-то со вкусом к оборудованию и некоторым опытом работы со своим компьютером может быть заинтересован в углублении. Если вы не заботитесь о внутренних деталях, вы можете перейти к следующему разделу.

Внутренности обработки прерываний на x86

Это описание о том, как это выглядит в ядрах версии 2.6, было экстраполировано из ***arch/i386/kernel/irq.c***, ***arch/i386/kernel/apic.c***, ***arch/i386/kernel/entry.S***, ***arch/i386/kernel/i8259.c*** и ***include/asm-i386/hw_irq.h***; хотя общая концепция остается той же, аппаратные подробности отличаться на других платформах.

Самый низкий уровень обработки прерываний можно найти в ***entry.S***, ассемблерном файле, который выполняет большую часть работы машинного уровня. Путём небольших ассемблерных трюков и некоторых макросов, каждому возможному прерыванию присвоен небольшой код. В каждом случае код помещает номер прерывания в стек и переходит к общему сегменту, который вызывает ***do_IRQ***, определённую в ***irq.c***.

Первое, что делает ***do_IRQ*** - подтверждает прерывание, чтобы контроллер прерываний мог переходить к другим вещам. Затем она получает спин-блокировку для данного номера прерывания, предотвращая таким образом обработку этого прерывания любым другим процессором. Она очищает несколько статусных битов (в том числе один, называемый **IRQ_WAITING**, который мы рассмотрим в ближайшее время) и затем ищет обработчик(и) для данного прерывания. Если обработчика нет, делать нечего; спин-блокировка освобождается, все ожидающие программные прерывания обработаны и ***do_IRQ*** возвращается.

Однако, обычно, если устройство создаёт прерывания, также есть по крайней мере один зарегистрированный обработчик для прерывания. Для реального вызова обработчиков вызывается функция ***handle_IRQ_event***. Если обработчик является медленной разновидностью (**SA_INTERRUPT** не установлен), прерывания в оборудовании снова разрешаются и вызывается обработчик. Затем, только для чистоты, выполняются

программные прерывания и происходит возврат к обычной работе. "Обычная работа" может также измениться в результате прерывания (обработчик мог ***wake_up*** (пробудить) процесс, например), поэтому последней вещью, которая происходит по возвращении из прерывания, является возможное перепланирование процессора.

Зондирование прерываний осуществляется установкой для каждого IRQ, для которого в настоящее время отсутствует обработчик, статусного бита **IRQ_WAITING**. Когда происходит прерывание, ***do_IRQ*** очищает этот бит и затем возвращается, потому что обработчик не зарегистрирован. ***probe_irq_off***, вызванной драйвером, необходимо только поискать прерывание, которое больше не имеет установленного **IRQ_WAITING**.

Реализация обработчика

До сих пор мы изучали регистрацию обработчика прерывания, но ничего не писали. Вообще-то, в обработчике нет ничего необычного -это обычный код Си.

Единственной особенностью является то, что обработчик работает во время прерывания и, следовательно, испытывает некоторые ограничения в том, что может делать. Эти ограничения являются такими же, какие мы видели для таймеров ядра. Обработчик не может передавать данные в или из пространства пользователя, так как не выполняется в контексте процесса. Обработчики также не могут делать ничего, что могло бы заснуть, например, вызвать ***wait_event***, выделять память ни с чем другим, кроме **GFP_ATOMIC**, или блокировать семафор. Наконец, обработчик не может вызвать ***schedule***.

Ролью обработчика прерываний является предоставление обратной связи устройству о приёме прерывания и прочитать или записать данные в зависимости от смысла обслуживаемого прерывания. Первый шаг обычно состоит в очистке бита на интерфейсной плате; большинство аппаратных устройств не будет генерировать другие прерывания, пока их бит "ожидание прерывания" не очищен. В зависимости от того, как работает ваше оборудование, этот шаг может выполняться последним, а не первым, здесь нет всеобъемлющего правила. Некоторые устройства не требуют этого шага, так как они не имеют бита "ожидание прерывания"; такие устройства составляют меньшинство, хотя параллельный порт является одним из них. По этой причине ***short*** ничего не делает для очистки такого бита.

Типичной задачей для обработчика прерывания является пробуждение спящих процессов устройства, если сигналы прерывания являются событиями, которые они ждут, такими, как появление новых данных.

В случае примера захвата кадров, процесс мог бы получать последовательность изображений постоянно читая устройство; перед чтением каждого фрейма вызов ***read*** блокируется, а обработчик прерывания пробуждает процесс при поступлении каждого нового кадра. Это предполагает, что захват прерывает процессор для сигнализации об успешном появлении каждого нового кадра.

Программист должен быть внимательным, чтобы написать процедуру, которая выполняется в минимальное количество времени, независимо от того, быстрый это или медленный обработчик. Если должно быть выполнено долгое вычисление, лучшим подходом является использование микрозадачи (tasklet) или очереди задач (workqueue) для выполнения вычислений в безопасное время (в разделе "Верхние и нижние половины" 262 мы увидим, как работа может быть отложена таким образом).

Наш код примера в ***short*** отвечает на прерывание вызывая ***do_gettimeofday*** и печатая текущее время в круговой буфер размером со страницу. Затем он пробуждает любой читающий процесс, поскольку теперь есть доступные для чтения данные.

```
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    int written;

    do_gettimeofday(&tv);

    /* Пишем 16 байтовый отчёт. Подразумеваем, что PAGE_SIZE кратна 16 */
    written = sprintf((char *)short_head,"%08u.%06u\n",
                      (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    BUG_ON(written != 16);
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* пробудить любой процесс чтения */
    return IRQ_HANDLED;
}
```

Этот код, хотя и прост, показывает типичную работу обработчика прерываний. Он, в свою очередь, вызывает ***short_incr_bp***, которая определена так:

```
static inline void short_incr_bp(volatile unsigned long *index, int delta)
{
    unsigned long new = *index + delta;
    barrier( ); /* Не оптимизировать эти две строки вместе */
    *index = (new >= (short_buffer + PAGE_SIZE)) ? short_buffer : new;
}
```

Эта функция была написана тщательно, чтобы обернуть указатель в круговом буфере никогда не получая неправильные значения. Чтобы заблокировать оптимизацию компилятором двух разных строчек функции, здесь используется вызов ***barrier***. Без барьера компилятор мог бы принять решение оптимизировать, удалив переменную **new**, и выполнить присвоение непосредственно ***index**. Такая оптимизация может привести к неправильному значение индекса в течение короткого периода, когда он оборачивается. Приняв меры для предотвращения получения некорректного значения другими потоками, мы можем безопасно, без блокирования, манипулировать указателем кругового буфера.

Файлом устройства, используемого для чтения заполненного во время прерывания буфера, является ***/dev/shortint***. Это специальный файл устройства не был рассмотрен в Главе 9 224 вместе с ***/dev/shortprint***, поскольку его использование является специфичным для обработки прерываний. Внутренняя организация ***/dev/shortint*** специально предназначена для генерации прерывания и отчётности. Запись в устройство генерирует одно прерывание для каждого байта; чтение устройства даёт время, когда было сообщено о каждом прерывании.

Если соединить вместе контакты 9 и 10 разъёма параллельного порта, вы сможете генерировать прерывания, устанавливая самый старший бит байта данных параллельного порта. Это может быть достигнуто записью бинарных данных в ***/dev/short0*** или записью чего-нибудь в ***/dev/shortint***. (* Устройство *shortint* выполняет свою задачу поочередно записывая в параллельный порт 0x00 и 0xFF.)

Следующий код реализует ***read*** и ***write*** для ***/dev/shortint***:

```
ssize_t short_i_read (struct file *filp, char __user *buf, size_t count,
loff_t *f_pos)
{
    int count0;
    DEFINE_WAIT(wait);

    while (short_head == short_tail) {
        prepare_to_wait(&short_queue, &wait, TASK_INTERRUPTIBLE);
        if (short_head == short_tail)
            schedule( );
        finish_wait(&short_queue, &wait);
        if (signal_pending (current)) /* сигнал поступил */
            return -ERESTARTSYS; /* предложить уровню файловой системы
обработать его */
    }
    /* count0 - число читаемых байтов данных */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_head + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    if (copy_to_user(buf, (char *)short_tail, count))
        return -EFAULT;
    short_incr_bp (&short_tail, count);
    return count;
}

ssize_t short_i_write (struct file *filp, const char __user *buf, size_t
count, loff_t *f_pos)
{
    int written = 0, odd = *f_pos & 1;
    unsigned long port = short_base; /* вывод в регистр-защёлку параллельных
данных */
    void *address = (void *) short_base;

    if (use_mem) {
        while (written < count)
            iowrite8(0xff * ((++written + odd) & 1), address);
    } else {
        while (written < count)
            outb(0xff * ((++written + odd) & 1), port);
    }

    *f_pos += count;
    return written;
}
```

Другой специальный файл устройства, ***/dev/shortprint***, использует параллельный порт для управления принтером; вы можете использовать его, если хотите избежать соединения контактов 9 и 10 разъёма D-25. Реализация ***write*** в ***shortprint*** использует круговой буфер для хранения данных, которые будут печататься, а реализация ***read*** аналогична только что показанной (так что вы можете читать время, которое забирает ваш принтер, чтоб съесть каждый символ).

В целях поддержки принтерных операций, обработчик прерывания был незначительно изменён относительно показанного добавлением возможности отправить следующий байт данных в принтер, если для передачи имеется больше данных.

Аргументы обработчика и возвращаемое значение

Обработчику прерывания передаются три аргумента: **irq**, **dev_id** и **regs**, хотя ***short*** и игнорирует их. Давайте посмотрим на роль каждого из них.

Номер прерывания (**int irq**) полезен как информация, которую вы можете напечатать в журнале сообщений, если таковой имеется. Второй аргумент, **void *dev_id**, является видом данных клиента; аргумент **void *** передаётся в ***request_irq*** и этот же указатель затем передаётся обратно как аргумент для обработчика, когда происходит прерывание. Обычно передаётся указатель на вашу структуру данных устройства в **dev_id**, так что драйверу, который управляет несколькими экземплярами одинаковых устройств, не требуется какого-либо дополнительного кода в обработчике прерывания, чтобы выяснить, какое устройство отвечает за текущее событие прерывания.

Типичное использование аргумента в обработчике прерывания выглядит следующим образом:

```
static irqreturn_t sample_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
    struct sample_dev *dev = dev_id;
    /* теперь `dev' указывает на правильный объект оборудования */
    /* .... */
}
```

Типичный код ***open***, связанный с этим обработчиком, выглядит следующим образом:

```
static void sample_open(struct inode *inode, struct file *filp)
{
    struct sample_dev *dev = hwinfo + MINOR(inode->i_rdev);
    request_irq(dev->irq, sample_interrupt,
                0 /* flags */, "sample", dev /* dev_id */);
    /*....*/
    return 0;
}
```

Последний аргумент, **struct pt_regs *regs**, используется редко. Он содержит снимок контекста процессора перед тем, как процессор вошёл в код прерывания. Регистры могут быть использованы для мониторинга и отладки; для обычных задач драйвера устройства они обычно не требуются.

Обработчики прерываний должны вернуть значение указывающее, было ли прерывание обработано на самом деле. Если обработчик обнаружил, что его устройство действительно требует внимания, он должен вернуть **IRQ_HANDLED**; в противном случае возвращаемое значение должно быть **IRQ_NONE**. Вы можете также сгенерировать возвращаемое значение этим макросом:

```
IRQ_RETVAL(handled)
```

где **handled** отличен от нуля, если вы были в состоянии обработать прерывание. Возвращаемое значение используется ядром для выявления и пресечения ложных прерываний. Если ваше устройство не даёт вам способа узнать, действительно ли прервало оно, вы должны вернуть **IRQ_HANDLED**.

Разрешение и запрет прерываний

Есть моменты, когда драйвер устройства должен блокировать генерацию прерываний на какой-то (в надежде на короткий) период времени (мы видели одну такую ситуацию в разделе "Спин-блокировки" 111 в Главе 5 101). Зачастую прерывания должны быть заблокированы при удержании спин-блокировки во избежании взаимоблокировки системы. Есть способы запрета прерываний, которые не осложняют спин-блокировки. Но прежде чем обсудить их, отметим, что запрет прерываний должен быть сравнительно редкой деятельностью даже в драйверах устройств и эта техника никогда не должна использоваться в качестве механизма взаимного исключения внутри драйвера.

Запрет одного прерывания

Иногда (но редко!) драйверу необходимо отключить доставку прерывания для определённой линии прерывания. Ядро предлагает для этой цели три функции, все они объявлены в ***<asm/irq.h>***. Эти функции являются частью API ядра, поэтому мы их опишем, но их использование не рекомендуется в большинстве драйверов. Среди прочего, вы не можете отключить разделяемые линии прерываний, а на современных системах разделяемые прерывания являются нормой. Как говорилось, вот они:

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

Вызов любой из этих функций может обновить маску указанного **irq** в программируемом контроллере прерываний (programmable interrupt controller, PIC), таким образом, запрещая или разрешая указанное прерывание на всех процессорах. Вызовы этих функций могут быть вложенными, если ***disable_irq*** вызвана два раза подряд, требуется дважды вызвать ***enable_irq*** для действительного разрешения прерывания заново. Можно вызывать эти функции из обработчика прерывания, но разрешение вашего собственного прерывания при его обработке обычно нехорошая практика.

disable_irq не только отключает данное прерывание, но также ждёт завершения выполняющегося в настоящее время обработчика прерывания, если таковой имеется. Помните, что если поток, вызывая ***disable_irq***, удерживает какой-либо ресурс (такой, как спин-блокировки), который необходим обработчику прерывания, система может заблокироваться. ***disable_irq_nosync*** отличается от ***disable_irq*** тем, что она возвращается немедленно. Таким образом, использование ***disable_irq_nosync*** немного быстрее, но может оставить драйвер открытым для состояний гонок.

Но зачем же отключать прерывание? Придерживаясь параллельного порта, давайте посмотрим на сетевой интерфейс ***plip***. Устройство ***plip*** использует простой параллельный порт для передачи данных. Так как из разъёма параллельного порта могут быть прочитаны только пять бит, они интерпретируются как четыре бита данных и тактовый/настроечный сигнал. Когда первые четыре бита пакета переданы инициатором (интерфейсом отправки пакета), для прерывания процессора в случае принимающего интерфейса поднимается тактовая линия.

Затем для обработки вновь прибывших данных вызывается обработчик ***plip***.

После приведения устройства в готовность осуществляется передача данных, используя линию настройки для тактирования новых данных для приёмного интерфейса (это может не быть лучшей реализацией, но это необходимо для совместимости с другими пакетными драйверами, использующими параллельный порт). Производительность была бы невыносимой, если бы принимающему интерфейсу пришлось обрабатывать два прерывания для каждого принимаемого байта. Таким образом, драйвер выключает прерывание во время приёма пакета; вместо этого для доставки данных используется цикл опроса и задержки.

Аналогичным образом, из-за того, что линия настройки от приёмника до передатчика используется для подтверждения приёма данных, передающий интерфейс запрещает свою линию прерывания во время передачи пакета.

Запрет всех прерываний

Что делать, если вам необходимо запретить ***все*** прерывания? В версии ядра 2.6 можно отключить всю обработку прерываний на текущем процессоре с одной из следующих двух функций (которые определены в ***<asm/system.h>***):

```
void local_irq_save(unsigned long flags);
void local_irq_disable(void);
```

Вызов ***local_irq_save*** запрещает доставку прерывания на текущий процессор после сохранения текущего состояния прерывания в **flags**. Обратите внимание, что **flags** передаётся непосредственно, не по указателю; так происходит потому что ***local_irq_save*** на самом деле реализована как макрос, а не как функция. ***local_irq_disable*** выключает местную доставку прерывания без сохранения состояния; вы должны использовать эту версию только если вы знаете, что прерывания уже не были запрещены где-то ещё.

Включение прерываний снова осуществляется с помощью:

```
void local_irq_restore(unsigned long flags);
void local_irq_enable(void);
```

Первая версия восстанавливает то состояние, которое было сохранено во **flags** функцией ***local_irq_save***, а ***local_irq_enable*** разрешает прерывания безоговорочно. В отличие от ***disable_irq***, ***local_irq_disable*** не отслеживает множественные вызовы. Если в цепочке вызова запретить прерывания может потребоваться более чем одной функции, должна быть использована ***local_irq_save***.

В ядре версии 2.6 не существует способа запретить все прерывания глобально во всей системе. Разработчики ядра решили, что цена отключения всех прерываний слишком высока и что в любом случае нет никакой необходимости для такой возможности. Если вы работаете со старым драйвером, который делает вызовы таких функций, как ***cli*** и ***sti***, необходимо обновить его для использования правильного блокирования, прежде чем он будет работать под версией 2.6.

Верхние и нижние половины

Одной из основных проблем при обработке прерывания является выполнение в обработчике длительных задач. Часто в ответ на прерывание устройства должна быть

проделана значительная часть работы, но обработчику прерывания необходимо завершиться быстро и не держать прерывания надолго заблокированными. Эти две потребности (работа и скорость) конфликтуют друг с другом, оставляя автора драйвера немного связанным.

Linux (наряду со многими другими системами) решает эту проблему разделяя обработчик прерывания на две половины. Так называемая ***верхняя половина*** является процедурой, которая на самом деле отвечает на прерывание, той, которую вы зарегистрировали с помощью ***request_irq***. ***Нижняя половина*** является процедурой, которая планируется верхней половиной, чтобы быть выполненной позднее, в более безопасное время. Большая разница между верхней половиной обработчика и нижней половиной в том, что во время выполнения нижней половины все прерывания разрешены, вот почему она работает в более безопасное время. В типичном сценарии верхняя половина сохраняет данные устройства в зависимый от устройства буфер, планирует свою нижнюю половину и выходит: эта операция очень быстрая. Затем нижняя половина выполняет всё то, что требуется, такое как пробуждение процессов, запуск другой операции ввода/вывода и так далее. Эта установка позволяет верхней половине обслужить новое прерывание, пока нижняя половина всё ещё работает.

Таким образом разделён почти каждый серьёзный обработчик прерывания. Например, когда сетевой интерфейс сообщает о появлении нового пакета, обработчик только извлекает данные и помещает их на уровень протокола; настоящая обработка пакетов выполняется в нижней половине.

Ядро Linux имеет два различных механизма, которые могут быть использованы для реализации обработки в нижней половине, оба они были представлены в Главе 7 174. Предпочтительным механизмом для обработки в нижней половине часто являются микрозадачи (tasklets), они очень быстры, но весь код микрозадачи должен быть атомарным. Альтернативой микрозадачам являются очереди задач (workqueues), могущие иметь большую задержку, но которые разрешают засыпать.

И снова, обсуждение работает с драйвером ***short***. Загружая ***short*** с соответствующей опцией, можно выполнять обработку прерывания в режиме верхней/нижней половины либо с помощью микрозадачи, либо с помощью очереди задач. В этом случае верхняя половина выполняется быстро; она просто запоминает текущее время и планирует обработку в нижней половине. Нижней половине затем поручено закодировать это время и пробудить любые пользовательские процессы, которые могут ожидать данные.

Микрозадачи

Вспомним, что микрозадачи являются специальной функцией, которая может быть запланирована для запуска в контексте программного прерывания в определяемое системой безопасное время. Они могут быть запланированы для запуска множество раз, но планирование микрозадачи не является накопительным; микрозадача работает только один раз, даже если перед запуском она была запрошена неоднократно. Микрозадача не работает даже параллельно сама с собой, так как она выполняются только один раз, но микрозадачи могут работать параллельно с другими микрозадачами на многопроцессорных системах. Таким образом, если ваш драйвер имеет несколько микрозадач, чтобы избежать конфликта друг с другом, они должны использовать какой-то вид блокировки.

Микрозадачи также гарантированно работают на том же процессоре, как и функция, которая первая запланировала их. Таким образом, обработчик прерывания может быть уверен, что микрозадача не начнёт выполнение перед завершением обработчика. Однако, безусловно, во время работы микрозадачи может быть доставлено другое прерывание, так что блокировка

между микрозадачей и обработчиком прерывания по-прежнему может быть необходима.

Микрозадачи должны быть объявлены с помощью макроса **DECLARE_TASKLET**:

```
DECLARE_TASKLET(name, function, data);
```

name является именем для передачи микрозадаче, **function** является функцией, которая вызывается для выполнения микрозадачи (она получает аргумент **unsigned long** и возвращает **void**) и **data** является значением **unsigned long**, которое будет передано функции микрозадачи.

Драйвер ***short*** декларирует свою микрозадачу следующим образом:

```
void short_do_tasklet(unsigned long);
DECLARE_TASKLET(short_tasklet, short_do_tasklet, 0);
```

Чтобы запланировать микрозадачу для работы, используется функция ***tasklet_schedule***. Если ***short*** загружен с **tasklet=1**, он устанавливает другой обработчик прерывания, который сохраняет данные и планирует микрозадачу следующим образом:

```
irqreturn_t short_tl_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    do_gettimeofday((struct timeval *) tv_head); /* приведение для остановки
предупреждения 'volatile' */
    short_incr_tv(&tv_head);
    tasklet_schedule(&short_tasklet);
    short_wq_count++; /* запоминаем, что поступило прерывание */
    return IRQ_HANDLED;
}
```

Фактическая процедура микрозадачи, ***short_do_tasklet***, будет выполнена в ближайшее время (скажем так), удобное для системы. Как упоминалось ранее, эта процедура выполняет основную работу обработки прерывания; она выглядит следующим образом:

```
void short_do_tasklet (unsigned long unused)
{
    int savecount = short_wq_count, written;
    short_wq_count = 0; /* мы уже удалены из очереди */
    /*
     * Нижняя половина читает массив tv, заполненный верхней половиной,
     * и печатает его в круговой буфер, который затем опустошается
     * читающими процессами
     */

    /* Сначала запишем число произошедших прерываний перед этой нижней
половиной (bh) */
    written = sprintf((char *)short_head,"bh after %6i\n",savecount);
    short_incr_bp(&short_head, written);

    /*
     * Затем запишем значения времени. Пишем ровно 16 байт за раз,
     * так что запись выровнена с PAGE_SIZE
     */

```

```
    do {
        written = sprintf((char *)short_head,"%08u.%06u\n",
                          (int)(tv_tail->tv_sec % 100000000),
                          (int)(tv_tail->tv_usec));
        short_incr_bp(&short_head, written);
        short_incr_tv(&tv_tail);
    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /* пробудить любой читающий процесс
*/
}
```

Среди прочего, этот такслет делает отметку о том, сколько пришло прерываний с момента последнего вызова. Устройства, такие как ***short***, могут генерировать много прерываний за короткий срок, поэтому нередко поступает несколько до выполнения нижней половины. Драйверы должны всегда быть готовы к этому и должны быть в состоянии определить объём работы на основе информации, оставленной верхней половиной.

В приведённом выше примере есть ошибка:

void short_do_tasklet (unsigned long unused)
{
   int savecount = short_wq_count, written;

Очищать переменную <short_wq_count> таким способом небезопасно, если в это время произойдёт аппаратное прерывание, мы потеряем один (или может быть больше) тиков для <short_wq_count>. Для решения этой проблемы следует использовать <spin_lock_irqsave>

short_wq_count = 0; /* мы уже удалены из очереди */

Ответ автора: вы правы - есть очень небольшое окошко, в которым происходит состязание с обработчиком прерывания. Это может исправлено, как и предложено, с помощью спин-блокировки, хотя, возможно, более подходящим инструментом в данной ситуации был бы seqlock, последовательная блокировка.

Очереди задач

Напомним, что очереди задач вызывают функцию когда-нибудь в будущем в контексте специального рабочего процесса. Поскольку функция очереди задач выполняется в контексте процесса, она может заснуть, если это будет необходимо. Однако, вы не можете копировать данные из очереди задач в пользовательское пространство, если вы не используете современные методики, которые мы покажем в Главе 15 395; рабочий процесс не имеет доступа к адресному пространству любого другого процесса.

Драйвер ***short***, если он загружен с опцией **wq**, установленной в ненулевое значение, для обработки в его нижней половине использует очередь задач. Он использует системную очередь задач по умолчанию, так что не требуется особого кода установки; если ваш драйвер имеет специальные требования латентности (задержки) (или может спать в течение длительного времени в функции очереди задач), вы можете создать свою собственную, предназначенную для этого очередь задач. Нам необходима структура **work_struct**, которая объявлена и проинициализирована так:

```
static struct work_struct short_wq;

    /* это строка в short_init( ) */
    INIT_WORK(&short_wq, (void (*)(void *)) short_do_tasklet, NULL);
```

Нашей рабочей функцией является ***short_do_tasklet***, которую мы уже видели в предыдущем разделе.

При работе с очередью задач ***short*** устанавливает ещё один обработчик прерывания, который выглядит следующим образом:

```
irqreturn_t short_wq_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    /* Получаем информацию о текущем времени. */
    do_gettimeofday((struct timeval *) tv_head);
    short_incr_tv(&tv_head);

    /* Помещаем нижнюю половину в очередь. Не беспокоимся о постановке в
очередь много раз */
    schedule_work(&short_wq);

    short_wq_count++; /* запоминаем, что поступило прерывание */
    return IRQ_HANDLED;
}
```

Как вы можете видеть, обработчик прерывания очень похож на версию с микрозадачей за исключением того, что для организации обработки в нижней половине он вызывает ***schedule_work***.

Разделяемые прерывания

Понятие конфликта прерываний является почти синонимом архитектуры ПК. В прошлом, линии прерываний на ПК не были способны обслужить более, чем одно устройство и их никогда не было достаточно. В результате, разочарованные пользователи часто проводили много времени с открытым корпусом своего компьютера, пытаясь найти способ сделать, чтобы вся их периферия хорошо жила вместе.

Конечно, современное техническое оснащение было разработано, чтобы позволить совместное использование прерываний; шина PCI требует этого. Таким образом, ядро Linux поддерживает совместное использование прерывания на всех шинах, даже на тех (таких, как шины ISA), где совместное использование традиционно не поддерживалось. Драйверы устройств для ядра версии 2.6 должны быть написаны для работы с разделяемыми прерываниями, если целевое оборудование поддерживает этот режим работы. К счастью, работа с разделяемыми прерываниями в большинстве случаев проста.

Установка обработчика разделяемого прерывания

Разделяемые прерывания устанавливаются через ***request_irq***, похожую на такую же для неразделяемых, но есть два отличия:

- В аргументе **flags** при запросе прерывания должен быть указан бит **SA_SHIRQ**.
- Аргумент **dev_id** должен быть уникальным. Он будет любым указателем в адресном пространстве модуля, но **dev_id**, безусловно, не может быть установлен в NULL.

Ядро хранит список обработчиков разделяемых прерываний, связанных с прерыванием, и **dev_id** может рассматриваться как подпись, которая их различает. Если бы два драйвера зарегистрировали **NULL** в качестве подписи на одном прерывании, всё могло бы запутаться во время выгрузки, в результате чего ядро при получении прерывания сказало бы Oops. По этой причине современные ядра громко жалуются, если при регистрации разделяемых прерываний в **dev_id** передан NULL. Когда запрашивается разделяемое прерывание ***request_irq*** успешна, если одно из следующего верно:

- Эта линия прерывания свободна.
- Все обработчики, уже зарегистрированные для этой линии, также уточнили, что это прерывание будет разделяться.

Всякий раз, когда два или более драйверов делят линию прерывания и оборудование прерывает процессор по этой линии, ядро вызывает каждый зарегистрированный обработчик для этого прерывания, передавая каждому его собственный **dev_id**. Таким образом, обработчик разделяемого прерывания должен быть в состоянии определить свои собственные прерывания и должен быстро выйти, если прерывание сгенерировало не его устройство. Будьте уверены, что при вызове вашего обработчика возвращаете **IRQ_NON**, если находите, что это устройство не прерывало.

Если вам необходимо проверить устройство до запроса линии прерывания, ядро не сможет вам помочь. Нет функции проверки для обработчиков разделяемых прерываний. Стандартный механизм проверки работает, если используемая линия свободна, но если линия уже занята другим драйвером с возможностью разделения, проба закончится неудачно, даже если ваш драйвер будет работать прекрасно. К счастью, большинство оборудования, разработанного для разделения прерывания, также способно сказать процессору, какое прерывание оно использует, тем самым устраняя необходимость явного зондирования.

Отключение обработчика производится в обычном порядке с использованием ***free_irq***. Для выбора правильного обработчика для отключения из списка обработчиков для разделяемого прерывания здесь используется аргумент **dev_id**. Вот почему указатель **dev_id** должен быть уникальным.

Драйверу, использующему обработчик разделяемого прерывания, необходимо позаботиться ещё об одном: он не может играть с ***enable_irq*** или ***disable_irq***. Если это произойдёт, всё может поломаться для других устройств, разделяющих эту же линию; запрет прерываний другого устройства даже на короткое время может создать задержки, которые являются проблематичными для такого устройства и его пользователя. Как правило, программист должен помнить, что его драйвер не является собственником такого прерывания и его поведение должно быть более "социальным", чем это необходимо, если он один владеет линией прерывания.

Работа обработчика

Как указывалось ранее, когда ядро получает прерывание, вызываются все зарегистрированные обработчики. Обработчики разделяемых прерываний должны быть в состоянии различить прерывания, которые необходимо обработать, и прерывания, генерируемые другими устройствами.

Загрузка ***short*** с вариантом **shared=1** устанавливает следующий обработчик вместо

используемого по умолчанию:

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value, written;
    struct timeval tv;

    /* Если это не для short, немедленно вернуться */
    value = inb(short_base);
    if (!(value & 0x80))
        return IRQ_NONE;

    /* очистить бит прерывания */
    outb(value & 0x7F, short_base);

    /* остальное остаётся неизменным */

    do_gettimeofday(&tv);
    written = sprintf((char *)short_head,"%08u.%06u\n",
                      (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    short_incr_bp(&short_head, written);
    wake_up_interruptible(&short_queue); /* разбудить любой читающий процесс
*/
    return IRQ_HANDLED;
}
```

Объяснение прямо здесь. Так как параллельный порт не имеет для проверки бита "ожидание прерывания", обработчик использует для этой цели бит ACK (подтверждение). Если бит установлен, прерывание предназначено для ***short*** и обработчик очищает этот бит.

Обработчик сбрасывает этот бит, обнуляя старший бит порта данных параллельного интерфейса - ***short*** предполагает, что контакты 9 и 10 соединены вместе. Если прерывание генерирует одно из других устройств, разделяющих прерывание с ***short***, ***short*** видит, что его собственная линия является неактивной и ничего не делает.

Полнофункциональные драйверы, вероятно, конечно, поделят работу на верхнюю и нижнюю половины, но это легко добавить и это не имеет никакого влияния на код, реализующий совместное использование прерывания. Вероятно, реальный драйвер будет также использовать аргумент **dev_id** для определения какое из, возможно многих, устройств могло сгенерировать прерывание.

Заметим, что если вы используете принтер (вместо провода для перемычки) для проверки управления прерываниями с ***short***, этот обработчик разделяемого прерывания не будет работать как рассказывалось, потому что протокол принтера не позволяет совместное использование и драйвер не сможет узнать, было ли прерывание от принтера.

Интерфейс /proc и разделяемые прерывания

Установка в систему обработчиков разделяемых прерываний не влияет на ***/proc/stat***, который даже не знает об обработчиках. Однако, ***/proc/interrupts*** немного изменяется.

Все обработчики, установленные для одного номера прерывания, появляются в одной строчке ***/proc/interrupts***. Следующий вывод (на системе x86_64) показывает, как

отображаются обработчики разделяемых прерываний:

```
           CPU0
 0:    892335412       XT-PIC  timer
 1:       453971       XT-PIC  i8042
 2:            0       XT-PIC  cascade
 5:            0       XT-PIC  libata, ehci_hcd
 8:            0       XT-PIC  rtc
 9:            0       XT-PIC  acpi
 10:    11365067       XT-PIC  ide2, uhci_hcd, uhci_hcd, SysKonnect SK-98xx,
EMU10K1
 11:     4391962       XT-PIC  uhci_hcd, uhci_hcd
 12:         224       XT-PIC  i8042
 14:     2787721       XT-PIC  ide0
 15:      203048       XT-PIC  ide1
NMI:       41234
LOC:   892193503
ERR:         102
MIS:           0
```

Эта система имеет несколько общих линий прерываний. IRQ 5 используется для контроллеров serial ATA и USB 2.0 ; IRQ 10 имеет несколько устройств, включая контроллер IDE, два USB контроллера, интерфейс Ethernet и звуковую карту; и IRQ 11 также используется двумя USB контроллерами.

Ввод/вывод, управлемый прерыванием

Всякий раз, когда передача данных в или из управляемого оборудования может быть отложена по любой причине, автору драйвера следует реализовывать буферизацию. Буферы данных помогают отделить передачу и приём данных от системных вызовов ***write*** и ***read***, а также повысить общую производительность системы.

Хороший буферный механизм приводит к ***вводу/выводу, управляемому прерываниями***, в котором входной буфер заполнен во время прерывания и очищается процессом, которые читает устройство; выходной буфер заполняется процессами, которые пишут в устройство, и опустошается во время прерывания. Примером управляемого прерыванием вывода является реализация ***/dev/shortprint***. Чтобы управляемая прерыванием передача данных происходила успешно, оборудование должно быть способно генерировать прерывания со следующей семантикой:

- Для ввода, устройство прерывает процессор, когда получены новые данные и они готовы для получения процессором системы. Фактические действия для выполнения зависят от того, использует ли устройство порты ввода/вывода, отображение на память, или DMA.
- Для вывода, устройство обеспечивает прерывание или когда оно готово принять новые данные, или для подтверждения успешной передачи данных. Устройства, использующие отображение на память, и DMA-совместимые устройства обычно генерируют прерывания, чтобы сообщить системе, что они завершили работу с буфером.

Отношения синхронизации между ***read*** или ***write*** и фактическим получением данных представлены в разделе "Блокирующие и неблокирующие операции" 143 в Главе 6. 128

Пример буферированной записи

Мы уже несколько раз упоминали драйвер ***shortprint***; теперь пришло время действительно посмотреть на него. Этот модуль реализует очень простой, ориентированной на вывод драйвер для параллельного порта; однако, этого достаточно, чтобы разрешить печать файлов. Однако, если вы решили проверить вывод этого драйвера, помните, что вы должны передать принтеру файл в формате, который он понимает; не все принтеры хорошо реагируют, когда получают поток произвольных данных.

Драйвер ***shortprint*** поддерживает одностраничный круговой буфера вывода. Когда процесс пользовательского пространства записывает данные в это устройство, эти данные поступают в буфер, но метод ***write*** не выполняет какой-либо фактический ввод/вывод. Вместо этого, ядро ***shortp_write*** выглядит следующим образом:

```
    while (written < count) {
        /* Выйти обратно, пока не освободится место в буфере. */
        space = shortp_out_space( );
        if (space <= 0) {
            if (wait_event_interruptible(shortp_out_queue,
                (space = shortp_out_space( )) > 0))
                goto out;
        }

        /* Переместить данные в буфер. */
        if ((space + written) > count)
            space = count - written;
        if (copy_from_user((char *) shortp_out_head, buf, space)) {
            up(&shortp_out_sem);
            return -EFAULT;
        }
        shortp_incr_out_bp(&shortp_out_head, space);
        buf += space;
        written += space;

        /* Если вывод неактивен, сделать его активным. */
        spin_lock_irqsave(&shortp_out_lock, flags);
        if (! shortp_output_active)
            shortp_start_output( );
        spin_unlock_irqrestore(&shortp_out_lock, flags);
    }

out:
    *f_pos += written;
```

Доступ к круговому буферу контролирует семафор (**shortp_out_sem**); ***shortp_write*** получает этот семафор только перед вышеприведённым фрагментом кода. Удерживая семафора, она пытается передать данные в круговой буфер. Функция ***shortp_out_space*** возвращает размер доступного непрерывного пространства (так что нет необходимости беспокоиться о переполнении буфера); если этот размер равен 0, драйвер ждёт, пока не освободится некоторое пространство. Затем копирует в буфер столько данных, сколько может.

Как только появились данные для вывода, ***shortp_write*** должен гарантировать, что данные записываются в устройство. Фактическая запись выполняется функцией ***workqueue***; ***shortp_write*** должен стартовать эту функцию, если она ещё не работает. После получения отдельной спин-блокировки, которая контролирует доступ к переменным, используемым на стороне потребителя выходного буфера (в том числе ***shortp_output_active***), она вызывает в

случае необходимости ***shortp_start_output***. Тогда это всего лишь вопрос пометки, сколько данных было "записано" в буфер, и возвращения. Функция, которая начинает процесс вывода, выглядит следующим образом:

```
static void shortp_start_output(void)
{
    if (shortp_output_active) /* Никогда не должно случиться */
         return;

    /* Установить на таймер 'пропущенное прерывание' */
    shortp_output_active = 1;
    shortp_timer.expires = jiffies + TIMEOUT;
    add_timer(&shortp_timer);

    /* И получить работу процесса. */
    queue_work(shortp_workqueue, &shortp_work);
}
```

Реальность работы с аппаратурой такова, что вы можете иногда терять прерывания от устройства. Когда это случается, вы действительно не хотите, чтобы ваш драйвер остановился навсегда до перезагрузки системы; это недружественный способ поведения. Намного лучше понять, что прерывание было пропущено, подхватить куски и продолжить работу. С этой целью ***shortprint*** устанавливает таймер ядра, когда выводит данные на устройство. Если таймер истекает, мы, возможно, пропустили прерывание. Мы вскоре рассмотрим таймерную функцию, но в данный момент давайте разберёмся с функциональностью основного вывода. Это реализовано в нашей функции ***workqueue***, которая, как вы можете видеть выше, планируется здесь. Суть этой функции выглядит следующим образом:

```
spin_lock_irqsave(&shortp_out_lock, flags);

/* Имеется ли что-то для записи? */
if (shortp_out_head == shortp_out_tail) { /* пусто */
    shortp_output_active = 0;
    wake_up_interruptible(&shortp_empty_queue);
    del_timer(&shortp_timer);
}
/* Нет, пишем другой байт */
else
    shortp_do_write( );

/* Если кто-то ждёт, можно разбудить их. */
if (((PAGE_SIZE + shortp_out_tail -shortp_out_head) % PAGE_SIZE) >
SP_MIN_SPACE)
{
    wake_up_interruptible(&shortp_out_queue);
}
spin_unlock_irqrestore(&shortp_out_lock, flags);
```

Поскольку мы имеем дело с общими переменными на стороне вывода, мы должны получить спин-блокировку. Затем посмотрим, есть ли дополнительные данные для отправки; если нет, мы обращаем внимание, что вывод больше не активен, удаляем таймер и пробуждаем любого, кто мог бы ожидать полной очистки очереди (такой вид ожидания завершается, когда устройство закрывается). Если, вместо этого, есть оставшиеся для записи данные, мы

вызываем ***shortp_do_write*** для фактической отправки байта в аппаратуру.

Затем, после того, как мы смогли освободить место в выходном буфере, мы учитываем пробуждение любых процессов, ожидающих, чтобы добавить данные в этот буфер. Однако, мы не выполняем безоговорочное пробуждение; вместо этого, мы ждём минимального количества доступного места. Нет никакого смысла в пробуждении записи каждый раз, когда мы забираем один байт из буфера; стоимость пробуждения процесса, планирования его для запуска и помещения его обратно в сон слишком высока для этого. Наоборот, мы должны подождать, пока этот процесс сможет переместить значительный объём данных в буфер за раз. Эта техника обычна в буферированных, управляемых прерываниями драйверах.

Для полноты, вот код, который записывает данные в порт:

```
static void shortp_do_write(void)
{
    unsigned char cr = inb(shortp_base + SP_CONTROL);

    /* Что-от произошло; сбросить таймер */
    mod_timer(&shortp_timer, jiffies + TIMEOUT);

    /* Стробировать вывод байта в устройство */
    outb_p(*shortp_out_tail, shortp_base+SP_DATA);
    shortp_incr_out_bp(&shortp_out_tail, 1);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr | SP_CR_STROBE, shortp_base+SP_CONTROL);
    if (shortp_delay)
        udelay(shortp_delay);
    outb_p(cr & ~SP_CR_STROBE, shortp_base+SP_CONTROL);
}
```

Здесь мы сбрасываем таймер, чтобы отразить тот факт, что мы добились некоторого прогресса, стробируем вывод байта в устройство и обновляем указатель кругового буфера.

Функция ***workqueue*** не повторяет сама себя непосредственно, поэтому в устройство будет записан только один байт. В определённый момент принтер, его медленным способом, будет потреблять байт и станет готовым к следующему; затем он прервёт процессор. Обработчик прерывания, используемых в ***shortprint***, короткий и простой:

```
static irqreturn_t shortp_interrupt(int irq, void *dev_id, struct pt_regs
*regs)
{
    if (! shortp_output_active)
        return IRQ_NONE;
    /* Запомнить время и запланировать паузу для функции workqueue */
    do_gettimeofday(&shortp_tv);
    queue_work(shortp_workqueue, &shortp_work);
    return IRQ_HANDLED;
}
```

Поскольку параллельный порт не требует явного подтверждения прерывания, всё, что обработчику прерывания действительно необходимо сделать, это сказать ядру снова запустить функцию ***workqueue***.

Что делать, если прерывание никогда не приходит? Код драйвера, который мы видели до сих пор, просто бы остановился. Чтобы этого не произошло, мы устанавливаем обратный таймер несколько страниц назад. Функция, которая выполняется, когда это время закончится:

```
static void shortp_timeout(unsigned long unused)
{
    unsigned long flags;
    unsigned char status;

    if (! shortp_output_active)
        return;
    spin_lock_irqsave(&shortp_out_lock, flags);
    status = inb(shortp_base + SP_STATUS);

    /* Если принтер всё ещё занят, мы просто сбрасываем таймер */
    if ((status & SP_SR_BUSY) == 0 || (status & SP_SR_ACK)) {
        shortp_timer.expires = jiffies + TIMEOUT;
        add_timer(&shortp_timer);
        spin_unlock_irqrestore(&shortp_out_lock, flags);
        return;
    }

    /* Иначе мы, видимо, пропустили прерывание. */
    spin_unlock_irqrestore(&shortp_out_lock, flags);
    shortp_interrupt(shortp_irq, NULL, NULL);
}
```

Если не предполагается активного вывода, функция таймера просто возвращается; это предохраняет таймер от повторного самостоятельного старта при выключении. Затем после получения блокировки мы запрашиваем состояние порта; если он утверждает, что занят, он просто ещё не удосужился нас прервать, так что мы сбрасываем таймер и возвращаемся. Принтеры могут иногда занять очень много времени, чтобы сделать себя готовыми; примите во внимание принтер, который стоит без бумаги, пока кто-нибудь не пришёл после длинных выходных. В такой ситуации нечего делать, кроме как терпеливо ждать, пока что-то изменится.

Однако, если принтер утверждает, что готов, мы, видимо, пропустили своё прерывание. В этом случае мы просто вызываем наш обработчик прерывания вручную, чтобы процесс вывода снова получил движение.

Драйвер ***shortprint*** не поддерживает чтение из порта; вместо этого он ведёт себя как ***shortint*** и возвращает информацию о времени, когда происходили прерывания. Впрочем, реализация управляемого прерыванием метода ***read*** будет очень похожа на то, что мы видели. Данные из устройства читались бы в буфер драйвера; он бы копировал наружу в пользовательское пространство, только когда в буфере накапливался значительный объём данных, полностью удовлетворял запрос ***read*** или происходил бы какой-то вид превышения времени ожидания.

Краткая справка

В этой главе были введены эти символы, связанные с управлением прерываниями:

#include <linux/interrupt.h>
int request_irq(unsigned int irq, irqreturn_t (*handler)( ), unsigned long flags, const

char *dev_name, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
Вызовы, которые регистрируют и отменяют регистрацию обработчика прерывания.

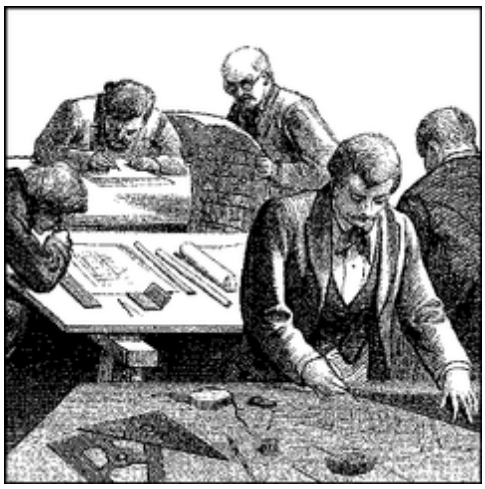
#include <linux/irq.h.h>
int can_request_irq(unsigned int irq, unsigned long flags);
Эта функция, доступная на i386 и x86_64, возвращает ненулевое значение, если попытка получить данную линию прерывания успешна.

#include <asm/signal.h>
SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM
Флаги для ***request_irq***. **SA_INTERRUPT** запрашивает установку быстрого обработчика (в отличие от медленного). **SA_SHIRQ** устанавливает обработчик разделяемого прерывания и третий флаг сообщает, что время прерывания может быть использовано для генерации энтропии в системе.

/proc/interrupts
/proc/stat
Узлы файловой системы, которые сообщают информацию об аппаратных прерываниях и установленных обработчиках.

unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
Функции, которые используются драйвером, когда он должен выполнить зондирование для определения, какая линия прерывания будет использоваться устройством. Результат ***probe_irq_on*** должен быть передан обратно в ***probe_irq_off*** после того, как прерывание было сгенерировано. Возвращаемое значение ***probe_irq_off*** является обнаруженным номером прерывания.
IRQ_NONE
IRQ_HANDLED
IRQ_RETVAL(int x)
Возможные возвращаемые значения из обработчика прерывания, указывающие, присутствовало ли фактическое прерывание от устройства.
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
Драйвер может разрешить или запретить сообщения о прерывании. Если оборудование пробует генерировать прерывание при запрещённых прерываниях, прерывание потеряно навсегда. Драйвер, использующий обработчик разделяемого прерывания, не должен использовать эти функции.
void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
Используйте ***local_irq_save*** для запрета прерываний на местном процессоре и запоминания их предыдущего состояния. Для восстановления предыдущего состояния прерывания в ***local_irq_restore*** может быть передан **flags**.
void local_irq_disable(void);
void local_irq_enable(void);
Функции, которые безоговорочного запрещают и разрешают прерывания на текущем процессоре.

Глава 11, Типы данных в ядре



Прежде чем перейти к более сложным темам, мы должны остановиться для быстрых замечаний о вопросах переносимости. Современные версии ядра Linux весьма переносимы, работают на большой числе различных архитектур. Учитывая мультиплатформенный характер Linux, драйверы, предназначенные для серьёзного использования также должны быть переносимыми.

Но основными проблемами с кодом ядра являются возможность доступа к объектам данных известной длина (например, структуры данных файловой системы или регистры на плате устройства) и эксплуатация возможностей разных процессоров (32-х разрядных и 64-х разрядной архитектур и, возможно, также 16-ти разрядных).

Некоторые из проблем, с которыми столкнулись разработчики ядра при переносе кода x86 на новые архитектуры, были связаны с неправильной типизацией данных. Соблюдением строгой типизации данных и компиляцией с флагами ***-Wall -Wstrict-prototypes*** можно предотвратить большинство ошибок.

Типы данных, используемые данными ядра, разделены на три основных класса: стандартные типы Си, такие как **int**, типы с явным размером, такие как **u32** и типы, используемые для определённых объектов ядра, такие как **pid_t**. Мы собираемся показать когда и каким образом должен быть использован каждый из трёх типовых классов. В заключительных разделах главы говорится о некоторых других типичных проблемах, с которыми можно столкнуться при переносе кода драйвера с x86 на другие платформы и представляется обобщённая поддержка связных списков, экспортируемых соответствующими заголовками ядра.

Если вы будете следовать предлагаем принципам, ваш драйвер должен компилироваться и работать даже на платформах, на которых вы не сможете его протестировать.

Использование стандартных типов языка Си

Хотя большинство программистов привыкли свободно использовать стандартные типы, такие как **int** и **long**, написание драйверов устройств требует некоторой осторожности, чтобы избежать конфликтов типов и неясных ошибок.

Проблема в том, что вы не можете использовать стандартные типы, когда необходим "2-х байтовый наполнитель" или "что-то, представляющее 4-х байтовые строки", потому что обычные типы данных Си не имеют одинакового размера на всех архитектурах. Чтобы показать размер данных различных типов Си, в файлы примеров, представленных на FTP сайте O'Reilly, в каталог ***misc-progs*** была включена программа ***datasize***. Это пример запуска программы на системе i386 (последние четыре показанных типа будут введены в следующем разделе):

```
morgana% misc-progs/datasize
arch   Size: char short int long ptr long-long u8 u16 u32 u64
i386           1    2    4    4   4      8        1   2   4   8
```

Программа может быть использована, чтобы показать, что целочисленные **long** и указатели имеют различные размеры на 64-х разрядных платформах, что демонстрирует запуск программы на другом компьютере с Linux:

```
arch   Size: char short int long ptr long-long u8 u16 u32 u64
i386           1    2    4    4   4      8        1   2   4   8
alpha          1    2    4    8   8      8        1   2   4   8
armv4l         1    2    4    4   4      8        1   2   4   8
ia64           1    2    4    8   8      8        1   2   4   8
m68k           1    2    4    4   4      8        1   2   4   8
mips           1    2    4    4   4      8        1   2   4   8
ppc            1    2    4    4   4      8        1   2   4   8
sparc          1    2    4    4   4      8        1   2   4   8
sparc64        1    2    4    4   4      8        1   2   4   8
x86_64         1    2    4    8   8      8        1   2   4   8
```

Интересно отметить, что архитектура SPARC 64 работает с 32-х разрядным пространством пользователя, имея там указатели размерностью 32 бита, хотя они и 64 бита в пространстве ядра. Это может быть проверено загрузкой модуля **kdatasize** (доступного в каталоге ***miscmodules*** файлов примеров). Модуль сообщает информацию о размерах во время загрузки используя ***printk*** и возвращает ошибку (то есть, нет необходимости его выгружать):

```
kernel: arch   Size: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64        1    2    4    8   8      8        1   2   4   8
```

Хотя вы должны быть осторожны при смешивании различных типов данных, иногда для этого бывают веские причины. Одной из таких ситуаций является адресация памяти, которая особо касается ядра. Хотя концептуально адреса являются указателями, управление памятью зачастую лучше выполняется с использованием типа беззнакового целого; ядро обрабатывает физическую память, как огромный массив, и адрес память является просто индексом внутри массива. Кроме того, указатель легко разыменовывается; при непосредственной работе с адресами памяти вы почти никогда не хотите их разыменовывать таким образом. Использование целочисленного типа мешает такому разыменованию, что позволяет избежать ошибок. Таким образом, обычные адреса памяти в ядре, как правило, **unsigned long**, эксплуатируя тот факт, что указатели и целые **long** всегда одного и того же размера, по крайней мере, на всех платформах в настоящее время поддерживаемых Linux.

Кому это интересно, стандарт C99 определяет для целочисленной переменной типы **intptr_t** и **uintptr_t**, которая может содержать значение указателя. Однако, эти типы почти не использовались в ядре версии 2.6.

Определение точного размера элементам данных

Иногда код ядра требует элементов данных определённого размера, может быть, чтобы соответствовать предопределённым бинарным структурам (* Это происходит при чтении таблицы разделов, когда запускается бинарный файл, либо при декодировании сетевого пакет.), для общения с пользовательским пространством, или выравнивания данных в структурах включением "добивочных" полей (отсылаем к разделу "Выравнивание данных" 280 для получения информации о вопросах согласований).

Когда вам необходимо знать размер ваших данных, ядро предлагает для использования следующие типы данных. Все типы объявлены в ***<asm/types.h>***, который, в свою очередь, подключается через ***<linux/types.h>***:

```
u8;  /* беззнаковый байт byte (8 бит) */
u16; /* беззнаковое слово (16 бит) */
u32; /* беззнаковая 32-х битовая переменная */
u64; /* беззнаковая 64-х битовая переменная */
```

Существуют соответствующие знаковые типы, но они требуются редко; если они вам необходимы, просто заменить в имени **u** на **s**.

Если программе пространства пользователя необходимо использовать эти типы, она может прибавить префикс к именам с двойным подчеркиванием: **__u8** и другие типы определяются независимо от **__KERNEL__**. Если, например, драйверу необходимо обменяться бинарными структурами с программой, работающей в пользовательском пространстве посредством ***ioctl***, файлы заголовков должны объявить 32-х разрядные поля в структурах как **__u32**.

Важно помнить, что эти типы специфичны для Linux, и их использование препятствует переносимости программ на другие виды Unix. Системы с последними компиляторами поддерживают типы стандарта C99, такие как **uint8_t** и **uint32_t**; если переносимость вызывает обеспокоенность, вместо разнообразных специфичных для Linux могут быть использованы эти типы.

Можно также отметить, что иногда ядро использует обычные типы, такие, как **unsigned int**, для элементов, размерность которых зависит от архитектуры. Это обычно сделано для обратной совместимости. Когда в версии 1.1.67 были введены **u32** и друзья, разработчики не могли изменить существующие структуры данных в соответствие с новыми типами, потому что компилятор выдаёт предупреждение, когда есть несоответствие между типом поля структуры и присваиваемой ему переменной (* На самом деле, компилятор сигнализирует о несоответствии типа, даже если два типа просто разные названия одного и того же объекта, такие как unsigned long и u32 на ПК.). Линус не ожидал, что операционная система (ОС), которую он написал для собственного пользования, станет мультиплатформенной; как следствие, старые структуры иногда типизированы небрежно.

Типы, специфичные для интерфейса

Некоторые из наиболее часто используемых типов данных в ядре имеют свои собственные операторы **typedef**, предотвращая таким образом любые проблемы переносимости. Например, идентификатор процесса (pid) обычно **pid_t**, вместо **int**. Использование **pid_t** маскирует любые возможные различия в фактическом типе данных. Мы используем выражение ***специфичные для интерфейса*** для обозначения типов, определённых библиотекой, с тем, чтобы предоставить интерфейс к определённой структуре данных.

Отметим, что в последнее время были определено сравнительно мало новых типов, специфичных для интерфейса. Использование оператора **typedef** вошло в немилость у многих разработчиков ядра, которые предпочли бы видеть реальную информацию об используемом типе непосредственно в коде, а не скрывающейся за определённым пользователем типом. Однако, многие старые специфичные для интерфейса типы остаются в ядре и они не уйдут в ближайшее время.

Даже тогда, когда нет определённого интерфейсного типа, всегда важно использовать правильный типа данных для совместимости с остальной частью ядра. Счётчик тиков, например, всегда **unsigned long**, независимо от его фактического размера, так что при работе с тиками всегда должен быть использован тип **unsigned long**. В этом разделе мы сосредоточимся на использовании типов **_t**.

Многие **_t** типы определены в ***<linux/types.h>***, но этот список редко бывает полезен. Когда вам потребуется определённый тип, вы будете находить его в прототипе функции, которую вам необходимо вызвать, или в структурах используемых данных.

Каждый раз, когда драйвер использует функции, которые требуют такого "заказного" типа, и вы не следуете соглашению, компилятор выдаёт предупреждение; если вы используете флаг компилятора ***-Wall*** и тщательно удаляете все предупреждения, вы можете быть уверены, что ваш код является переносимым.

Основной проблемой с элементами данных **_t** является то, что когда вам необходимо их распечатать, не всегда просто выбрать правильный формат ***printk*** или ***printf*** и предупреждения, которые вы убираете на одной архитектуре, снова появляются на другой. Например, как бы вы напечатали **size_t**, который является **unsigned long** на одних платформах и **unsigned int** на других?

Если вам необходимо распечатать некоторые специфичные для интерфейса данные, наилучшим способом сделать это является приведение типа значения к наибольшему возможному типу (как правило, **long** или **unsigned long**) и затем печать их через соответствующий формат. Такой вид корректировки не будет генерировать ошибки или предупреждения, потому что формат соответствует типу, и вы не потеряете биты данных, потому что приведение либо ничего не делает, либо увеличивает объект до большего типа данных.

На практике, объекты данных, о которых мы говорим, обычно не предназначены для печати, так что вопрос распространяется только на сообщения об отладке. Чаще всего коду необходимо только запоминать и сравнивать типы, специфичные для интерфейса, в дополнение к передаче их в качестве аргумента в библиотечные функции или функции ядра.

Хотя **_t** типы являются правильным решением для большинства ситуаций, иногда правильного типа не существует. Это происходит на старых интерфейсах, которые ещё не были очищены.

Одним неоднозначным моментом, который мы нашли в заголовках ядра, является типизация данных для функций ввода/вывода, которые определены небрежно (смотрите раздел "Зависимость от платформы" 231 в Главе 9 224). Небрежные типы в основном существуют по историческим причинам, но это может создать проблемы при написании кода. Например, можно попасть в беду, меняя аргументы функций, таких как **outb**; если бы существовал тип

port_t, компилятор бы нашёл этот тип ошибки.

Другие вопросы переносимости

В дополнение к типам данным есть несколько других программных вопросов, чтобы иметь ввиду при написании драйвера, если вы хотите, чтобы он был переносим между платформами Linux. Общее правило состоит в том, чтобы относиться с подозрением к явным постоянным значениям. Обычно код параметризован помощью макросов препроцессора. В этом разделе перечисляются наиболее важные проблемы переносимости. Всякий раз, когда вы сталкиваетесь другими значениями, которые были параметризованы, вы можете найти подсказки в файлах заголовков и драйверах устройств, распространяемых с официальным ядром.

Интервалы времени

Когда речь идёт о временных интервалах, не думайте, что есть 1000 тиков в секунду. Хотя в настоящее время для архитектуры i386 это справедливо, не каждая платформа Linux работает на этой скорости. Предположение может быть ложным даже для x86, если вы играете со значением **HZ** (как делают некоторые люди), и никто не знает, что произойдёт в будущих ядрах. Всякий раз, когда вы рассчитываете интервалы времени используя тики, масштабируйте ваши времена с помощью **HZ** (число прерываний таймера в секунду). Например, чтобы проверить ожидание в пол-секунды, сравнивайте прошедшее время с **HZ/2**. Более широко, число тиков, соответствующее **msec** миллисекунд, всегда **msec*HZ/1000**.

Размер страницы

При играх с памятью помните, что память страницы - **PAGE_SIZE** байт, а не 4 Кб. Предполагая, что размер страницы составляет 4 Кбайт и явное указание этого значения является распространённой ошибкой среди программистов ПК, вместо этого, поддерживаемые платформы показывают размер страницы от 4 Кб до 64 Кб и иногда они различаются между разными реализациями одной и той же платформы. Соответствующими макросами являются **PAGE_SIZE** и **PAGE_SHIFT**. Последний содержит число битов для сдвига адреса, чтобы получить номер страницы. В настоящее время число составляет 12 или больше для страниц, которые 4 Кб и более. Макросы определены в ***<asm/page.h>***; программы пространства пользователя могут использовать библиотечную функцию ***getpagesize***, если им когда-нибудь потребуется такая информация.

Давайте посмотрим на нетривиальную ситуацию. Если драйверу необходимо 16 Кб для временных данных, не следует указывать **order** как **2** для ***get_free_pages***. Вам необходимо переносимое решение. Такое решение, к счастью, был написано разработчиками ядра и называется ***get_order***:

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

Помните, что аргумент **get_order** должно быть степенью двойки.

Порядок байт

Будьте внимательны, чтобы не делать предположений о порядке байт. Если ПК сохраняют многобайтовые величины начиная с младшего байта (сначала младший конец, то есть little-

endian), некоторые высокоуровневые платформы делают это другим способом (big-endian). Когда это возможно, ваш код должен быть написан так, чтобы не заботиться о порядке байт в данных, которыми он манипулирует. Однако, иногда драйверу необходимо построить целое число из раздельный байтов или сделать обратное, или он должен взаимодействовать с устройством, которое ожидает определённый порядок.

Подключаемый файл ***<asm/byteorder.h>*** определяет либо **__BIG_ENDIAN**, либо **__LITTLE_ENDIAN**, в зависимости от порядка байт в процессоре. Имея дело с вопросами порядка байт, вы могли бы написать кучу условий **#ifdef __LITTLE_ENDIAN**, но есть путь лучше. Ядро Linux определяет набор макросов, которые занимаются переводом между порядком байтов процессора и теми данными, которые необходимо сохранять или загружать с определённым порядком байтов. Например:

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

Эти два макроса преобразуют значение от любого используемого процессором в **unsigned**, **little-endian**, 32-х разрядное и обратно. Они работают независимо от того, использует ли ваш процессор **big-endian** или **little-endian** и, если на то пошло, является ли он 32-х разрядным процессором или нет. Они возвращают их аргумент неизменным в тех случаях, где нечего делать. Использование этих макросов позволяет легко писать переносимый код без необходимости использовать большое количество конструкций условной компиляции.

Существуют десятки подобных процедур; вы можете увидеть их полный список в ***<linux/byteorder/big_endian.h>*** и ***<linux/byteorder/little_endian.h>***. Спустя некоторое время шаблону не трудно следовать. **be64_to_cpu** преобразует **unsigned**, **big-endian**, 64-х разрядное значение во внутреннее представление процессора. **le16_to_cpus** вместо этого обрабатывает **signed**, **little-endian**, 16-ти разрядное значение. При работе с указателями вы можете также использовать функции, подобные ***cpu_to_le32p***, которые принимают указатель на значение, которое будет преобразовано, вместо самого значения указателя. Для всего остального смотрите подключаемый файл .

Выравнивание данных

Последней проблемой, заслуживающей рассмотрения при написании переносимого кода, является то, как получить доступ к невыровненным данным, например, как прочитать 4-х байтовое значение, хранящееся по адресу, который не кратен 4-м байтам. Пользователи i386 часто адресуют невыровненные элементы данных, но не все архитектуры позволяют это. Многие современные архитектуры генерируют исключение каждый раз, когда программа пытается передавать невыровненные данные; передача данных обрабатывается обработчиком исключения с большой потерей производительности. Если вам необходимо получить доступ невыровненным данным, вам следует использовать следующие макросы:

```
#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);
```

Эти макросы безтиповые и работают для каждого элемента данных, будь они один, два, четыре, или восемь байт длиной. Они определяются в любой версии ядра.

Другой проблемой, связанная с выравниванием, является переносимость структур данных между разными платформами. Такая же структура данных (как определена в исходном файле

на языке Си) может быть скомпилирована по-разному на разных платформах. Компилятор передвигает поля структуры, для соответствия соглашениям, которые отличаются от платформы к платформе.

Для записи структур данных для элементов данных, которые могут перемещаться между архитектурами, вы должны всегда следовать естественному выравниванию элементов данных в дополнение к стандартизации на определённый порядок байтов. ***Естественное выравнивание*** означает хранение объектов данных по адресу, кратному их размеру (например, 8-ми байтовые объекты располагаются по адресам, кратным 8). Для принудительного естественного выравнивания, чтобы не допустить организацию полей компилятором непредсказуемым образом, вы должны использовать поля-заполнители, во избежание оставления пустот в структуре данных.

Чтобы показать, как компилятором выполняется выравнивание, в каталоге ***misc-progs*** примеров кода распространяется программа ***dataalign*** и эквивалентный модуль ***kdataalign***, как часть ***misc-modules***. Это результат работы программы на нескольких платформах и результат работы модуля на SPARC64:

```
arch Align: char short int long ptr long-long u8 u16 u32 u64
i386          1    2    4    4   4      4       1   2   4   4
i686          1    2    4    4   4      4       1   2   4   4
alpha         1    2    4    8   8      8       1   2   4   8
armv4l        1    2    4    4   4      4       1   2   4   4
ia64          1    2    4    8   8      8       1   2   4   8
mips          1    2    4    4   4      8       1   2   4   8
ppc           1    2    4    4   4      8       1   2   4   8
sparc         1    2    4    4   4      8       1   2   4   8
sparc64       1    2    4    4   4      8       1   2   4   8
x86_64        1    2    4    8   8      8       1   2   4   8

kernel: arch Align: char short int long ptr long-long u8 u16 u32 u64
kernel: sparc64       1    2    4    8   8      8       1   2   4   8
```

Интересно отметить, что не на всех платформах 64-х разрядные значения выровнены по 64-х битной границе, так что вам необходимо заполнить поля для обеспечения выравнивания и обеспечения переносимости.

Наконец, необходимо учитывать, что компилятор может спокойно вставить заполнитель в структуру сам, чтобы обеспечить выравнивание каждого поля для хорошей производительности на целевом процессоре. Если вы определяете структуру, которая призвана соответствовать структуре, ожидаемой устройством, это автоматическое заполнение может помешать вашей попытке. Способом преодоления этой проблемы является сказать компилятору, что структура должна быть "упакована" без добавления наполнителей. Например, файл заголовка ядра ***<linux/edd.h>*** определяет несколько структур данных, используемых для взаимодействия с BIOS x86, и включает в себя следующие определения:

```
struct {
    u16 id;
    u64 lun;
    u16 reserved1;
    u32 reserved2;
} __attribute__ ((packed)) scsi;
```

Без такого **__attribute__ ((packed))** полю **lun** предшествовало бы два заполняющих байта или шесть, если бы мы компилировали структуру на 64-х разрядной платформе.

Указатели и значения ошибок

Многие внутренние функции ядра возвращают вызывающему значение указателя. Многие из этих функций также могут закончится неудачно. В большинстве случаев отказ указывается возвращением указателя со значением **NULL**. Этот метод работает, но он не может сообщить точную природу проблемы. Некоторым интерфейсам действительно необходимо вернуть собственно код ошибки, с тем, чтобы вызвавший мог сделать правильное решение, основанное на том, что на самом деле не так.

Некоторые интерфейсы ядра возвращают эту информацию кодируя код ошибки в значении указателя. Такие функции должны использоваться с осторожностью, поскольку их возвращаемое значение нельзя просто сравнить с **NULL**. Чтобы помочь в создании и использовании подобного вида интерфейса, предоставлен небольшой набор функций (в ***<linux/err.h>***).

Функция, возвращающая тип указателя может, вернуть значение ошибки с помощью:

```
void *ERR_PTR(long error);
```

где **error** является обычным отрицательным кодом ошибки. Для проверки, является ли возвращённый указатель кодом ошибки или нет, вызвавший может использовать ***IS_ERR***:

```
long IS_ERR(const void *ptr);
```

Если вам необходим настоящий код ошибки, он может быть извлечён с помощью:

```
long PTR_ERR(const void *ptr);
```

Вы должны использовать ***PTR_ERR*** только для значения, для которого ***IS_ERR*** возвращает значение "истина"; любое другое значение является правильным указателем.

Связные списки

Ядрам операционной системы, как и многим другим программам, часто необходимо вести списки структур данных. Ядро Linux, порой, содержит в одно и то же время несколько реализаций связного списка. Чтобы уменьшить количество дублирующегося кода, разработчики ядра создали стандартную реализацию кругового, двойного связного списка; другим нуждающимся в манипулировании списками рекомендуется использовать это средство.

При работе с интерфейсом связного списка всегда следует иметь в виду, что функции списка выполняют без блокировки. Если есть вероятность того, что драйвер может попытаться выполнить на одном списке конкурентные операции, вашей обязанностью является реализация схемы блокировки. Альтернативы (повреждённые структуры списка, потеря данных, паники ядра), как правило, трудно диагностировать.

Чтобы использовать механизм списка, ваш драйвер должен подключить файл ***<linux/list.h>***. Этот файл определяет простую структуру типа **list_head**:

```
struct list_head {
    struct list_head *next, *prev;
};
```

Связные списки, используемые в реальном коде, почти неизменно составлены из структуры одного типа, каждая из которых описывает одну запись в списке. Для использования в вашем коде средства списка Linux, необходимо лишь вставлять **list_head** внутри структур, входящих в список. Если ваш драйвер, скажем, поддерживает список того, что делать, его декларация будет выглядеть следующим образом:

```
struct todo_struct {
    struct list_head list;
    int priority; /* зависит от драйвера */
    /* ... добавить другие зависимые от драйвера поля */
};
```

Головой списка, как правило, является автономная структура **list_head**. Рисунок 11-1 показывает, как для поддержания списка данных структур используется простая структура **list_head**.

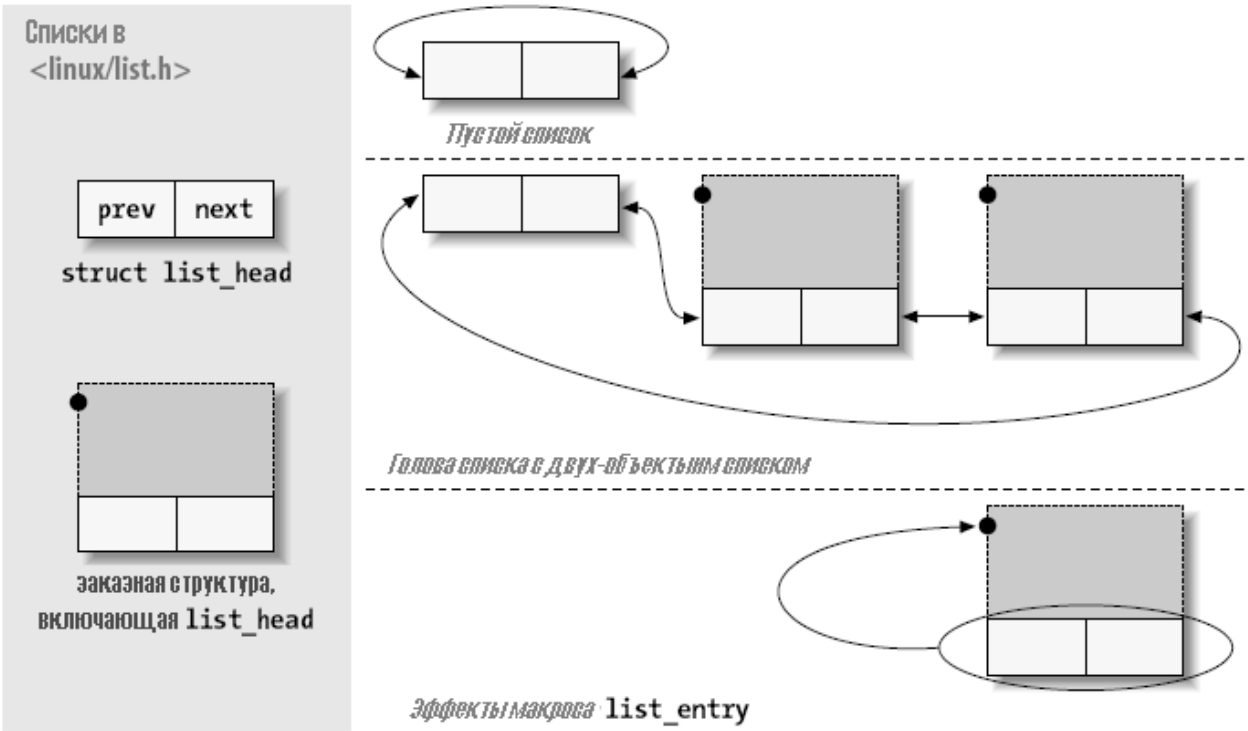


Рисунок 11-1. Структура данных list_head

Заголовки списков должны быть проинициализированы перед использованием с помощью макроса **INIT_LIST_HEAD**. Заголовок списка "что-нибудь сделать" может быть объявлен и проинициализирован так:

```
struct list_head todo_list;

INIT_LIST_HEAD(&todo_list);
```

Альтернативно, списки могут быть проинициализированы при компиляции:

```
LIST_HEAD(todo_list);
```

Некоторые функции для работы со списками определены в ***<linux/list.h>***:

```
list_add(struct list_head *new, struct list_head *head);
```

добавляет новую запись сразу же после головы списка, как правило, в начало списка. Таким образом, она может быть использована для создания стеков. Однако, следует отметить, что голова не должна быть номинальной головой списка; если вы передадите структуру **list_head**, которая окажется где-то в середине списка, новая запись пойдёт сразу после неё. Так как списки Linux являются круговыми, голова списка обычно не отличается от любой другой записи.

list_add_tail(struct list_head *new, struct list_head *head);
Добавляет элемент **new** перед головой данного списка - в конец списка, другими словами. ***list_add_tail*** может, таким образом, быть использована для создания очередей первый вошёл - первый вышел.

list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
Данная запись удаляется из списка. Если эта запись может быть когда-либо вставленной в другой список, вы должны использовать ***list_del_init***, которая инициализирует заново указатели связного списка.

list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
Данная запись удаляется из своего текущего списка и добавляется в начало головы. Чтобы поместить запись в конце нового списка, используйте взамен неё ***list_move_tail***.

list_empty(struct list_head *head);
Возвращает ненулевое значение, если данный список пуст.

list_splice(struct list_head *list, struct list_head *head);
Объединение двух списков вставкой списка сразу после головы.

Структуры **list_head** хороши для реализации списка, подобного структурам, но использующие его программы, как правило, больше заинтересованы в более крупных структурах, которые представляют список как целое. Предусмотрен макрос **list_entry**, который связывает указатель структуры **list_head** обратно с указателем на структуру, которая его содержит. Он вызывается следующим образом:

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

где **ptr** является указателем на используемую структуру **list_head**, **type_of_struct** является типом структуры, содержащей этот **ptr**, и **field_name** является именем поля списка в этой структуре. В нашей предыдущей структуре **todo_struct** поле списка называется просто **list**. Таким образом, мы бы хотели превратить запись в списке в соответствующую структуру такими строчками:

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

Макрос **list_entry** требует немного времени, чтобы привыкнуть, но его не так сложно использовать. Обход связных списков достаточно прост: надо только использовать указатели **prev** и **next**. В качестве примера предположим, что мы хотим сохранить список объектов **todo_struct**, отсортированный в порядке убывания. Функция добавления новой записи будет выглядеть примерно следующим образом:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    for (ptr = todo_list.next; ptr != &todo_list; ptr = ptr->next) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_list)
}
```

Однако, как правило, лучше использовать один из набора предопределённых макросов для создание циклов, которые перебирают списки. Например, предыдущий цикл мог бы быть написать так:

```
void todo_add_entry(struct todo_struct *new)
{
    struct list_head *ptr;
    struct todo_struct *entry;

    list_for_each(ptr, &todo_list) {
        entry = list_entry(ptr, struct todo_struct, list);
        if (entry->priority < new->priority) {
            list_add_tail(&new->list, ptr);
            return;
        }
    }
    list_add_tail(&new->list, &todo_list)
}
```

Использование предусмотренных макросов помогает избежать простых ошибок программирования; разработчики этих макросов также приложили некоторые усилия, чтобы они выполнялись хорошо. Существует несколько вариантов:

list_for_each(struct list_head *cursor, struct list_head *list)
: Этот макрос создаёт цикл **for**, который выполняется один раз с **cursor**, указывающим на каждую последовательную позицию в списке. Будьте осторожны с изменением списка при итерациях через него.

list_for_each_prev(struct list_head *cursor, struct list_head *list)
: Эта версия выполняет итерации назад по списку.

list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)
Если ваш цикл может удалить записи в списке, используйте эту версию. Он просто сохраняет следующую запись в списке в **next** для начала цикла, поэтому не запутается, если запись, на которую указывает **cursor**, удаляется.

list_for_each_entry(type *cursor, struct list_head *list, member)
list_for_each_entry_safe(type *cursor, type *next, struct list_head *list, member)
Эти макросы облегчают процесс просмотр списка, содержащего структуры данного **type**. Здесь, **cursor** является указателем на содержащий структуру тип, и **member** является именем структуры **list_head** внутри содержащей структуры. С этими макросами нет необходимости помещать внутри цикла вызов **list_entry**.

Если вы загляните внутрь ***<linux/list.h>***, вы увидите некоторые дополнительные декларации. Тип **hlist** является двойным связным списком с отдельным, одно-указательным списком заголовка; он часто используется для создания хэш-таблиц и аналогичных структур. Есть также макросы для итерации через оба типа списков, которые предназначены для работы с механизмом прочитать-скопировать-обновить (описанным в разделе "Прочитать-скопировать-обновить" 123 в Главе 5 101). Эти примитивы вряд ли будут полезны в драйверах устройств; смотрите заголовочный файл, если вам потребуется дополнительная информация о том, как они работают.

Краткая справка

В этой главе были представлены следующие символы:

#include <linux/types.h>
typedef u8;
typedef u16;
typedef u32;
typedef u64;
Тип гарантированно являющиеся 8-ми, 16-ми, 32-х и 64-х разрядными беззнаковыми целыми величинами. Также существуют эквивалентные знаковые типы. В пользовательском пространстве вы можете ссылаться на эти типы как **__u8**, **__u16** и так далее.

#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT
Символы, которые определяют количество байт на страницу для существующей архитектуры и число битов на смещение страницы (12 для 4 Кб страниц и 13 для 8 Кб страниц).

#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN
В зависимости от архитектуры определяется только один из этих двух символов.

#include <asm/byteorder.h>
u32 __cpu_to_le32 (u32);

u32 __le32_to_cpu (u32);

Функции, которые выполняют преобразование между известными порядками байтов и таковыми в процессоре. Есть более чем 60 таких функций; для полного списка и способов, которыми они определены, смотрите разные файлы в ***include/linux/byteorder/***.

#include <asm/unaligned.h>
get_unaligned(ptr);
put_unaligned(val, ptr);

Некоторым архитектурам необходимо с использованием этих макросов защитить доступ к невыровненным данным. Макросы преобразуются в обычное разыменование указателя для архитектур, которые позволяют вам получить доступ к невыровненным данным.

#include <linux/err.h>
void *ERR_PTR(long error);
long PTR_ERR(const void *ptr);
long IS_ERR(const void *ptr);

Функции позволяют функциям, которые возвращают значение указателя, возвращать коды ошибок.

#include <linux/list.h>
list_add(struct list_head *new, struct list_head *head);
list_add_tail(struct list_head *new, struct list_head *head);
list_del(struct list_head *entry);
list_del_init(struct list_head *entry);
list_empty(struct list_head *head);
list_entry(entry, type, member);
list_move(struct list_head *entry, struct list_head *head);
list_move_tail(struct list_head *entry, struct list_head *head);
list_splice(struct list_head *list, struct list_head *head);

Функции, которые манипулируют круговыми, двойными связными списками.

list_for_each(struct list_head *cursor, struct list_head *list)
list_for_each_prev(struct list_head *cursor, struct list_head *list)
list_for_each_safe(struct list_head *cursor, struct list_head *next, struct list_head *list)
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)

Удобные макросы для перебора связных списков.

Глава 12, PCI драйверы

В то время, как Глава 9 224 познакомила с самыми низкими уровнями управления оборудованием, эта глава предлагает обзор высокоуровневых шинных архитектур. Шина состоит одновременно из электрического интерфейса и интерфейса программирования. В этой главе мы имеем дело с программным интерфейсом.

Эта глава охватывает несколько шинных архитектур. Однако, основной упор делается на функции ядра, которые обеспечивают доступ к периферийным устройствам Peripheral Component Interconnect (PCI, взаимосвязь периферийных компонентов), потому что в эти дни шина PCI является наиболее часто используемой периферийной шиной на ПК и больших компьютерах. Эта шина имеет наилучшую поддержку ядром. ISA по-прежнему характерна для любителей электроники и описана ниже, хотя это почти чисто аппаратный вид шины и мало чего можно сказать в дополнение к тому, что описано в Главах 9 224 и 10 246.

Интерфейс PCI

Хотя многие пользователи компьютеров думают о PCI как о способе подключения электрических проводов, на самом деле это полный набор спецификаций, определяющих, как должны взаимодействовать разные части компьютера.

Спецификация PCI охватывает большинство вопросов, связанных с компьютерными интерфейсами. Мы не собираемся покрыть здесь их все; в этом разделе мы в основном касаемся того, как драйвер PCI может найти своё оборудование и получить к нему доступ. Методы зондирования, обсуждаемые в разделе "Параметры модуля" 33 в Главе 2 14 и "Автоопределение номера прерывания" 252 в Главе 10 246, могут использоваться с устройствами PCI, но спецификация предлагает альтернативу, которая более предпочтительна, чем зондирование.

Архитектура PCI был разработана в качестве замены стандарту ISA с тремя основными целями: получить лучшую производительность при передаче данных между компьютером и его периферией, быть независимой от платформы, насколько это возможно, и упростить добавление и удаление периферийных устройств в системе.

Шина PCI достигает лучшей производительности за счёт использования более высокой тактовой частоты, чем ISA; она работает на 25 или 33 МГц (фактическое значение зависит от

частоты системы), и недавно были развернуты также 66 МГц и даже 133 МГц реализации. Кроме того, она оснащена 32-х разрядной шиной данных и в спецификацию было включено 64-х разрядное расширение. Независимость от платформы является частой целью в разработке компьютерной шины и это особенно важная особенность PCI, поскольку в мире ПК всегда доминировали стандарты интерфейсов, зависимые от процессора. В настоящее время PCI широко используется на системах IA-32, Alpha, PowerPC, SPARC64 и IA-64, а также некоторые других платформах.

Однако, наиболее актуальной для автора драйвера является поддержка PCI автоопределения интерфейса плат. PCI устройства безджамперные (в отличие от большинства старой периферии) и настраивается автоматически во время загрузки. Затем драйвер устройства должен быть в состоянии получить доступ к информации о конфигурации в устройстве с целью завершения инициализации. Это происходит без необходимости совершать какое-либо тестирование.

Адресация в PCI

Каждое периферийное устройство PCI идентифицируется номером шины, номером устройства и номером функции. Спецификация PCI позволяет одной системе содержать до 256 шин, но из-за того, что 256 шин не является достаточным для многих больших систем, Linux теперь поддерживает домены PCI. Каждый домен PCI может содержать до 256 шин. Каждая шина содержит до 32 устройств и каждое устройство может быть многофункциональной платой (такой, как аудио-устройство с сопровождающим приводом CD-ROM) с максимум восемью функциями. Поэтому каждая функция может быть идентифицирована на аппаратном уровне 16-ти разрядным адресом, или ключом. Однако, драйверам устройств, написанным для Linux, не требуется иметь дело с этими двоичными адресами, потому что они используют для работы с устройствами специальную структуру данных, названную **pci_dev**.

Последние рабочие станции имеют по крайней мере две шины PCI. Подключение более одной шины в одной системе выполняется с помощью мостов, периферии PCI специального назначения, задачей которой является объединение двух шин. Общая схема системы PCI представляет собой дерево, где каждая шина связана с шиной верхнего уровня, вплоть до шины 0 в корне дерева. Система карт ПК CardBus (стандарт шины для PCMCIA) также подключена к системе PCI через мосты. Типичная система PCI представлена на Рисунке 12-1, где подсвечены различные мосты.

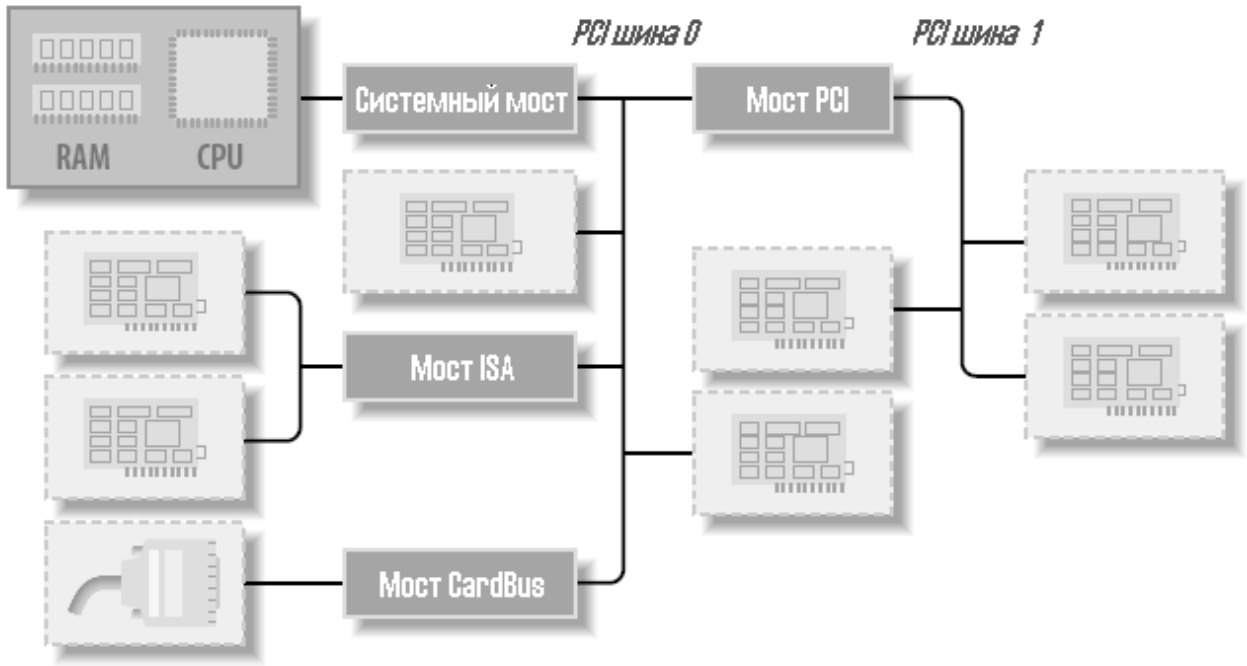


Рисунок 12-1. Схема типичной системы PCI

16-ти разрядные аппаратные адреса, связанные с периферийными устройствами PCI, чаще всего скрыты в объекте **struct pci_dev**, но иногда всё ещё видимы, особенно когда используются списки устройств. Одной из таких ситуаций является вывод ***lspci*** (часть пакета ***pciutils***, доступного в большинстве дистрибутивов) и расположение информации в ***/proc/pci*** и ***/proc/bus/pci***. Представление устройств PCI в sysfs также показывает эту схему адресации с добавлением информации о домене PCI. (* Некоторые архитектуры также показывают информацию PCI домена в файлах */proc/pci* и */proc/bus/pci*.) При отображении аппаратный адрес может быть показан в виде двух значений (8-ми разрядный номер шины и 8-ми разрядные номера устройства и функции), как три значения (шина, устройство и функция), или как четыре значения (домен, шина, устройство и функция); все значения, как правило, отображаются в шестнадцатеричном виде.

Например, ***/proc/bus/pci/devices*** используют одно 16-ти разрядное поле (для облегчения анализа и сортировки), а ***/proc/bus/busnumber*** разделяет адрес на три поля. Следующий снимок демонстрирует, как выглядят эти адреса, показано только начало строк вывода:

```
$ lspci | cut -d: -f1-3
0000:00:00.0 Host bridge
0000:00:00.1 RAM memory
0000:00:00.2 RAM memory
0000:00:02.0 USB Controller
0000:00:04.0 Multimedia audio controller
0000:00:06.0 Bridge
0000:00:07.0 ISA bridge
0000:00:09.0 USB Controller
0000:00:09.1 USB Controller
0000:00:09.2 USB Controller
0000:00:0c.0 CardBus bridge
0000:00:0f.0 IDE interface
0000:00:10.0 Ethernet controller
```

```
0000:00:12.0 Network controller
0000:00:13.0 FireWire (IEEE 1394)
0000:00:14.0 VGA compatible controller
$ cat /proc/bus/pci/devices | cut -f1
0000
0001
0002
0010
0020
0030
0038
0048
0049
004a
0060
0078
0080
0090
0098
00a0
$ tree /sys/bus/pci/devices/
/sys/bus/pci/devices/
|-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
|-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
|-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
`-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
```

Все три списка устройств отсортированы в одном порядке, поскольку ***lspci*** использует как источник информации файлы ***/proc***. Используя видеоконтроллер VGA в качестве примера, 0x00a0 представляется как 0000:00:14.0 при разделении на домен (16 бит), шину (8 бит), устройство (5 бит) и функцию (3 бита).

Аппаратная схема каждой периферийной платы отвечает на запросы, относящиеся к трём адресным пространствам: ячейкам памяти, портам ввода/вывода и регистрам конфигурации. Первые два адресных пространства являются общими для всех устройств на одной шине PCI (то есть, когда вы обращаетесь к памяти, все устройства на этой шине PCI видят этот цикл шины одновременно). Пространство конфигурации, с другой стороны, использует ***географическую адресацию***. Запросы конфигурации в один момент времени адресуют только один слот, поэтому они никогда не конфликтуют.

Что же касается драйвера, области памяти и ввода/вывода доступны обычными способами через ***inb***, ***readb*** и так далее. С другой стороны, операции по конфигурации выполняются

вызовом для доступа к регистрам конфигурации определённых функций ядра. Что касается прерываний, каждый слот PCI имеет четыре контакта прерывания и каждая функция устройства может воспользоваться одной из них, не заботясь о том, как эти контакты маршрутизируются в процессор. Такая маршрутизация является обязанностью компьютерной платформы и осуществляется вне шины PCI. Так как спецификация PCI требует, чтобы линии прерывания были разделяемыми, даже процессор с ограниченным количеством линий прерывания, такой как x86, может принимать много интерфейсных плат PCI (каждая с четырьмя контактами прерываний).

В шине PCI пространство ввода/вывода использует 32-х разрядную шину адреса (что ведёт к 4 Гб портов ввода/вывода), в то время как пространство памяти может быть доступно или с 32-х разрядными, или 64-х разрядными адресами. 64-х разрядные адреса доступны на более новых платформах. Адреса должны быть уникальными для одного устройства, но программное обеспечение может ошибочно настроить два устройства на один адрес, делая невозможным доступ к любому из них. Но такой проблемы никогда не случится, если драйвер не будет играть с регистрами, которых не должен касаться. Хорошая новость в том, что каждая область памяти и адреса ввода/вывода, предлагаемые интерфейсной платой, могут быть переназначены с помощью операций по конфигурации. То есть встроенное программное обеспечение при загрузке системы инициализирует оборудование PCI, отображая каждую область на другой адрес во избежание конфликтов. (* На самом деле, такая конфигурация не ограничивается временем загрузки системы; например, устройства, подключаемые без выключения системы, не могут быть доступны во время загрузки и вместо этого появляются позже. Основным моментом здесь является то, что драйвер устройства не должен менять области адреса ввода/вывода или памяти.) Адреса, по которым эти регионы в настоящее время отображаются, могут быть прочитаны из конфигурационного пространства, поэтому драйвер Linux может получить доступ к своим устройствам без зондирования. После чтения регистров конфигурации драйвер может иметь безопасный доступ к своему оборудованию.

Пространство конфигурации PCI состоит из 256 байт для каждой функции устройства (за исключением устройств PCI Express, которые имеют 4 Кб конфигурационного пространства для каждой функции) и стандартизированную схему регистров конфигурации. Четыре байта конфигурационного пространства содержать уникальный ID функции, поэтому драйвер может определить своё устройство, глядя на заданный для такой периферии ID. (* Вы найдёте ID любого устройства в своём руководстве для оборудования. Перечень включён в файл pci.ids, часть пакета pciutils, и исходные тексты ядра; он не претендует на полноту, это просто список наиболее известных поставщиков и устройств. Версия ядра этого файла не будет включена в будущие серии ядра.) Таким образом, каждая плата устройства адресуема географически для получения её регистров конфигурации; затем информация в этих регистрах может быть использована для выполнения обычного доступа ввода/вывода, без необходимости дальнейшей географической адресации.

Из этого описания должно быть ясно, что основным новшеством стандартна интерфейса PCI перед ISA является пространство конфигурации адресов. Таким образом, в дополнение к обычному коду драйвера, драйверу PCI необходима возможность доступа к пространству конфигурации, чтобы предохранить себя от рискованных задач зондирования.

В оставшейся части этой главы мы используем слово ***устройство*** для ссылки на функцию устройства, потому что каждая функция в многофункциональной плате выступает в качестве независимого субъекта. Когда мы говорим об устройстве, мы имеем в виду набор "номера домена, номер шины, номер устройства и номер функции".

Момент загрузки

Чтобы увидеть, как работает PCI, мы начинаем с загрузки системы, поскольку устройства настраиваются именно тогда.

При подаче питания на устройство PCI его оборудование остаётся неактивным. Другими словами, устройство реагирует только на операции по конфигурации. При включении питания устройство не имеет памяти и портов ввода/вывода, связанных с адресным пространством компьютера; все другие функции, зависящие от устройства, такие как генерация прерываний, также отключены. К счастью, все материнские платы с PCI оснащены осведомлённым о PCI встроенным программным обеспечением, называемым BIOS, NVRAM, или PROM, в зависимости от платформы. Встроенное программное обеспечение обеспечивает доступ к адресному пространству конфигурации устройства чтением и записью регистров контроллера PCI.

Во время загрузки системы встроенное программное обеспечение (или ядро Linux, если так настроено) выполняет операции по конфигурации с каждой периферией PCI для того, чтобы выделить безопасное место для каждого адресуемого региона, предлагаемого ей. К тому времени, когда драйвер устройства обращается к устройству, его области памяти и ввода/вывода уже отображены в адресное пространство процессора. Драйвер может изменить это значение по умолчанию, но он никогда не должен этого делать.

Как предложено, пользователь может посмотреть список устройств PCI и регистры конфигурации устройства читая ***/proc/bus/pci/devices*** и ***/proc/bus/pci/*/****. Первый из них является текстовым файлом с (шестнадцатеричной) информацией об устройстве, а последние являются бинарными файлами, которые показывают снимки регистров конфигурации каждого устройства, один файл на устройство. Отдельный каталог PCI устройства в дереве sysfs может быть найден в ***/sys/bus/pci/devices***. Каталог PCI устройства содержит несколько различных файлов:

```
$ tree /sys/bus/pci/devices/0000:00:10.0
/sys/bus/pci/devices/0000:00:10.0
|-- class
|-- config
|-- detach_state
|-- device
|-- irq
|-- power
|   `-- state
|-- resource
|-- subsystem_device
|-- subsystem_vendor
`-- vendor
```

Файл конфигурации представляет собой бинарный файл, который позволяет считать из устройства сырую информации о конфигурации PCI (подобно обеспечиваемой ***/proc/bus/pci/*/****.) Каждый из файлов ***vendor***, ***device***, ***subsystem_device***, ***subsystem_vendor*** и ***class*** ссылается на определённые значения этого PCI устройства (все PCI устройства предоставляют эту информацию.) Файл ***irq*** показывает текущее прерывание, назначенное для этого PCI устройства, и файл ***resource*** показывает текущие ресурсы памяти, выделенные этим устройством.

Регистры конфигурации и инициализация

В этом разделе мы рассмотрим регистры конфигурации, которые содержат PCI устройства. Все PCI устройства содержат по крайней мере 256 байт адресного пространства. Первые 64 байт стандартизованы, а остальные зависят от устройства. Рисунок 12-2 показывает схему не зависящего от устройства конфигурационного пространства.

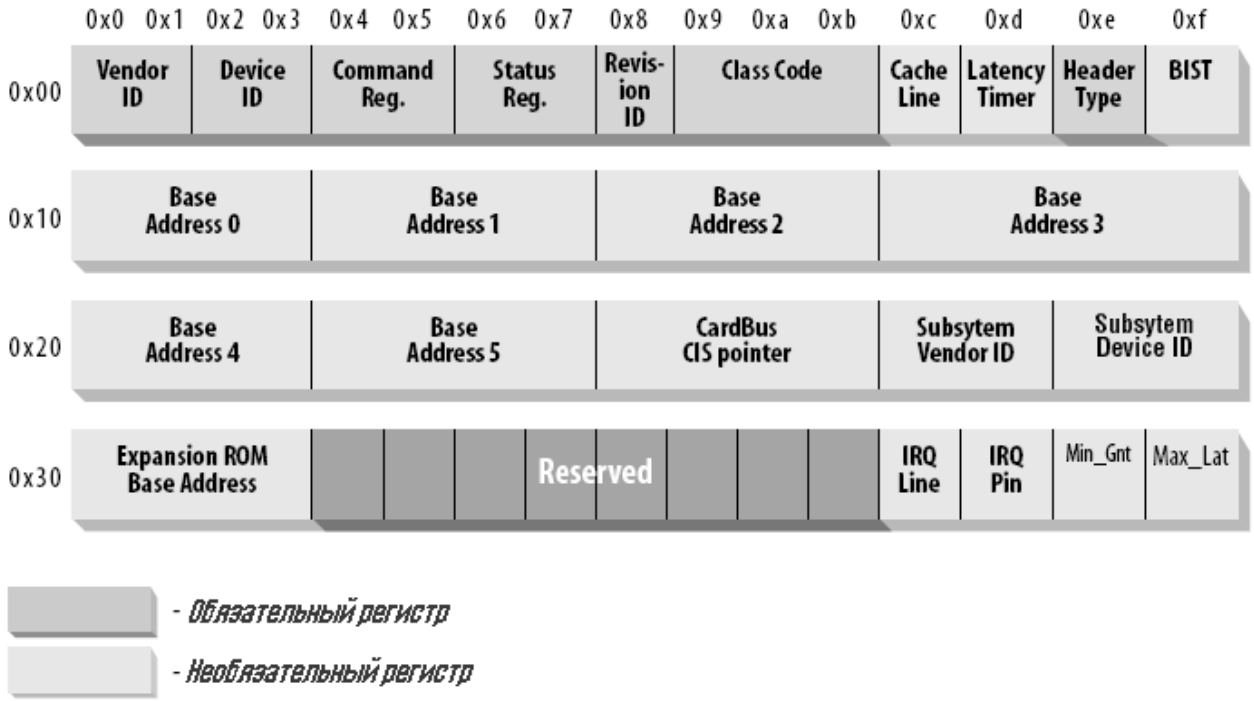


Рисунок 12-2. Стандартизированные конфигурационные регистры PCI

Как видно из рисунка, некоторые регистры конфигурации PCI необходимы и некоторые являются необязательными. Каждое устройство PCI должно содержать поддающиеся интерпретации значения в необходимых регистрах, а содержание необязательных регистров зависит от реальных возможностей периферии. Необязательные поля не используются, пока содержимое необходимых полей не покажет, что они являются действительными. Таким образом, обязательные поля декларируют возможности платы, в том числе, пригодность других полей.

Интересно отметить, что регистры PCI всегда little-endian (сначала младший). Хотя стандарт разрабатывался, чтобы быть независимым от архитектуры, разработчики PCI иногда показывают склонность в сторону среды ПК. Автор драйвера должен быть аккуратен с порядком байтов при доступе к многобайтовым регистрам конфигурации; код, который работает на ПК, может не работать на других платформах. Разработчики Linux позаботились о проблеме порядка байт (смотрите следующий раздел "Доступ к пространству конфигурации" 300), но об этом необходимо помнить. Если вам когда-нибудь понадобиться преобразовать данные от порядка на платформе в порядок PCI, или наоборот, вы можете прибегнуть к функциям, определённым в ***<asm/byteorder.h>***, введённым в Главе 11 275, зная, что порядок байт в PCI little-endian.

Описание всех элементов конфигурации выходит за рамки этой книги. Как правило, техническая документация, выпущенная с каждым устройством, описывает поддерживаемые

регистры. То, чем мы интересуемся, каким образом драйвер может увидеть своё устройство и как он может получить доступ к пространству конфигурации устройства.

Устройство идентифицируют три или пять регистров PCI: **vendorID**, **deviceID** и **class** являются теми тремя, которые используются всегда. Каждый производитель PCI присваивает собственные значения этим регистрам, предназначенным только для чтения, и драйвер может использовать их для поиска устройства. Кроме того, с целью дальнейшего различия похожих устройств поставщик иногда устанавливает поля **subsystem vendorID** и **subsystem deviceID**.

Давайте посмотрим на эти регистры более подробно:

vendorID
Этот 16-ти разрядный регистр идентифицирует изготовителя оборудования. Например, каждое устройство Intel отмаркировано одним и тем же числом поставщика, 0x8086. Существует глобальный реестр таких чисел, который ведёт PCI Special Interest Group, и производители должны обратиться туда, чтобы получить уникальный номер.

deviceID
Это другой 16-ти разрядный регистр, выбранный производителем; для ID устройства не требуется никакой официальной регистрации. Этот ID, как правило, используется в паре с ID поставщика, создавая уникальный 32-х разрядный идентификатор аппаратного устройства. Мы используем слово ***сигнатура*** для обращения к ID поставщика и ID устройства в паре. Драйвер устройства обычно полагается на сигнатуру, чтобы определить своё устройство; вы можете найти, какое значение искать, в руководстве оборудования для целевого устройства.

class
Каждое периферийное устройство относится к **class (классу)**. Регистр **class** является 24-х разрядным значением, чьи старшие 8 бит идентифицируют "базовый класс" (или группу). Например, "ethernet" и "token ring" являются двумя классами, принадлежащими к группе "network", а классы "serial" и "parallel" относятся к группе "communication". Некоторые драйверы могут поддерживать несколько аналогичных устройств, каждое из них имеет свою сигнатуру, однако все они принадлежат к одному классу; эти драйверы могут рассчитывать на регистр **class** для определения своих периферийных устройства, как показано ниже.

subsystem vendorID
subsystem deviceID
Эти поля могут быть использованы для дальнейшей идентификации устройства. Если такая микросхема является обычным чипом интерфейса для локальной (на плате) шины, она часто используется в нескольких совершенно разных ролях и драйвер должен определить фактическое устройство для общения. Идентификаторы subsystem (подсистема) и используются для этой цели.

Используя эти разные идентификаторы, драйвер PCI может сказать ядру, какие виды устройств он поддерживает. Чтобы определить список различных типов устройств PCI, которые поддерживает драйвер, используется структура **struct pci_device_id**. Эта структура содержит следующие поля:

__u32 vendor;

__u32 device;
Они определяют идентификаторы поставщика PCI и устройства. Если драйвер может работать с любым идентификатором поставщика или устройства, для этих полей должно быть использовано значение **PCI_ANY_ID**.

__u32 subvendor;
__u32 subdevice;
Эти определяют идентификаторы поставщика PCI подсистемы и подсистемы устройства для устройства. Если драйвер может обрабатывать любой тип ID подсистемы, для этих полей должно быть использовано значение **PCI_ANY_ID**.

__u32 class;
__u32 class_mask;
Эти два значения позволяют драйверу определить, какой тип устройства PCI класса он поддерживает. Разные классы устройств PCI (VGA контроллер является одним из примеров) описаны в спецификации PCI. Если драйвер может обрабатывать любой класс, для этих полей должно быть использовано значение 0.

kernel_ulong_t driver_data;
Это значение не используется для соответствия устройству, но используется для хранения информации, которую драйвер PCI может использовать, чтобы различать устройства, если он этого хочет.

Существуют две вспомогательные макроса, которые должны использоваться для инициализации структуры **struct pci_device_id**:

PCI_DEVICE(vendor, device)
Этот создаёт структуру **pci_device_id**, которая соответствует только заданным идентификаторам поставщика и устройства. Макрос устанавливает поля структуры **subvendor** и **subdevice** в **PCI_ANY_ID**.

PCI_DEVICE_CLASS(device_class, device_class_mask)
Этот создаёт структуру **pci_device_id**, которая соответствует определённому классу PCI.

Пример использования этих макросов для определения типа устройств, которые поддерживает драйвер, может быть найден в следующих файлах ядра:

```
drivers/usb/host/ehci-hcd.c:

static const struct pci_device_id pci_ids[ ] = { {
    /* работаем с любым контроллером USB 2.0 EHCI */
    PCI_DEVICE_CLASS(((PCI_CLASS_SERIAL_USB << 8) | 0x20), ~0),
    .driver_data = (unsigned long) &ehci_driver,
    },
    { /* последняя: все нули */ }
};

drivers/i2c/busses/i2c-i810.c:

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, },
};
```

Эти примеры создают список структур **struct pci_device_id** с пустой структурой, проинициализированной нулями в качестве последнего значения в списке. Этот массив идентификаторов используется в структуре **pci_driver** (описанной ниже) и используется также, чтобы указать пользовательскому пространству, какие устройства поддерживает этот специфический драйвер.

MODULE_DEVICE_TABLE

Структура **pci_device_id** должна быть экспортирована в пользовательское пространство, чтобы позволить системам горячего подключения и загрузки модулей узнать, с какими устройствами работает модуль. Эту задачу решает макрос **MODULE_DEVICE_TABLE**. Пример:

```
MODULE_DEVICE_TABLE(pci, i810_ids);
```

Этот оператор создаёт локальную переменную, называемую **__mod_pci_device_table**, которая указывает на список **struct pci_device_id**. Позже, в процессе сборки ядра, программа ***depmod*** просматривает все модули на символ **__mod_pci_device_table**. Если этот символ найден, она вынимает данные из модуля и добавляет их в файл ***/lib/modules/KERNEL_VERSION/modules.pcimap***. После завершения работы ***depmod***, все PCI устройства, поддерживаемые модулем в ядре, вместе с именами их модулей, перечисляются этом в файле. Когда ядро сообщает системе горячего подключения, что было найдено новое устройство PCI, система горячего подключения использует файл ***modules.pcimap***, чтобы найти для загрузки правильный драйвер.

Регистрация PCI драйвера

Основной структурой, которую должны создать все драйверы PCI для того, чтобы быть правильно зарегистрированными в ядре, является структура **struct pci_driver**. Эта структура состоит из ряда функций обратного вызова и переменных, описывающих драйвер PCI для ядра PCI. Вот поля в этой структуре, о которых должен знать драйвер PCI:

const char *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 находится в ядре, оно появляется в sysfs в ***/sys/bus/pci/drivers/***.

const struct pci_device_id *id_table;
Указатель на таблицу **struct pci_device_id**, описанную ранее в этой главе.

int (*probe) (struct pci_dev *dev, const struct pci_device_id *id);
Указатель на зондирующую функцию в драйвере PCI. Эта функция вызывается ядром PCI, когда оно имеет **struct pci_dev** и думает, что этот драйвер хочет контролировать её. Указатель на **struct pci_device_id**, который используется ядром PCI, чтобы сделать

это решение, также передаётся в эту функцию. Если драйверу PCI требуется передать ей **struct pci_dev**, он должен правильно проинициализировать устройство и вернуть 0. Если драйвер не хочет заявлять устройство или произошла ошибка, он должен вернуть отрицательное значение ошибки. Подробнее об этой функции далее в этой главе.

void (*remove) (struct pci_dev *dev);
Указатель на функцию, которую вызывает ядро PCI, когда **struct pci_dev** удаляется из системы, или когда PCI драйвер выгружается из ядра. Подробнее об этой функции далее в этой главе.

int (*suspend) (struct pci_dev *dev, u32 state);
Указатель на функцию, которую вызывает ядро PCI, когда **struct pci_dev** в настоящее время приостановлена. Состояние приостановки передаётся в переменной **state**. Эта функция не является обязательной; драйвер не обязан предоставлять её.

int (*resume) (struct pci_dev *dev);
Указатель на функцию, которую вызывает ядро PCI, когда **struct pci_dev** возобновляется. Она всегда вызывается после того, как была вызвана ***suspend***. Эта функция не является обязательной; драйвер не обязан предоставлять её.

Таким образом, чтобы создать правильную структуру **struct pci_driver** должны быть проинициализированы только четыре поля :

```
static struct pci_driver pci_driver = {
    .name = "pci_skel",
    .id_table = ids,
    .probe = probe,
    .remove = remove,
};
```

Чтобы зарегистрировать **struct pci_driver** в ядре PCI, выполняется вызов ***pci_register_driver*** с указателем на **struct pci_driver**. Это традиционно делается в коде инициализации модуля PCI драйвера:

```
static int __init pci_skel_init(void)
{
    return pci_register_driver(&pci_driver);
}
```

Обратите внимание, что функция ***pci_register_driver*** возвращает или отрицательное число ошибки, или **0**, если всё было успешно зарегистрировано. Она не возвращает количество устройств, которые были связаны с драйвером или номер ошибки, если нет устройств, связанных с драйвером. Это является изменением от предыдущих ядер к релизу версии 2.6 и было сделано из-за следующих ситуаций:

- В системах с поддержкой горячего подключения PCI или системах CardBus, устройство PCI может появиться или исчезнуть в любой момент времени. Полезно, если драйверы могут быть загружены прежде, чем появится устройство, чтобы сократить время, необходимое для инициализации устройства.
- Ядро версии 2.6 позволяет новым идентификаторам PCI быть динамически выделенными для драйвера после того, как он был загружен. Это делается через файл ***new_id***, который создаётся во всех каталогах PCI драйверов в sysfs. Это очень полезно, если используется

новое устройство, о котором ядро пока ещё не знает. Пользователь может записать PCI ID значения в файл ***new_id***, а затем драйвер связывается с новым устройством. Если бы драйверу не была разрешена загрузка, пока устройство не присутствует в системе, этот интерфейс бы не был в состоянии работать.

Когда PCI драйвер должен быть выгружен, необходимо разрегистрировать **struct pci_driver** в ядре. Это делается с помощью вызова ***pci_unregister_driver***. Когда происходит этот вызов, любые PCI устройства, которые в настоящее время связаны с этим драйвером, удаляются, и перед возвращением функции ***pci_unregister_driver*** вызывается функция ***remove*** этого драйвера PCI.

```
static void __exit pci_skel_exit(void)
{
    pci_unregister_driver(&pci_driver);
}
```

Старый способ зондирования PCI

В более старых версиях ядра, эта функция, ***pci_register_driver***, не всегда использовалась PCI драйверами. Вместо этого они бы либо вручную проходили по списку устройств PCI в системе, или вызывали бы функцию, которая могла бы выполнить поиск заданного PCI устройства. Способность драйвера проходить по списку PCI устройств в системе была удалена из ядра версии 2.6, чтобы предотвратить сбой драйверов ядра, если произошла модификация списков устройств PCI во время удаления устройства.

Если способность находить определённое устройства PCI действительно необходима, доступны следующие функции:

struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);

Эта функция сканирует список устройств PCI, присутствующих в системе в настоящее время, и если входные параметры соответствуют указанным идентификаторам **vendor** и **device**, она увеличивает счётчик ссылок на найденную переменную **struct pci_dev** и возвращает его вызывающему. Это предотвращает исчезновение структуры без предварительного уведомления и гарантирует, что ядро не выдаст Oops. После того, как драйвер завершает работу со **struct pci_dev**, возвращённой функцией, он должен вызвать функцию ***pci_dev_put*** для правильного обратного уменьшения счётчика использования, чтобы разрешить ядру очистить устройство, если оно удаляется.
Чтобы завладеть несколькими устройствами с одинаковой сигнатурой, используется аргумент **from**; аргумент должен указывать на последнее найденное устройство, чтобы поиск мог продолжаться, вместо перезапуска с начала списка. Чтобы найти первое устройство, **from** определяется как **NULL**. Если не найдено (больше) устройств, возвращается **NULL**.
Пример правильного использования этой функции:

```
struct pci_dev *dev;
dev = pci_get_device(PCI_VENDOR_FOO, PCI_DEVICE_FOO, NULL);
if (dev) {
    /* Используем это PCI устройство */
    ...
    pci_dev_put(dev);
}
```

struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device,
unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);

Эта функция работает подобно ***pci_get_device***, но она позволяет указать для поиска устройства идентификаторы поставщика подсистемы и подсистемы устройства. Эта функция не может быть вызвана из контекста прерывания. Если это случилось, в системный журнал печатается предупреждение.

struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);

Эта функция выполняет поиск списка устройств PCI в системе на заданную **struct pci_bus** для указанного устройства и номера функции устройства PCI. Если обнаружено соответствующее устройство, его счётчик ссылок увеличивается и возвращается указатель на него. Когда вызывающий закончил обращение к **struct pci_dev**, он должен вызвать ***pci_dev_put***.

Все эти функции не могут быть вызваны из контекста прерывания. Если это случилось, в системный журнал печатается предупреждение.

Разрешение устройства PCI

В функции ***probe*** драйвера PCI, прежде чем драйвер сможет получить доступ к любому ресурсу устройства (область ввода/вывода или прерывание) данного PCI устройства, драйвер должен вызвать функцию ***pci_enable_device***:

int pci_enable_device(struct pci_dev *dev);

Эта функция фактически разрешает устройство. Она будит устройство и в некоторых случаях также задаёт ему линию прерывания и области ввода/вывода. Это происходит, например, с устройствами CardBus (которые были сделаны полностью эквивалентными PCI на уровне драйвера).

Доступ в пространство конфигурации

После обнаружения устройства драйвером, обычно необходимо читать или записывать в три адресные пространства: память, порт и конфигурацию. В частности, доступ к конфигурационному пространству является жизненно важным для драйвера, потому что только так он может узнать, как устройство отображается в памяти и пространстве ввода/вывода.

Поскольку микропроцессор не имеет возможности получить доступ к конфигурационному пространству напрямую, поставщик компьютера должен обеспечить способ сделать это. Для доступа к конфигурационному пространству процессор должен читать и записывать в регистры контроллера PCI, но точная реализация является зависимой от поставщика и не имеет отношения к этому обсуждению, потому что Linux предлагает стандартный интерфейс для доступа к конфигурационному пространству.

Что же касается драйвера, доступ в конфигурационное пространство можно получить через 8-ми разрядные, 16-ти разрядные или 32-х разрядные передачи данных. Прототипы соответствующих функций в ***<linux/pci.h>***:

int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);

Прочитать один, два или четыре байта из конфигурационного пространства устройства, идентифицируемого **dev**. Аргумент **where** является байтовым смещением от начала конфигурационного пространства. Значение, извлекаемое из конфигурационного пространства, возвращается через указатель **val** и возвращаемое значение функции является кодом ошибки. Функции ***word*** и ***dword*** преобразуют только что прочитанное значение из little-endian (сначала младший) в родной для процессора порядок байтов, так что нет необходимости заботится о порядке байтов.

int pci_write_config_byte(struct pci_dev *dev, int where, u8 val);
int pci_write_config_word(struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword(struct pci_dev *dev, int where, u32 val);
Записывают один, два или четыре байта в конфигурационное пространство. Устройство идентифицируется **dev**, как обычно, а записываемое значение передаётся как **val**. Функции ***word*** и ***dword*** преобразуют значение в little-endian перед записью на периферийное устройство.

Все предыдущие функции реализованы в виде встраиваемых функций, которые на самом деле вызывают следующие функции. Не стесняйтесь использовать эти функции взамен вышеприведённых в случае, если в какой-то момент времени драйвер не имеет доступа к **struct pci_dev**:

int pci_bus_read_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 *val);
int pci_bus_read_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16 *val);
int pci_bus_read_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32 *val);
Подобны функциям ***pci_read_***, но вместо **struct pci_dev *** необходимы переменные **struct pci_bus *** и **devfn**.

int pci_bus_write_config_byte (struct pci_bus *bus, unsigned int devfn, int where, u8 val);
int pci_bus_write_config_word (struct pci_bus *bus, unsigned int devfn, int where, u16 val);
int pci_bus_write_config_dword (struct pci_bus *bus, unsigned int devfn, int where, u32 val);
Подобны функциям ***pci_write_***, но вместо **struct pci_dev *** необходимы переменные **struct pci_bus *** и **devfn**.

Лучшим способом адресации переменных конфигурации с помощью функций ***pci_read_*** является использование символических имён, определённых в ***<linux/pci.h>***. Например, следующая небольшая функция получает идентификатор ревизии устройства, передавая его символическое имя в ***pci_read_config_byte***:

```
static unsigned char skel_get_revision(struct pci_dev *dev)
{
    u8 revision;

    pci_read_config_byte(dev, PCI_REVISION_ID, &revision);
    return revision;
}
```

Доступ к пространствам ввода/вывода и памяти

PCI устройство реализует до шести областей адресов ввода/вывода. Каждый регион состоит или из адресов памяти или из адресов ввода/вывода. Большинство устройств содержат свои регистры ввода/вывода в областях памяти, потому что это наиболее разумный подход (как описано в разделе "Порты ввода/вывода и память ввода/вывода" 224 в Главе 9 224). Однако, в отличие от обычной памяти, регистры ввода/вывода не должны кэшироваться процессором, поскольку каждый доступ может иметь побочные эффекты. PCI устройство, которое реализует регистры ввода/вывода как область памяти, отмечает различие значением бита “memory-is-prefetchable” ("память с упреждающей выборкой") в его регистре конфигурации. (* Информация живёт в одном из младших битов базового адреса регистров PCI. Эти биты определены в <*linux/pci.h*>.) Если область память обозначена как "с упреждением", процессор может кэшировать её содержимое и выполнять с ним все виды оптимизации; доступ к памяти без упреждающей выборки, с другой стороны, не может быть оптимизирован, так как каждый доступ может иметь побочные эффекты, так же как и с портами ввода/вывода. Периферия, которая связывает свои регистры управления с диапазоном адресов памяти, декларирует этот диапазон как "без упреждения", тогда как что-то вроде видео-памяти на плате PCI является "упреждающим". В этом разделе мы используем слово ***регион***, чтобы сослаться на то, что общее адресное пространство ввода/вывода является отображаемым на память или на порты.

Интерфейсная плата сообщает размер и текущее расположение своих регионов используя регистры конфигурации, эти шесть 32-х разрядных регистров показаны на Рисунке 12-2, их символические имена от **PCI_BASE_ADDRESS_0** до **PCI_BASE_ADDRESS_5**. Так как пространство ввода/вывода, определяемое PCI, является 32-х разрядным адресным пространством, для памяти и ввода/вывода имеет смысл использовать тот же интерфейс конфигурации. Если устройство использует 64-х разрядную шину адреса, оно может декларировать регионы в 64-х разрядном пространстве памяти с помощью двух последовательных регистров **PCI_BASE_ADDRESS** для каждого региона, младшими битами вперёд. Одно устройство может предлагать одновременно и 32-х разрядные и 64-х разрядные регионы.

В ядре регионы ввода/вывода PCI устройств были интегрированы в общее управление ресурсами. По этой причине вам не требуется обращаться к переменным конфигурации, чтобы узнать, где ваше устройство отображено на память или пространство ввода/вывода. Предпочтительный интерфейс для получения информации о регионе состоит из следующих функций:

unsigned long pci_resource_start(struct pci_dev *dev, int bar);
Функция возвращает первый адрес (адрес памяти или номер порта ввода/вывода), связанный с одним из шести регионов ввода/вывода PCI. Регион выбирается целым числом **bar** (регистр базового адреса) в диапазоне 0 - 5 (включительно).

unsigned long pci_resource_end(struct pci_dev *dev, int bar);
Функция возвращает последний адрес, который является частью региона ввода/вывода значения **bar**. Заметим, что это последний используемый адрес, а не первый адрес после этого региона.

unsigned long pci_resource_flags(struct pci_dev *dev, int bar);
Эта функция возвращает флаги, связанные с этим ресурсом.

Флаги ресурсов, используемые для определения некоторых особенностей отдельного ресурса. Для ресурсов PCI, связанных регионами ввода/вывода PCI, эта информация извлекается из регистров базовых адресов, но может прийти из других мест для ресурсов, не связанных с PCI устройствами.

Все ресурсные флаги определены в ***<linux/ioport.h>***; наиболее важными являются:

IORESOURCE_IO
IORESOURCE_MEM
: Если связанный регион ввода/вывода существует, один и только один из этих флагов установлен.

IORESOURCE_PREFETCH
IORESOURCE_READONLY
: Эти флаги определяют, является ли область памяти упреждающей и/или защищённой от записи. Последний флаг для ресурсов PCI никогда не устанавливается.

Пользуясь функциями ***pci_resource_***, драйвер устройства может полностью игнорировать нижележащие регистры PCI, так как система уже использовала их для структурирования информации ресурса.

PCI прерывания

Что касается прерывания, PCI прост в обращении. Ко времени загрузки Linux встроенное программное обеспечение компьютера уже присвоило уникальный номер прерывания устройству и драйверу просто необходимо его использовать. Номер прерывания хранится в регистре конфигурации 60 (PCI_INTERRUPT_LINE), который имеет размер в один байт. Это позволяет иметь 256 линий прерываний, но фактический предел зависит от используемого процессора. Драйверу нет необходимости беспокоиться о проверке номера прерывания, так как значение, найденное в **PCI_INTERRUPT_LINE**, гарантированно будет правильным.

Если устройство не поддерживает прерывания, регистр 61 (PCI_INTERRUPT_PIN) равен 0; в противном случае он не равен нулю. Однако, поскольку драйвер знает, является ли его устройство управляемым прерыванием или нет, читать **PCI_INTERRUPT_PIN** обычно необходимости нет.

Таким образом, коду PCI, предназначенному для работы с прерываниями, просто необходимо прочитать байт конфигурации, чтобы получить номер прерывания, который сохраняется в локальную переменную, как показано на следующем коде. Помимо этого, применяется информация, содержащаяся в Главе 10 246.

```
result = pci_read_config_byte(dev, PCI_INTERRUPT_LINE, &myirq);
if (result) {
    /* имеем дело с ошибкой */
}
```

В оставшейся части этого раздела приводится дополнительная информация для любопытных читателей, но не требующаяся для написания драйверов.

Разъём PCI имеет четыре контакта прерывания и периферийные платы могут использовать любой или их все. Каждый контакт подключён индивидуально к контроллеру прерываний в

материнской плате, так что прерывания могут быть разделяемыми без каких-либо электрических проблем. Контроллер прерываний ответственен за связь проводов (контактов) прерывания с аппаратной частью процессора; эта зависящая от платформы операция остаётся в контроллере для достижения платформо-независимости в самой шине.

Регистр конфигурации, доступный только для чтения и расположенный по **PCI_INTERRUPT_PIN**, используется, чтобы сообщить компьютеру, какой единственный контакт действительно используются. Стоит вспомнить, что каждая плата устройства может содержать до восьми устройств; каждое устройство использует один контакт прерывания и сообщает о нём в собственном регистре конфигурации. Разные устройства на одной плате могут использовать разные контакты прерывания или совместно использовать одно.

Регистр **PCI_INTERRUPT_LINE**, с другой стороны, является читаемым/записываемым. Когда компьютер загружается, встроенное программное обеспечение сканирует PCI устройства и устанавливает этот регистр для каждого устройства в соответствии с тем, какой контакт прерывания подключен к этому PCI слоту. Значение присваивается встроенным программным обеспечением, потому что только оно знает, как материнская плата маршрутизирует разные контакты прерывания в процессор. Однако, для драйвера устройства регистр **PCI_INTERRUPT_LINE** является только читаемым. Интересно, что последние версии ядра Linux при некоторых обстоятельствах могут назначить линии прерывания не прибегая к BIOS.

Аппаратные абстракции

Мы завершаем обсуждение PCI делая быстрый взгляд на то, как система работает со множеством контроллеров PCI, имеющихся на рынке. Это просто информационный раздел, призванный показать любопытному читателю, как объектно-ориентированная схема ядра простирается вплоть до самых низких уровней.

Механизм, используемый для реализации аппаратной абстракции, является обычной структурой, содержащей методы. Это мощная техника, которая добавляет только минимальные накладные расходы разыменования указателя к обычным накладным расходам вызова функции. В случае с управлением PCI, единственными аппаратно-зависимыми операциями являются те, которые считывают и записывают конфигурационные регистры, потому что всё остальное выполняется в мире PCI непосредственным чтением и записью в пространства ввода/вывода и адресов памяти, и те находятся под прямым контролем центрального процессора.

Таким образом, соответствующая структура для доступа к конфигурационному регистру включает в себя только два поля:

```
struct pci_ops {
    int (*read)(struct pci_bus *bus, unsigned int devfn, int where, int size,
u32 *val);
    int (*write)(struct pci_bus *bus, unsigned int devfn, int where, int
size, u32 val);
};
```

Структура определена в ***<linux/pci.h>*** и используется в ***drivers/pci/pci.c***, где определяются фактические функции для общего доступа.

Эти две функции, которые воздействуют на конфигурационное пространство PCI, имеют больше накладных расходов, чем разыменование указателя; они используют каскадные

указатели связи из-за высокой объектной ориентированности кода, но накладные расходы не является проблемой в операциях, которые выполняются довольно редко и никогда не являются критичными к скорости. Фактическая реализация из ***pci_read_config_byte(dev, where, val)***, например, преобразуется в:

```
dev->bus->ops->read(bus, devfn, where, 8, val);
```

Разные шины PCI в системе обнаруживаются при загрузке системы и вот тогда создаются объекты **struct pci_bus** и связываются с их функциями, в том числе, поле **ops**.

Реализация аппаратной абстракции через структуры данных "аппаратные операции" является типичной в ядре Linux. Одним из важных примеров является структура данных **struct alpha_machine_vector**. Она определена в ***<asm-alpha/machvec.h>*** и заботится обо всём, что может изменяться между разными компьютерами на основе Alpha.

Взгляд назад: ISA

Шина ISA довольно стара по дизайну и является заведомо низкоэффективной, но всё ещё занимает значительную часть рынка для устройств расширения. Если скорость не важна и вы хотите поддержать старые материнские платы, реализация ISA предпочтительнее PCI. Дополнительным преимуществом этого старого стандарта является то, что если вы любитель электроники, вы можете легко создавать свои собственные ISA устройства, что определённо невозможно с PCI.

С другой стороны, большим недостатком ISA является то, что она тесно связана с архитектурой ПК; интерфейсная шина имеет все ограничения процессора 80286 и причина бесконечной боли системных программистов. Другой большой проблемой с дизайном ISA (унаследованной от оригинального IBM PC) является отсутствие географической адресации, которая привела ко многим проблемам и длительным циклам отключите-переставьте перемычки-подключите-проверьте при добавлении новых устройств. Интересно отметить, что даже на самых старых компьютерах Apple II уже использовалась географическая адресация, и они уже поддерживали безджамперные платы расширения.

Несмотря на свои большие недостатки, ISA всё ещё используется в нескольких неожиданных местах. Например, серия VR41xx процессоров MIPS, используемая в нескольких карманных компьютерах, включает ISA-совместимую шину расширения, как это ни странно. Причиной этих неожиданных использований ISA является крайне низкая стоимость некоторого устаревшего оборудования, такого как сетевые карты на базе 8390, так что процессор с электрическими сигналами ISA может легко использовать ужасные, но дешёвые устройства ПК.

Аппаратные ресурсы

ISA устройство может быть оснащено портами ввода/вывода, областями памяти и линиями прерывания. Даже несмотря на то, что x86 процессоры поддерживают 64 Кб пространство портов ввода/вывода (то есть процессор имеет 16 адресных линий), некоторое старое оборудование ПК декодирует только младшие 10 адресных линий. Это ограничивает используемое адресное пространство до 1024 портов, потому что любой адрес в диапазоне от 1 Кб до 64 Кб ошибочно принимается за более низкий адрес любым устройством, которое декодирует только младшие адресные линии. Некоторая периферия обходит это ограничение отображая только один порт в младший килобайт и используя старшие адресные линии для выбора между разными регистрами устройства. Например, устройство, отображаемое на 0x340, может безопасно использовать порт 0x740, 0xB40 и так далее.

Если количество портов ввода/вывода ограничено, с доступом к памяти всё ещё хуже. Устройство ISA можно использовать только память в диапазоне от 640 Кб до 1 Мб, а также между 15 Мб и 16 Мб для регистра ввода/вывода и управления устройством. Диапазон от 640 Кб до 1 Мб используется BIOS ПК, в VGA-совместимых видеокартах и различными другими устройствами, оставляя мало пространства для новых устройств. Память на 15 Мб, с другой стороны, непосредственно не поддерживает Linux и требуется изменение ядра для такой поддержки, в настоящее время это бесполезный расход времени программирования.

Третьим ресурсом, доступным для плат ISA устройств, являются линии прерывания. К шине ISA подключено ограниченное число линий прерывания и они являются общими для всех интерфейсных плат. В результате, если устройства не были настроены правильно, они могут оказаться использующими одни и те же линии прерывания.

Хотя оригинальная спецификация ISA не разрешает разделение прерывания между устройствами, большинство плат устройств позволяют это. (* Проблема с разделением прерывания идёт из электротехники: если устройство, управляющее сигнальной линией неактивно, применяя уровень напряжения с малым сопротивлением, прерывание не может быть общим. Если, с другой стороны, устройство использует нагрузочный резистор для неактивного логического уровня, возможно совместное использование. В настоящее время это норма. Однако, всё ещё есть потенциальный риск потери событий прерывания, так как прерывания ISA запускаются перепадом, вместо срабатывания по уровню. Запускаемые по перепаду прерывания проще для реализации на оборудовании, но не поддаются безопасному совместному использованию.) Совместное использование прерывания на программном уровне описано в разделе "Разделяемые прерывания" 266 в Главе 10 246.

Программирование ISA

Что же касается программирования, для облегчения доступа к ISA устройствам нет никакой особой помощи в ядре или в BIOS (как есть, например, для PCI). Единственными средствами, которые вы можете использовать, являются регистрации портов ввода/вывода и линий прерывания, описанные в разделе "Установка обработчика прерываний" 247 в Главе 10 246.

Приёмы программирования, показанные в первой части этой книги, распространяются на ISA устройства; драйвер может проверять порты ввода/вывода, а линия прерывания должна определяться автоматически одним из методов, показанных в разделе "Автоопределение номера прерывания" 252 в Главе 10 246.

Вспомогательные функции ***isa_readb*** и друзья были кратко представлены в разделе "Использование памяти ввода/вывода" 237 в Главе 9 224 и сказать о них больше нечего.

Спецификация Plug-and-Play

Некоторые новые платы устройств ISA следуют своеобразным правилам проектирования и требуют специальной последовательности инициализации для упрощения установки и настройки дополнительный интерфейсных плат. Эта спецификация называется ***plug and play*** (PnP, включай и работай) и состоит из набора громоздких правил для создания и конфигурирования безджамперных ISA устройств. Устройства PnP реализуют перемещаемые регионы ввода/вывода; BIOS в ПК, отвечающая за перемещение, напоминает PCI.

Короче говоря, целью PnP является получение такой же гибкости, как в устройствах PCI, без изменения основного электрического интерфейса (шины ISA). С этой целью спецификации

определяют набор аппаратно-независимых регистров конфигурации и способ географической адресации интерфейсных плат, хотя и физическая шина не содержит по-платной (географической) проводки - каждая сигнальная линия ISA подключается к каждому имеющемуся слоту.

Географическая адресация работает назначая малое целое число, называемое card select number (CSN, номер выбора карты), для каждой периферии PnP в компьютере. Каждое устройство PnP имеет уникальный серийный идентификатор размером 64 бита, который установлен аппаратно в периферийной плате. CSN определение использует уникальный серийный номер, чтобы определить PnP устройства. Но CSN-ы могут быть назначены безопасно только во время загрузки системы, что требует BIOS, осведомлённый о PnP. По этой причине старые компьютеры требуют от пользователя получить и вставить специальную конфигурационную дискету, даже если устройство совместимо с PnP.

Интерфейсные платы, использующие спецификации PnP, сложны на аппаратном уровне. Они гораздо более сложные, чем платы PCI и требуют сложного программного обеспечения. Столкновение с трудностями при установке этих устройств не необычно и даже если установка идёт хорошо, вы-прежнему сталкиваетесь с ограничениями производительности и ограниченным пространством ввода/вывода шины ISA. Гораздо лучше вместо этого установить PCI устройства, когда возможно, и наслаждаться новыми технология.

Если вы заинтересованы в программной конфигурации PnP, вы можете просмотреть ***drivers/net/3c509.c***, чьи зондирующие функции имеют дело с устройствами PnP. Ядро версии 2.6 вобрало много работы в области поддержки устройств PnP, так что по сравнению с предыдущими версиями ядра было убрано много не гибких интерфейсов.

PC/104 и PC/104+

В настоящее время в промышленных странах достаточно модны две шинных архитектуры: PC/104 и PC/104 +. Оба являются стандартом в одноплатных компьютерах класса ПК.

Оба стандарта относятся к особым форм-факторам для печатных плат, а также электрическим/механическим спецификациям для межплатных соединений. Практическим преимуществом этих шин является то, что они позволяют подключать платы вертикально друг на друга, используя для соединения разъём на одной стороне устройства.

Электрическая и логическая схема этих двух шин идентична ISA (PC/104) и PCI (PC/104 +), так что программное обеспечение не заметит никакой разницы между обычными шинами настольного компьютера и этими двумя.

Другие шины ПК

PCI и ISA являются наиболее часто используемыми периферийными интерфейсами в мире персональных компьютеров, но они не являются единственными. Вот краткий обзор особенностей других шин, найденных на рынке ПК.

MCA

Micro Channel Architecture (MCA, микроканальная архитектура) является стандартом IBM, используемым в компьютерах PS/2 и некоторых ноутбуках. На аппаратном уровне Micro Channel имеет больше функций, чем ISA. Он поддерживает multimaster DMA (многоабонентский ПДП), 32-х разрядные адреса и линии передачи данных, разделяемые линии прерываний, а также географическую адресацию для доступа к находящимся на плате регистрам

конфигурации. Такие регистры называются Programmable Option Select (POS, выбор программируемых опций), но они не имеют всех возможностей регистров PCI. Поддержка в Linux для Micro Channel включает в себя функции, которые экспортируются для модулей.

Драйвер устройства может прочитать целое значение **MCA_bus**, чтобы увидеть, работает ли он на компьютере с Micro Channel. В качестве макроса препроцессора так же определён макрос **MCA_bus__is_a_macro**. Если **MCA_bus__is_a_macro** не определён, то **MCA_bus**, являющееся целой переменной, экспортируется в код модуля. Оба символа, **MCA_BUS** и **MCA_bus__is_a_macro** определены в ***<asm/processor.h>***.

EISA

Шина Extended ISA (EISA, расширенная ISA) является 32-х разрядным расширением ISA, с совместимым интерфейсным разъёмом; платы устройств ISA могут быть подключены к разъёму EISA. Дополнительные провода разведены под контактами ISA (отдельный небольшой разъём).

Подобно PCI и MCA, шина EISA разработана для подключения безджамперного устройства и она имеет те же возможности, как MCA: 32-х разрядная адресная и линия передачи данных, многоабонентский DMA и разделяемые линии прерываний. Устройства EISA настраиваются программным обеспечением, но им не требуется какая-либо особенная поддержка от операционной системы. Драйверы EISA уже существуют в ядре Linux для сетевых устройств и контроллеров SCSI.

Драйвер EISA проверяет значение **EISA_bus**, чтобы определить, поддерживает ли данный компьютер шину EISA. Как и **MCA_bus**, **EISA_bus** является либо макросом, либо переменной, в зависимости, определён ли **EISA_bus__is_a_macro**. Оба символа определены в ***<asm/processor.h>***.

Ядро имеет полную поддержку EISA для устройств с sysfs и функциональностью управления ресурсами. Это находится в каталоге ***drivers/eisa***.

VLB

Ещё одним расширением для ISA является интерфейсная шина VESA Local Bus (VLB), которая расширяет разъёмы ISA, добавляя третий продольный слот. Устройство можно подключить только в этот дополнительный разъём (без подключения двух связанных с ним разъёмов ISA), потому что слот VLB дублирует все важные сигналы от разъёмов ISA. Такие "автономные" периферийные устройства VLB не использующие слот ISA редки, так как большинству устройств необходимо достичь задней панели, чтобы их внешние разъёмы стали доступны.

Шина VESA является гораздо более ограниченной по своим возможностям, чем шины EISA, MCA и PCI и исчезает с рынка. Не существует специальной поддержки в ядре для VLB. Однако, оба драйвера, Lance Ethernet и IDE дисков, в Linux 2.0 могут иметь дело с VLB версиями своих устройств.

SBus

Хотя большинство компьютеров в настоящее время оснащаются PCI или интерфейсной шиной ISA, большинство старых рабочих станций на базе SPARC используют для подключения своей периферии SBus.

SBus является вполне современной разработкой, хотя она была в ходу в течение долгого времени. Она предназначена для процессорно-независимых (хотя её используют только компьютеры SPARC) и оптимизирована для периферийных плат ввода/вывода. Другими словами, вы не можете подключить дополнительную оперативную память в слоты SBus (платы расширения памяти уже давно забыты даже в мире ISA, а PCI не поддерживает их совсем). Такая оптимизация предназначена для упрощения разработки и аппаратных средств и системного программного обеспечения за счёт некоторого дополнительного усложнения в материнской плате.

Такое смещение в сторону ввода/вывода приводит к использованию в периферии ***виртуальных*** адресов для передачи данных, минуя, таким образом, необходимость выделять непрерывный буфер DMA. Материнская плата отвечает за декодирование виртуальных адресов и связи их с физическими адресами. Это требует подключения к шине MMU (memory management unit, блок управления памяти); набор микросхем, отвечающих за эту задачу, называется IOMMU. Хотя это что-то более сложное, чем использование на интерфейсной шине физических адресов, эта конструкция существенно упрощается тем фактом, что процессоры SPARC всегда разрабатывались с поддержкой отдельного от основного ядра процессора ядра MMU (физически или, по крайней мере, концептуально). Собственно, этот конструкторский выбор является общим для других разработок мощных процессоров и выгодно полный. Ещё одной особенностью этой шины является то, что платы устройств используют обширную географическую адресацию, так что нет необходимости реализации дешифратора адреса в каждой периферии или решения конфликтов адресации.

Периферия SBus использует язык Форт в их ПЗУ для самостоятельной инициализации. Форт был выбран потому, что интерпретатор является простым и, следовательно, может быть легко реализован в прошивке любой компьютерной системы. Кроме того, спецификация SBus описывает процесс загрузки, так что совместимые устройства ввода/вывода легко устанавливаются в систему и распознаются во время загрузки системы. Это было большим шагом для поддержки мультиплатформенных устройств; это совершенно другой от ПК мир, где мы привыкли использовать ISA. Однако она не стала успешной по ряду коммерческих причин.

Хотя текущие версии ядра предлагают совершенно полнофункциональную поддержку устройств SBus, эта шина используется в наши дни так мало, что не стоит подробного описания здесь. Заинтересованные читатели могут взглянуть на исходные файлы в ***arch/sparc/kernel*** и ***arch/sparc/mm***.

NuBus

Другой интересной, но почти забытой интерфейсной шиной является NuBus. Она существует на старых компьютерах Mac (тех, которые используют семейство процессоров M68k).

Все такие шины являются отображёнными на память (как и всё с M68k) и такие устройства адресуются только географически. Это хорошо и типично для Apple, как что много более старый Apple II уже имел аналогичную схему шины. Что плохо, так это то, что почти невозможно найти документацию по Nubus, благодаря политики закрытия всего, которой Apple всегда следовал со своими компьютерами Mac (и в отличие от предыдущих Apple II, исходный код которого и схемы были доступны за небольшую плату).

Файл ***drivers/nubus/nubus.c*** включает в себя почти всё, что мы знаем об этой шине и это интересно почитать; он показывает, насколько сложное реверсивное проектирование должны были сделать разработчики.

Внешние шины

Одной из самых последних записей в области интерфейса шин является целый класс внешний шин. Он включает в себя USB, FireWire, и IEEE1284 (внешняя шина на основе параллельного порта). Эти интерфейсы немного похожи на старые и не очень внешние технологии, такие как PCMCIA/CardBus и даже SCSI.

Концептуально эти шины не являются ни полнофункциональными интерфейсными шинами (какой является PCI), ни молчаливыми каналами связи (такими, как последовательные порты). Трудно классифицировать программное обеспечение, которому необходимо использовать их возможности, как это обычно делается делением на два уровня: драйвер для аппаратного контроллера (подобно драйверам для адаптеров PCI SCSI или PCI контроллеров, представленных в разделе "Интерфейс PCI") и драйвер для определённого "клиентского" устройства (подобно тому, как ***sd.c*** обрабатывает обычные диски SCSI и так называемые PCI драйверы, имеющие дело с картами, подключаемыми к шине).

Краткая справка

В этом разделе просуммированы символы, представленные в этой главе:

#include <linux/pci.h>
Заголовок, который подключает символические имена регистров PCI и несколько идентификаторов поставщиков и устройств.

struct pci_dev;
Структура, которая представляет PCI устройство в ядре.

struct pci_driver;
Структура, которая представляет драйвер PCI. Все драйверы PCI должны её определять.

struct pci_device_id;
Структура, которая описывает типы устройств PCI, которые поддерживает этот драйвер.

int pci_register_driver(struct pci_driver *drv);
int pci_module_init(struct pci_driver *drv);
void pci_unregister_driver(struct pci_driver *drv);
Функции, которые регистрируют или отменяют регистрацию PCI драйвера в ядре.

struct pci_dev *pci_find_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
struct pci_dev *pci_find_device_reverse(unsigned int vendor, unsigned int device, const struct pci_dev *from);
struct pci_dev *pci_find_subsys (unsigned int vendor, unsigned int device, unsigned int ss_vendor, unsigned int ss_device, const struct pci_dev *from);
struct pci_dev *pci_find_class(unsigned int class, struct pci_dev *from);
Функции, которые ищут устройство в списке устройств с заданной сигнатурой или принадлежащие к определённому классу. Если ничего не найдено, возвращается значение **NULL**. **from** используется для продолжения поиска; она должно быть NULL при первом вызове любой функции и она должна указывать на только что найденное устройство, если вы ищете несколько устройств. Эти функции не рекомендуется использовать, используйте вместо них варианты ***pci_get_***.

struct pci_dev *pci_get_device(unsigned int vendor, unsigned int device, struct pci_dev *from);
struct pci_dev *pci_get_subsys(unsigned int vendor, unsigned int device,
unsigned int ss_vendor, unsigned int ss_device, struct pci_dev *from);
struct pci_dev *pci_get_slot(struct pci_bus *bus, unsigned int devfn);

Функции, которые ищут список устройств для устройств с заданной сигнатурой или принадлежащих к определённому классу. Если ничего не найдено, возвращается значение **NULL**. **from** используется для продолжения поиска; она должна быть **NULL** при первом вызове любой функции и она должна указывать на только что найденное устройство, если вы ищете несколько устройств. Возвращаемая структура увеличивает свой счётчик ссылок и после завершения работы с ней вызывающего должна быть вызвана функция ***pci_dev_put***.

int pci_read_config_byte(struct pci_dev *dev, int where, u8 *val);
int pci_read_config_word(struct pci_dev *dev, int where, u16 *val);
int pci_read_config_dword(struct pci_dev *dev, int where, u32 *val);
int pci_write_config_byte (struct pci_dev *dev, int where, u8 val);
int pci_write_config_word (struct pci_dev *dev, int where, u16 val);
int pci_write_config_dword (struct pci_dev *dev, int where, u32 val);

Функции, которые читают или пишут реестр PCI конфигурации. Хотя ядро Linux заботится о порядке байтов, программист должен быть внимателен с порядком байтов при сборке многобайтовых значений из отдельных байтов. Шина PCI использует little-endian (сначала младший).

int pci_enable_device(struct pci_dev *dev);

Разрешение PCI устройства.

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_flags(struct pci_dev *dev, int bar);

Функции, которые обрабатывают ресурсы PCI устройства.

Глава 13, USB драйверы

Universal Serial Bus (USB, Универсальная Последовательная Шина) является соединением между компьютером и несколькими периферийными устройствами. Первоначально она была создана для замены широкого круга медленных и различных шин, параллельной, последовательной и клавиатурного соединений, на один тип шины, чтобы к ней могли подключаться все устройства. (* Части этой главы основаны на находящейся в ядре документации для USB кода ядра Linux, который был написан разработчиками ядра USB и выпущен под GPL.) USB развилась позже этих медленных соединений и в настоящее время поддерживает практически любой тип устройства, который может быть подключен к ПК. Последние версии спецификации USB добавили высокоскоростное соединение с теоретическим пределом скорости в 480 Мбит/с.

Топологически, подсистема USB не выходит наружу как шина; скорее, это дерево, построенное из нескольких соединений "точка-точка". Соединения представляют собой четырёх-проводные кабели (земля, питание и два сигнальных провода), которые соединяют устройство и концентратор (хаб), подобно витой паре Ethernet. Контроллер узла (хост) USB отвечает на запросы каждого USB устройства, если оно имеет какие-либо данные для передачи. Из-за такой топологии USB стройство никогда не может начать передачу данных не будучи сначала запрошенным хост-контроллером. Такая конфигурация позволяет очень легко реализовать систему plug-and-play (включай и работай), в которой устройства могут автоматически конфигурироваться хост-компьютером.

Шина очень проста на технологическом уровне, поскольку это реализация с одним мастером, в которой хост-компьютер опрашивает разнообразные периферийные устройства. Несмотря на это внутреннее ограничение, шина имеет несколько интересных особенностей, таких как возможность для устройства запросить фиксированную полосу пропускания для передачи данных для надёжной поддержки ввода/вывода видео и аудио. Другой важной особенностью USB является то, что она выступает лишь в качестве канала связи между устройством и хостом, не требуя особого понимания или структур для доставляемых данных. (* На самом деле, некоторые структуры есть, но это в основном сводится к требованию, чтобы взаимодействие вписывались в один из нескольких предопределённых классов: например, клавиатура не будет выделять полосу пропускания, в то время как некоторые видео-камеры будут.)

Спецификации протокола USB определяют набор стандартов, которым может следовать

любое устройство определённого типа. Если устройство следует такому стандарту, специальный драйвер для этого устройства не требуется. Эти разные типы называются классами и состоят из таких вещей, как устройства хранения данных, клавиатуры, мыши, джойстики, сетевые устройства и модемы. Другие типы устройств, которые не вписываются в эти классы, требуют от поставщика собственного драйвера, написанного для данного специфического устройства. Видео устройства и устройства для преобразования USB в последовательный интерфейс являются хорошим примером, когда не существует определённого стандарта и каждому устройству от разных производителей необходим драйвер.

Эти особенности, вместе с присущей ей конструкции возможности горячего подключения, делают USB удобным, недорогим механизмом для подключения (и отключения) нескольких устройств к компьютеру без необходимости выключения системы, открывания крышки и ругани, имея дело с винтами и проводами.

Ядро Linux поддерживает два основных типа драйверов USB: драйверы на хост-системе и драйверы на устройстве. USB драйверы для хост-системы управляют USB-устройствами, которые к ней подключены, с точки зрения хоста (обычно хостом USB является персональный компьютер.) USB-драйверы в устройстве контролируют, как одно устройство видит хост-компьютер в качестве устройства USB. Поскольку термин "драйверы устройств USB" очень запутывает, разработчики USB создали термин "драйверы USB гаджетов (приспособлений)", чтобы описать, что этот драйвер управляет устройством USB, которое подключается к компьютеру (вспомните, что Linux также работает в тех крошечных встраиваемых устройствах). В этой главе подробно рассказывается, каким образом на персональном компьютере работает система USB. Драйверы USB приспособлений в данный момент времени находятся за рамками этой книги.

Как показано на Рисунке 13-1, USB драйверы находятся между различными подсистемами ядра (блочными, сетевыми, символьными и так далее) и аппаратными USB контроллерами. Ядро USB предоставляет интерфейс для драйверов USB, используемый для доступа и управления USB оборудованием, без необходимости беспокоится о различных типах аппаратных контроллеров USB, которые присутствуют в системе.

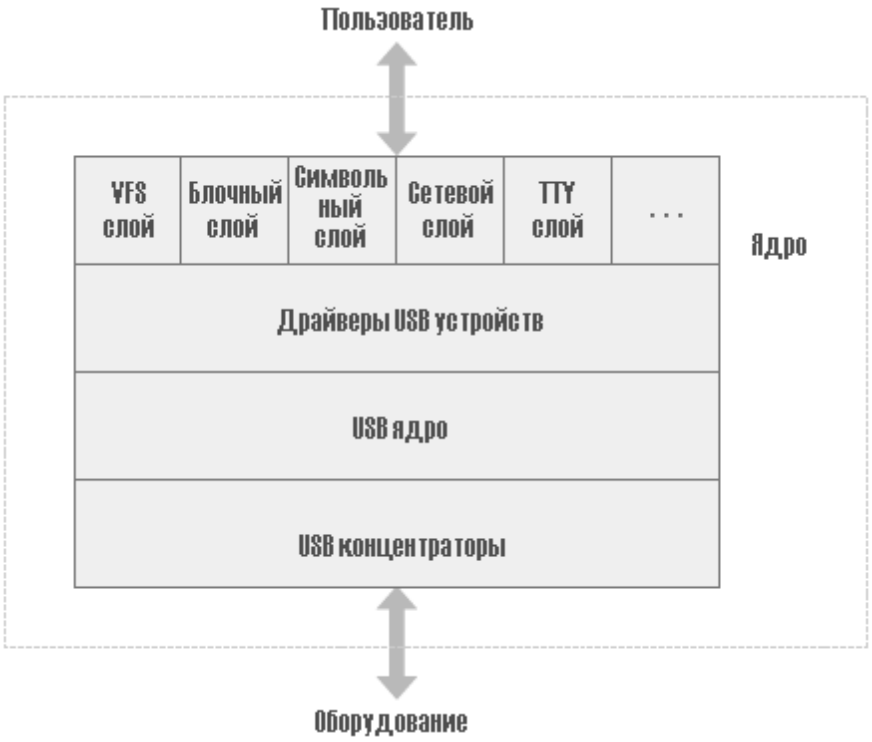


Рисунок 13-1. Обзор USB драйвера

Основы USB устройства

USB-устройство представляет собой очень сложную вещь, как описано в официальной документации USB (доступной на http://www.usb.org). К счастью, ядро Linux предоставляет подсистему, называемую ядром USB, чтобы справляться с большинством сложностей. В этой главе описывается взаимодействие между драйвером и USB ядром. Рисунок 13-2 показывает, как USB-устройства состоят из конфигураций, интерфейсов и оконечных точек и как USB драйверы связаны с интерфейсами USB, а не всего устройства USB.

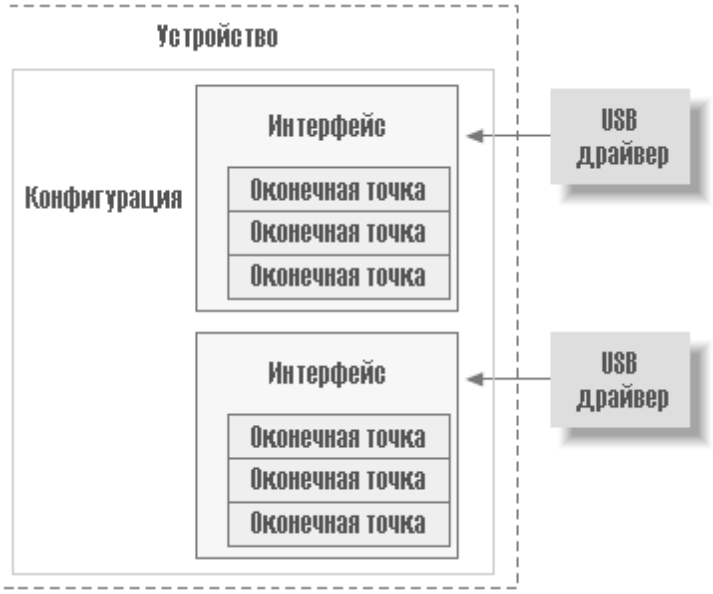


Рисунок 13-2. Обзор USB устройства

Оконечные точки

Самый основной формой USB взаимодействия является то, что называется ***endpoint (оконечная точка)***. Оконечная точка USB может переносить данные только в одном направлении, либо со стороны хост-компьютера на устройство (называемая оконечной точкой ***OUT***) или от устройства на хост-компьютер (называемая оконечной точкой ***IN***). Оконечные точки можно рассматривать как однонаправленные трубы.

Оконечная точка USB может быть одной из четырёх типов, которые описывают, каким образом передаются данные:

CONTROL (Управляющая)

Управляющие оконечные точки используются для обеспечения доступа к различным частям устройства USB. Они широко используются для настройки устройства, получения информации об устройстве, посылке команд в устройство, или получения статусных сообщений устройства. Эти оконечные точки, как правило, малы по размеру. Каждое устройство имеет управляющую оконечную точку, называемую "endpoint 0", которая используется ядром USB для настройки устройства во время подключения. Эти передачи гарантируются протоколом USB, чтобы всегда иметь достаточную зарезервированную пропускную способность шины для их передачи на устройство.

INTERRUPT (Прерывание)

Оконечные точки прерывания передают небольшие объёмы данных с фиксированной частотой каждый раз, когда USB хост запрашивает устройство для передачи данных. Эти оконечные точки являются основным транспортным методом для USB клавиатур и мышей. Они также часто используются для передачи данных на USB устройства для управления устройством, но обычно не используются для передачи больших объёмов данных. Эти передачи гарантируются протоколом USB, чтобы всегда иметь достаточную зарезервированную пропускную способность для их передачи.

BULK (Поточная)

Поточные оконечные точки передают большие объёмы данных. Эти оконечные точки, как

правило, значительно больше (они могут содержать больше символов за один раз), чем оконечные точки прерывания. Они являются обычными для устройств, которые должны передавать любые данные, которые должны пройти через шину, без потери данных. Этим передачам протокол USB не гарантирует выполнения в определённые сроки. Если на шине нет достаточного места, чтобы отправить целый пакет BULK, он распадается на несколько передач в или из устройства. Эти оконечные точки общеприняты на принтерах, устройствах хранения и сетевых устройствах.

ISOCHRONOUS (Изохронная)

Изохронные оконечные точки также передают большие объёмы данных, но этим данным не всегда гарантирована доставка. Эти оконечные точки используются в устройствах, которые могут обрабатывать потери данных, и больше полагаются на сохранение постоянного потока поступающих данных. При сборе данных в реальном времени, таком, как аудио- и видео-устройства, почти всегда используются такие оконечные точки.

Управляющие и поточные оконечные точки используются для асинхронной передачи данных, когда драйвер решает их использовать. Оконечные точки прерывания и изохронные точки являются периодическими. Это означает, что эти оконечные точки созданы для передачи данных непрерывно за фиксированное время, что приводит к тому, что их пропускная способность защищена ядром USB.

Оконечные точки USB описаны в ядро структурой **struct usb_host_endpoint**. Эта структура содержит реальную информацию оконечной точки в другой структуре, названной **struct usb_endpoint_descriptor**. Последняя структура содержит все специфичные USB данные точно в том формате, который указало само устройство. Полями этой структуры, с которыми имеют дело драйверы, являются:

bEndpointAddress

Это адрес USB этой данной оконечной точки. Также в это 8-ми разрядное значение включено направление оконечной точки. В это поле могут быть помещены битовые маски **USB_DIR_OUT** и **USB_DIR_IN**, чтобы определить, куда направлены данные этой оконечной точки, в устройство или в хост.

bmAttributes

Это тип оконечной точки. В этом значении должна присутствовать битовая маска **USB_ENDPOINT_XFERTYPE_MASK**, чтобы определить, является ли конечная точка типом **USB_ENDPOINT_XFER_ISOC**, **USB_ENDPOINT_XFER_BULK** или типом **USB_ENDPOINT_XFER_INT**. Эти макросы определяют изохронные, поточные и оконечные точки прерывания, соответственно.

wMaxPacketSize

Это максимальный размер в байтах, который эта оконечная точка может обработать за раз. Заметим, что возможно для драйвера отправить объём данных в оконечной точке, превышающий это значение, но во время фактической передачи на устройство эти данные будут разделены на куски по **wMaxPacketSize**. Для высокоскоростных устройств это поле может быть использовано для поддержки режима высокой пропускной способности для оконечной точки с помощью нескольких дополнительных битов в верхней части значения. Для более подробной информации о том, как это делается, смотрите спецификацию USB .

bInterval

Если эта оконечная точка имеет тип прерывания, это значение является установкой

интервала для оконечной точки, то есть времени между запросами прерывания для оконечной точки. Это значение представляется в миллисекундах.

Поля этой структуры не имеют "традиционной" для ядра Linux схемы именования. Это происходит потому, что эти поля напрямую соотносятся с именами полей в спецификации USB. Программисты ядра USB посчитали, что было более важно использовать специфицированные имена, чтобы избежать путаницы при чтении спецификации, чем иметь имена переменных, которые выглядят знакомо для программистов Linux.

Интерфейсы

Оконечные точки USB завёрнуты в ***интерфейсы***. USB интерфейсы обрабатывают только один тип логического USB соединения, такого как мышь, клавиатура или аудио поток. Некоторые USB устройства имеют несколько интерфейсов, таких как USB динамик, который может состоять из двух интерфейсов: USB клавиатура для кнопок и USB аудио поток. Поскольку USB интерфейс представляет собой основную функциональность, каждый драйвер USB управляет интерфейсом; так что в случае динамика, Linux необходимы два разных драйвера для одного аппаратного устройства.

USB интерфейсы могут иметь дополнительные параметры настройки, которые представляют собой разные наборы параметров интерфейса. Первоначальное состояние интерфейса находится в первой настройке под номером 0. Другие параметры могут быть использованы, чтобы управлять отдельными оконечными точками по-разному, например, резервировать разные размеры полосы пропускания USB устройства. Каждое устройство с изохронной оконечной точкой использует дополнительные параметры на том же самом интерфейсе.

USB интерфейсы описаны в ядре структурой **struct usb_interface**. Эта структура является тем, что ядро USB передаёт в USB драйверы и тем, за чьё управление затем отвечает драйвер USB. Важными полями в этой структуре являются:

struct usb_host_interface *altsetting
: Массив интерфейсных структур, содержащий все дополнительные настройки, которые могут быть выбраны для этого интерфейса. Каждая структура **struct usb_host_interface** состоит из набора конфигураций оконечной точки, определённой структурой **struct usb_host_endpoint**, описанной выше. Заметим, что эти интерфейсные структуры не имеют какого-то определённого порядка.

unsigned num_altsetting
: Количество дополнительных параметров, на которые указывает указатель **altsetting**.

struct usb_host_interface *cur_altsetting
: Указатель на массив **altsetting**, обозначающий активные в настоящее время настройки для этого интерфейса.

int minor
: Если драйвер USB, связанный с этим интерфейсом, использует старший номер USB, эта переменная содержит младший номер, присвоенный интерфейсу USB ядром. Это справедливо только после успешного вызова ***usb_register_dev*** (описываемого далее в этой главе).

В структуре **struct usb_interface** есть и другие поля, но USB драйверам знания о них не требуются.

Конфигурации

Сами USB интерфейсы завёрнуты в ***конфигурации***. USB устройство может иметь множество конфигураций и может переключаться между ними с целью изменения состояния устройства. Например, некоторые устройства, которые позволяют загружать в них программное обеспечение, для решения этой задачи содержат несколько конфигураций. В один момент времени может быть разрешена только одна конфигурация. Linux не очень хорошо обрабатывает множественную конфигурацию USB устройства, но, к счастью, они встречаются редко.

Linux описывает USB конфигурации структурой **struct usb_host_config** и устройства USB в целом структурой **struct usb_device**. Драйверам устройств USB обычно даже не требуется читать или записывать какие-то значения в эти структуры, поэтому подробности их здесь не описываются. Любопытный читатель может найти их описания в файле ***include/linux/usb.h*** в дереве исходных текстов ядра.

Драйверу USB устройства обычно требуется преобразовать данные из заданной структуры **struct usb_interface** в структуру **struct usb_device**, которая необходима ядру USB для широкого круга функциональных вызовов. Для этого предоставляется функция ***interface_to_usbdev***. Надеемся, что в будущем все USB вызовы, которым в настоящее время требуется **struct usb_device** будут преобразованы для приёма параметра ***struct usb_interface*** и не будут требовать от драйверов выполнения преобразования.

Так что подводя итоги, USB устройства являются довольно сложными и состоят из множества разных логических единиц. Отношения между этими частями можно описать просто следующим образом:

- Устройство обычно имеют одну или более конфигураций.
- Конфигурации часто имеют один или несколько интерфейсов.
- Интерфейсы обычно имеет один или несколько наборов параметров.
- Интерфейсы имеют ноль или более оконечных точек.

USB и Sysfs

Из-за сложности одного физического устройства USB представление такого устройства в sysfs также достаточно сложное. Как физическое устройство USB (представленное **struct usb_device**), так и индивидуальные интерфейсы USB (представленные **struct usb_interface**) показаны в sysfs как отдельные устройства. (Так происходит потому, что обе эти структуры содержат структуру **struct device**.) Например, для простой USB мыши, которая содержит только один USB интерфейс, для этого устройства было бы следующее дерево каталогов sysfs:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
|-- 2-1:1.0
|  |-- bAlternateSetting
|  |-- bInterfaceClass
|  |-- bInterfaceNumber
|  |-- bInterfaceProtocol
|  |-- bInterfaceSubClass
```

```
| |-- bNumEndpoints
| |-- detach_state
| |-- iInterface
| `-- power
|     `-- state
|-- bConfigurationValue
|-- bDeviceClass
|-- bDeviceProtocol
|-- bDeviceSubClass
|-- bMaxPower
|-- bNumConfigurations
|-- bNumInterfaces
|-- bcdDevice
|-- bmAttributes
|-- detach_state
|-- devnum
|-- idProduct
|-- idVendor
|-- maxchild
|-- power
|   `-- state
|-- speed
`-- version
```

struct usb_device представлена в дереве по адресу:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1
```

хотя USB интерфейс для мыши, интерфейс, который содержится в драйвере USB мыши, находится в каталоге:

```
/sys/devices/pci0000:00/0000:00:09.0/usb2/2-1/2-1:1.0
```

Чтобы понять, что означает этот длинный путь устройства, мы расскажем, как ядро маркирует USB устройства.

Первое USB устройство представляет собой корневой концентратор (хаб, узел). Это контроллер USB, как правило, содержится в PCI устройстве. Контроллер назван так, поскольку он контролирует всю связанную с ним USB шину. Контроллер является мостом между шиной PCI и шиной USB, а также первым устройством USB на этой шине.

Всем корневым узлам USB ядро присваивает уникальный номер. В нашем примере корневой узел называется **usb2**, поскольку он является вторым корневым узлом, зарегистрированным USB ядром. Не существует ограничений на количество корневых узлов, которые могут содержаться в одной системе в любой момент.

Каждое устройство, которое подключено к шине USB, получает номер корневого концентратора в качестве первого номера в его названии. Затем за ним следует символ **-**, а затем номер порта, к которому подключено устройство. Так как устройство в нашем примере подключено к первому порту, к названию добавлена **1**. Таким образом, именем устройства для основного устройства USB мыши является **2-1**. Так как это USB устройство содержит один интерфейс, это вызывает добавление другого устройства в дерево к sysfs пути. До этого момента схема именования для USB интерфейсов аналогична: в нашем примере это **2-1**, затем

двоеточие и номер USB конфигурации, затем точка и номер интерфейса. Так что для этого примера, имя устройства **2-1:1.0**, потому что это первая конфигурация и имеет интерфейс номер ноль.

Так что в итоге USB устройства в sysfs имеют схему обозначения:

```
корневой_узел-порт_узла:конфигурация.интерфейс
```

Так как устройства двигаются дальше в дереве USB и используется всё больше и больше USB концентраторов, номер порта концентратора добавляется в строку после предыдущего номера порта концентратора в цепочке. Для дерева глубиной в два, название устройства выглядит следующим образом:

```
корневой_узел-порт_узла-порт_узла:конфигурация.интерфейс
```

Как можно видеть в предыдущей распечатке каталога USB устройства и интерфейса, вся специальная USB информация доступна непосредственно через sysfs (например, информация **idVendor**, **idProduct** и **bMaxPower**). Один из этих файлов, ***bConfigurationValue***, может быть перезаписан в целях изменения используемой активной конфигурации USB. Это полезно для устройств, которые имеют несколько конфигураций, когда ядро не в состоянии определить, какую конфигурацию выбрать, чтобы должным образом управлять устройством. Ряду USB модемов необходимо иметь правильное значение конфигурации, записанное в этот файл, чтобы для связи с устройством использовать подходящий USB драйвер.

Sysfs не раскрывает всех различных частей устройства USB, так как она останавливается на уровне интерфейс. Любые дополнительные конфигурации, которые может содержать устройство, не показываются, так же как и сведения об оконечных точках, связанных с интерфейсами. Эту информацию можно найти в файловой системе ***usbfs***, которая монтируется в каталог системы ***/proc/bus/usb/***. Файл ***/proc/bus/usb/devices*** показывает всю ту же информацию, представленную в sysfs, а также дополнительную конфигурацию и информацию об оконечных точках для всех USB устройств, которые присутствуют в системе. ***usbfs*** также позволяет программам пользовательского пространства непосредственно общаться с USB устройствами, что позволило многим драйверам ядра переехать в пользовательское пространство, где их легче поддерживать и отлаживать. Драйвер USB сканера является хорошим примером этого, так как его уже давно нет в ядре, поскольку его функциональность теперь содержится в библиотечных программах SANE пользовательского пространства.

Блоки запроса USB

Код USB в ядре Linux взаимодействует со всеми устройствами USB помощью так называемых ***urb*** (USB request block, блок запроса USB). Этот блок запроса описывается структурой **struct urb** и может быть найден в файле ***include/linux/usb.h***.

urb используется для передачи или приёма данных в или из заданной оконечной точки USB на заданное USB устройство в асинхронном режиме. Он используется так же, как в асинхронном коде ввода/вывода в файловой системе используется структура **kiocb** или как в сетевом коде используется **struct skbuff**. В зависимости от потребностей, драйвер USB устройства может выделить для одной оконечной точке много urb-ов или может повторно использовать один urb для множества разных оконечных точек. Каждая оконечная точка в устройстве может обрабатывать очередь urb-ов, так что перед тем, как очередь опустеет, к одной оконечной точке может быть отправлено множество urb-ов. Типичный жизненный цикл urb выглядит следующим образом:

- Создание драйвером USB.
- Назначение в определённую оконечную точку заданного USB устройства.
- Передача драйвером USB устройства в USB ядро.
- Передача USB ядром в заданный драйвер контроллера USB узла для указанного устройства.
- Обработка драйвером контроллера USB узла, который выполняет передачу по USB в устройство.
- После завершения работы с urb драйвер контроллера USB узла уведомляет драйвер USB устройства.

Urb-ы также могут быть отменены в любое время драйвером, который передал urb, или ядром USB, если устройство удалено из системы. urb-ы создаются динамически и содержат внутренний счётчик ссылок, что позволяет им быть автоматически освобождаемыми, когда последний пользователь urb-а отпускает его.

Процедура, описанная в этой главе для обработки urb-ов является полезной, поскольку позволяет потоковое и других сложные, совмещённые взаимодействия, которые позволяют драйверам достигать максимально возможных скоростей передачи данных. Но доступны менее громоздкие процедуры, если вы просто хотите отправить отдельные потоковые или управляющие сообщения и не заботитесь о пропускной способности. (Смотрите раздел "USB передачи без Urb-ов" 340.)

struct urb

Полями структуры **struct urb**, которые имеют значение для драйвера USB устройства являются:

struct usb_device *dev
Указатель на **struct usb_device**, в которую послан этот urb. Эта переменная должна быть проинициализирована драйвером USB перед тем, как urb может быть отправлен в ядро USB.

unsigned int pipe
Информация оконечной точки для указанной **struct usb_device**, которую этот usb будет передавать. Эта переменная должна быть проинициализирована драйвером USB перед тем, как urb может быть отправлен в ядро USB.
Чтобы установить поля этой структуры, драйвер использует по мере необходимости следующие функции, в зависимости от направления передачи. Обратите внимание, что каждая оконечная точка может быть только одного типа.

unsigned int usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВЫХОДНУЮ управляющую оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВХОДНУЮ управляющую оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВЫХОДНУЮ потоковую оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВХОДНУЮ потоковую оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_sndintpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВЫХОДНУЮ оконечную точку прерывания для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_rcvintpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВХОДНУЮ оконечную точку прерывания для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_sndisocpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВЫХОДНУЮ изохронную оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int usb_rcvisocpipe(struct usb_device *dev, unsigned int endpoint)
Указывает ВХОДНУЮ изохронную оконечную точку для указанного USB устройства с указанным номером оконечной точки.

unsigned int transfer_flags
Эта переменная может быть установлена в несколько различных битовых значений, в зависимости от того, что драйвер USB хочет, чтобы происходило с urb. Возможными значениями являются:

URB_SHORT_NOT_OK
Если установлен, он указывает, что любое неполноценное чтение во ВХОДНОЙ оконечной точке, которое может произойти, должно рассматриваться ядром USB как ошибка. Это значение используется только для urb-ов, которые читаются из USB устройства, но не для записи urb-ов.

URB_ISO_ASAP
Если urb изохронный, этот бит может быть установлен, если драйвер хочет, чтобы urb был запланирован, как только это позволит ему использование полосы пропускания, и установить переменную **start_frame** в urb в этой точке. Если для изохронного urb этот бит не установлен, драйвер должен указать значение **start_frame** и должен быть в состоянии восстановить его должным образом, если передача не может начаться в тот момент. Для дополнительной информации об изохронных urb-ах смотрите последующий раздел.

URB_NO_TRANSFER_DMA_MAP
Должен быть установлен, когда urb содержит для передачи буфер DMA. Ядро USB использует как указатель на буфер переменную **transfer_dma**, а не указатель буфера в виде переменной **transfer_buffer**.

URB_NO_SETUP_DMA_MAP
Как и бит **URB_NO_TRANSFER_DMA_MAP**, этот бит используется для управления urb-ми, которые имеют уже установленный буфер DMA. Если он установлен, ядро USB использует буфер, на который указывает переменная **setup_dma**, вместо переменной **setup_packet**.

URB_ASYNC_UNLINK
Если установлен, вызов ***usb_unlink_urb*** для этого urb возвращается почти немедленно и этот urb отсоединяется в фоновом режиме. В противном случае функция

ждёт перед возвратом, пока urb отсоединится полностью и завершится. Используйте этот бит с осторожностью, так как это может сделать вопросы синхронизации очень сложными для отладки.

URB_NO_FSBR
Используется только драйвером UHCI USB хост-контроллера и указывает ему не попробовать выполнять логику Front Side Bus Reclamation. Этот бит, как правило, не должен быть установлен, так как машины с UHCI хост-контроллером создают большие накладные расходы для процессора и шина PCI насыщается ожиданием urb-а, который устанавливает этот бит.

URB_ZERO_PACKET
Если установлен, urb выходного потока заканчивается посылкой короткого пакета не содержащего данных, когда данные выравнивается по границе пакета оконечной точки. Это необходимо некоторым нестандартным USB устройствам (таким, как ряд USB ИК-устройства) для того, чтобы работать правильно.

URB_NO_INTERRUPT
Если установлен, оборудование не может генерировать прерывание после завершения urb-а. Этот бит следует использовать с осторожностью и только тогда, когда в очереди к одной оконечной точке находится множество urb-ов. Функции USB ядра используют это для того, чтобы выполнять передачи DMA буфера.

void *transfer_buffer
Указатель на буфер, который будет использоваться при передаче данных в устройство (для ВЫХОДНОГО urb) или при получении данных из устройства (для ВХОДНОГО urb). Для того, чтобы хост-контроллер правильно получал доступ к этому буферу, он должен быть создан вызовом ***kmalloc***, а не на стеке или статически. Для управляющих оконечных точек этот буфер для стадии передачи данных.

dma_addr_t transfer_dma
Буфер для использования для передачи данных в USB устройство с помощью DMA.

int transfer_buffer_length
Длина буфера, на который указывает переменная **transfer_buffer** или **transfer_dma** (только одна из них может быть использована для urb). Если она 0, USB ядром никакие буферы передачи не используются.
Для ВЫХОДНОЙ оконечной точки, если максимальный размер оконечной точки меньше значения, указанного в этой переменной, передача в USB устройство разбивается на мелкие куски, чтобы правильно передать данные. Это большая передача происходит в последовательных кадрах USB. Гораздо быстрее поместить большой блок данных в один urb и предоставить контроллеру USB узла разделить его на мелкие куски, чем отправить маленькие буферы в последовательно.

unsigned char *setup_packet
Указатель на установочный пакет для управляющего urb-а. Он передаётся перед данными в буфере передачи. Эта переменная действительна только для управляющих urb-ов.

dma_addr_t setup_dma
DMA буфер для установочного пакета для управляющего urb-а. Он передается перед данными в обычном буфере передачи. Эта переменная действительна только для управляющих urb-ов.

usb_complete_t complete

Указатель на завершающую функцию обработки, которая вызывается USB ядром, когда urb полностью передан или при возникновении ошибки с urb-ом. В этой функции драйвер USB может проверить urb, освободить его, или использовать повторно для другой передачи. (Смотрите раздел "Завершение Urb-ов: завершающий обработчик с обратным вызовом" 330 для более подробной информации о завершающем обработчике.)

Тип ***usb_complete_t*** определён как:

```
typedef void (*usb_complete_t)(struct urb *, struct pt_regs *);
```

void *context

Указатель на данные blob (Binary Large Object, большой двоичный объект), который может быть установлены USB драйвером. Он может быть использован в завершающем обработчике, когда urb возвращается в драйвер. Смотрите следующий раздел для более подробной информации об этой переменной.

int actual_length

После завершения urb-а эта переменная равна фактической длине данных или отправленных urb-ом (для ВЫХОДНЫХ urb-ов) или полученных urb-ом (для ВХОДНЫХ urb-ов). Для ВХОДНЫХ urb-ов, она должна быть использована вместо переменной **transfer_buffer_length**, поскольку полученные данные могут быть меньше, чем размер всего буфера.

int status

После завершения urb-а или его обработки USB ядром, эта переменная содержит текущий статус urb-а. Единственным временем, когда USB драйвер может безопасно получить доступ к этой переменной, является функция обработчика завершения urb-а (описанная в разделе "Завершение Urb-ов: завершающий обработчик с обратным вызовом" 330). Это ограничение является защитой от состояний гонок, которые происходят в то время, как urb обрабатывается USB ядром. Для изохронных urb-ов успешное значение (0) в этой переменной указывает лишь, был ли urb отсоединён. Для получения подробного статуса изохронных urb-ов должны быть проверены переменные **iso_frame_desc**.

Допустимые значения для этой переменной включают:

0
Передача urb-а была успешной.

-ENOENT
urb был остановлен вызовом ***usb_kill_urb***.

-ECONNRESET
urb были отсоединён вызовом ***usb_unlink_urb*** и переменная **transfer_flags** в urb-е была установлена в **URB_ASYNC_UNLINK**.

-EINPROGRESS
urb всё ещё обрабатывается контроллерами узлов USB. Если ваш драйвер когда-нибудь увидит это значение, это является ошибкой в вашем драйвере.

-EPROTO
С этим urb-ом произошла одна из следующих ошибок:
- Во время передачи произошла ошибка некорректной последовательности битов (bitstuff).
- Оборудованием своевременно не был получен ответный пакет.

-EILSEQ
Было несоответствие контрольной суммы (CRC) при передаче urb-а.

-EPIPE
Оконечная точка сейчас застряла. Если данная оконечная точка не является управляющей оконечной точкой, эта ошибка может быть сброшена вызовом функции ***usb_clear_halt***.

-ECOMM
Данные в течение передачи были получены быстрее, чем они могли быть записаны в память системы. Эта ошибка случается только с ВХОДНЫМИ urb-ами.

-ENOSR
Данные не могут быть получены из системной памяти при передаче достаточно быстро, чтобы сохранять запрошенную скорость передачи данных USB. Эта ошибка случается только с ВЫХОДНЫМИ urb-ами.

-EOVERFLOW
С urb-ом произошла ошибка "помеха". Ошибка "помеха" происходит, когда оконечная точка принимает больше информации, чем указанный максимальный размер пакета оконечной точки.

-EREMOTEIO
Возникает только если в переменной urb-а **transfer_flags** установлен флаг **URB_SHORT_NOT_OK** и означает, что весь объём данных, запрошенный этим urb-ом, не был принят.

-ENODEV
USB устройство является теперь отключенным от системы.

-EXDEV
Происходит только для изохронного urb-а и означает, что передача была выполнена лишь частично. Для того, чтобы определить, что было передано, драйвер должен посмотреть на статус отдельного фрейма.

-EINVAL
С urb-ом произошло что-то очень плохое. Документация USB ядра описывает, что это значение означает:

```
ISO madness, if this happens: Log off and go home
ISO обезумело, если это происходит: завершите работу и идите домой
```

Это также может произойти, если неправильно установлен какой-то параметр в структуре urb-а или если в USB ядро urb поместил неправильный параметр функции в вызове ***usb_submit_urb***.

-ESHUTDOWN
Был серьёзные ошибки в драйвере контроллера USB узла; теперь он запрещён или устройство было отключено от системы, а urb был получен после удаления устройства. Она может также возникнуть, если во время помещения urb-а в устройство для данного устройства была изменена конфигурация.

Как правило, значения ошибок **-EPROTO**, **-EILSEQ** и **-EOVERFLOW** указывают на проблемы с оборудованием, встроенным программным обеспечением устройства, или кабелем, соединяющим устройство и компьютер.

int start_frame
Устанавливает или возвращает для использования номер начального фрейма для изохронной передачи.

int interval
Интервал, с которым собираются urb-ы. Это справедливо только для urb-ов прерывания или изохронных. Единицы значения существенно отличаются в зависимости от скорости устройства. Для низкоскоростных и полноскоростных устройств единицами являются фреймы, которые эквивалентны миллисекундам. Для высокоскоростных устройств единицами являются микрофреймы, что эквивалентно единицам в 1/8 миллисекунды. Эта значение должно быть установлено драйвером USB для изохронных urb-ов или urb-ов прерывания до того, как urb посылается в USB ядро.

int number_of_packets
Имеет смысл только для изохронных urb-ов и определяет число изохронных буферов передачи, которые должен обработать этот urb. Эта величина должна быть установлена драйвером USB для изохронных urb-ов передачи до того, как urb посылается в USB ядро.

int error_count
Устанавливается USB ядром только для изохронных urb-ов после их завершения. Она определяет число изохронных передач, которые сообщили о любых ошибках.

struct usb_iso_packet_descriptor iso_frame_desc[0]
Имеет смысл только для изохронных urb-ов. Эта переменная представляет собой массив из структур **struct usb_iso_packet_descriptor**, которые составляют этот urb. Такая структура позволяет сразу одним urb-ом определить число изохронных передач. Она также используется для сбора статуса передачи каждой отдельной передачи.

struct usb_iso_packet_descriptor состоит из следующих полей:

unsigned int offset
Смещение в буфере передачи (начиная с 0 для первого байта), где расположены данные этого пакета.
unsigned int length
Размер буфера передачи для этого пакета.
unsigned int actual_length
Длина данных, полученных в буфере передачи для этого изохронного пакета.
unsigned int status
Статус отдельной изохронной передачи этого пакета. Он может иметь такие же возвращаемые значения, как основная переменная статуса структуры **struct urb**.

Создание и уничтожение Urb-ов

Структура **struct urb** не должна быть создана статически в драйвере или внутри другой структуры, потому что это нарушит схему подсчёта ссылок, используемую USB ядром для urb-ов. Она должна быть создана вызовом функции ***usb_alloc_urb***. Эта функция имеет такой прототип:

```
struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
```

Первый параметр, **iso_packets**, это число изохронных пакетов, которые должен содержать этот urb. Если вы не хотите создать изохронный urb, эта переменная должна быть установлена в 0. Второй параметр, **mem_flags**, имеет такой же тип флага, который передаётся в функцию ***kmalloc*** для выделения ядром памяти (для подробной информации об этих флагах смотрите раздел "Аргумент flags" 203 в Главе 8 203). Если функция успешно выделила для urb-а достаточно

пространства, вызывающему возвращается указатель на urb. Если возвращаемое значение равно NULL, внутри ядра USB произошла какая-то ошибка и драйверу необходимо правильно выполнить очистку.

После того, как urb был создан, он должен быть правильно проинициализирован, прежде чем он может быть использован USB ядром. Как проинициализировать различные типы urb-ов, смотрите следующие разделы.

Для того, чтобы сказать USB ядру, что драйвер закончил работу с urb-ом, драйвер должен вызвать функцию ***usb_free_urb***. Эта функция имеет только один аргумент:

```
void usb_free_urb(struct urb *urb);
```

Аргумент является указателем на **struct urb**, которую вы хотите освободить. После вызова этой функции, структура urb-а уходит, и драйвер не может больше получить к ней доступ.

Urb-ы прерывания

Функция ***usb_fill_int_urb*** является вспомогательной функцией для правильной инициализации urb-а, посылаемого в оконечную точку прерывания USB устройства:

```
void usb_fill_int_urb(struct urb *urb, struct usb_device *dev,
                      unsigned int pipe, void *transfer_buffer,
                      int buffer_length, usb_complete_t complete,
                      void *context, int interval);
```

Эта функция содержит много параметров:

struct urb *urb
Указатель на urb для инициализации.

struct usb_device *dev
USB устройство, которому этот urb будет отправлен.

unsigned int pipe
Указывает оконечную точку USB устройства, которому отправляется этот urb. Это значение создаётся упоминавшимся ранее функциями ***usb_sndintpipe*** или ***usb_rcvintpipe***.

void *transfer_buffer
Указатель на буфер, из которого берутся исходящие данные или в который принимаются входящие данные. Заметим, что это не может быть статический буфер и он должен быть создан вызовом ***kmalloc***.

int buffer_length
Размер буфера, на который указывает указатель **transfer_buffer**.

usb_complete_t complete
Указатель на завершающий обработчик, который вызывается при завершении этого urb-а.

void *context

Указатель на массив двоичных данных, который добавляется в структуру urb-а для последующего извлечения с помощью функции обработчика завершения.

int interval
Интервал с которым этот urb должен быть запланирован. Чтобы найти правильные единицы измерения для этого значения, смотрите предыдущее описание структуры **struct urb**.

Поточные Urb-ы

Поточные urb-ы инициализируются в основном так же, как urb-ы прерывания. Функцией, которая делает это, является ***usb_fill_bulk_urb***, и она выглядит следующим образом:

```
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev,
                       unsigned int pipe, void *transfer_buffer,
                       int buffer_length, usb_complete_t complete,
                       void *context);
```

Все параметры функции такие же, как в функции ***usb_fill_int_urb***. Однако, нет параметра **interval**, поскольку поточные urb-ы не имеют значения интервала. Пожалуйста, заметьте, что переменная **unsigned int pipe** должна быть проинициализирована вызовом функций ***usb_sndbulkpipe*** или ***usb_rcvbulkpipe***.

Функция ***usb_fill_bulk_urb*** не устанавливает переменную **transfer_flags** в urb, так что любые изменения этого поля предстоит сделать самому драйверу.

Управляющие Urb-ы

Управляющие urb-ы инициализируются почти так же, как поточные urb-ы, вызовом функции ***usb_fill_control_urb***:

```
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev,
                unsigned int pipe, unsigned char *setup_packet,
                void *transfer_buffer, int buffer_length,
                usb_complete_t complete, void *context);
```

Все параметры функции такие же, как в функции ***usb_fill_bulk_urb***, за исключением того, что есть новый параметр, **unsigned char *setup_packet**, который должен указывать на настроечный пакет данных, который должен быть передан в оконечную точку. Кроме того, переменная **unsigned int pipe** должна быть проинициализирована вызовом функций ***usb_sndctrlpipe*** или ***usb_rcvictrlpipe***.

Функция ***usb_fill_control_urb*** не устанавливает переменную **transfer_flags** в urb, поэтому любое изменение этого поля предстоит сделать самому драйверу. Большинство драйверов не используют эту функцию, так как намного проще использовать синхронные вызовы API, как описывается в разделе "USB передачи без Urb-ов" 340.

Изохронные Urb-ы

Изохронные urb-ы, к сожалению, не имеют функции инициализации, как urb-ы прерывания, управления и поточные urb-ы. Поэтому они должны быть проинициализированы "вручную" в драйвере до того, как они могут быть отправлены в USB ядро. Ниже приводится пример того,

как правильно инициализировать этот тип urb. Он был взят из драйвера ядра ***konicawc.c***, находящегося в каталоге ***drivers/usb/media*** в основном дереве исходных текстов ядра.

```
urb->dev = dev;
urb->context = uvd;
urb->pipe = usb_rcvisocpipe(dev, uvd->video_endp-1);
urb->interval = 1;
urb->transfer_flags = URB_ISO_ASAP;
urb->transfer_buffer = cam->sts_buf[i];
urb->complete = konicawc_isoc_irq;
urb->number_of_packets = FRAMES_PER_DESC;
urb->transfer_buffer_length = FRAMES_PER_DESC;
for (j=0; j < FRAMES_PER_DESC; j++) {
    urb->iso_frame_desc[j].offset = j;
    urb->iso_frame_desc[j].length = 1;
}
```

Отправка Urb-ов

После того, как urb был надлежащим образом создан и проинициализирован USB драйвером, он готов быть отправленным в USB ядро для передачи в USB устройство. Это делается с помощью вызова функции ***usb_submit_urb***:

```
int usb_submit_urb(struct urb *urb, int mem_flags);
```

Параметр **urb** является указателем на urb, который будет передан в устройство. Параметр **mem_flags** эквивалентен тому же параметру, который передаётся при вызове ***kmalloc*** и используется, чтобы указать USB ядру, как выделять любые буферы памяти в данный момент времени.

После того, как urb был успешно отправлен в USB ядро, никогда не следует пытаться обращаться к любым полям структуры urb-а до вызова функции ***complete***.

Поскольку функция ***usb_submit_urb*** может быть вызвана в любое время (в том числе в контексте прерывания), спецификация переменной **mem_flags** должна быть правильной. Есть только три действительно допустимых значения, которые должны быть использованы в зависимости от того, когда вызывается ***usb_submit_urb***:

GFP_ATOMIC
Это значение должно быть использовано всегда при выполнении следующих условий:
- Вызывающий находится в завершающем обработчике urb, прерывании, нижней половине, микрозадаче или обратном вызове таймера.
- Вызывающий удерживает спин-блокировку или блокировку чтения/записи. Заметим, что если удерживается семафор, это значение не является необходимым.
- **current->state** не **TASK_RUNNING**. Состояние всегда **TASK_RUNNING**, пока драйвер не изменил текущее состояние сам.

GFP_NOIO
Это значение должно быть использовано, если драйвер находится в блокирующем вводе/выводе. Оно также должно быть использовано в пути обработки ошибок всех типов устройств хранения.

GFP_KERNEL
Это должно быть использовано во всех других ситуациях, которые не подпадают под одну из вышеупомянутых категорий.

Завершение Urb-ов: завершающий обработчик с обратным вызовом

Если вызов ***usb_submit_urb*** был успешен, передавая контроль над urb в USB ядро, функция возвращает 0; иначе возвращается отрицательное число ошибки. Если функция завершается успешно, завершающий обработчик urb (задаваемый указателем на функцию ***complete***) вызывается только один раз, когда urb завершается. Когда вызывается эта функция, ядро USB завершает работу с URB и контроль над ним теперь возвращается драйверу устройства.

Есть только три пути, как urb может быть завершён и как может быть вызвана функция ***complete***:

- urb успешно отправлен в устройство и устройство возвращает правильное подтверждение. Для ВЫХОДНОГО urb-а, данные были успешно отправлены, а для ВХОДНОГО urb-а, запрошенные данные были успешно получены. Если это произошло, статусная переменная в urb-е установлена в 0.
- Какая-то ошибка произошла во время передачи и приёма данных из устройства. Это отмечено значением ошибки в статусной переменной в структуре urb-а.
- urb был "отсоединён" от USB ядра. Это происходит либо когда драйвер приказывает USB ядру отменить отправку urb-а вызовом ***usb_unlink_urb*** или ***usb_kill_urb***, или когда устройство удаляется из системы и urb были отправлен ему.

Пример того, как проверяются разные возвращаемые значения в течение завершающего вызова urb-а, показан далее в этой главе.

Отмена Urb-ов

Чтобы остановить urb, который был отправлен в USB ядро, должна называться функции ***usb_kill_urb*** или ***usb_unlink_urb***:

```
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);
```

Параметр **urb** для этих обоих функций является указателем на urb, который должен быть отменён.

Когда функцией является ***usb_kill_urb***, жизненный цикл urb-а останавливается. Эта функция обычно используется, когда устройство отключается от системы, в обратном вызове отключения.

Для некоторых драйверов должна быть использована функция ***usb_unlink_urb***, чтобы приказать USB ядру остановить urb. Эта функция не ожидает перед возвращением к вызывающему, пока urb будет полностью остановлен. Это полезно для остановки urb-а в обработчике прерывания или во время удержания спин-блокировки, тогда как ожидание полной остановки urb-а требует способность USB ядра поместить вызывающий процесс в сон. Эта функция требует в urb-е установки значения флага **URB_ASYNC_UNLINK**, чтобы правильно

отработать запрос на остановку.

Написание USB драйвера

Подход к написанию драйвера USB устройства аналогичен драйверу PCI: драйвер регистрирует свой объект драйвера с USB подсистемой и затем использует идентификаторы поставщика и устройства для сообщения, когда его оборудование было установлено.

Какие устройства поддерживает драйвер?

Структура **struct usb_device_id** содержит список различных типов USB устройств, которые поддерживает этот драйвер. Этот список используется ядром USB, чтобы решить, какой драйвер предоставить устройству, или скриптами горячего подключения, чтобы решить, какой драйвер автоматически загрузить, когда устройство подключается к системе.

Структура **struct usb_device_id** определена со следующими полями:

__u16 match_flags
Определяет, какие из следующих полей в структуре устройства должны сопоставляться. Это битовое поле определяется разными значениями **USB_DEVICE_ID_MATCH_***, указанными в файле ***include/linux/mod_devicetable.h***. Это поле, как правило, никогда не устанавливается напрямую, а инициализируется с помощью макросов типа **USB_DEVICE**, описываемых ниже.

__u16 idVendor
Идентификатор поставщика USB для устройства. Этот номер присваивается форумом USB для своих членов и не может быть присвоен кем-то еще.

__u16 idProduct
Идентификатор продукта USB для устройства. Все поставщики, которые имеют выданный им идентификатор поставщика, могут управлять своими идентификаторами продукта, как они предпочитают.

__u16 bcdDevice_lo
__u16 bcdDevice_hi
Определяют нижнюю и верхнюю границу диапазона назначаемого поставщиком номера версии продукта. Значения **bcdDevice_hi** является включительным; его значение является значением наибольшего номера устройства. Обе эти величины представлены в двоично-десятичной (BCD) форме. Эти переменные в сочетании с **idVendor** и **idProduct** используются для определения данного варианта устройства.

__u8 bDeviceClass
__u8 bDeviceSubClass
__u8 bDeviceProtocol
Определяют класс, подкласс и протокол устройства, соответственно. Эти номера присваиваются форумом USB и определены в спецификации USB. Эти значения определяют поведение для всего устройства, в том числе все интерфейсы на этом устройстве.

__u8 bInterfaceClass
__u8 bInterfaceSubClass

__u8 bInterfaceProtocol
Подобно зависимым от устройства вышеприведённым величинам, эти определяют класса, подкласс и протокол отдельного интерфейса, соответственно. Эти номера присваиваются форумом USB и определены в спецификации USB.

kernel_ulong_t driver_info
Это значение не используется для сравнения, но оно содержит информацию о том, что драйвер может использовать, чтобы отличать разные устройства друг от друга в функции обратного вызова ***probe*** драйвера USB.

Как и с PCI устройствами, существует ряд макросов, которые используются для инициализации этой структуры:

USB_DEVICE(vendor, product)
Создаёт **struct usb_device_id**, которая может быть использована только для соответствия указанными значениям идентификаторов поставщика и продукта. Она очень часто используется для устройств USB, которым необходим специальный драйвер.

USB_DEVICE_VER(vendor, product, lo, hi)
Создаёт **struct usb_device_id**, которая может быть использована только для соответствия указанным значениям идентификаторов поставщика и продукта внутри диапазона версий.

USB_DEVICE_INFO(class, subclass, protocol)
Создаёт **struct usb_device_id**, которая может быть использованы для соответствия определённому классу USB устройств.

USB_INTERFACE_INFO(class, subclass, protocol)
Создаёт **struct usb_device_id**, которая может быть использована для соответствия определённому классу USB интерфейсов.

Итак, для простого драйвера USB устройства, который управляет только одним USB устройством от одного поставщика, таблица **struct usb_device_id** будет определяться как:

```
/* таблица устройств, которые работают с этим драйвером */
static struct usb_device_id skel_table [ ] = {
    { USB_DEVICE(USB_SKEL_VENDOR_ID, USB_SKEL_PRODUCT_ID) },
    { }         /* Завершающая запись */
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

Как и с драйвером PCI, необходим макрос **MODULE_DEVICE_TABLE**, чтобы разрешить инструментам пространства пользователя выяснить, какими устройствами может управлять этот драйвер. Но для USB драйверов первым значением в этом макросе должна быть строка **usb**.

Регистрация USB драйвера

Основной структурой, которую должны создать все USB драйверы, является **struct usb_driver**. Эта структура должна быть заполнена драйвером USB и состоит из ряда функций

обратного вызова и переменных, описывающих USB драйвер для кода USB ядра:

struct module *owner
Указатель на модуль владельца этого драйвера. Ядро USB использует его для правильного подсчёта ссылок на этот драйвер USB, чтобы он не выгружался в несвоевременные моменты. Переменной должен быть присвоен макрос **THIS_MODULE**.

const char *name
Указатель на имя драйвера. Он должен быть уникальным среди всех USB драйверов в ядре и, как правило, установлен на такое же имя, что и имя модуля драйвера. Оно проявляется в sysfs в ***/sys/bus/usb/drivers/***, когда драйвер находится в ядре.

const struct usb_device_id *id_table
Указатель на таблицу **struct usb_device_id**, которая содержит список всех различных видов устройств USB, которые драйвер может распознать. Если эта переменная не установлена, функция обратного вызова ***probe*** в драйвере USB никогда не вызывается. Если вы хотите, чтобы ваш драйвер всегда вызывался для каждого USB устройства в системе, создайте запись, которая устанавливает только поле **driver_info**:

```
static struct usb_device_id usb_ids[ ] = {
    {.driver_info = 42},
    { }
};
```

int (*probe) (struct usb_interface *intf, const struct usb_device_id *id)
Указатель на зондирующую функцию в USB драйвере. Эта функция (описанная в разделе "probe и disconnect в деталях" 334) вызывается USB ядром, когда оно думает, что оно имеет структуру **usb_interface**, которую этот драйвер может обработать. Указатель на **struct usb_device_id**, который использовало USB ядро, чтобы принять это решение также передается в эту функцию. Если USB драйвер признаёт переданную ему структуру **usb_interface**, он должен правильно проинициализировать устройство и вернуть 0. Если драйвер не хочет признавать устройство или произошла ошибка, он должен вернуть отрицательное значение ошибки.

void (*disconnect) (struct usb_interface *intf)
Указатель на функцию отключения в USB драйвере. Эта функция (описанная в разделе "probe и disconnect в деталях" 334) вызывается USB ядром, когда структура **usb_interface** была удалена из системы, или когда драйвер выгружается из ядра USB.

Таким образом, чтобы создать значимую структуру **struct usb_driver**, должны быть проинициализированы только пять полей:

```
static struct usb_driver skel_driver = {
    .owner = THIS_MODULE,
    .name = "skeleton",
    .id_table = skel_table,
    .probe = skel_probe,
    .disconnect = skel_disconnect,
};
```

struct usb_driver содержит несколько больше обратных вызовов, которые, как правило, очень часто не используются, и не требуются для правильной работы USB драйвера:

int (*ioctl) (struct usb_interface *intf, unsigned int code, void *buf)
Указатель на функцию ***ioctl*** в USB драйвере. Если он присутствует, то вызывается, когда программа пользовательского пространства делает вызов ***ioctl*** для записи файловой системы устройств ***usbfs***, связанной с устройством USB, относящемуся к этому USB драйверу. На практике только драйвер USB концентратора использует этот ***ioctl***, так как любому другому USB драйверу нет иной реальной необходимости его использовать.

int (*suspend) (struct usb_interface *intf, u32 state)
Указатель на функцию приостановки в USB драйвере. Она вызывается, когда работа устройства должна быть приостановлена USB ядром.

int (*resume) (struct usb_interface *intf)
Указатель на функцию возобновления в USB драйвере. Она вызывается, когда работа устройства возобновляется USB ядром.

Чтобы зарегистрировать **struct usb_driver** в USB ядре, выполняется вызов ***usb_register_driver*** с указателем на **struct usb_driver**. Для USB драйвера это традиционно делается в коде инициализации модуле:

```
static int __init usb_skel_init(void)
{
    int result;

    /* регистрируем этот драйвер в подсистеме USB */
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);

    return result;
}
```

Когда драйвер USB будет выгружаться, необходимо разрегистрировать **struct usb_driver** в ядре. Это делается с помощью вызова ***usb_deregister***. Когда происходит этот вызов, любые USB интерфейсы, которые в настоящее время связаны с этим драйвером, отключаются и для них вызывается функция ***disconnect***.

```
static void __exit usb_skel_exit(void)
{
    /* отменяем регистрацию этого драйвера в подсистеме USB */
    usb_deregister(&skel_driver);
}
```

probe и disconnect в деталях

В структуре **struct usb_driver structure**, описанной в предыдущем разделе, драйвер указывает две функции, которые в соответствующее время вызывает ядро USB. Функция ***probe*** вызывается, когда установлено устройство, которым, как думает ядро USB, должен управлять этот драйвер; функция ***probe*** должна выполнять проверки информации, переданной ей об устройстве, и решать, действительно ли этот драйвер подходит для этого устройства. Функция ***disconnect*** вызывается, когда по каким-то причинам драйвер не должен больше управлять устройством и может делать очистку.

Оба функции обратного вызова ***probe*** и ***disconnect*** вызываются в контексте потока USB узла ядра, так что засыпать в них допускается. Тем не менее, рекомендуется, чтобы большая часть работы выполнялась, когда устройство открыто пользователем, если это возможно, чтобы сократить время зондирования USB к минимуму. Такое требование появляется потому, что USB ядро обрабатывает добавление и удаление устройств USB в одном потоке, так что любой медленный драйвер устройства может привести к замедлению обнаружения USB устройства и это станет заметно для пользователя.

В функции обратного вызова ***probe***, USB драйвер должен проинициализировать любые локальные структуры, которые он может использовать для управления USB устройством. Следует также сохранить в локальные структуры любую необходимую информацию об устройстве, так как это обычно легче сделать в данное время. Например, USB драйверы обычно хотят обнаружить адрес оконечной точки и размеры буферов для данного устройства, так как они необходимы для общения с устройством. Вот пример некоторого кода, который определяет две оконечные точки ВХОДА и ВЫХОДА поточного типа и сохраняет некоторую информацию о них в локальной структуре устройства:

```
/* установить информацию оконечной точки */
/* используем только первые поточные точки входа и выхода */
iface_desc = interface->cur_altsetting;
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;

    if (!dev->bulk_in_endpointAddr &&
        (endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
        == USB_ENDPOINT_XFER_BULK)) {
        /* мы нашли оконечную точку входного потока */
        buffer_size = endpoint->wMaxPacketSize;
        dev->bulk_in_size = buffer_size;
        dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
        dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
        if (!dev->bulk_in_buffer) {
            err("Could not allocate bulk_in_buffer");
            goto error;
        }
    }

    if (!dev->bulk_out_endpointAddr &&
        !(endpoint->bEndpointAddress & USB_DIR_IN) &&
        ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
        == USB_ENDPOINT_XFER_BULK)) {
        /* мы нашли оконечную точку выходного потока */
        dev->bulk_out_endpointAddr = endpoint->bEndpointAddress;
    }
}
if (!(dev->bulk_in_endpointAddr && dev->bulk_out_endpointAddr)) {
    err("Could not find both bulk-in and bulk-out endpoints");
    goto error;
}
```

Этот блок кода сначала в цикле обходит каждую оконечную точку, которая присутствует в этом интерфейсе, и назначает локальный указатель для структуры оконечной точки для

облегчения доступа впоследствии:

```
for (i = 0; i < iface_desc->desc.bNumEndpoints; ++i) {
    endpoint = &iface_desc->endpoint[i].desc;
```

Затем, после того, как мы получили оконечную точку, и если мы уже не нашли ВХОДНУЮ оконечную точку поточного типа, мы проверяем, является ли направление этой оконечной точки ВХОДНЫМ. Это может быть проверено просмотром, содержится ли битовая маска **USB_DIR_IN** в переменной **bEndpointAddress** оконечной точки. Если это так, мы определяем, имеет ли оконечная точки тип поточной или нет, сначала накладывая битовую маску **USB_ENDPOINT_XFERTYPE_MASK** на переменную **bmAttributes**, а затем проверяя, совпадает ли она со значением **USB_ENDPOINT_XFER_BULK**:

```
if (!dev->bulk_in_endpointAddr &&
    (endpoint->bEndpointAddress & USB_DIR_IN) &&
    ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
    == USB_ENDPOINT_XFER_BULK)) {
```

Если все эти тесты успешны, драйвер знает, что нашёл оконечную точку правильного типа и может сохранить информацию об оконечной точке, в которой позднее будет нуждаться для общения через неё, в локальной структуре:

```
/* мы нашли оконечную точку входного потока */
buffer_size = endpoint->wMaxPacketSize;
dev->bulk_in_size = buffer_size;
dev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
dev->bulk_in_buffer = kmalloc(buffer_size, GFP_KERNEL);
if (!dev->bulk_in_buffer) {
    err("Could not allocate bulk_in_buffer");
    goto error;
}
```

Поскольку драйверу USB позднее в жизненном цикле устройства необходимо получать локальные структуры данных, связанные с этой **struct usb_interface**, может быть вызвана функция ***usb_set_intfdata***:

```
/* сохраняем наш указатель на данные в этом интерфейсе устройства */
usb_set_intfdata(interface, dev);
```

Эта функция принимает указатель на любой тип данных и сохраняет его в структуре **struct usb_interface** для последующего доступа. Для получения данных должна быть вызвана функция ***usb_get_intfdata***:

```
struct usb_skel *dev;
struct usb_interface *interface;
int subminor;
int retval = 0;

subminor = iminor(inode);

interface = usb_find_interface(&skel_driver, subminor);
if (!interface) {
    err ("%s - error, can't find device for minor %d",
```

```
                __FUNCTION__, subminor);
        retval = -ENODEV;
        goto exit;
    }

    dev = usb_get_intfdata(interface);
    if (!dev) {
        retval = -ENODEV;
        goto exit;
    }
```

usb_get_intfdata обычно вызывается в функции ***open*** USB драйвера и снова в функции ***disconnect***. Благодаря этим двум функциям USB драйверам не требуется держать статический массив указателей, которые хранят отдельные структуры устройства для всех текущих устройств в системе. Косвенная ссылка на информацию об устройстве позволяет любому USB драйверу поддерживать неограниченное количество устройств.

Если USB драйвер не связан с другим типом подсистемы, которая обрабатывает взаимодействие пользователя с устройством (такой, как ввод, терминал, видео и так далее), драйвер может использовать старший номер USB, чтобы использовать традиционный интерфейс символьного драйвера с пользовательским пространством. Чтобы сделать это, драйвер USB должен вызвать функцию ***usb_register_dev*** в функции ***probe***, когда он хочет зарегистрировать устройство в USB ядре. Убедитесь, что устройство и драйвер находятся в надлежащем состоянии, чтобы выполнить желание пользователя получить доступ к устройству, как только вызвана эта функция.

```
/* мы можем зарегистрировать это устройство сейчас, так как оно готово */
retval = usb_register_dev(interface, &skel_class);
if (retval) {
    /* что-то помешало зарегистрировать этот драйвер */
    err("Not able to get a minor for this device.");
    usb_set_intfdata(interface, NULL);
    goto error;
}
```

Функция ***usb_register_dev*** требует указатель на **struct usb_interface** и указатель на **struct usb_class_driver**. **struct usb_class_driver** используется для определения ряда различных параметров, о которых драйвер USB желает, чтобы их знало USB ядро при регистрации на младший номер. Эта структура состоит из следующих переменных:

char *name
: Имя, которое использует sysfs для описания устройства. Головное имя пути, если присутствует, используется только в devfs и в этой книге не рассматривается. Если ряду устройств необходимо быть в этом имени, в строке имени должны быть символы **%d**. Например, чтобы создать в devfs имя **usb/foo1** и в sysfs имя класса foo1, строка имени должна быть установлена как **usb/foo%d**.

struct file_operations *fops;
: Указатель на **struct file_operations**, которую этот драйвер определил, чтобы использовать для регистрации в качестве символьного устройства. Смотрите Главу 3 39 для получения дополнительной информации об этой структуре.

mode_t mode;

Режим для файла devfs, который будет создан для этого драйвера; иначе неиспользуемый. Типичный установкой для этой переменной будет значение **S_IRUSR** в сочетании со значением **S_IWUSR**, которыми владелец файла устройства предоставит доступ только для чтения и записи.

int minor_base;

Это начало установленного младшего диапазона для этого драйвера. Все устройства, связанные с этим драйвером, создаются с уникальными, увеличивающимися младшими номерам, начиная с этого значения. Если в ядре была включена опция конфигурации **CONFIG_USB_DYNAMIC_MINORS**, в любой момент допускается только 16 устройств, связанных с этим драйвером. Если это так, эта переменная игнорируется и все младшие номера для этого устройства распределяются по принципу "первый пришёл, первым обслужен". Рекомендуется, чтобы системы, которые имеют эту опцию разрешённой, использовали такие программы, как ***udev*** для управления узлами устройств в системе, так как статическое дерево ***/dev*** не будет работать должным образом.

После отключения USB устройства, все ресурсы, связанные с устройством должны быть очищены, если это возможно. В это время, если в функции ***probe*** для выделения младшего номера для этого USB устройства была вызвана ***usb_register_dev***, должна быть вызвана функция ***usb_deregister_dev***, чтобы вернуть USB ядру младший номер обратно.

В функции ***disconnect*** также важно извлечь из этого интерфейса все данные, которые была ранее установлены вызовом ***usb_set_intfdata***. Затем установить указатель на данные в структуре **struct usb_interface** в **NULL**, чтобы предотвратить дальнейшие ошибки при доступе к данным ненадлежащим образом:

```
static void skel_disconnect(struct usb_interface *interface)
{
    struct usb_skel *dev;
    int minor = interface->minor;

    /* предохраняем skel_open( ) от гонки со skel_disconnect( ) */
    lock_kernel( );

    dev = usb_get_intfdata(interface);
    usb_set_intfdata(interface, NULL);

    /* возвращаем наш младший номер */
    usb_deregister_dev(interface, &skel_class);

    unlock_kernel( );

    /* уменьшаем наш счётчик использования */
    kref_put(&dev->kref, skel_delete);

    info("USB Skeleton #%d now disconnected", minor);
}
```

Обратите внимание на вызов ***lock_kernel*** в предыдущем фрагменте кода. Он получает большую блокировку ядра, чтобы обратный вызов ***disconnect*** не находился в состоянии гонки с вызовом ***open*** при попытке получить указатель на правильную структуру данных интерфейса. Поскольку ***open*** вызывается с полученной большой блокировкой ядра, если ***disconnect*** также

получает эту блокировку, только одна часть драйвера может получить доступ, и затем установить указатель данных интерфейса.

Перед вызовом для устройства USB функции ***disconnect*** все urb-ы, которые в настоящее время находятся в процессе передачи для устройства, будут отменены ядром USB, поэтому драйвер не должен явно вызывать ***usb_kill_urb*** для этих urb-ов. Если драйвер пытается отправить urb в USB устройство после того, как оно было отключено вызовом ***usb_submit_urb***, отправка завершится неудачно с ошибочным значением **-EPIPE**.

Отправка и управление Urb

Когда драйвер имеет данные для передачи в USB устройство (как обычно бывает в функции записи драйвера), для передачи данных на устройство должен быть создан urb:

```
urb = usb_alloc_urb(0, GFP_KERNEL);
if (!urb) {
    retval = -ENOMEM;
    goto error;
}
```

После успешного создания urb-а, для отправки данных в устройство наиболее эффективным образом также должен быть создан буфер DMA и данные, которые переданы в драйвер, должны быть скопированы в этот буфер:

```
buf = usb_buffer_alloc(dev->udev, count, GFP_KERNEL, &urb->transfer_dma);
if (!buf) {
    retval = -ENOMEM;
    goto error;
}
if (copy_from_user(buf, user_buffer, count)) {
    retval = -EFAULT;
    goto error;
}
```

После того как данные должным образом скопированы из пространства пользователя в локальный буфер, urb должен быть правильно проинициализирован, прежде чем он может быть отправлен в ядро USB:

```
/* проинициализируем urb надлежащим образом */
usb_fill_bulk_urb(urb, dev->udev,
            usb_sndbulkpipe(dev->udev, dev->bulk_out_endpointAddr),
            buf, count, skel_write_bulk_callback, dev);
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
```

Теперь, когда должным образом выделен urb, должным образом скопированы данные и urb проинициализирован соответствующим образом, он может быть отправлен в ядро USB для передачи в устройство:

```
/* отправляем данные из поточного порта */
retval = usb_submit_urb(urb, GFP_KERNEL);
if (retval) {
    err("%s - failed submitting write urb, error %d", __FUNCTION__, retval);
    goto error;
```

```
}
```

После того, как urb успешно передан в USB устройство (или что-то произошло при передаче), USB ядром выполняется обратный вызов urb. В нашем примере мы проинициализировали urb для указания на функцию ***skel_write_bulk_callback*** и это та самая функция, которая вызывается:

```
static void skel_write_bulk_callback(struct urb *urb, struct pt_regs *regs)
{
    /* сообщения об синхронные/асинхронные разъединениях не являются ошибками
*/
    if (urb->status &&
        !(urb->status == -ENOENT ||
        urb->status == -ECONNRESET ||
        urb->status == -ESHUTDOWN)) {
        dbg("%s - nonzero write bulk status received: %d",
                __FUNCTION__, urb->status);
    }

    /* освобождаем наш выделенный буфер */
    usb_buffer_free(urb->dev, urb->transfer_buffer_length, urb-
>transfer_buffer, urb->transfer_dma);
}
```

Первое вещью, которую делает функция обратного вызова, является проверка состояния urb-а для определения, завершён ли этот urb успешно или нет. Ошибочные значения, **-ENOENT**, **-ECONNRESET** и **-ESHUTDOWN** являются не реальными ошибками передачи, а просто сообщают об условиях, сопровождающих успешную передачу. (Смотрите список возможных ошибок для urb-ов, подробно изложенный в разделе "struct urb" 321.) Затем обратный вызов освобождает выделенный буфер, который был выделен для передачи этого urb-а.

Для другого urb-а характерно быть отправленным в устройство во время выполнения функции обратного вызова urb-а. Это полезно, когда в устройство передаются потоковые данные. Помните, что обратный вызов urb-а работает в контексте прерывания, поэтому он не должен выполнять любые выделения памяти, удерживать какие-либо семафоры, или не делать чего-то другого, что может привести процесс к засыпанию. При отправке urb-а изнутри обратного вызова используйте флаг **GFP_ATOMIC**, чтобы приказать USB ядру не засыпать, если необходимо выделять новые куски памяти во время процесса отправки.

USB передачи без Urb-ов

Иногда драйвер USB не хочет проходить через все хлопоты по созданию **struct urb**, её инициализации и затем дожидаться работы функции завершения urb-а, а просто отправить или получить некоторые простые USB данные. Предоставить простой интерфейс готовы две функции.

usb_bulk_msg

usb_bulk_msg создаёт потоковый urb для USB и отправляет его в указанное устройство, затем ждёт его завершения, прежде чем вернуться к вызывающему. Она определяется как:

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe,
```

```
                    void *data, int len, int *actual_length,
                    int timeout);
```

Параметры этой функции:

struct usb_device *usb_dev
Указатель на USB устройство для передачи потокового сообщения.

unsigned int pipe
Задаёт оконечную точку USB устройства, которой должно быть отправлено это потоковое сообщение. Это значение создаётся с помощью вызова либо ***usb_sndbulkpipe***, либо ***usb_rcvbulkpipe***.

void *data
Указатель на данные для отправки в устройство, если это ВЫХОДНАЯ оконечная точка. Если это ВХОДНАЯ оконечная точка, он указывает, где эти данные должны быть размещены после чтения из устройства.

int len
Размер буфера, на который указывает параметр **data**.

int *actual_length
Указывает туда, где функция размещает фактическое количество байт, которые были либо переданы в устройство или получены от устройства, в зависимости от направления оконечной точки.

int timeout
Количество времени, в тиках, которое необходимо подождать до выхода. Если это значение равно 0, функция ждёт завершения сообщения бесконечно.

Если функция завершилась успешно, возвращается значение 0; в противном случае возвращается отрицательный номер ошибки. Этот номер ошибки соответствует номерам ошибок ранее описанных для urb-ов в разделе "struct urb" 321. При успешном выполнении параметр ***actual_length*** содержит число байт, которое было передано или получено из этого сообщения.

Ниже приведен пример использования вызова этой функции:

```
/* делаем блокирующее поточное чтение для получения данных из устройства */
retval = usb_bulk_msg(dev->udev,
usb_rcvbulkpipe(dev->udev, dev->bulk_in_endpointAddr),
                dev->bulk_in_buffer,
                min(dev->bulk_in_size, count),
                &count, HZ*10);

/* если чтение было успешным, копируем данные в пользовательское пространство
*/
if (!retval) {
    if (copy_to_user(buffer, dev->bulk_in_buffer, count))
        retval = -EFAULT;
    else
        retval = count;
}
```

Этот пример показывает простое потоковое чтение во ВХОДНОЙ оконечной точке. Если чтение прошло успешно, данные копируются в пространство пользователя. Обычно это делается для USB драйвера в функции ***read***.

Функция ***usb_bulk_msg*** не может быть вызвана из контекста прерывания или при удержании спин-блокировки. Кроме того, эта функция не может быть отменена любой другой функцией, так что будьте внимательны при её использовании; убедитесь, что функция ***disconnect*** вашего драйвера знает достаточно, чтобы дождаться завершения вызова, прежде чем позволить себе быть выгруженной из памяти.

usb_control_msg

Функция ***usb_control_msg*** работает подобно функции ***usb_bulk_msg***, за исключением того, что позволяет драйверу отправлять и получать управляющие сообщения USB:

```
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
                    __u8 request, __u8 requesttype,
                    __u16 value, __u16 index,
                    void *data, __u16 size, int timeout);
```

Параметры этой функции являются почти такими же, как у ***usb_bulk_msg***, с несколькими существенными различиями:

struct usb_device *dev
Указатель на USB устройство для передачи ему управляющего сообщения.

unsigned int pipe
Определяет оконечную точку USB устройства, которой это управляющее сообщение будет отправлено. Это значение создаётся с помощью вызова либо ***usb_sndctrlpipe***, либо ***usb_rcvctrlpipe***.

__u8 request
Значение USB запроса для управляющего сообщения.

__u8 requesttype
Значение типа запроса USB для управляющего сообщения.

__u16 value
Значение USB сообщения для управляющего сообщения.

__u16 index
Значение индекса сообщения USB для управляющего сообщения.

void *data
Указатель на данные для отправки в устройство, если это ВЫХОДНАЯ оконечная точка. Если это ВХОДНАЯ оконечная точка, он указывает, где должны быть размещены данные после чтения с устройства.

__u16 size
Размер буфера, на который указывает параметр **data**.

int timeout
Количество времени, в тиках, которое необходимо подождать до выхода. Если это значение равно 0, функция ждёт завершения сообщения бесконечно.

Если функция завершилась успешно, она возвращает количество байт, которые были переданы в или из устройства. Если она завершилась не успешно, возвращается отрицательный номер ошибки.

Все параметры, **request**, **requesttype**, **value** и **index**, напрямую привязаны со спецификацией USB в том, каким образом определяются управляющие сообщения USB. Для получения дополнительной информации о допустимых значениях этих параметров и как они используются, смотрите спецификации USB в Главе 13 344.

Подобно функции ***usb_bulk_msg***, функция ***usb_control_msg*** не может быть вызвана из контекста прерывания или при удержании спин-блокировки. Также, эта функция не может быть отменена любой другой функцией, поэтому будьте осторожны при её использовании; убедитесь, что функция ***disconnect*** вашего драйвера знает достаточно, чтобы дождаться завершения вызова, прежде чем позволить себе быть выгруженной из памяти.

Другие функции для работы с данными USB

Для получения стандартной информации из всех USB устройств может быть использован ряд вспомогательных функции в ядре USB. Эти функции не могут быть названы в контексте прерывания или при удержании спин-блокировки.

Функция ***usb_get_descriptor*** возвращает указанный дескриптор USB из указанного устройства. Функция определяется как:

```
int usb_get_descriptor(struct usb_device *dev, unsigned char type,
                       unsigned char index, void *buf, int size);
```

Эта функция может быть использована USB драйвером для извлечения из структуры **struct usb_device** любого из дескрипторов устройства, которых уже нет в существующих структурах **struct usb_device** и **struct usb_interface**, таких как аудио дескрипторы или другой зависящей от класса информации. Параметры функции:

struct usb_device *usb_dev
Указатель на USB устройство, из которого должно быть извлечён дескриптор.

unsigned char type
Тип дескриптора. Этот тип описан в спецификации USB и может быть одним из следующих типов:
USB_DT_DEVICE
USB_DT_CONFIG
USB_DT_STRING
USB_DT_INTERFACE
USB_DT_ENDPOINT
USB_DT_DEVICE_QUALIFIER
USB_DT_OTHER_SPEED_CONFIG
USB_DT_INTERFACE_POWER
USB_DT_OTG

USB_DT_DEBUG
USB_DT_INTERFACE_ASSOCIATION
USB_DT_CS_DEVICE
USB_DT_CS_CONFIG
USB_DT_CS_STRING
USB_DT_CS_INTERFACE
USB_DT_CS_ENDPOINT

unsigned char index
Количество дескрипторов, которые должны быть извлечены из устройства.

void *buf
Указатель на буфер, в который вы копируете дескриптор.

int size
Размер памяти, на которую указывает переменная **buf**.

Если эта функция завершилась успешно, она возвращает количество байтов, считанных из устройства. В противном случае, она возвращает отрицательный номер ошибки, возвращённый нижележащим вызовом ***usb_control_msg***, который выполняет эта функция.

Одним из наиболее часто используемых вызовов ***usb_get_descriptor*** является извлечение строки из устройства USB. Так как это происходит довольно часто, для этого есть вспомогательная функция, называемая ***usb_get_string***:

```
int usb_get_string(struct usb_device *dev, unsigned short langid,
                   unsigned char index, void *buf, int size);
```

В случае успеха эта функция возвращает количество байт, полученных устройством для строки. В противном случае, она возвращает отрицательный номер ошибки, возвращённый нижележащим вызовом ***usb_control_msg***, который выполняет эта функция.

Если эта функция завершилась успешно, она возвращает строку в кодировке формата UTF-16LE (Unicode, 16 бит на символ, с порядком байтов little-endian, сначала младший) в буфере, на который указывает параметр **buf**. Поскольку этот формат обычно не очень удобен, есть ещё одна функция, называемая ***usb_string***, которая возвращает строку, которая считывается из USB устройства и уже преобразована в формат строки ISO 8859-1. Этот набор символов является 8-ми разрядным подмножеством Unicode и самый распространённый формат для строк в английском и других западноевропейских языках. Так как это обычно тот формат, в котором USB устройства имеют строки, рекомендуется, чтобы вместо функции ***usb_get_string*** использовалась функция ***usb_string***.

Краткая справка

В этом разделе просуммированы символы, введённые в этой главе:

#include <linux/usb.h>
Заголовок файла, где находится всё, связанное с USB. Он должен подключаться всеми драйверами USB устройств.
struct usb_driver;
Структура, которая описывает USB драйвер.
struct usb_device_id;

Структура, которая описывает типы USB устройств, которые поддерживает драйвер.

int usb_register(struct usb_driver *d);
void usb_deregister(struct usb_driver *d);

Функции, которые используются для регистрации и отмены регистрации драйвера USB из USB ядра.

struct usb_device *interface_to_usbdev(struct usb_interface *intf);

Извлекает управляющую **struct usb_device *** из **struct usb_interface ***.

struct usb_device;

Структура, которая управляет целиком USB устройством.

struct usb_interface;

Главная структура USB устройства, которую используют все драйверы USB для взаимодействия с ядром USB.

void usb_set_intfdata(struct usb_interface *intf, void *data);
void *usb_get_intfdata(struct usb_interface *intf);

Функции для установки и получения доступа к закрытому разделу указателей данных внутри **struct usb_interface**.

struct usb_class_driver;

Структура, которая описывает USB драйвер, который хочет использовать старший номер USB для связи с программами пользовательского пространства.

int usb_register_dev(struct usb_interface *intf, struct usb_class_driver *class_driver);
void usb_deregister_dev(struct usb_interface *intf, struct usb_class_driver *class_driver);

Функции, которые используются для регистрации и отмены регистрации указанной структуры **struct usb_interface *** со структурой **struct usb_class_driver ***.

struct urb;

Структура, которая описывает передачу данных USB.

struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
void usb_free_urb(struct urb *urb);

Функции, которые используются для создания и уничтожения **struct usb urb** *.

int usb_submit_urb(struct urb *urb, int mem_flags);
int usb_kill_urb(struct urb *urb);
int usb_unlink_urb(struct urb *urb);

Функции, которые используются для запуска и остановки передачи данных USB.

void usb_fill_int_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer,
int buffer_length, usb_complete_t complete, void *context, int interval);
void usb_fill_bulk_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe, void *transfer_buffer,
int buffer_length, usb_complete_t complete, void *context);
void usb_fill_control_urb(struct urb *urb, struct usb_device *dev, unsigned int pipe,
unsigned char *setup_packet, void *transfer_buffer, int buffer_ length,
usb_complete_t complete, void *context);

Функции, которые используются для инициализации **struct urb** до её отправки в USB ядро.

int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, int *actual_length, int timeout);
int usb_control_msg(struct usb_device *dev, unsigned int pipe,
__u8 request, __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout);

Функции, использующиеся для передачи или приёма данных USB, не прибегая к использованию **struct urb**.

Глава 14, Модель устройства в Linux

Одной из заявленных целей для цикла разработки версии 2.5 было создание для ядра унифицированной модели устройства. Предыдущие ядра не имели единой структуры данных, к которой они могли обращаться для получения информации о том, как система собрана воедино. Вопреки такой нехватки информации, в течение некоторого времени всё работало хорошо. Требования новейших систем, с их более сложными топологиями и необходимостью в поддержке таких функций, как управление питанием, тем не менее, делают ясным, что необходима общая абстракция, описывающая структуру системы.

Модель устройства версии 2.6 обеспечивает такую абстракцию. Сейчас она широко используется в ядре для поддержки широкого спектра задач, включая:

Управление питанием и выключение системы

Это требует понимания структуры системы. Например, адаптер USB узла не может быть закрыт до закрытия всех устройств, подключенных к этому адаптеру. Модель устройства позволяет обход системного оборудования в правильном порядке.

Взаимодействие с пользовательским пространством

Реализация виртуальной файловой системы sysfs тесно связана с моделью устройства и выставляет структуру, предоставленную ей. Предоставление информации о системе для пользовательского пространства и ручек управления для изменения рабочих параметров всё чаще выполняется через sysfs и, следовательно, через модель устройства.

Устройства, подключаемые без выключения системы

Компьютерное оборудование становится все более динамичным; периферийные устройства могут приходить и уходить по прихоти пользователя. Механизм горячего подключения, используемый в ядре для обработки и (особенно) взаимодействия с пользовательским пространством при подключении и отключении устройств, управляется с помощью модели устройства.

Классы устройств

Многие части системы мало заинтересованы в знании того, как устройства подключаются, но им необходимо знать, какие устройства доступны. Модель устройства включает в себя механизм для присвоения устройствам ***классов***, которые описывают эти устройства на более высоком, функциональном уровне и позволяют им быть

обнаруженными из пространства пользователя.

Жизненные циклы объектов

Многие функции, описанные выше, имеющие поддержку горячего подключения и sysfs, осложняют создание и манипулирование объектами, созданными в ядре. Реализация модели устройства требует создания набора механизмов, чтобы иметь дело с жизненным циклом объектов, их взаимоотношениями и их представлением в пространстве пользователя.

Модель устройства Linux представляет собой сложную структуру данных. Рассмотрим, например, Рисунок 14-1, который показывает (в упрощённой форме) небольшую часть структуры модели устройства, связанной с USB мышью. Вниз по центру диаграммы мы видим часть дерева ядра "устройств", которое показывает, как мышь подключена к системе. Дерево "шины" прослеживает, что подключено к каждой шине, в то время как поддерево "классы" озабочено функциями, предоставляемыми устройствами, независимо от того, как они подключены. Дерево модели устройства даже на простой системе содержит сотни узлов, похожих на показанные на диаграмме; оно является сложной структурой данных для визуализации в целом.

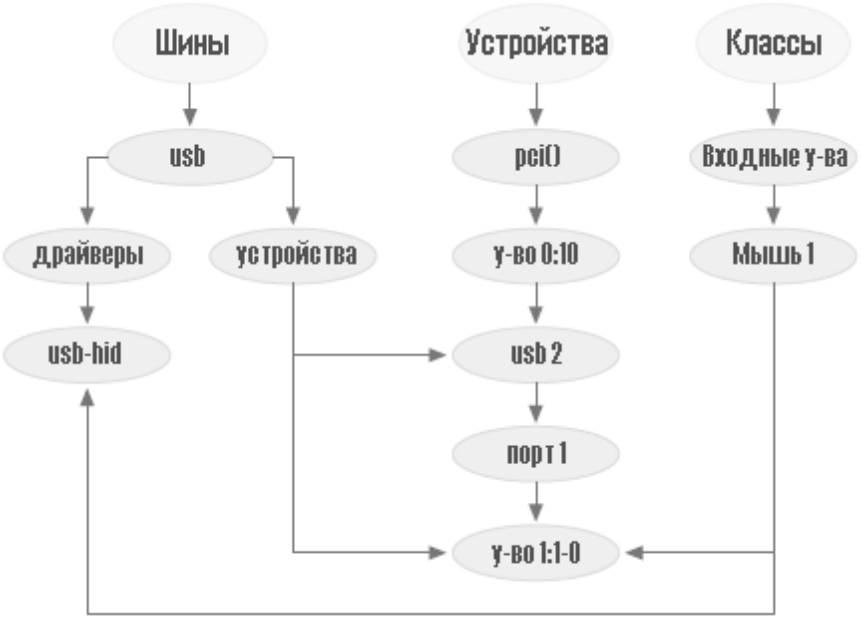


Рисунок 14-1. Небольшая часть модели устройства

По большей части, код модели устройства в Linux заботится обо всех этих деталях без навязывания себя авторам драйверов. Оно в основном находится на заднем плане; прямым взаимодействием с моделью устройства, как правило, занимается логика шинного уровня и другие разные подсистемы ядра. Как результат, многие авторы драйверов могут полностью игнорировать модель устройства и доверить ей позаботиться о себе самой.

Однако, есть моменты, когда хорошо иметь понимание модели устройства. Есть моменты, когда модель устройства "выходит наружу" из-под других слоёв; например, обычный код DMA (который мы встретим в Главе 15 395) работает со **struct device**. Вы можете захотеть использовать некоторые из возможностей модели устройства, таких как подсчёт ссылок и соответствующие возможности, предоставляемые **kobjects**. Взаимодействие с пользовательским пространством через sysfs также является функцией модели устройства;

эта глава рассказывает, как работает такое взаимодействие.

Начнём, однако, с презентации модели устройства снизу вверх. Сложность модели устройства делает её труднопонимаемой при начале просмотра с высоких уровней. Мы надеемся, что показывая, как работают низкоуровневые компоненты устройства, мы сможем подготовить вас к постижению, как используются эти компоненты для построения большой структуры.

Многие читатели могут рассматривать эту главу как дополнительный материал, которые не требует прочтения до конца в первый раз. Тех, кто заинтересовался работой модели устройства Linux призываем, однако, двигаться вперёд, так как мы углубляемся в низкоуровневые детали.

Kobject-ы, Kset-ы и Subsystem-ы

kobject является основополагающей структурой, которая содержит модель устройства всю вместе. Она первоначально была задумана как простой счётчик ссылок, но её обязанности выросли с течением времени, так же как и её поля. Задачи, решаемые **struct kobject** и её поддерживающим кодом, теперь включают в себя:

Подсчёт ссылок объектов

Часто, когда создаётся объект ядра, нет никакого способа узнать, насколько долго он будет существовать. Одним из способов отслеживания жизненного цикла таких объектов является использование подсчёта ссылок. Если нет кода в ядре, удерживающего ссылку на данный объект, этот объект закончил свою полезную жизнь и может быть удалён.

Представление в sysfs

Каждый объект, который появляется в sysfs, имеет под собой **kobject**, который взаимодействует с ядром для создания его видимого представления.

Связующий элемент структуры данных

Модель устройства является, в целом, чрезвычайно сложной структурой данных, состоящей из множества иерархий с многочисленными связями между ними. **kobject** реализует эту структуру и удерживает всё вместе.

Обработка событий горячего подключения

Подсистема **kobject** обрабатывает поток событий, которые уведомляют пользовательское пространство о входящем и уходящем из системы оборудовании.

Как можно было бы заключить из предыдущего списка, **kobject** представляет собой сложную структуру. Это было бы правильным. Однако, рассматривая по одному кусочку за раз, можно понять эту структуру и как она работает.

Основы kobject

kobject имеет тип **struct kobject**; он определён в ***<linux/kobject.h>***. Этот файл подключает также декларацию ряда других структур, связанных с kobject-ами и, конечно, длинный список функций для работы с ними.

Внедрение kobject-ов

Прежде чем мы углубимся в детали, стоит уделить время, чтобы понять, как используются kobject-ы. Если вы посмотрите назад на список функций, поддерживаемых kobject-ами, то увидите, что все они являются услугами, выполняемыми от имени других объектов. Другими словами, kobject мало интересен сам по себе; он существует только для того, чтобы связать высокоуровневые объекты в модель устройства.

Таким образом, редко (даже неизвестно), чтобы код ядра создавал автономный kobject; Вместо этого, kobject-ы используются для управления доступом к большому, зависимому от области определения объекту. С этой целью kobject-ы находятся внедрёнными в другие структуры. Если вы привыкли думать о вещах в объектно-ориентированных терминах, kobject-ы можно рассматривать как высокоуровневый, абстрактный класс, от которого порождаются все остальные классы. kobject реализует набор возможностей, которые не особенно полезны сами по себе, но которые приятно иметь в других объектах. Язык Си не позволяет прямое выражение наследования, поэтому должны быть использованы другие методы, такие как вложение одной структуры в другую.

В качестве примера давайте посмотрим назад на **struct cdev**, с которой мы столкнулись в Главе 3 39. Эта структура, найденная в ядре версии 2.6.10, выглядит следующим образом:

```
struct cdev {
    struct kobject kobj;
    struct module *owner;
    struct file_operations *ops;
    struct list_head list;
    dev_t dev;
    unsigned int count;
};
```

Как мы видим, эта структура **cdev** имеет внедрённый в неё kobject. Если у вас есть одна из этих структур, получение встроенного в неё kobject - это всего лишь вопрос использования поля **kobj**. Однако, код, который работает с kobject-ами, часто имеет противоположную проблему: как имея указатель на **struct kobject** получить указатель на содержащую его структуру? Вам следует избегать трюков (такого, как предположение, что kobject находится в начале структуры), а вместо этого использовать макрос ***container_of*** (введённый в разделе "Метод open" 54 в Главе 3 39). Таким образом, способом преобразования указателя на **struct kobject**, названного **kp** и встроенного в структуру **cdev**, будет:

```
struct cdev *device = container_of(kp, struct cdev, kobj);
```

Программисты часто определяют простой макрос для "обратного приведения типа" указателей kobject к содержащему их типу.

Инициализация kobject

В этой книге представлены несколько типов с простыми механизмами для инициализации на этапе компиляции или во время выполнения. Инициализация kobject-а является немного более сложной, особенно когда используются все его функции. Однако, независимо от того, каким образом используется kobject, должны быть выполнены несколько шагов.

Первым из них является просто установка всего kobject в 0, как правило, вызовом ***memset***. Часто такая инициализация происходит как часть обнуления структуры в которую встроен kobject. Отказ от обнуления kobject часто приводит в конечном счёте к очень странным сбоям; это не тот шаг, который вы захотите пропустить.

Следующий шаг заключается в создании некоторых внутренних полей вызовом ***kobject_init ()***:

```
void kobject_init(struct kobject *kobj);
```

Среди прочего, ***kobject_init*** устанавливает счётчик ссылок kobject-а в единицу. Однако, вызова ***kobject_init*** не достаточно. Пользователи kobject-а должны, как минимум, установить имя kobject; это имя, которое используется в записях sysfs. Если вы пороетесь в исходном коде ядра, то сможете найти код, который копирует строку непосредственно в поле **name** kobject-а, но такого подхода следует избегать. Вместо этого используйте:

```
int kobject_set_name(struct kobject *kobj, const char *format, ...);
```

Эта функция принимает переменный список аргументов в стиле ***printk***. Верите или нет, на самом деле возможно, что эта операция закончится неудачей (она может попытаться выделить память); добросовестный код должен проверять возвращаемое значение и реагировать соответствующим образом.

Другими полями kobject-а, которые должны быть установлены создателем прямо или косвенно, являются **ktype**, **kset** и **parent**. Мы узнаем о них далее в этой главе.

Манипуляция счётчиком ссылок

Одной из ключевых функций kobject-а является обслуживание счётчика ссылок для объекта, в который он внедрён. Пока ссылки на объект существуют, объект (и код, который его содержит) должен продолжать существование. Низкоуровневыми функциями для манипулирования счётчиками ссылок kobject-а являются:

```
struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
```

Вызов kobject_get увеличивает счётчик ссылок kobject-а и возвращает указатель на kobject.
Старая версия абзаца:
Успешный вызов *kobject_get* увеличивает счётчик ссылок kobject-а и возвращает указатель на kobject. Если, однако, kobject уже находится в процессе уничтожения, операция заканчивается неудачно и *kobject_get* возвращает NULL. Этот код возврата всегда должен быть проверен, или это может привести к бесконечным неприятным состояниям гонок.
Видимо, также следует исправить и приведённый ниже пример?

При освобождении ссылки вызов ***kobject_put*** уменьшает счётчик ссылок и, возможно, освобождает объект. Помните, что ***kobject_init*** устанавливает счётчик ссылок в единицу; таким образом, когда вы создаёте kobject, вы должны убедиться, что соответствующий вызов ***kobject_put*** производится, когда эта начальная ссылка уже больше не требуется.

Отметим, что во многих случаях счётчика ссылок в kobject-е самого по себе не может быть достаточно, чтобы предотвратить состояния гонок. Например, существование kobject-а (и содержащей его структуры) может, конечно, требовать дальнейшего существования модуля,

который создал этот kobject. Он бы не допустил выгрузку этого модуля, пока kobject всё ещё где-то используется. Вот почему структура **cdev**, которую мы видели выше, содержит указатель **struct module**. Подсчёт ссылок на **struct cdev** осуществляется следующим образом:

```
struct kobject *cdev_get(struct cdev *p)
{
    struct module *owner = p->owner;
    struct kobject *kobj;

    if (owner && !try_module_get(owner))
        return NULL;
    kobj = kobject_get(&p->kobj);
    if (!kobj)
        module_put(owner);
    return kobj;
}
```

Создание ссылки на структуру **cdev** требует также создания ссылки на модуль, которому она принадлежит. Таким образом, ***cdev_get*** использует ***try_module_get*** для попытки увеличения счётчика использования модуля. Если эта операция успешна, ***kobject_get*** также используется для увеличения счётчика ссылок kobject-а. Эта операция может быть, конечно, неудачной, так что код проверяет возвращаемое значения ***kobject_get*** и освобождает его ссылку на модуль, если что-то не получилось.

Функции освобождения и типы kobject

Одной важной вещью, всё ещё пропущенной в ходе обсуждения, является то, что происходит с kobject-ом, когда его счётчик ссылок становится 0. Код, который создал kobject, обычно не знает, когда это произойдёт; если бы знал, во-первых, не было бы никакого смысла в использовании счётчика ссылок. Даже жизненные циклы предсказуемых объектов становятся более сложными, когда они включены в sysfs; программы пользовательского пространства могут хранить ссылку на kobject (на поддержание открытым одного из связанных с ним файлов в sysfs) произвольный период времени.

Конечным результатом является то, что структура, защищённая kobject-ом, не может быть освобождена в какой-либо одной, предсказуемой точке в жизненном цикле драйвера, а только в коде, который должен быть готов к запуску в любом момент, когда счётчик ссылок kobject-а стал 0. Счётчик ссылок не находится под прямым контролем кода, который создал этот kobject. Так что код должен получить асинхронное уведомление всякий раз, когда последняя ссылка на один из его kobject-ов уходит.

Такое уведомление осуществляется посредством метода ***release*** kobject-а. Как правило, этот метод имеет такую форму:

```
void my_object_release(struct kobject *kobj)
{
    struct my_object *mine = container_of(kobj, struct my_object, kobj);
    /* Выполнить любую дополнительную очистку в этом объекте, затем... */
    kfree(mine);
}
```

Нельзя недооценить один важный момент: каждый kobject должен иметь метод ***release*** и

kobject должен сохраняться (в целостном состоянии), пока вызван этот метод. Если эти ограничения не выполняются, код имеет изъяны. Он рискует освободить объект, когда он всё ещё используется, или это может привести к ошибке при освобождении объекта после возвращения последней ссылки.

Интересно, что метод ***release*** не хранится в самом kobject-е; вместо этого он связан с типом структуры, которая содержит kobject. Этот тип отслеживается структурой типа **struct kobj_type**, часто называемой просто "ktype". Эта структура выглядит следующим образом:

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

Поле **release** в **struct kobj_type** это, конечно, указатель на метод ***release*** для этого типа kobject-а. Мы вернёмся к двум другим полям (**sysfs_ops** и **default_attrs**) позже в этой главе.

Каждый kobject должен иметь связанную с ним структуру **kobj_type**. Как ни странно, указатель на эту структуру может быть найден в двух разных местах. Структура kobject-а сама содержит поле (называемое **ktype**), которое может содержать этот указатель. Если, однако, этот kobject является членом kset-а, вместо этого указатель **kobj_type** обеспечивается kset-ом. (Мы будем рассматривать kset-ы в следующем разделе). Тем временем, макрос:

```
struct kobj_type *get_ktype(struct kobject *kobj);
```

находит указатель kobj_type для данного kobject-а.

Иерархии kobject-а, kset-ы и subsystem-ы

Структура kobject-а часто используется, чтобы связать объекты воедино в иерархическую структуру, которая соответствует структуре моделируемой подсистемы. Есть два отдельных механизма для такого связывания: указатель **parent** и kset-ы.

Поле **parent (родитель)** в **struct kobject** является указателем на другой kobject, который представляет следующий более высокий уровень в иерархии. Если, например, kobject представляет USB устройство, его указатель **parent** может указывать на объект, представляющий концентратор, в который подключено устройство .

Основным использованием для указателя **parent** является позиционирование объекта в иерархии sysfs. Мы рассмотрим, как это работает в разделе "Низкоуровневые операции в sysfs" 356.

Kset-ы

Во многих отношениях kset выглядит как расширение структуры **kobj_type**; kset является коллекцией kobject-ов, встроенных в структуры того же типа. Однако, если **struct kobj_type** имеет отношение к типу объекта, **struct kset** связана с агрегацией и сбором. Эти два понятия были разделены, чтобы объекты идентичных типов могли появляться в различных наборах.

Таким образом, основной функцией kset-а является агрегация; она может рассматриваться

как контейнерный класс верхнего уровня для kobject-ов. В самом деле, каждый kset содержит внутри свои собственные kobject и он может, во многом, обрабатываться теми же способами, как и kobject. Стоит отметить, что kset-ы всегда представлены в sysfs; после того, как kset был создан и добавлен в систему, он будет присутствовать в каталоге sysfs. Kobject-ы не обязательно показаны в sysfs, но каждый kobject, являющийся членом kset-а, там представлен. Добавление kobject-а к kset-у обычно делается при создании объекта; это двух этапный процесс. Поле **kset** kobject-а должно указывать на интересующий kset; затем этот kobject должен быть передан в:

```
int kobject_add(struct kobject *kobj);
```

Как всегда, программисты должны понимать, что эта функция может потерпеть неудачу (в этом случае она возвращает отрицательный код ошибки) и реагировать соответствующим образом. Существует удобная функция, предоставляемая ядром:

```
extern int kobject_register(struct kobject *kobj);
```

Эта функция - просто комбинация ***kobject_init*** и ***kobject_add***.

Когда kobject передается в ***kobject_add***, счётчик ссылок увеличивается. Содержимое в kset, в конце концов, является ссылкой на объект. В определенный момент kobject, вероятно, должен быть удалён из kset для очистки этой ссылки; это делается следующим образом:

```
void kobject_del(struct kobject *kobj);
```

Существует также функция ***kobject_unregister***, которая представляет собой комбинацию ***kobject_del*** и ***kobject_put***.

kset держит своих детей в стандартном связном списке ядра. Почти во всех случаях содержащиеся kobject-ы также имеют указатели на kset (или, точнее, его встроенный kobject) в полях **parent** (родитель). Таким образом, как правило, kset и его kobject-ы выглядят как что-то похожее на то, что вы видите на Рисунке 14-2. Имейте в виду, что:

- Все содержащиеся на диаграмме kobject-ы фактически внедрены в некоторый другой тип, возможно, даже другие kset-ы.
- Не обязательно, чтобы родитель kobject-а содержал kset (хотя любая другая организация была бы странной и редкой).

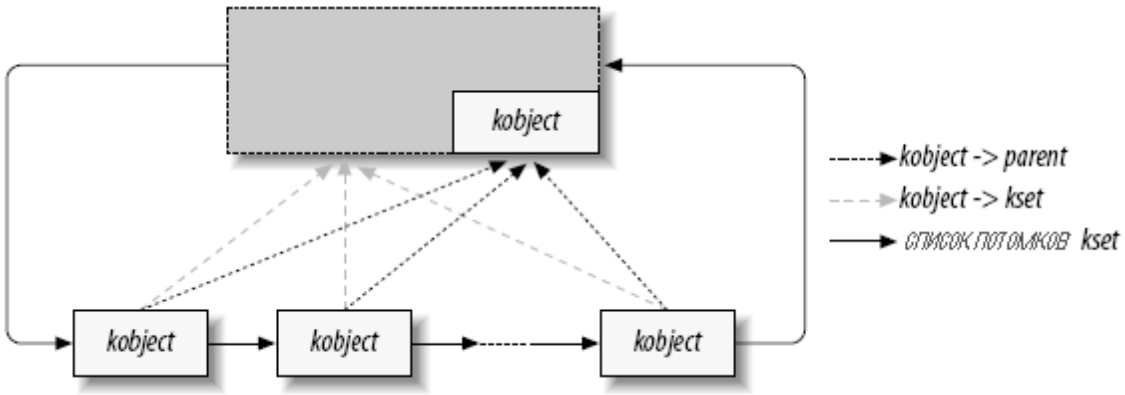


Рисунок 14-2. Простая иерархия kset

Операции с kset-ами

Для инициализации и настройки kset-ы имеют интерфейс, очень похожий на такой у kobject-ов.

Существуют следующие функции:

```
void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
```

По большей части эти функции просто вызывают аналогичную функции ***kobject_*** у встроенного в kset kobject-а.

С управлением счётчиками ссылок у kset-ов ситуация примерно такая же:

```
struct kset *kset_get(struct kset *kset);
void kset_put(struct kset *kset);
```

kset также имеет имя, которое сохраняется во встроенном kobject-е. Таким образом, если у вас есть kset, названный **my_set**, вы должны установить его имя так:

```
kobject_set_name(&my_set->kobj, "The name");
```

Kset-ы имеют также указатель (в поле **ktype**) на структуру **kobj_type**, описывающую kobject-ы, которые он содержит. Этому типу отдаётся предпочтение перед полем **ktype** в самом kobject-е. Как результат, при обычном использовании поле **ktype** в **struct kobject** остаётся NULL, потому что на самом деле используются аналогичное поле внутри этого kset.

Наконец, kset содержит указатель на subsystem (подсистему) (называемый **subsys**). Так что пришло время поговорить о подсистемах.

Subsystem-ы

Subsystem (подсистема) является представлением для высокоуровневой части ядра в целом. Подсистемы обычно (но не всегда) отображаются на верху иерархии sysfs. Несколько примеров подсистем в ядре включают **block_subsys** (***/sys/block***, для блочных устройств), **devices_subsys** (***/sys/devices***, основная иерархия устройств), а также специальные подсистемы для каждого типа шины, известной ядру. Автору драйвера почти никогда не требуется создавать новую подсистему; если вы чувствуете соблазн сделать это, подумайте ещё раз. Что вы, вероятно, захотите в конце концов, так это добавить новый класс, как это описано в разделе "Классы".

Подсистема представляет собой простую структуру:

```
struct subsystem {
    struct kset kset;
    struct rw_semaphore rwsem;
};
```

Подсистема, таким образом, является на самом деле просто обёрткой kset-а с

добавленным в неё семафором.

Каждый kset должен принадлежать к подсистеме. Членство в подсистеме помогает установить позицию kset-а в иерархии, но, что более важно, **rwsem** семафор подсистемы используется для организации последовательного доступа к внутреннему связному списку kset-а. Это членство представлено указателем **subsys** в **struct kset**. Таким образом, можно найти каждую содержащую kset подсистему из структуры kset-а, но не возможно найти несколько kset-ов, содержащейся в подсистеме, непосредственно из структуры подсистемы.

Подсистемы часто декларируется специальным макросом:

```
decl_subsys(name, struct kobj_type *type, struct kset_hotplug_ops
*hotplug_ops);
```

Этот макрос создаёт **struct subsystem** с именем, сформированным из переданного в макрос **name** с добавленным к нему **_subsys**. Макрос также инициализирует внутренний kset с заданным **type** и **hotplug_ops**. (Мы обсудим операции hotplug (горячего подключения) позднее в этом главы).

Подсистемы имеют обычный список функции установки и демонтажа:

```
void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)
void subsys_put(struct subsystem *subsys);
```

Большинство из этих операций воздействуют только на kset подсистемы.

Низкоуровневые операции в sysfs

Kobject-ы являются механизмом, стоящим за виртуальной файловой системой sysfs. Для каждого каталога, находящегося в sysfs, существует kobject, скрывающийся где-то в ядре. Каждый kobject интересен также экспортом одного или более ***атрибутов***, которые появляются в каталоге sysfs kobject-а как файлы, содержащие генерируемую ядром информацию. В этом разделе рассматривается, как kobject-ы и sysfs взаимодействуют на низком уровне.

Код, который работает с sysfs, должен подключать ***<linux/sysfs.h>***.

Появление kobject-а в sysfs - это просто вопрос вызова ***kobject_add***. Мы уже видели эту функцию, как способ добавить kobject к kset-у; создание записей в sysfs также является частью её работы. Есть несколько моментов, которые стоит знать, как создаётся запись в sysfs:

- Записями sysfs для kobject-ов всегда являются каталоги, поэтому вызов ***kobject_add*** в результате создаёт каталог в sysfs. Обычно это каталог, содержащий один или более атрибутов; в ближайшее время мы увидим, как определяются эти атрибуты.
- Имя, присвоенное kobject-у (с помощью ***kobject_set_name***), является именем, используемым для каталога в sysfs. Таким образом, kobject-ы, которые появляются в одной части иерархии sysfs, должны иметь уникальные имена. Имена, данные kobject-ам, должны быть также разумными именами файлов: они не могут содержать символ "косая

черта" (slash), а также настоятельно не рекомендуется использование пробелов.
- Запись sysfs находится в каталоге, соответствующем указателю **parent** kobject-a. Если при вызове ***kobject_add*** **parent** является **NULL**, он устанавливается на kobject, внедрённый в новый kset kobject-a; таким образом, иерархия sysfs обычно соответствует внутренней иерархии, созданной kset-ми. Если и **parent** и **kset** установлены в NULL, каталог sysfs создаётся на высшем уровне, который почти наверняка не то, что вы хотите.

Используя механизмы, которые мы описывали до сих пор, мы можем использовать kobject для создания пустого каталога в sysfs. Как правило, вы хотите сделать что-то более интересное, чем это, так что пришло время посмотреть на реализацию атрибутов.

Атрибуты по умолчанию

При создании каждого kobject-a даётся набор атрибутов по умолчанию. Эти атрибуты задаются структурой **kobj_type**. Напомним, что эта структура выглядит следующим образом:

```
struct kobj_type {
    void (*release)(struct kobject *);
    struct sysfs_ops *sysfs_ops;
    struct attribute **default_attrs;
};
```

Поле **default_attrs** перечисляет атрибуты, которые будут созданы для каждого kobject-a этого типа, а **sysfs_ops** предоставляет методы для реализации этих атрибутов. Мы начнём с **default_attrs**, который указывает на массив указателей на структуры **attribute**:

```
struct attribute {
    char *name;
    struct module *owner;
    mode_t mode;
};
```

В этой структуре **name** является именем атрибута (как он показывается в каталог sysfs kobject-a), **owner** является указателем на модуль (если таковой имеется), который отвечает за реализацию этого атрибута, и **mode** (режим) представляет собой защитные биты, которые будут применяться к этому атрибуту. Режим, как правило, **S_IRUGO** для атрибутов, предназначенных только для чтения; если атрибут доступен для записи, вы можете добавить в него **S_IWUSR**, чтобы дать доступ на запись только суперпользователю (макросы для режимов определены в ***<linux/stat.h>***). Последняя запись в списке **default_attrs** должна быть заполнена нулями.

Массив **default_attrs** говорит, какие атрибуты есть, но не говорит sysfs, как на самом деле реализовать эти атрибуты. Эта задача возложена на поле **kobj_type->sysfs_ops**, которое указывает на структуру, определённую следующим образом:

```
struct sysfs_ops {
    ssize_t (*show)(struct kobject *kobj, struct attribute *attr,
                    char *buffer);
    ssize_t (*store)(struct kobject *kobj, struct attribute *attr,
                     const char *buffer, size_t size);
};
```

Всякий раз, когда из пользовательского пространства читается атрибут, вызывается метод ***show*** с указателем на kobject и соответствующую структуру **attribute**. Этот метод должен закодировать значение данного атрибута в буфер, будучи уверенным, что его не переполнил (он имеет **PAGE_SIZE** байт), и вернуть фактическую длину возвращаемых данных. Соглашения для sysfs заявляют, что каждый атрибут должен содержать единственное, доступное для человеческого понимания значение; если у вас для возвращения есть много информации, вы можете захотеть рассмотреть возможность её разделения на несколько атрибутов.

Для всех атрибутов, связанных с данным kobject-ом, используется тот же самый метод ***show***. Указатель **attr**, передаваемый в функцию, может быть использован для определения, какой атрибут был запрошен. Те же методы ***show*** включают в себя наборы тестов по имени атрибута. Другие реализации внедряют структуру **attribute** внутрь другой структуры, которая содержит информацию, необходимую для возврата значения атрибута; в этом случае для получения указателя на вложенную структуру внутри метода ***show*** может быть использована ***container_of***.

Метод ***store*** аналогичен; он должен декодировать данные, хранящиеся в **buffer** (**size** содержит длину этих данных, которая не превышает **PAGE_SIZE**), сохранить и отреагировать на новое значение любым имеющим смысл способом и вернуть количество фактически декодированных байт. Метод ***store*** может быть вызван только если разрешения атрибута позволяют запись. При написании метода ***store*** никогда не забывайте, что вы получаете произвольную информацию из пространства пользователя; вы должны проверять её очень внимательно, прежде чем в ответ выполнить какие-либо действия. Если входные данные не соответствуют ожиданиям, возвращение отрицательного значения ошибки лучше, чем возможность делать что-нибудь нежелательное и непоправимое. Если ваше устройство экспортирует атрибут **self_destruct**, вы должны требовать, чтобы для вызова такой функциональности была написана определённая строка; неожиданная, случайная запись должна приносить только ошибки.

Нестандартные атрибуты

Во многих случаях поле **default_attrs** типа kobject описывает все атрибуты, которые kobject будет когда-либо иметь. Но при разработке это не является ограничением; по желанию атрибуты могут быть добавлены и удалены из kobject-ов. Если вы желаете добавить новый атрибут в каталоге kobject-а в sysfs, просто заполните структуру **attribute** и передайте её в:

```
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
```

Если всё идёт хорошо, создаётся файл с именем, заданным в структуре **attribute** и возвращаемым значением является 0; в противном случае, возвращается обычный отрицательный код ошибки.

Обратите внимание, что для реализации операций на новом атрибуте вызываются те же самые функции ***show( )*** и ***store( )***. Прежде чем добавить новый, нестандартный атрибут к kobject-у, вы должны принять все необходимые меры для обеспечения того, чтобы эти функции знали, как реализовать этот атрибут.

Чтобы удалить атрибуты, вызывайте:

```
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
```

После этого вызова атрибут больше не появляется в записи kobject-а в sysfs. Знайте, однако, что процесс в пользовательском пространстве мог бы иметь открытый дескриптор файла для этого атрибута и что после удаления атрибута вызовы ***show*** и ***store*** по-прежнему возможны.

Двоичные атрибуты

Соглашения sysfs призывают, чтобы все атрибуты содержали единственное значение в удобном для восприятия человеком текстовом формате. Тем не менее, есть отдельная, редкая необходимость для создания атрибутов, которые могут обрабатывать большие куски двоичных данных. Такая необходимость возникает только в случаях, когда между пространством пользователем и устройством данные должны быть переданы нетронутыми. Например, эта функция требуется для загрузки в устройство программы (прошивка). Когда в системе встречается такое устройство, может быть запущена программа пользовательского пространства (с помощью механизма горячего подключения); эта программа затем передаёт код программы в ядро через бинарный атрибут в sysfs, как это показано в разделе "Интерфейс ядра для встроенного программного обеспечения" 390.

Двоичные атрибуты описываются структурой **bin_attribute**:

```
struct bin_attribute {
    struct attribute attr;
    size_t size;
    ssize_t (*read)(struct kobject *kobj, char *buffer, loff_t pos, size_t
size);
    ssize_t (*write)(struct kobject *kobj, char *buffer, loff_t pos, size_t
size);
};
```

Здесь, **attr** является структурой **attribute**, задающей имя, владельца и разрешения для бинарного атрибута и **size** является максимальным размером бинарного атрибута (или 0, если максимума не существует). Методы ***read*** и ***write*** работают по аналогии с обычными эквивалентами символьного драйвера; они могут быть вызваны множество раз в течение одной загрузки с максимальным размером данных в каждом вызове в одну страницу. Для sysfs нет возможности просигнализировать о последней из набора операции записи, поэтому код реализации бинарных атрибутов должен быть в состоянии определить конец данных каким-то другим способом.

Двоичные атрибуты должны быть созданы явно; они не могут быть установлены как атрибуты по умолчанию. Для создания бинарного атрибута вызовите:

```
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);
```

Двоичные атрибуты могут быть удалены с помощью:

```
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
```

Символические ссылки

Файловая система sysfs имеет обычную древовидную структуру, отражающую иерархическую организацию kobject-ов, которые она представляет. Однако, отношения между объектами в ядре часто более сложные, чем эти. Например, одно поддерево sysfs (***/sys/devices***) представляет все известные в системе устройства, в то время как другие поддеревья

(в ***/sys/bus***) представляют драйверы устройств. Эти деревья, однако, не показывают отношения между драйверами и устройствами, которыми они управляют. Показ этих дополнительных отношений требует дополнительных указателей, которые реализуются в sysfs через символические ссылки.

Создание символической ссылки в sysfs является простым:

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char
*name);
```

Эта функция создаёт ссылку (названную **name**, имя), указывающую на запись **target**-а (цели) в sysfs как атрибут kobj-а. Она является относительной ссылкой, так что она работает независимо от того, где в любой специфической системе смонтирована sysfs.

Ссылка сохраняется, даже если **target** удалён из системы. Если вы создаёте символические ссылки на другие kobject-ы, вам следует, вероятно, иметь способ узнать об изменениях в этих kobject-ах, или какую-то гарантию того, что целевые kobject-ы не исчезнут. Последствия (мёртвые символические ссылки внутри sysfs) не особенно серьёзные, но они показывают не лучший стиль программирования и могут создать путаницу в пространстве пользователя.

Символические ссылки могут быть удалены с помощью:

```
void sysfs_remove_link(struct kobject *kobj, char *name);
```

Генерация события горячего подключения

Событие горячего подключения является уведомлением пользовательского пространства от ядра, что в конфигурации системы что-то изменилось. Они генерируются при создании или уничтожении kobject-а. Такие события генерируются, например, когда цифровая камера подключается при помощи кабеля USB, когда пользователь переключает режимы консоли, или когда диск заново разбит на разделы. События горячего подключения превращаются в вызов ***/sbin/hotplug***, который может реагировать на каждое событие загрузкой драйверов, созданием узлов устройств, монтированием разделов, или любыми другими действиями, которые являются необходимыми.

Последней важной функцией kobject-а, которую мы рассмотрим, является генерация этих событий. Фактическая генерация события имеет место, когда kobject передаётся в ***kobject_add*** или ***kobject_del***. Перед тем, как событие передано в пространство пользователя, код, связанный с kobject (или, более конкретно, kset, к которому он принадлежит), имеет возможность добавить информацию для пространства пользователя или полностью запретить генерацию события.

Операции горячего подключения

Фактический контроль событий горячего подключения осуществляется путём набора методов, хранящихся в структуре **kset_hotplug_ops**:

```
struct kset_hotplug_ops {
    int (*filter)(struct kset *kset, struct kobject *kobj);
    char *(*name)(struct kset *kset, struct kobject *kobj);
```

```
    int (*hotplug)(struct kset *kset, struct kobject *kobj,
                   char **envp, int num_envp, char *buffer,
                   int buffer_size);
};
```

Указатель на эту структуру можно найти в поле **hotplug_ops** структуры kset-а. Если данный kobject не содержится в kset-е, ядро осуществляет поиск вверх по иерархии (с помощью указателя **parent**), пока не найдёт kobject, который имеет kset; затем используются операции горячего подключения этого kset-а.

Операция ***filter*** горячего подключения вызывается всякий раз, когда ядро рассматривает возможность генерации события для данного kobject-а. Если ***filter*** возвращает 0, событие не создаётся. Этот метод, таким образом, даёт коду kset-а возможность определить, какие события следует передать в пользовательское пространство, а какие нет.

В качестве примера того, как может быть использован этот метод, рассмотрим блочную подсистему. Есть по крайней мере три типа используемых в ней kobject-ов, представляющих диски, разделы и очереди запросов. Пользовательское пространство может захотеть реагировать на добавление диска или раздела, но оно обычно не заботится об очередях запросов. Таким образом, метод ***filter*** разрешает генерацию события только для kobject-ов, представляющих диски и разделы. Выглядит это примерно так:

```
static int block_hotplug_filter(struct kset *kset, struct kobject *kobj)
{
    struct kobj_type *ktype = get_ktype(kobj);

    return ((ktype == &ktype_block) || (ktype == &ktype_part));
}
```

Здесь достаточно быстрого теста на тип kobject, чтобы решить, должно ли событие быть сгенерированным или нет.

При вызове программы горячего подключения пользовательского пространства, в качестве единственного параметра ей передаётся имя соответствующей подсистемы. Метод ***name*** горячего подключения отвечает за предоставление этого имени. Он должен возвращать простую строку, подходящую для передачи в пространство пользователя.

Всё остальное, что скрипт горячего подключения может захотеть узнать, передаётся в окружении. Последний метод горячего подключения (***hotplug***) даёт возможность добавлять полезные переменные окружения перед вызовом этого скрипта. Повторим, что прототипом метода является:

```
int (*hotplug)(struct kset *kset, struct kobject *kobj,
               char **envp, int num_envp, char *buffer,
               int buffer_size);
```

Как обычно, **kset** и **kobject** описывают объект, для которого генерируется событие. Массив **envp** является местом для сохранения дополнительных определений переменных окружения (в обычном формате **ИМЯ=значение**); в нём доступны записи **num_envp**. Сами переменные должны быть закодированы в буфере размером **buffer_size** байт. Если вы добавляете любую переменную к **envp**, не забудьте после вашего последнего добавления добавить запись **NULL**, чтобы ядро знало, где находится конец. Возвращаемое значение, как правило, 0; любая

ненулевое возвращаемое значение прерывает генерацию события горячего подключения.

Генерация событий горячего подключения (как значительная часть работы в модели устройства), как правило, обрабатываться с помощью логики на уровне шины драйвера.

Шины, устройства и драйверы

До сих пор мы видели немало низкоуровневой инфраструктуры и относительно мало примеров. Мы постараемся компенсировать это в оставшейся части этой главы, как только мы перейдём на более высокие уровни модели устройства Linux. С этой целью мы вводим новую виртуальную шину, которую мы называем ***lddbus*** (* Логичным именем для этой шины, конечно, было бы "sbus", но это название уже было взято настоящей, физической шиной.), и изменяем драйвер ***scullp*** для "подключения" к этой шине. Ещё раз, большая часть прочитанного здесь никогда не потребуется многим авторам драйверов. Подробности этого уровня, как правило, обрабатываются на уровне шины, и небольшому числу авторов потребуется добавить новый тип шины. Однако, эта информация является полезной для тех, кому интересно, что происходит внутри PCI, USB и других уровнях или кому требуется сделать изменения на этом уровне.

Шины

Шина является каналом между процессором и одним или несколькими устройствами. Для целей модели устройства, все устройства подключаются через шину, даже если она является внутренней, виртуальной, "инструментальной" шиной. Шины могут подключаться друг в друга - контроллер USB, к примеру, обычно представляет собой PCI устройство. Модель устройства отражает фактические связи между шинами и устройствами, которыми они управляют.

В модели устройства Linux шина представлена структурой **bus_type**, определённой в ***<linux/device.h>***. Эта структура выглядит следующим образом:

```
struct bus_type {
    char *name;
    struct subsystem subsys;
    struct kset drivers;
    struct kset devices;
    int (*match)(struct device *dev, struct device_driver *drv);
    struct device *(*add)(struct device * parent, char * bus_id);
    int (*hotplug) (struct device *dev, char **envp,
                    int num_envp, char *buffer, int buffer_size);
    /* Некоторые поля опущены */
};
```

Поля **name** является именем этой шины, что-нибудь такое, как pci. Вы можете видеть из структуры, что каждая шина имеет собственную подсистему; однако, эти подсистемы не живут на верхнем уровне в sysfs. Вместо этого, они находятся под шинной подсистемой. Шина содержит два kset-а, представляющих известные драйверы для этой шины и все устройства, подключенные к шине. Затем существует набор методов, которые мы получим в ближайшее время.

Регистрация шины

Как мы уже отмечали, исходник примера включает реализацию виртуальной шины под названием ***lddbus***. Эта шина создаёт свою структуру **bus_type** следующим образом:

```
struct bus_type ldd_bus_type = {
    .name = "ldd",
    .match = ldd_match,
    .hotplug = ldd_hotplug,
};
```

Обратите внимание, что очень немногие из полей **bus_type** требуют инициализации; большинство из них обрабатывается ядром модели устройства. Однако, мы должны указать имя этой шины и все методы, которые его сопровождают.

Безусловно, новая шина должна быть зарегистрирована в системе с помощью вызова ***bus_register***. Код ***lddbus*** делает это таким образом:

```
ret = bus_register(&ldd_bus_type);
if (ret)
    return ret;
```

Этот вызов может, конечно, оказаться неудачным, так что возвращаемое значение всегда должно быть проверено. Если он успешен, новая шинная подсистема была добавлена в систему; это видно в sysfs в ***/sys/bus*** и можно начинать добавлять устройства.

Если необходимо удалить шину из системы (когда, например, удаляется связанный с ней модуль), должна вызываться ***bus_unregister***:

```
void bus_unregister(struct bus_type *bus);
```

Методы шины

Есть несколько методов, определённых для структуры **bus_type**; они позволяют коду шины выступать в качестве посредника между ядром устройств и отдельными драйверами. Методами, определёнными в ядре версии 2.6.10 являются:

int (*match)(struct device *device, struct device_driver *driver);
Этот метод вызывается, возможно, несколько раз, каждый раз, когда к этой шины добавляется новое устройство или драйвер. Она должна вернуть ненулевое значение, если данный **device** может быть обработан данным драйвером. (Мы скоро узнаем детали структур драйвера **device** и **device_**). Эта функция должна обрабатываться на уровне шины, потому что именно здесь существует соответствующая логика; основное ядро не может знать, как соответствовать устройствам и драйверам для всех возможных типов шины.

int (*hotplug) (struct device *device, char **envp, int num_envp, char *buffer, int buffer_size);
Этот метод позволяет шине добавить переменные в окружение перед генерацией события горячего подключения в пользовательском пространстве. Его параметры такие же, как и для метода ***hotplug*** kset-а (описанных в предыдущем разделе "Генерация события горячего подключения" 360).

Драйвер ***lddbus*** имеет очень простую функцию совпадения, которая просто сравнивает имена драйвера и устройства:

```
static int ldd_match(struct device *dev, struct device_driver *driver)
{
    return !strncmp(dev->bus_id, driver->name, strlen(driver->name));
}
```

Когда речь идёт о реальном оборудовании, функция ***match*** обычно делает какое-то сравнение между идентификационным номером оборудования, предоставляемым самим устройством, и идентификаторами, поддерживаемыми драйвером.

Метод горячего подключения в ***lddbus*** выглядит следующим образом:

```
static int ldd_hotplug(struct device *dev, char **envp, int num_envp, char
*buffer, int buffer_size)
{
    envp[0] = buffer;
    if (snprintf(buffer, buffer_size, "LDDBUS_VERSION=%s",
                    Version) >= buffer_size)
        return -ENOMEM;
    envp[1] = NULL;
    return 0;
}
```

Здесь мы добавляем в него номер текущей версии исходного кода ***lddbus***, на случай, если кто-нибудь полюбопытствует.

Перебор устройств и драйверов

Если вы пишете код шинного уровня, вам может потребоваться выполнять некоторые операции со всеми устройствами или драйверами, которые были зарегистрированы на вашей шине. Может быть заманчиво копаться непосредственно в структурах в структуре **bus_type**, но лучше использовать предоставленные вспомогательные функции.

Для работы с каждым устройством, известном шине, используйте:

```
int bus_for_each_dev(struct bus_type *bus, struct device *start,
                     void *data, int (*fn)(struct device *, void *));
```

Эта функция перебирает все устройства на шине, передавая связанную структуру **device** в **fn**, вместе со значением, передаваемым как **data**. Если **start** является NULL, итерация начинается с первого устройства на шине; в противном случае итерация начинается с первого устройства после **start**. Если **fn** возвращает ненулевое значение, итерация останавливается и такое значение является возвращённым из ***bus_for_each_dev***.

Существует аналогичная функция для перебора драйверов:

```
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start,
                     void *data, int (*fn)(struct device_driver *, void *));
```

Эта функция работает подобно ***bus_for_each_dev***, за исключением, конечно, что вместо этого она работает с драйверами.

Следует отметить, что обе эти функции в течение работы удерживают шинный семафор подсистемы чтения/записи. Так что попытка использовать их обе вместе приведёт к

взаимоблокировке, каждая будет пытаться получить один и тот же семафор. Операции, которые изменяют шину (такие, как отмена регистрации устройств), также будут блокироваться. Таким образом, используйте функцию ***bus_for_each*** с некоторой осторожностью.

Атрибуты шины

Почти каждый слой в модели устройства Linux предоставляет интерфейс для добавления атрибутов и уровень шины не является исключением. Тип **bus_attribute** определён в ***<linux/device.h>*** следующим образом:

```
struct bus_attribute {
    struct attribute attr;
    ssize_t (*show)(struct bus_type *bus, char *buf);
    ssize_t (*store)(struct bus_type *bus, const char *buf, size_t count);
};
```

Мы уже видели **struct attribute** в разделе "Атрибуты по умолчанию" 357. Тип **bus_attribute** также включает в себя два метода для отображения и установки значения атрибута. Большинство уровней модели устройства над уровнем kobject-а работают таким образом. Для создания во время компиляции и инициализации структуры **bus_attribute** был предусмотрен удобный макрос:

```
BUS_ATTR(name, mode, show, store);
```

Этот макрос декларирует структуру, создавая её название, предваряя заданное **name** строкой **bus_attr_**.

Любые атрибуты, принадлежащие шине, должны быть созданы явно с помощью ***bus_create_file***:

```
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
```

Атрибуты также могут быть удалены с помощью:

```
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
```

Драйвер ***lddbus*** создаёт простой файл атрибута, опять же содержащий номер версии исходника. Метод ***show*** и структура **bus_attribute** создаются следующим образом:

```
static ssize_t show_bus_version(struct bus_type *bus, char *buf)
{
    return snprintf(buf, PAGE_SIZE, "%s\n", Version);
}
static BUS_ATTR(version, S_IRUGO, show_bus_version, NULL);
```

Создание атрибутного файла производится на время загрузки модуля:

```
if (bus_create_file(&ldd_bus_type, &bus_attr_version))
    printk(KERN_NOTICE "Unable to create version attribute\n");
```

Этот вызов создаёт атрибутный файл (***/sys/bus/ldd/version***), содержащее номер ревизии кода ***lddbus***.

Устройства

На самом низком уровне, каждое устройство в системе Linux представлено экземпляром **struct device**:

```
struct device {
    struct device *parent;
    struct kobject kobj;
    char bus_id[BUS_ID_SIZE];
    struct bus_type *bus;
    struct device_driver *driver;
    void *driver_data;
    void (*release)(struct device *dev);
    /* Некоторые поля опущены */
};
```

Есть много других полей **struct device**, которые представляют интерес только для кода ядра устройства. Эти поля, однако, стоит знать:

struct device *parent
Устройство-"родитель" для устройства - то устройство, к которому оно подключено. В большинстве случаев родительским устройством является какая-то шина или хост-контроллер. Если **parent** является NULL, устройство является устройством верхнего уровня, которое обычно не то, что вы хотите.

struct kobject kobj;
Kobject, который представляет это устройство и подсоединяет его в эту иерархию. Заметим, что как правило **device->kobj->parent** эквивалентен **&device-parent->kobj**.

char bus_id[BUS_ID_SIZE];
Строка, которая однозначно идентифицирует это устройство на этой шине. PCI устройства, например, используют стандартный формат PCI идентификатора, содержащий домен, шину, устройство и функциональные номера.

struct bus_type *bus;
Определяет, на каком виде шины находится устройство.

struct device_driver *driver;
Драйвер, который управляет этим устройством; мы изучаем **struct device_driver** в следующем разделе.

void *driver_data;
Поле собственных данных, которые могут быть использованы драйвером устройства.

void (*release)(struct device *dev);
Метод вызывается, когда удаляется последняя ссылка на устройство; он вызывается из метода ***release*** встроенного kobject-а. Все структуры устройства, зарегистрированные в ядре, должны иметь метод ***release***, или ядро распечатывает страшные жалобы.

Перед тем, как структура устройства может быть зарегистрирована, должны быть установлены, по крайней мере, поля **parent**, **bus_id**, **bus** и **release**.

Регистрация устройства

Существует обычный набор функций регистрации и её отмены:

```
int device_register(struct device *dev);
void device_unregister(struct device *dev);
```

Мы видели, как код ***lddbus*** регистрирует свой тип шины. Тем не менее, настоящая шина является устройством и должна быть зарегистрирована отдельно. Для простоты, модуль ***lddbus*** поддерживает только одну виртуальную шину, так что драйвер создаёт свои устройства во время компиляции:

```
static void ldd_bus_release(struct device *dev)
{
    printk(KERN_DEBUG "lddbus release\n");
}

struct device ldd_bus = {
    .bus_id = "ldd0",
    .release = ldd_bus_release
};
```

Это шина верхнего уровня, поэтому поля **parent** и **bus** остались NULL. У нас есть простой метод ***release***, не делающий ничего, и, как первая (и единственная) шина, она имеет имя **ldd0**. Это устройство шины регистрируется так:

```
ret = device_register(&ldd_bus);
if (ret)
    printk(KERN_NOTICE "Unable to register ldd0\n");
```

Как только этот вызов будет завершён, в sysfs в каталоге ***/sys/devices*** можно увидеть новую шину. Любые устройства, добавленные к этой шине, показываются затем в ***/sys/devices/ldd0/***.

Атрибуты устройства

Записи устройств в sysfs могут иметь атрибуты. Соответствующей структурой является:

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

Эти структуры атрибутов могут быть созданы во время компиляции с помощью этого макроса:

```
DEVICE_ATTR(name, mode, show, store);
```

Результирующая структура называется, предваряя **dev_attr_** заданное **name**. Фактическое управление файлами атрибутов осуществляется обычной парой функций:

```
int device_create_file(struct device *device, struct device_attribute
*entry);
void device_remove_file(struct device *dev, struct device_attribute *attr);
```

Поле **dev_attrs** в **struct bus_type** указывает на список атрибутов по умолчанию, создаваемых для каждого добавляемого к шине устройства.

Внедрение структуры устройства

Структура **device** содержит информацию, которая необходима ядру модели устройства для моделирования системы. Однако, большинство подсистем отслеживают дополнительную информацию о тех устройств, которые они содержат. Как результат, редкие устройства представлены голыми структурами **device**; вместо этого, эта структура, как и структуры kobject, как правило, встроены в представление устройства более высокого уровня. Если вы посмотрите на определения **struct pci_dev** или **struct usb_device**, вы найдете похороненную внутри **struct device**. Как правило, низкоуровневые драйверы даже не знают об этой **struct device**, но здесь могут быть исключения.

Драйвер ***lddbus*** создаёт свой собственный тип устройства (**struct ldd_device**) и ожидает, что отдельные драйверы устройств зарегистрировали свои устройства, используя этот тип. Он является простой структурой:

```
struct ldd_device {
    char *name;
    struct ldd_driver *driver;
    struct device dev;
};

#define to_ldd_device(_dev) container_of(_dev, struct ldd_device, dev);
```

Эта структура позволяет драйверу обеспечить актуальное имя для устройства (которое может отличается от его ID шины, сохранённом в структуре **device**) и указатель на информацию драйвера. Структуры для настоящих устройств обычно также содержат информацию о поставщике, модели устройства, конфигурации устройства, используемых ресурсах и так далее. Хорошие примеры могут быть найдены в **struct pci_dev** (***<linux/pci.h>***) или **struct usb_device** (***<linux/usb.h>***). Для **struct ldd_device** также определён удобный макрос (***to_ldd_device***), чтобы сделать простым преобразование указателей во встроенной структуре **device** в указатели **ldd_device**.

Интерфейс регистрации, экспортируемый ***lddbus***, выглядит следующим образом:

```
int register_ldd_device(struct ldd_device *ldddev)
{
    ldddev->dev.bus = &ldd_bus_type;
    ldddev->dev.parent = &ldd_bus;
    ldddev->dev.release = ldd_dev_release;
    strncpy(ldddev->dev.bus_id, ldddev->name, BUS_ID_SIZE);
    return device_register(&ldddev->dev);
}
EXPORT_SYMBOL(register_ldd_device);
```

Здесь мы просто заполняем некоторые поля встроенной структуры **device** (о которых

отдельным драйверам не необходимости знать) и регистрируем устройство в драйверном ядре. Если бы мы хотели добавить в устройство зависимые от шины атрибуты, мы могли бы сделать это здесь.

Чтобы показать, как используется этот интерфейс, давайте познакомимся с другим примером драйвера, который мы назвали ***sculld***. Он является ещё одним вариантом драйвера ***scullp***, впервые представленного в Главе 8 203. Он реализует обычное устройство в области памяти, но ***sculld*** также работает с моделью устройства в Linux через интерфейс ***lddbus***.

Драйвер ***sculld*** добавляет собственный атрибут к своей записи устройства; этот атрибут, названный **dev**, просто содержит связанный с ним номер устройства. Этот атрибут может быть использован скриптом загрузки модуля или подсистемой горячего подключения для автоматического создания узлов устройства, когда устройство добавляется в систему. Установка этого атрибута следует обычным моделям:

```
static ssize_t sculld_show_dev(struct device *ddev, char *buf)
{
    struct sculld_dev *dev = ddev->driver_data;

    return print_dev_t(buf, dev->cdev.dev);
}

static DEVICE_ATTR(dev, S_IRUGO, sculld_show_dev, NULL);
```

Затем, во время инициализации, устройство регистрируется и создается атрибут **dev** посредством следующей функции:

```
static void sculld_register_dev(struct sculld_dev *dev, int index)
{
    sprintf(dev->devname, "sculld%d", index);
    dev->ldev.name = dev->devname;
    dev->ldev.driver = &sculld_driver;
    dev->ldev.dev.driver_data = dev;
    register_ldd_device(&dev->ldev);
    device_create_file(&dev->ldev.dev, &dev_attr_dev);
}
```

Обратите внимание, что мы используем поле **driver_data** для сохранения указателя на нашу собственную внутреннюю структуру устройства.

Драйверы устройств

Модель устройства отслеживает все драйверы, известные в системе. Основной причиной для этого отслеживания является необходимость разрешить драйверному ядру сопоставлять драйверы с новыми устройствами. Однако, хотя драйверы представляют собой известные объекты внутри системы, стал возможным ряд других вещей. Драйверы устройств могут, например, экспортировать переменные информации и конфигурации, которые не зависят от какого-либо особого устройства.

Драйверы определяются следующей структурой:

```
struct device_driver {
    char *name;
```

```
    struct bus_type *bus;
    struct kobject kobj;
    struct list_head devices;
    int (*probe)(struct device *dev);
    int (*remove)(struct device *dev);
    void (*shutdown) (struct device *dev);
};
```

Снова, некоторые поля этой структуры были опущены (смотрите ***<linux/device.h>*** для полной информации). Здесь, **name** является именем драйвера (оно появляется в sysfs), **bus** является типом шины, с которой работает этот драйвер, **kobj** является неизбежным kobject, **devices** является списком всех устройств в настоящее время связанных с этим драйвером, ***probe*** является функцией, вызываемой для запроса наличия определённого устройства (и может ли этот драйвер работать с ним), ***remove*** вызывается, когда устройство удаляется из системы, и ***shutdown*** вызывается во время выключения для перевода устройства в пассивное состояние.

Форма функций для работы со структурами **device_driver** должны теперь выглядеть уже знакомо (так что мы рассматриваем их очень быстро). Функциями регистрации являются:

```
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
```

Существует обычная структура атрибута:

```
struct driver_attribute {
    struct attribute attr;
    ssize_t (*show)(struct device_driver *drv, char *buf);
    ssize_t (*store)(struct device_driver *drv, const char *buf, size_t
count);
};
DRIVER_ATTR(name, mode, show, store);
```

И атрибутные файлы создаются обычным образом:

```
int driver_create_file(struct device_driver *drv, struct driver_attribute
*attr);
void driver_remove_file(struct device_driver *drv, struct driver_attribute
*attr);
```

Структура **bus_type** содержит поле (**drv_attrs**), которое указывает на набор атрибутов по умолчанию, создаваемых для всех драйверов, связанных с этой шиной.

Внедрение структуры драйвера

Как и в случае с большинством структур драйверного ядра, структура **device_driver** обычно внедрена в высокоуровневую, зависимую от шины структуру. Подсистема ***lddbus*** никогда не пойдёт против такой тенденции, поэтому она определяет свою собственную структуру **ldd_driver**:

```
struct ldd_driver {
    char *version;
    struct module *module;
```

```
    struct device_driver driver;
    struct driver_attribute version_attr;
};

#define to_ldd_driver(drv) container_of(drv, struct ldd_driver, driver);
```

Здесь мы требуем, чтобы каждый драйвер предоставлял свою текущую версию программного обеспечения, а ***lddbus*** экспортирует эту строку версии для каждого драйвера о котором она знает. Шинно-зависимой функцией регистрации драйвера является:

```
int register_ldd_driver(struct ldd_driver *driver)
{
    int ret;
    driver->driver.bus = &ldd_bus_type;
    ret = driver_register(&driver->driver);
    if (ret)
        return ret;
    driver->version_attr.attr.name = "version";
    driver->version_attr.attr.owner = driver->module;
    driver->version_attr.attr.mode = S_IRUGO;
    driver->version_attr.show = show_version;
    driver->version_attr.store = NULL;
    return driver_create_file(&driver->driver, &driver->version_attr);
}
```

Первая половина функции просто регистрирует в ядре низкоуровневую структуру **device_driver**; остальная устанавливает атрибут **version**. Так как этот атрибут создаётся во время выполнения, мы не можем использовать макрос **DRIVER_ATTR**; вместо этого структура **driver_attribute** должна быть заполнена вручную. Обратите внимание, что мы установили владельцем атрибута модуль драйвера, а не модуль ***lddbus***; причину этого можно увидеть в реализации функции ***show*** для этого атрибута:

```
static ssize_t show_version(struct device_driver *driver, char *buf)
{
    struct ldd_driver *ldriver = to_ldd_driver(driver);

    sprintf(buf, "%s\n", ldriver->version);
    return strlen(buf);
}
```

Можно подумать, что владельцем атрибута должен быть модуль ***lddbus***, поскольку функция, которая реализует данный атрибут, определяется здесь. Однако, эта функция работает со структурой **ldd_driver**, созданной (и принадлежащей) самим драйвером. Если бы эта структура ушла в то время, как процесс в пользовательском пространстве попытался прочитать номер версии, всё могло бы поломаться. Назначение модуля драйвера в качестве владельца атрибута предохраняет модулю от выгрузки, пока пользовательское пространство удерживает файл атрибута открытым. Так как каждый модуль драйвера создаёт ссылку на модуль ***lddbus***, мы можем быть уверены, что ***lddbus*** не будет выгружена в неподходящее время.

Для полноты, ***sculld*** создаёт свою структуру **ldd_driver** следующим образом:

```
static struct ldd_driver sculld_driver = {
```

```
    .version = "$Revision: 1.1 $",
    .module = THIS_MODULE,
    .driver = {
        .name = "sculld",
    },
};
```

В систему её добавляет простой вызов ***register_ldd_driver***. После завершения инициализации информацию драйвера можно увидеть в sysfs:

```
$ tree /sys/bus/ldd/drivers
/sys/bus/ldd/drivers
`-- sculld
    |-- sculld0 -> ../../../../devices/ldd0/sculld0
    |-- sculld1 -> ../../../../devices/ldd0/sculld1
    |-- sculld2 -> ../../../../devices/ldd0/sculld2
    |-- sculld3 -> ../../../../devices/ldd0/sculld3
    `-- version
```

Классы

Последней концепцией модели устройства, которую мы рассмотрим в этой главе, является ***класс***. Класс является высокоуровневым представлением устройство, которое резюмирует низкоуровневые детали реализации. Драйверы могут видеть SCSI диск или диск ATA, но на уровне класса они все просто диски. Классы позволяют пользовательскому пространству работать с устройствами, базируясь на том, что они делают, а не как они подключены или как они работают.

Почти все классы отображаются в sysfs в каталоге ***/sys/class***. Так, например, все сетевые интерфейсы, независимо от типа интерфейса, могут быть найдены в ***/sys/class/net***. Устройства ввода могут быть найдены в ***/sys/class/input***, а последовательных устройства находятся в ***/sys/class/tty***. Единственным исключением являются блочные устройства, которые можно найти в каталоге ***/sys/block*** по историческим причинам.

Членство в классе обычно обрабатывается высокоуровневым кодом без необходимости явной поддержки со стороны драйверов. Когда драйвер ***sbull*** (смотрите Главу 16 445) создаёт виртуальный диск устройства, он автоматически появляется в ***/sys/block***. Сетевому драйверу ***snull*** (смотрите Главу 17 478) не требуется делать ничего специального для представления его интерфейсов в ***/sys/class/net***. Однако, будут случаи, когда в конечном итоге драйверы будут иметь дело с классами напрямую.

Во многих случаях подсистема классов является наилучшим способом экспорта информации для пользовательского пространстве. Когда подсистема создаёт класс, она принадлежит классу полностью, поэтому нет необходимости беспокоиться, каким модулям принадлежат атрибуты, находящиеся там. Немного времени надо, чтобы побродив в более аппаратно-ориентированных частях sysfs понять, что это может быть недружественным местом для прямого просмотра. Пользователи более успешно находят информацию в ***/sys/class/some-widget***, чем, скажем, в ***/sys/devices/pci0000:00/0000:00:10.0/usb2/2-0:1.0***. Драйверное ядро экспортирует два различных интерфейса для управления классами. Процедуры ***class_simple*** разработаны, чтобы сделать добавление новых классов в систему как можно более простым; их основная цель обычно состоит в том, чтобы показать атрибуты, содержащие номера устройства, разрешить автоматическое создание узлов устройств.

Формальный интерфейс класса более сложный, но предлагает также и больше функций. Мы начинаем с простой версии.

Интерфейс class_simple

Начиная с версии 2.6.13 интерфейс class_simple в ядрах больше не присутствует. Все упоминания его и его функций в данной книге устарели.

Интерфейс ***class_simple*** должен был стать настолько простым в использовании, чтобы никто не имел бы никакого предлога для отказа от экспорта, как минимум, атрибута, содержащего номер, назначенный устройству. Использование этого интерфейса является просто вопросом нескольких вызовов функций с небольшим количеством обычного шаблона, связанного с моделью устройства Linux.

Первым шагом является создание самого класса. Это осуществляется с помощью вызова ***class_simple_create***:

```
struct class_simple *class_simple_create(struct module *owner, char *name);
```

Эта функция создаёт класс с заданным **name** (именем). Конечно, операция может потерпеть неудачу, так что возвращаемое значение всегда должно быть проверено (с использованием IS_ERR, описанного в разделе "Указатели и значения ошибок" 282 в Главе 11 286), прежде чем продолжить.

Простой класс может быть уничтожен с помощью:

```
void class_simple_destroy(struct class_simple *cs);
```

Настоящая цель создания простого класса состоит в добавлении к нему устройства; эта задача решается с помощью:

```
struct class_device *class_simple_device_add(struct class_simple *cs,
                dev_t devnum,
                struct device *device,
                const char *fmt, ...);
```

Здесь, **cs** является ранее созданным простым классом, **devnum** является присвоенным номером устройства, **device** является **struct device**, представляющей данное устройство и остальными параметрами являются строка формата в стиле printk и аргументы для создания имени устройства. Этот вызов добавляет запись к классе, содержащую один атрибут, **dev**, который содержит номер устройства. Если параметр **device** не NULL, символическая ссылка (с названием из **device**) указывает на запись устройства в ***/sys/devices***.

Можно добавить другие атрибуты к записи устройства. Это всего лишь вопрос использования ***class_device_create_file***, которую мы обсудим в следующем разделе вместе с остальной частью подсистемы полного класса.

Классы генерируют события горячего подключения, когда устройства приходят и уходят. Если вашему драйверу необходимо добавить переменные в окружение для обработчика события в пространстве пользователя, можно создать обратный вызов горячего подключения с помощью:

```
int class_simple_set_hotplug(struct class_simple *cs,
                int (*hotplug)(struct class_device *dev,
                char **envp, int num_envp,
                char *buffer, int buffer_size));
```

Когда устройство уходит, запись класса должна быть удалена с помощью:

```
void class_simple_device_remove(dev_t dev);
```

Обратите внимание, что структура **class_device**, возвращаемая ***class_simple_device_add***, здесь не требуется; достаточно номера устройства (который должен быть, конечно, уникальным).

Полный интерфейс класса

Для многих потребностей достаточно интерфейса ***class_simple***, но иногда требуется больше гибкости. Следующее обсуждение описывает, как использовать полноценный механизм класса, на котором основан ***class_simple***. Оно является кратким: функции и структуры класса следуют тем же шаблонам, как и остальная модель устройства, поэтому действительно нового здесь мало.

Управление классами

Класс определяется экземпляром **struct class**:

```
struct class {
    char *name;
    struct class_attribute *class_attrs;
    struct class_device_attribute *class_dev_attrs;
    int (*hotplug)(struct class_device *dev, char **envp,
                   int num_envp, char *buffer, int buffer_size);
    void (*release)(struct class_device *dev);
    void (*class_release)(struct class *class);
    /* Некоторые поля опущены */
};
```

Каждому классу требуется уникальное **name** (имя), которое определяет, как этот класс представляется в ***/sys/class***. После регистрации класса, все атрибуты, перечисленные в (заканчивающемся NULL) массиве, указываемом **class_attrs**, созданы. Для каждого устройства, добавляемого в класс, существует также набор атрибутов по умолчанию; на них указывает **class_dev_attrs**. Существует обычная функция горячего подключения для добавления переменных в окружение при генерации события. Есть также два метода ***release***: ***release*** вызывается, когда устройство удаляется из класса, а ***class_release*** вызывается, когда освобождается сам класс.

Функциями регистрации являются:

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

Интерфейс для работы с атрибутами не будет здесь ни для кого сюрпризом:

```
struct class_attribute {
```

```
    struct attribute attr;
    ssize_t (*show)(struct class *cls, char *buf);
    ssize_t (*store)(struct class *cls, const char *buf, size_t count);
};

CLASS_ATTR(name, mode, show, store);

int class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute
*attr);
```

Устройства класса

Настоящая цель класса - служить контейнером для устройств, которые являются членами этого класса. Член представлен **struct class_device**:

```
struct class_device {
    struct kobject kobj;
    struct class *class;
    struct device *dev;
    void *class_data;
    char class_id[BUS_ID_SIZE];
};
```

Поле **class_id** содержит название этого устройства, как оно появляется в sysfs. Указатель **class** должен указывать на класс, содержащий это устройство и **dev** должен указывать на связанную с устройством структуру. Настройка **dev** не является обязательной; если он не NULL, он используется для создания символической ссылки из записи класса на соответствующую запись в */sys/devices*, что позволяет легко найти запись устройства в пространстве пользователя. Класс может использовать **class_data** для содержания своего собственного указателя.

Представлены обычные функции регистрации:

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
```

Интерфейс устройств класса позволяет также переименование уже зарегистрированной записи:

```
int class_device_rename(struct class_device *cd, char *new_name);
```

Записи устройства класса имеют атрибуты:

```
struct class_device_attribute {
    struct attribute attr;
    ssize_t (*show)(struct class_device *cls, char *buf);
    ssize_t (*store)(struct class_device *cls, const char *buf, size_t
count);
};

CLASS_DEVICE_ATTR(name, mode, show, store);
```

```
int class_device_create_file(struct class_device *cls, const struct
class_device_attribute *attr);
void class_device_remove_file(struct class_device *cls, const struct
class_device_attribute *attr);
```

Когда регистрируется класс устройств, в поле класса **class_dev_attrs** создаётся набор атрибутов по умолчанию; для создания дополнительных атрибутов может быть использована ***class_device_create_file***. Атрибуты также могут быть добавлены и к классу устройств, созданных с помощью интерфейса ***class_simple***.

Интерфейсы класса

Подсистема классов имеет дополнительную концепцию, не встречающуюся в других частях модели устройства Linux. Этот механизм назван интерфейсом, но это, пожалуй, лучше думать о нём, как своего рода спусковом механизме, который может быть использован для получения уведомлений, когда устройство входит или покидает класс.

Интерфейс представлен следующим образом:

```
struct class_interface {
    struct class *class;
    int (*add) (struct class_device *cd);
    void (*remove) (struct class_device *cd);
};
```

Интерфейсы могут быть зарегистрированными и разрегистрированными с помощью:

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```

Функционирование интерфейса является простым. Всякий раз, когда устройство класса добавляется в **class**, указанный в структуре **class_interface**, вызывается функция интерфейса ***add***. Эта функции может выполнять любые дополнительные настройки, необходимые для этого устройства; эта настройка часто принимает форму добавления дополнительных атрибутов, но возможны и другие применения. Когда устройство удаляется из класса, для выполнения всей необходимой очистки вызывается метод ***remove***.

Для класса могут быть зарегистрированы множество интерфейсов.

Собираем всё вместе

Чтобы лучше понять, что делает драйверная модель, давайте пройдём по шагам жизненный цикл устройства в ядре. Мы опишем, как с драйверной моделью взаимодействует подсистема PCI, основные понятия добавления и удаления драйвера, и как добавляются и удаляются из системы устройства. Эти сведения, а при описании PCI кода ядра в частности, распространяются на все другие подсистемы, которые используют драйверное ядро для управления своими драйверами и устройствами.

Как показано на Рисунке 14-3, взаимодействие между PCI ядром, драйверным ядром и отдельными драйверами PCI является довольно сложным.

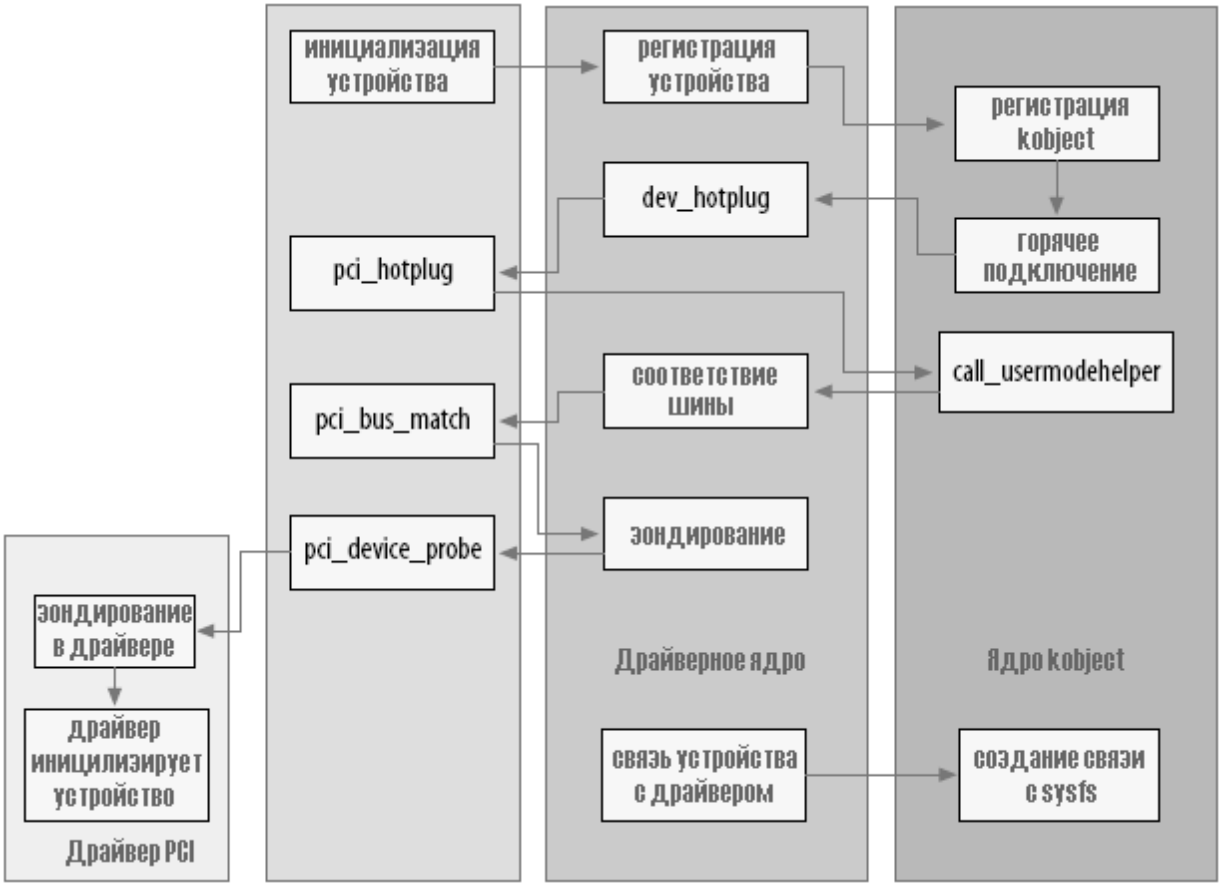


Рисунок 14-3. Процесс создания устройства

Добавление устройства

Подсистема PCI декларирует единственную **struct bus_type**, названную **pci_bus_type**, которая инициализируется следующими значениями:

```
struct bus_type pci_bus_type = {
    .name      = "pci",
    .match     = pci_bus_match,
    .hotplug   = pci_hotplug,
    .suspend   = pci_device_suspend,
    .resume    = pci_device_resume,
    .dev_attrs = pci_dev_attrs,
};
```

Эта переменная **pci_bus_type** регистрируется в драйверном ядре, когда подсистема PCI загружается в ядро с помощью вызова ***bus_register***. Когда это происходит, драйверное ядро создаёт в sysfs каталог ***/sys/bus/pci***, который состоит из двух каталогов: ***devices*** и ***drivers***.

Все драйверы PCI должны определить переменную **struct pci_driver**, которая определяет различные функции, которые может выполнять этот драйвер PCI (для получения дополнительной информации о подсистеме PCI и как написать драйвер PCI, смотрите Главу 12 288). Эта структура содержит **struct device_driver**, которая затем инициализируется ядром

PCI при регистрации драйвера PCI:

```
/* инициализация обычных полей драйвера */
drv->driver.name = drv->name;
drv->driver.bus = &pci_bus_type;
drv->driver.probe = pci_device_probe;
drv->driver.remove = pci_device_remove;
drv->driver.kobj.ktype = &pci_driver_kobj_type;
```

Этот код устанавливает шину для драйвера задавая указатель на **pci_bus_type** и указатели на функции ***probe*** и ***remove***, чтобы они указывали на функции в ядре PCI. Чтобы файлы атрибутов драйвера PCI работали соответствующим образом, ktype для kobject-а драйвера устанавливается на переменную **pci_driver_kobj_type**. Затем ядро PCI регистрирует PCI драйвер в драйверном ядре:

```
/* регистрация в ядре */
error = driver_register(&drv->driver);
```

Драйвер теперь готов быть связанным со всеми PCI устройствами, которые он поддерживает.

Ядро PCI, с помощью архитектурно-зависимого кода, который фактически общается с шиной PCI, начинает зондирование адресного пространства PCI, осуществляя поиск всех устройств PCI. Когда устройство PCI найдено, ядро PCI создаёт в памяти новую переменную типа **struct pci_dev**. Часть структуры **struct pci_dev** выглядит следующим образом:

```
struct pci_dev {
    /* ... */
    unsigned int devfn;
    unsigned short vendor;
    unsigned short device;
    unsigned short subsystem_vendor;
    unsigned short subsystem_device;
    unsigned int class;
    /* ... */
    struct pci_driver *driver;
    /* ... */
    struct device dev;
    /* ... */
};
```

Шинно-зависимые поля этого PCI устройства инициализируются ядром PCI (**devfn**, **vendor**, **device** и другие поля) и переменная **parent** переменной **struct device** является установленной на устройство шины PCI, в которой живёт это устройство PCI. Переменная **bus** устанавливается для указания на структуру **pci_bus_type**. Затем устанавливаются переменные **name** и **bus_id**, в зависимости от названия и ID, считанного из PCI устройства.

После инициализации структуры устройства PCI устройство регистрируется в драйверном ядре вызовом:

```
device_register(&dev->dev);
```

В функции ***device_register*** драйверное ядро инициализирует ряд полей устройства,

регистрирует kobject устройства в kobject-е ядра (которое вызывает генерацию события горячего подключения, но мы обсудим это позже в этой главе) и затем добавляет устройство в список устройств, который содержится в родителе устройства. Это делается так, чтобы все устройства можно было обойти в надлежащем порядке, всегда зная, где в иерархии устройств живёт каждое из них.

Устройство добавляется затем в зависимый от шины список всех устройств, в этом примере, в список **pci_bus_type**. Затем обходится список всех драйверов, которые зарегистрированы на шине, и для каждого драйвера вызывается функция шины ***match***, с указанием этого устройства. Для шины **pci_bus_type**, прежде, чем устройство было передано драйверному ядру, функция ***match*** была установлена ядром PCI для указания на функцию ***pci_bus_match***.

Функция ***pci_bus_match*** приводит переданную ему драйверным ядром **struct device** обратно к **struct pci_dev**. Она также приводит **struct device_driver** обратно к **struct pci_driver** и затем смотрит на зависимую от устройства PCI информацию устройства и драйвер, чтобы увидеть, если драйвер утверждает, что он может поддерживать устройство такого вида. Если соответствие не успешно, функция обратно в драйверное ядро возвращает **0** и драйверное ядро переходит к следующему драйверу в своём списке.

Если соответствие найдено успешно, функция возвращает обратно в драйверное ядро **1**. Это вызывает установку драйверным ядром указателя **driver** на **struct device** для указания на этот драйвер и затем оно вызывает функцию ***probe***, которая указана в **struct device_driver**.

Ранее, до того, как PCI драйвер был зарегистрирован в драйверном ядре, переменная **probe** была установлена для указания на функцию ***pci_device_probe***. Эта функция приводит (ещё раз) **struct device** обратно к **struct pci_dev** и **struct driver**, которая обратно становится устройством (device) в **struct pci_driver**. Она вновь проверяет, что этот драйвер заявляет, что он может поддерживать это устройство (которая представляется избыточной дополнительной проверкой по какой-то неизвестной причине), увеличивает счётчик ссылок устройства и затем вызывает функцию ***probe*** PCI драйвера с указателем на структуру **struct pci_dev**, к которой она должна привязываться.

Если функция ***probe*** драйвера PCI определяет, что он по какой-то причине не может работать с этим устройством, она возвращает отрицательное значение ошибки, которое передаётся обратно в драйверное ядро и заставляет его продолжать просматривать список драйверов, чтобы проверить для этого устройства следующий. Если функция ***probe*** может претендовать на устройство, она выполняет всю необходимую инициализацию для поддержки устройства должным образом и затем обратно в драйверное ядро возвращает 0. Это приводит к добавлению драйверным ядром устройства в список всех устройств, связанных в настоящее время с этим определённым драйвером, и создаёт для устройства, которым он теперь управляет, в каталоге драйвера в sysfs символическую ссылку. Эта символическая ссылка позволяет пользователям точно видеть, с какими устройствами какие устройства связаны. Это можно увидеть как:

```
$ tree /sys/bus/pci
/sys/bus/pci/
|-- devices
|   |-- 0000:00:00.0 -> ../../../devices/pci0000:00/0000:00:00.0
|   |-- 0000:00:00.1 -> ../../../devices/pci0000:00/0000:00:00.1
```

```
|   |-- 0000:00:00.2 -> ../../../devices/pci0000:00/0000:00:00.2
|   |-- 0000:00:02.0 -> ../../../devices/pci0000:00/0000:00:02.0
|   |-- 0000:00:04.0 -> ../../../devices/pci0000:00/0000:00:04.0
|   |-- 0000:00:06.0 -> ../../../devices/pci0000:00/0000:00:06.0
|   |-- 0000:00:07.0 -> ../../../devices/pci0000:00/0000:00:07.0
|   |-- 0000:00:09.0 -> ../../../devices/pci0000:00/0000:00:09.0
|   |-- 0000:00:09.1 -> ../../../devices/pci0000:00/0000:00:09.1
|   |-- 0000:00:09.2 -> ../../../devices/pci0000:00/0000:00:09.2
|   |-- 0000:00:0c.0 -> ../../../devices/pci0000:00/0000:00:0c.0
|   |-- 0000:00:0f.0 -> ../../../devices/pci0000:00/0000:00:0f.0
|   |-- 0000:00:10.0 -> ../../../devices/pci0000:00/0000:00:10.0
|   |-- 0000:00:12.0 -> ../../../devices/pci0000:00/0000:00:12.0
|   |-- 0000:00:13.0 -> ../../../devices/pci0000:00/0000:00:13.0
|   `-- 0000:00:14.0 -> ../../../devices/pci0000:00/0000:00:14.0
`-- drivers
    |-- ALI15x3_IDE
    |   `-- 0000:00:0f.0 -> ../../../../devices/pci0000:00/0000:00:0f.0
    |-- ehci_hcd
    |   `-- 0000:00:09.2 -> ../../../../devices/pci0000:00/0000:00:09.2
    |-- ohci_hcd
    |   |-- 0000:00:02.0 -> ../../../../devices/pci0000:00/0000:00:02.0
    |   |-- 0000:00:09.0 -> ../../../../devices/pci0000:00/0000:00:09.0
    |   `-- 0000:00:09.1 -> ../../../../devices/pci0000:00/0000:00:09.1
    |-- orinoco_pci
    |   `-- 0000:00:12.0 -> ../../../../devices/pci0000:00/0000:00:12.0
    |-- radeonfb
    |   `-- 0000:00:14.0 -> ../../../../devices/pci0000:00/0000:00:14.0
    |-- serial
    `-- trident
        `-- 0000:00:04.0 -> ../../../../devices/pci0000:00/0000:00:04.0
```

Удаление устройства

PCI устройство может быть удалено из системы несколькими различными способами. Все устройства Card-Bus в действительности PCI устройства в другом физическом форм-факторе и ядро PCI в ядре не делает различий между ними. Системы, которые позволяют удаление или добавление PCI устройств во время работы машины, становятся всё более популярными и Linux их поддерживает. Существует также фальшивый драйвер PCI горячего подключения, который позволяет разработчикам тестирование, чтобы увидеть, что их PCI драйвер правильно обрабатывает удаление устройства во время работы системы. Этот модуль называется **fakephp** и заставляет ядро думать, что PCI устройство ушло, но не позволяет пользователям физически удалить из системы PCI устройство, которое не имеет надлежащего оборудования, чтобы сделать это. Смотрите документацию этого драйвера для получения дополнительной информации о том, как его использовать для тестирования PCI драйверов.

Ядро PCI прилагает гораздо меньше усилий для удаления устройства, чем для его добавления. Когда PCI устройство должно быть удалено, вызывается функция ***pci_remove_bus_device***. Эта функция выполняет некоторые специфические для PCI очистки и служебные действия и затем вызывает функцию ***device_unregister*** с указателем на член **struct device** в **struct pci_dev**.

В функции ***device_unregister*** драйверное ядро лишь отсоединяет файлы sysfs от драйвера, связанного с устройством (если таковое имеется), удаляет устройство из своего внутреннего

списка устройств и вызывает ***kobject_del*** с указателем на **struct kobject**, которая содержится в структуре **struct device**. Эта функция делает вызов горячего подключения в пользовательское пространство сообщая, что kobject теперь удалён из системы и затем она удаляет все файлы sysfs, связанные с kobject, и сам каталог в sysfs, который был первоначально создан kobject-ом.

Функция ***kobject_del*** также удаляет ссылку kobject-а самого устройства. Если эта ссылка была последней (что означает, что в пространстве пользователя нет файлов, которые были бы открыты в записи устройства в sysfs), то вызывается функция ***release*** в самом PCI устройстве, ***pci_release_dev***. Эта функция просто освобождает память, которую занимала **struct pci_dev**.

После этого удаляют все записи в sysfs, связанные с устройством, и память, связанная с устройством, освобождается. PCI устройство теперь полностью удалено из системы.

Добавление драйвера

Драйвер PCI добавляется в ядро PCI при вызове функции ***pci_register_driver***. Эта функция просто инициализирует структуру **struct device_driver**, которая содержится в структуре **struct pci_driver**, как уже упоминалось в разделе о добавлении устройства. Затем ядро PCI вызывает в драйверном ядре функцию ***driver_register*** с указателем на структуру **struct device_driver**, содержащуюся в структуре **struct pci_driver**.

Функция ***driver_register*** инициализирует несколько блокировок в структуре **struct device_driver** и затем вызывает функцию ***bus_add_driver***. Эта функция выполняет следующее шаги:

- Ищет шину, с которой должен быть связан драйвер. Если эта шина не найдена, функция мгновенно возвращается.
- Создаётся каталог драйвера в sysfs на основе имени драйвера и шины, с которой он связан.
- Захватывается внутренняя блокировка шины и затем обходятся все устройства, которые были зарегистрированы на шине, и для них вызывается функция совпадения (match), как при добавлении нового устройства. Если эта функция функция совпадения (match) завершается успешно, то происходит оставшаяся часть процесса подключения, как описано в предыдущем разделе.

Удаление драйвера

Удаление драйвера представляет собой очень простое действие. Для PCI драйвера, драйвер вызывает функцию ***pci_unregister_driver***. Эта функция просто вызывает функцию ***driver_unregister*** драйверного ядра с указателем на структуру **struct device_driver**, часть структуры **struct pci_driver**, переданной ему.

Функция ***driver_unregister*** выполняет некоторую основную служебную работу, очищая некоторые атрибуты в sysfs, которые были связаны с записью драйвера в дереве sysfs. Затем она перебирает все устройства, которые были прикреплены к этому драйверу и вызывает для них функцию ***release***. Это происходит так же, как упоминалось ранее для функции ***release***, когда из системы удаляется устройство.

После того, как все устройства отсоединены от драйвера, код драйвера выполняет этот

уникальный кусочек логики:

```
down(&drv->unload_sem);
up(&drv->unload_sem);
```

Это делается непосредственно перед возвращением к вызвавшему эту функцию. Выполняется захват этой блокировки, потому что код должен подождать, пока все счётчики ссылок на этот драйвер упадут до 0, чтобы сделать его возвращение безопасным. Это необходимо, потому что функция ***driver_unregister*** чаще всего вызывается по выходному пути выгружаемого модуля. Модуль должен оставаться в памяти так долго, пока на драйвер ссылаются устройства и ожидать освобождения этой блокировки, это позволяет ядру узнать, когда можно безопасно удалить драйвер из памяти.

Горячее подключение

Существуют два различных способа рассматривать горячее подключение. Ядро рассматривает горячее подключение как взаимодействие между оборудованием, ядром и драйвером ядра. Пользователи рассматривают горячее подключение как взаимодействие между ядром и пользовательским пространством в рамках программы, называемой ***/sbin/hotplug***. Эта программа вызывается ядром, когда оно хочет уведомить пространство пользователя, что в ядре только что случился какой-то тип события горячего подключения.

Динамические устройства

Наиболее частое использование значение термина "горячее подключение" происходит при обсуждении того факта, что большинство всех компьютерных систем теперь может обрабатывать устройства, которые появляются или исчезают, когда система включена. Это очень отличается от компьютерных систем лишь несколько лет назад, когда программисты знали, что им необходимо сканировать все устройства только во время загрузки и им никогда не приходилось беспокоиться о своих устройствах, исчезающих при отключении питания для всей машины. Теперь, с появлением USB, CardBus PCMCIA, IEEE1394 и PCI контроллеров горячего подключения ядру Linux необходимо иметь способность работать надежно, независимо от того, какое оборудование добавляется или удаляется из системы. Это ложится дополнительным бременем на автора драйвера устройства, поскольку теперь они должны всегда работать с устройством, внезапно вырываемым из подчинения без предварительного уведомления.

Каждый тип шины обрабатывает потерю устройства по-разному. Например, когда PCI, CardBus или PCMCIA устройство удаляется из системы, это обычно происходит до того, как драйвер был уведомлен об этом действии через свою функцию ***remove***. Прежде, чем это случается, все чтения из PCI шины возвращают все биты установленными. Это означает, что драйверам необходимо всегда проверять значение данных, которые они прочитали из шины PCI и быть в состоянии должным образом обработать значение **0xff**.

Пример этого можно увидеть в драйвере ***drivers/usb/host/ehci-hcd.c***, который представляет собой PCI драйвер для платы контроллера USB 2.0 (High-Speed). Он имеет следующий код в своём основном цикле установления связи для обнаружения, что плата контроллера была удалена из системы:

```
result = readl(ptr);
if (result == ~(u32)0) /* карта удалена */
    return -ENODEV;
```

Для драйверов USB, когда устройство, с которым связан USB драйвер удалено из системы, все ожидающие urb-ы, которые были отправлены в устройство, сначала заканчиваются неудачей с ошибкой **-ENODEV**. Драйвер должен распознать эту ошибку и надлежащим образом очистить весь ожидающий ввод/вывод, если он имеет место.

Устройства горячего подключения не ограничены только традиционными устройствами, такими как мышь, клавиатуры и сетевые карты. Есть много систем, которые теперь поддерживают удаление и добавление целиком процессоров и карт памяти. К счастью, ядро Linux должным образом обрабатывает добавление и удаление таких основных "системных" устройств, так что отдельные драйверы устройств не должны обращать внимание на эти вещи.

Утилита /sbin/hotplug

Как упоминалось ранее в этой главе, когда устройство добавляется или удаляется из системы, генерируется "событие горячего подключения". Это означает, что ядро вызывает программу пользовательского пространства ***/sbin/hotplug***. Эта программа, как правило, очень небольшой скрипт bash, который просто передаёт выполнение списку других программ, которые находятся в дереве каталога ***/etc/hotplug.d/***. Для большинства дистрибутивов Linux этот скрипт выглядит следующим образом:

```
DIR="/etc/hotplug.d"
for I in "${DIR}/$1/"*.hotplug "${DIR}/"default/*.hotplug ; do
    if [ -f $I ]; then
        test -x $I && $I $1 ;
    fi
done
exit 1
```

Другими словами, скрипт ищет все программы имеющие суффикс ***.hotplug***, которые могут быть заинтересованы в этом событии и вызывает их, передавая им ряд различных переменных окружения, которые были установлены ядром. Более подробная информация о работе скрипта ***/sbin/hotplug*** можно найти в комментариях к программе и на странице руководства ***hotplug(8)***.

Как упоминалось ранее, ***/sbin/hotplug*** вызывается при создании или уничтожении kobject-а. Программа горячего подключения вызывается с одним аргументом командной строки, представляющим название для данного события. Основное ядро и определённая подсистема также участвуют в установке набора переменных окружения (смотрите ниже) с информацией о том, что только что произошло. Эти переменные используются в программах горячего подключения, чтобы определить, что только что произошло в ядре, и есть ли какое-то специальное действие, которое должно иметь место.

Аргумент командной строки, переданный в ***/sbin/hotplug***, является именем, связанным с этим событием горячего подключения, как определено kset-ом, назначенным для kobject. Это имя может быть установлено вызовов функции ***name***, которая является частью структуры **hotplug_ops** kset-а, описанной ранее в этой главе; если эта функция отсутствует или никогда не вызывалась, используется название самого kset-а.

Переменными окружения по умолчанию, которые всегда устанавливаются для программы ***/sbin/hotplug***, являются:

ACTION
Строка **add** (добавить) или **remove** (удалить), в зависимости от того, был ли данный объект только что создан или уничтожен.

DEVPATH
Путь к каталогу в файловой системе sysfs, который указывает на kobject, который в настоящее время либо создан, либо уничтожен. Обратите внимание, что точка монтирования файловой системы sysfs не добавлена к этому пути, так что её определение предоставлено сделать программе пользовательского пространства.

SEQNUM
Порядковый номер для этого события горячего подключения. Порядковый номер представляет собой 64-х разрядное число, которое увеличивается с каждым генерируемым событием горячего подключения. Это позволяет пользовательскому пространству отсортировать события горячего подключения в том порядке, в котором их генерирует ядро, так как для программ пространства пользователя возможна работа не по порядку.

SUBSYSTEM
Та же строка, передаваемая в качестве аргумента командной строки, как описано выше.

Ряд различных шинных подсистем для вызова ***/sbin/hotplug*** добавляют свои собственные переменные окружения, когда связанное с шиной устройство было добавлено или удалено из системы. Они делают это в своём обратном вызове горячего подключения, указанном в **struct kset_hotplug_ops**, назначенной этой шине (как описано в разделе "Операции горячего подключения" 360). Это позволяет пользовательскому пространству иметь возможность автоматической загрузки необходимых модулей, которые могут быть необходимы для управления устройством, которое было обнаружено на шине. Вот список разных типов шин и переменных окружения, которые они добавляют для вызова ***/sbin/hotplug***.

IEEE1394 (FireWire)

Все устройства на шине IEEE1394, также известной как FireWire, имеют параметр имени для ***/sbin/hotplug*** и переменная окружения **SUBSYSTEM** устанавливается в значение **ieee1394**. Подсистема Ieee1394 также всегда добавляет следующие четыре переменные окружения:

VENDOR_ID
24-х разрядный идентификатор поставщика для устройства IEEE1394.
MODEL_ID
24-х разрядный идентификатор модели для устройства IEEE1394.
GUID
64-х разрядный GUID для этого устройства.
SPECIFIER_ID
24-х разрядное значение, определяющее владельца спецификации протокола для этого устройства
VERSION
Значение, которое определяет версию спецификации протокола для этого устройства.

Сеть

Все сетевые устройства создают сообщение горячего подключения, когда устройство зарегистрировано или разрегистрировано в ядре. Вызов */sbin/hotplug* имеет параметр имени и переменная окружения **SUBSYSTEM** устанавливается в значение **net** и добавляет только следующую переменную окружения:

INTERFACE
Имя интерфейса, который был зарегистрирован или разрегистрирован из ядра. Примерами его являются **lo** и **eth0**.

PCI

Любые устройства на шине PCI имеют параметр имени и переменная окружения SUBSYSTEM устанавливается в значение **pci**. Подсистема PCI также всегда добавляет следующие четыре переменные окружения:

PCI_CLASS
Номер PCI класса для данного устройства, в шестнадцатеричном виде.
PCI_ID
Идентификаторы поставщика и устройства PCI для данного устройства, в шестнадцатеричном виде, объединенные в формате **vendor:device**.
PCI_SUBSYS_ID
Идентификаторы поставщика и подсистемы PCI, объединенные в формате **subsys_vendor:subsys_device**.
PCI_SLOT_NAME
"Имя" слота PCI, которое даётся устройству ядром в формате **domain:bus:slot:function**. Примером может быть **0000:00:0d.0**.

Ввод

Для всех устройств ввода (мышь, клавиатуры, джойстики и так далее), сообщение горячего подключения генерируется, когда устройство добавляется и удаляется из ядра. Параметр **/sbin/hotplug** и переменная окружения **SUBSYSTEM** устанавливаются в значение **input**. Подсистема ввода также всегда добавляет следующие переменные окружения:

PRODUCT
Многозначная строка, перечисляющая значения в шестнадцатеричном виде, без ведущих нулей, в формате **bustype:vendor:product:version**.

Следующие переменные окружения могут присутствовать, если устройство их поддерживает:

NAME
Название устройства ввода, как задано устройством.
PHYS
Физический адрес устройства, который подсистема ввода дала этому устройству. Он должен быть стабильным, зависящим от местонахождения шины, на которую было подключено устройство.
EV
KEY
REL
ABS

MSC
LED
SND
FF
Все они происходят из дескриптора устройства ввода и устанавливаются в соответствующие значения, если данное устройство ввода его поддерживает.

USB

Любые устройства на шине USB имеют параметр имени и переменная окружения SUBSYSTEM устанавливается в значение **usb**. Подсистема USB также всегда добавляет следующие переменные окружения:

PRODUCT
Строка в формате ***idVendor/idProduct/bcdDevice***, которая определяет эти зависимые от устройства USB поля.
TYPE
Строка в формате ***bDeviceClass/bDeviceSubClass/bDeviceProtocol***, которая определяет эти зависимые от устройства USB поля.

Если поле **bDeviceClass** установлено в 0, также устанавливается следующая переменная окружения:

INTERFACE
Строка в формате ***bInterfaceClass/bInterfaceSubClass/bInterfaceProtocol***, которая определяет эти зависимые от устройства USB поля.

Если выбрана опция сборки ядра **CONFIG_USB_DEVICEFS**, который выбирает, что файловая система **usbfs** будет собрана в ядре, также устанавливается следующая переменная окружения:

DEVICE
Строка, которая показывает, где находится устройство в файловой системе **usbfs**. Эта строка имеет формат **/proc/bus/usb/USB_BUS_NUMBER/ SB_DEVICE_NUMBER**, в котором **USB_BUS_NUMBER** является трёхзначным номером шины USB, к которой подключено устройство, а **USB_DEVICE_NUMBER** является трёхзначным номером, который был назначен ядром для этого USB устройства.

SCSI

Все SCSI устройства создают событие горячего подключения, когда SCSI устройство создано или удалено из ядра. Вызов ***/sbin/hotplug*** имеет параметр имени и переменная окружения **SUBSYSTEM** установлена в значение **scsi** для каждого SCSI устройства, которое добавляется или удаляется из системы. Никакие дополнительные переменные окружения не добавляться системой SCSI, но она упоминается здесь потому, что существует специальный SCSI скрипт в пространстве пользователя, который может определить, что SCSI драйверы (дисковода, ленточного накопителя, обычный и т.д.) должны быть загружены для указанного устройства SCSI.

Установочные станции ноутбуков

Если поддерживающая Plug-and-Play установочная (док) станция ноутбука добавлена или удалена из работающей системы Linux (путём включения ноутбука в станцию, или его удаления), создаётся событие горячего подключения. Вызов ***/sbin/hotplug*** имеет параметр имени и переменная окружения **SUBSYSTEM** установлена в значение **dock**. Никакие другие переменные окружения не установлены.

S/390 и zSeries

На архитектуре S/390, архитектура канальной шины поддерживает широкий спектр оборудования, каждое из которых генерирует события ***/sbin/hotplug***, когда они добавляются или удаляются из виртуальной системы Linux. Все эти устройства имеют для ***/sbin/hotplug*** параметр имени и переменная окружения **SUBSYSTEM** установлена в значение **dasd**. Никакие другие переменные окружения не установлены.

Использование /sbin/hotplug

Теперь, когда ядро Linux вызывает ***/sbin/hotplug*** для каждого добавляемого или удаляемого из ядра устройства, чтобы воспользоваться этим, в пользовательском пространстве были созданы ряд очень полезных инструментов. Двумя из наиболее популярных инструментов являются скрипты Linux горячего подключения и **udev**.

Скрипты горячего подключения Linux

Скрипты горячего подключения Linux начались в качестве самого первого пользователя вызова ***/sbin/hotplug***. Эти скрипты смотрят на разные переменные окружения, которые ядро устанавливает для описание устройства, которое было только что обнаружено и затем пытаются найти модуль ядра, который соответствует этому устройству.

Как уже говорилось ранее, когда драйвер использует макрос **MODULE_DEVICE_TABLE**, программа, **depmod**, принимает эту информацию и создаёт файлы, находящиеся в ***/lib/module/KERNEL_VERSION/modules.*map***. Знак * является различием, в зависимости от типа шины, которую поддерживает драйвер. В настоящее время файлы модульной карты создаются для драйверов, которые работают с устройствами с поддержкой подсистем PCI, USB, IEEE1394, INPUT, ISAPNP и CCW.

Скрипты горячего подключения используют эти текстовые файлы модульной карты для определения модуля, чтобы попытаться загрузить его для поддержки устройства, которое было недавно обнаружено ядро. Они загружают все модули и не останавливаться на первом соответствии, с тем, чтобы позволить ядру выбрать, какой модуль лучше подходит. Эти скрипты не выгружают все модули при удалении устройств. Если бы они попытались это сделать, они могли бы случайно выключить устройства, которые также управляются тем драйвером устройства, который был удалён.

Обратите внимание, теперь, когда программа **modprobe** может читать информацию **MODULE_DEVICE_TABLE** непосредственно из модулей без необходимости файлов модульной карты, скрипты горячего подключения могут быть сокращены до небольшой обёртки вокруг программы **modprobe**.

udev

Одной из основных причин для создания единой модели драйвера в ядре было позволить пользовательскому пространству управлять деревом ***/dev*** в динамическом стиле. Раньше это было сделано в пользовательском пространстве реализацией **devfs**, но эта кодовая база постепенно сгнила из-за отсутствия активного сопровождающего и некоторых неисправимых базовых ошибок. Несколько разработчиков ядра поняли, что если бы всю информацию устройства экспортировать в пространство пользователя, оно могло бы выполнять всё необходимое управление деревом ***/dev***.

devfs имеет в своём дизайне некоторые весьма существенные недостатки. Она требует от каждого драйвера устройства быть измененным для её поддержки и она требует от драйвера устройства указать имя и местоположение в дереве ***/dev***, где он помещён. Она также не надлежащим образом обрабатывает динамические старшие и младшие номера, заставляя политику именования устройств принадлежать ядру, а не пространству пользователя. Разработчики ядра Linux действительно ненавидят иметь политику в ядре и так как политика именования devfs не следует спецификации Linux Standard Base, это действительно их беспокоит.

С тех пор, как ядро Linux начало устанавливаться на огромных серверах, многие пользователи столкнулись с проблемой, как управлять очень большим количеством устройств. Массивы дисковых накопителей из более 10.000 уникальных устройств представляют очень сложную задачу обеспечения того, чтобы каждый диск всегда был проименован тем же точным именем, где бы он ни был помещен в дисковом массиве или когда он был обнаружен ядром. Эта та же проблема, от которой страдают пользователи настольных компьютеров, пытающиеся подключить два USB принтера к своей системе и затем понимающие, что они не имели возможности обеспечить, чтобы принтер, известный как ***/dev/lpt0***, не был бы изменён и отнесён к другому принтеру в случае перезагрузки системы.

Таким образом, был создан ***udev***. Он опирается на всю информацию устройства, экспортируемую в пользовательское пространство через sysfs и на уведомление через ***/sbin/hotplug***, что устройство было добавлено или удалено. Политические решения, такие, как какое имя дать устройству, могут быть указаны в пространстве пользователя, вне ядра. Это гарантирует, что политика именования удалена из ядра и позволяет большую степень гибкости при именовании каждого устройства.

Для получения дополнительной информации по использованию ***udev*** и как его настроить, пожалуйста, смотрите документацию, которая поставляется включённой в пакет ***udev*** в вашем дистрибутиве.

Всё, что драйверу устройства необходимо сделать, чтобы ***udev*** правильно с ним работал, является обеспечение того, чтобы любые старшие и младшие номера, присвоенные устройству, управляемому драйвером, экспортировались в пользовательское пространство через sysfs. Для любого драйвера, который использует подсистему для присвоения ему старшего и младшего номера, это уже сделано подсистемой и драйвер не должен делать никакой работы. Примерами подсистем, которые делают это, являются подсистемы: tty, misc, usb, input, scsi, block, i2c, network и frame buffer. Если ваш драйвер самостоятельно обрабатывает получение старшего и младшего номера через вызов функции ***cdev_init*** или устаревшей функции ***register_chrdev***, драйвер должен быть изменён, чтобы ***udev*** работал с ним должным образом.

udev ищет в дереве ***/class/*** в sysfs файл с именем **dev**, чтобы определить, какой старший и младший номер присвоен данному устройству, когда оно вызывается ядром через интерфейс ***/sbin/hotplug***. Драйверу устройства просто необходимо создать такой файл для каждого устройства, которым он управляет. Как правило, интерфейс ***class_simple*** - самый простой способ это сделать.

Как уже упоминалось в разделе "Интерфейс class_simple" 373, первым шагом в использовании интерфейса **class_simple** является создание **struct class_simple** с помощью вызова функции ***class_simple_create***:

```
static struct class_simple *foo_class;
...
foo_class = class_simple_create(THIS_MODULE, "foo");
if (IS_ERR(foo_class)) {
    printk(KERN_ERR "Error creating foo class.\n");
    goto error;
}
```

Этот код создаёт каталог в sysfs в ***/sys/class/foo***.

Всякий раз, когда драйвер находит новое устройство и вы присваиваете ему младший номер, как описано в Главе 3 39, драйвер должен вызывать функцию ***class_simple_device_add***:

```
class_simple_device_add(foo_class, MKDEV(FOO_MAJOR, minor), NULL, "foo%d",
minor);
```

Этот код вызывает создание в ***/sys/class/foo*** поддиректории, названной ***fooN***, где ***N*** - младший номер для этого устройства. В этом каталоге создаётся один файл, ***dev***, и это именно то, что необходимо ***udev***, чтобы создать узел устройства для вашего устройства. Когда ваш драйвер освобождается от устройства и вы отказываетесь от младшего номера, который был за ним закреплён, для удаления записи в sysfs для этого устройства необходим вызов ***class_simple_device_remove***:

```
class_simple_device_remove(MKDEV(FOO_MAJOR, minor));
```

Позже, когда весь ваш драйвер выключается, для удаления класса, который вы первоначально создали вызовом ***class_simple_create***, является необходимым вызов ***class_simple_destroy*** :

```
class_simple_destroy(foo_class);
```

Файл ***dev***, который создаётся вызовом ***class_simple_device_add***, состоит из старшего и младшего номера, разделенных символом **:**. Если ваш драйвер не хочет использовать интерфейс **class_simple**, потому что вы хотите предоставить для подсистемы другие файлы внутри каталога класса, используйте функцию ***print_dev_t*** для правильного формата старшего и младшего номера для каждого устройства.

Работа со встроенным программным обеспечением

Как автор драйвера, вы можете столкнуться с устройством, которое должно иметь встроенное программное обеспечение (прошивку), загружаемую в него перед тем, как оно будет функционировать должным образом. Конкуренции во многих частях рынка оборудования

настолько сильна, что даже стоимость небольшой EEPROM для программы управления устройством будет больше, чем производитель готов потратить. Таким образом, встроенное программное обеспечение поставляется на компакт-дисках вместе с оборудованием и операционная система ответственна за передачу программы в само устройство.

Вы можете попытаться решить проблему прошивки такой декларацией, так:

```
static char my_firmware[ ] = { 0x34, 0x78, 0xa4, ... };
```

Однако, такой подход почти наверняка является ошибкой. Кодирование прошивки в драйвере раздувает код драйвера, делает обновление прошивки трудным и весьма вероятно заработать проблемы лицензирования. Весьма маловероятно, что поставщик выпустил образ прошивки под GPL, так что смешивание его с лицензированным под GPL кодом обычно является ошибкой.

По этой причине устройства, содержащие внедрённую прошивку, вряд ли будут приняты в главную ветку ядра или включаться дистрибьюторами Linux.

Интерфейс ядра для встроенного программного обеспечения

Правильным решением является получение прошивки из пользовательского пространства, когда это действительно необходимо. Однако, пожалуйста, сопротивляйтесь искушению попробовать открыть файл, содержащий прошивку, непосредственно из пространства ядра; это подверженная ошибкам операция и она помещает политику (в форма имени файла) в ядро. Вместо этого корректный подход заключается в использовании интерфейса прошивки, который был создан только для этой цели:

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name, struct device
*device);
```

Вызов ***request_firmware*** просит пользовательское пространство найти и предоставить образ прошивки в ядро; мы рассмотрим подробности, как это работает, за минуту. **name** (имя) должно идентифицировать желаемую прошивку; при нормальном использовании это имя файла прошивки, предоставленной производителем. Что-то вроде ***my_firmware.bin*** является типичным. Если прошивка успешно загружена, возвращаемое значение равно **0** (в противном случае, возвращается обычный код ошибки), и аргумент **fw** указывает на одну из таких структур:

```
struct firmware {
    size_t size;
    u8 *data;
};
```

Эта структура содержит актуальную прошивку, которая теперь может быть загружена в устройство. Будьте осторожны, эта прошивка является непроверенными данными из пространства пользователя; вы должны применять любые и все тесты которые можно придумать, чтобы убедить себя, что она является правильным образом прошивки перед отправкой его в оборудование. Прошивка устройства обычно содержит идентификационные строки, контрольные суммы и так далее; проверяйте их все перед тем, как доверять данным.

После того, как вы отправили прошивку в устройство, вы должны освободить структуру в

ядре:

```
void release_firmware(struct firmware *fw);
```

Так как ***request_firmware*** запрашивает для помощи пространство пользователя, это гарантирует сон перед возвратом. Если ваш драйвер не имеет возможности заснуть, когда он должен запросить прошивку, может быть использован асинхронный вариант:

```
int request_firmware_nowait(struct module *module,
            char *name, struct device *device, void *context,
            void (*cont)(const struct firmware *fw, void *context));
```

Дополнительными аргументами здесь являются **module** (который почти всегда будет **THIS_MODULE**), **context** (свой указатель данных, который не используется подсистемой прошивки) и **cont**. Если все пойдёт хорошо, ***request_firmware_nowait*** начинает процесс загрузки прошивки и возвращает 0. В какой-то момент в будущем, будет вызвана ***cont*** с результатом загрузки. Если по каким-то причинам загрузка прошивки не удаётся, **fw** является NULL.

Как это работает

Подсистема прошивки работает с механизмами sysfs и горячего подключения. Когда сделан вызов ***request_firmware***, с использованием имени вашего устройства в ***/sys/class/firmware*** создаётся новый каталог. Этот каталог содержит три атрибута:

loading
: Этот атрибут должен быть установлен в единицу процессом пользовательского пространства, который загружает прошивку. Когда процесс загрузки завершится, он должен быть установлен в **0**. Запись значения **-1** для загрузки прерывает процесс загрузки прошивки.

data
: **data** является бинарным атрибутом, который получает сами данные прошивки. После настройки загрузки, процесс пользовательского пространства должен записать прошивку в этот атрибут.

device
: Этот атрибут является символической ссылкой на соответствующую запись в ***/sys/devices***.

После того, как были созданы записи в sysfs, ядро генерирует для вашего устройства событие горячего подключения. Окружение, передаваемое в обработчик горячего подключения, включает в себя переменную **FIRMWARE**, которая установлена в имя, указанное в ***request_firmware***. Обработчик должен найти файл прошивки и скопировать его в ядро с помощью предоставленных атрибутов. Если файл не может быть найден, обработчик должен установить атрибут **loading** в **-1**.

Если запрос на прошивку не обслуживается в течение 10 секунд, ядро прерывает его и возвращает статус неисполнения драйверу. Этот период ожидания может быть изменён с помощью атрибута ***/sys/class/firmware/timeout*** в sysfs.

Использование интерфейса ***request_firmware*** позволяет вам распространять прошивки с вашим драйвером. При правильной интеграции в механизм горячего подключения, подсистема загрузки прошивки позволяет устройствам работает сразу "из коробки". Это явно лучший способ решения проблемы.

Пожалуйста, будьте к нам снисходительны, однако, так как мы передаём ещё одно предупреждение: прошивка устройства не должна распространяться без разрешения производителя. Многие производители согласны лицензировать свои прошивки на разумных условиях, если их вежливо попросить; некоторые другие могут быть менее сговорчивы. В любом случае, копирование и распространение их прошивки без разрешения является нарушением закона об авторском праве и способом получить неприятности.

Краткая справка

В этой главе было введено множество функций, вот краткая сумма их всех.

Kobject-ы

#include <linux/kobject.h>
Подключает файл, содержащий определения kobjects, связанные с ними структуры и функции.

void kobject_init(struct kobject *kobj);
int kobject_set_name(struct kobject *kobj, const char *format, ...);
Функции для инициализации kobject-а.

struct kobject *kobject_get(struct kobject *kobj);
void kobject_put(struct kobject *kobj);
Функции, которые управляют счётчиком ссылок для kobject-ов.

struct kobj_type;
struct kobj_type *get_ktype(struct kobject *kobj);
Представляет тип структуры, внутрь которой встроен kobject. Используйте **get_ktype** для получения **kobj_type**, связанного с данным kobject-ом.

int kobject_add(struct kobject *kobj);
extern int kobject_register(struct kobject *kobj);
void kobject_del(struct kobject *kobj);
void kobject_unregister(struct kobject *kobj);
kobject_add добавляет kobject в систему, обрабатывая членство в kset, представление в sysfs и генерацию события горячего подключения. ***kobject_register*** является удобной функцией, которая сочетает в себе ***kobject_init*** и ***kobject_add***. Для удаления kobject-а используйте ***kobject_del*** или ***kobject_unregister***, которая объединяет ***kobject_del*** и ***kobject_put***.

void kset_init(struct kset *kset);
int kset_add(struct kset *kset);
int kset_register(struct kset *kset);
void kset_unregister(struct kset *kset);
Функции инициализации и регистрации для kset-ов.

decl_subsys(name, type, hotplug_ops);
Макрос, который упрощает декларацию подсистем.

void subsystem_init(struct subsystem *subsys);
int subsystem_register(struct subsystem *subsys);
void subsystem_unregister(struct subsystem *subsys);
struct subsystem *subsys_get(struct subsystem *subsys)

void subsys_put(struct subsystem *subsys);
Операции над подсистемами.

Операции в sysfs

#include <linux/sysfs.h>
Подключает файл, содержащий декларации для sysfs.
int sysfs_create_file(struct kobject *kobj, struct attribute *attr);
int sysfs_remove_file(struct kobject *kobj, struct attribute *attr);
int sysfs_create_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_remove_bin_file(struct kobject *kobj, struct bin_attribute *attr);
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
void sysfs_remove_link(struct kobject *kobj, char *name);
Функции для создания и удаления атрибутных файлов, связанных с kobject-ом.

Шины, устройства, и драйверы

int bus_register(struct bus_type *bus);
void bus_unregister(struct bus_type *bus);
Функции, которые выполняют регистрации и отмену регистрации шин в модели устройства.
int bus_for_each_dev(struct bus_type *bus, struct device *start, void *data, int (*fn)(struct device *, void *));
int bus_for_each_drv(struct bus_type *bus, struct device_driver *start, void *data, int (*fn)(struct device_driver *, void *));
Функции, которые выполняют итерацию по каждому из устройств и драйверов, соответственно, которые подключены к данной шине.
BUS_ATTR(name, mode, show, store);
int bus_create_file(struct bus_type *bus, struct bus_attribute *attr);
void bus_remove_file(struct bus_type *bus, struct bus_attribute *attr);
Макрос **BUS_ATTR** может быть использован для объявления структуры **bus_attribute**, которая затем может быть добавлена и удалена двумя указанными выше функциями.
int device_register(struct device *dev);
void device_unregister(struct device *dev);
Функции, которые обрабатывают регистрацию устройства.
DEVICE_ATTR(name, mode, show, store);
int device_create_file(struct device *device, struct device_attribute *entry);
void device_remove_file(struct device *dev, struct device_attribute *attr);
Макросы и функции, которые имею дело с атрибутами устройств.
int driver_register(struct device_driver *drv);
void driver_unregister(struct device_driver *drv);
Функции, которые регистрируют и отменяют регистрацию драйвера устройства.
DRIVER_ATTR(name, mode, show, store);
int driver_create_file(struct device_driver *drv, struct driver_attribute *attr);
void driver_remove_file(struct device_driver *drv, struct driver_attribute *attr);
Макросы и функции, которые управляют атрибутами драйвера.

Классы

struct class_simple *class_simple_create(struct module *owner, char *name);

```
void class_simple_destroy(struct class_simple *cs);
struct class_device *class_simple_device_add(struct class_simple *cs,
                dev_t devnum, struct device *device, const char *fmt, ...);
void class_simple_device_remove(dev_t dev);
int class_simple_set_hotplug(struct class_simple *cs,
                int (*hotplug)(struct class_device *dev, char **envp,
                int num_envp, char *buffer, int buffer_size));
```

Функции, реализующие интерфейс ***class_simple***; они управляют простыми записями классов, содержащими атрибут **dev** и небольшое число других. Начиная с версии 2.6.13 данный интерфейс в ядрах больше не присутствует.

```
int class_register(struct class *cls);
void class_unregister(struct class *cls);
```

Регистрация и отмена регистрации классов.

```
CLASS_ATTR(name, mode, show, store);
int class_create_file(struct class *cls, const struct class_attribute *attr);
void class_remove_file(struct class *cls, const struct class_attribute *attr);
```

Обычные макросы и функции для работы с атрибутами классов.

```
int class_device_register(struct class_device *cd);
void class_device_unregister(struct class_device *cd);
int class_device_rename(struct class_device *cd, char *new_name);
CLASS_DEVICE_ATTR(name, mode, show, store);
int class_device_create_file(struct class_device *cls, const struct
class_device_attribute *attr);
void class_device_remove_file(struct class_device *cls, const struct
class_device_attribute *attr);
```

Функции и макросы, которые реализуют интерфейс класса устройств.

```
int class_interface_register(struct class_interface *intf);
void class_interface_unregister(struct class_interface *intf);
```

Функции, которые добавляют к классу интерфейс (или удаляют его).

Встроенное программное обеспечение

```
#include <linux/firmware.h>
int request_firmware(const struct firmware **fw, char *name, struct device
*device);
int request_firmware_nowait(struct module *module, char *name,
                struct device *device, void *context,
                void (*cont)(const struct firmware *fw, void *context));
void release_firmware(struct firmware *fw);
```

Функции, которые реализуют интерфейс ядра для загрузки встроенного программного обеспечения (прошивки).

Глава 15, Отображение памяти и DMA

В этой главе углубляется в область управления памятью в Linux, с акцентом на методах, которые являются полезными для автора драйвера устройства. Многие типы программирования драйвера требуют некоторого понимания того, как работает виртуальная подсистема памяти; материал, который мы рассмотрим в этой главе не раз пригодится, когда мы перейдём к некоторым из наиболее сложных и критических по производительности подсистемам. Подсистема виртуальной памяти является также весьма интересной частью основного ядра Linux и, следовательно, она заслуживает внимания.

Материал в этой главе состоит из трёх разделов:

- Первый рассматривает реализацию системного вызова ***mmap***, который позволяет отображение памяти устройства непосредственно в адресное пространство пользовательского процесса. Не все устройства требуют поддержки ***mmap***, однако, для некоторых отображение памяти устройства может дать значительный прирост производительности.
- Затем мы посмотрим на пересечение границы с другой стороны, обсуждая прямой доступ к страницам пользовательского пространства. Такая возможность необходима сравнительно небольшому числу драйверов; во многих случаях ядро выполняет такого рода отображение, когда драйвер даже не подозревает об этом. Но понимание того, как отображать память пользовательского пространства в ядро (используя ***get_user_pages***) может быть полезным.
- Заключительный раздел рассматривает операции ввода/вывода с прямым доступом к памяти (DMA), которые обеспечивают периферии прямой доступ к системной памяти. Конечно, все эти методы требуют понимания того, как в Linux работает управление памятью, поэтому мы начнём с обзора этой подсистемы.

Управление памятью в Linux

Вместо того, чтобы описывать теорию управления памятью в операционных системах, этот раздел попытается выявить основные особенности её реализации в Linux. Хотя вам не требуется быть гуру в виртуальный памяти Linux для выполнения ***mmap***, основной обзор того, как всё это работает, полезен. Далее следует довольно пространное описание структур данных, используемых ядром для управления памятью. После получения необходимых

предварительных знаний мы сможем приступить к работе с этими структурами.

Типы адресов

Linux, конечно же, система с виртуальной памятью, а это означает, что адреса, видимые пользовательскими программами, не соответствуют напрямую физическим адресам, используемым оборудованием. Виртуальная память вводит слой косвенности, который позволяет ряд приятных вещей. С виртуальной памятью программы, выполняющиеся в системе, могут выделить гораздо больше памяти, чем доступно физически; более того, даже один процесс может иметь виртуальное адресное пространство больше физической памяти системы. Виртуальная память позволяет также программе использовать разные ухищрения с адресным пространством процесса, в том числе отображение памяти программы в память устройства.

До сих пор мы говорили о виртуальных и физических адресах, но подробности умалчивались. Система Linux имеет дело с несколькими типами адресов, каждый со своей собственной семантикой. К сожалению, в коде ядра не всегда чётко понятно, какой именно тип адреса используется в каждой ситуации, так что программист должен быть осторожным.

Ниже приведён список типов адресов используемых в Linux. Рисунок 15-1 показывает, как эти типы адресов связаны с физической памятью.

Пользовательские виртуальные адреса

Это обычные адреса, видимые программами пространства пользователя. Пользовательские адреса имеют размерность 32 или 64 бита, в зависимости от архитектуры используемого оборудования, и каждый процесс имеет своё собственное виртуальное адресное пространство.

Физические адреса

Адреса, используемые между процессором и памятью системы. Физические адреса 32-х или 64-х разрядные; даже 32-х разрядные системы могут использовать большие физические адреса в некоторых ситуациях.

Адреса шин

Адреса, используемые между периферийными шинами и памятью. Зачастую они такие же, как физические адреса, используемые процессором, но это не обязательно так. Некоторые архитектуры могут предоставить блок управления памятью ввода/вывода (I/O memory management unit, IOMMU), который переназначает адреса между шиной и оперативной памяти. IOMMU может сделать жизнь легче несколькими способами (делая разбросанный в памяти буфер выглядящим непрерывным для устройства, например), но программирование IOMMU является дополнительным шагом, который необходимо выполнить при настройке DMA операций. Конечно, адреса шин сильно зависят от архитектуры.

Логические адреса ядра

Они составляют обычное адресное пространство ядра. Эти адреса отображают какую-то часть (возможно, всю) основной памяти и часто рассматриваются, как если бы они были физическими адресами. На большинстве архитектур логические адреса и связанные с ними физические адреса отличаются только на постоянное смещение. Логические адреса используют родной размер указателя оборудования и, следовательно, могут быть не в

состоянии адресовать всю физическую память на 32-х разрядных системах, оборудованных в большей степени. Логические адреса обычно хранятся в переменных типа **unsigned long** или **void *** . Память, возвращаемая ***kmalloc***, имеет логический адрес ядра.

Виртуальные адреса ядра

Виртуальные адреса ядра похожи на логические адреса в том, что они являются отображением адреса пространства ядра на физический адрес. Однако, виртуальные адреса ядра не всегда имеют линейную, взаимно-однозначную связь с физическими адресами, которая характеризует логическое адресное пространство. Все логические адреса ***являются*** виртуальными адресами ядра, но многие виртуальные адреса ядра не являются логическими адресами. Так, например, память, выделенная ***vmalloc***, имеет виртуальный адрес (но без прямого физического отображения). Функция ***kmap*** (описываемая далее в этой главе) также возвращает виртуальные адреса. Виртуальные адреса обычно хранятся в переменных указателей.

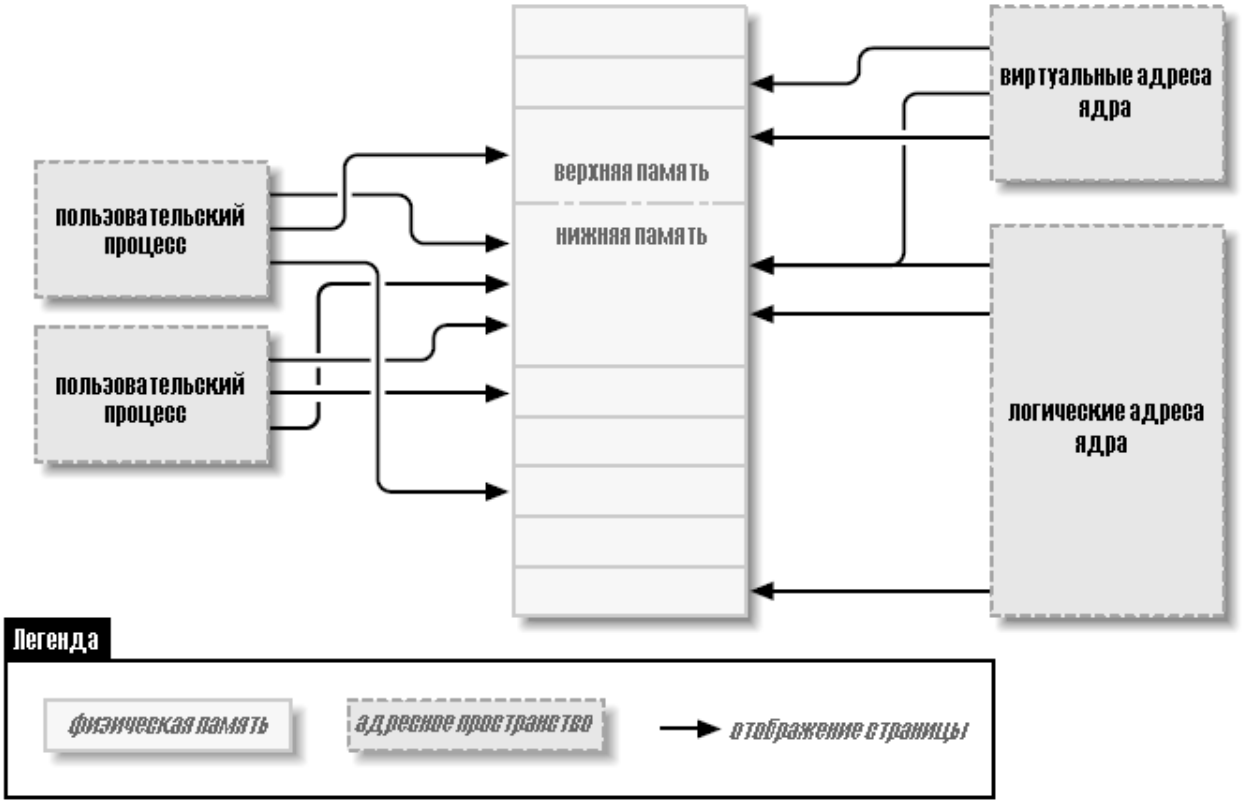


Рисунок 15-1. Типы адресов, используемые в Linux

Если у вас есть логический адрес, макрос ***__pa( )*** (определённый в ***<asm/page.h>***) возвращает соответствующий ему физический адрес. Физические адреса могут быть преобразованы обратно в логические адреса с помощью ***__va( )***, но только для нижних страниц памяти.

Различные функции ядра требуются разные типы адресов. Было бы неплохо, если бы были определены различные типы Си, так, чтобы необходимый тип адреса был бы явным, но этого нет. В этой главе мы стараемся ясно указывать, какой тип адресов где используется.

Физические адреса и страницы

Физическая память разделена на отдельные модули, называемые ***страницами***. Значительная часть внутренней системной обработки памяти производится на постраничной основе. Размер страницы варьируется от одной архитектуры к другой, хотя большинство систем в настоящее время использует страницы по 4096 байт. Постоянная **PAGE_SIZE** (определённая в ***<asm/page.h>***) задаёт размер страницы для любой архитектуры.

Если вы посмотрите на адрес памяти, виртуальный или физический, он делится на номер страницы и смещение внутри этой страницы. Например, если используются страницы по 4096 байт, 12 младших значащих бит являются смещением, а остальные, старшие биты, указывают номер страницы. Если отказаться от смещения и сдвинуть оставшуюся часть адреса вправо, результат называют ***номером страничного блока*** (page frame number, PFN). Сдвиг битов для конвертации между номером страничного блока и адресами является довольно распространённой операцией; макрос **PAGE_SHIFT** сообщает, сколько битов должны быть смещены для выполнения этого преобразования.

Верхняя и нижняя память

Разница между логическими и виртуальными адресами ядра хорошо видна на 32-х разрядных системах, которые оборудованы большими объёмами памяти. Используя 32 бита можно адресовать 4 ГБ памяти. Однако, Linux на 32-х разрядных системах до недавнего времени был ограничен значительно меньшей памятью, чем это, из-за способа инициализации виртуального адресного пространства.

Ядро (на архитектуре x86, в конфигурации по умолчанию) разделяет 4 ГБ виртуальное адресное пространство между пространством пользователя и ядром; в обоих контекстах используется один и тот же набор отображений. Типичное разделение выделяет 3 Гб для пространства пользователя и 1 ГБ для пространства ядра. (* Многие не-x86 архитектуры способны эффективно обходиться без описанного здесь разделения на ядро/ пользовательское пространство, так что они могут работать с адресным пространством ядра до 4 ГБ на 32-х разрядных системах. Однако, ограничения, описанные в этом разделе, до сих пор относятся к таким системам, где установлено более 4 ГБ оперативной памяти.) Код ядра и структуры данных должны вписываться в это пространство, но самым большим потребителем адресного пространства ядра является виртуальное отображение на физическую память. Ядро не может напрямую управлять памятью, которая не отображена в адресное пространство ядра. Ядру, другими словами, необходим свой виртуальный адрес для любой памяти, с которой оно должно непосредственно соприкасаться. Таким образом, на протяжении многих лет, максимальным объёмом физической памяти, которая могла быть обработана ядром, было значение, которое могло быть отображено в часть для ядра виртуального адресного пространства, минус пространство, необходимое для самого кода ядра. В результате, базирующиеся на x86 системы Linux могли работать максимально с немногим менее 1 ГБ физической памяти.

В ответ на коммерческое давление, чтобы поддержать больше памяти, не нарушая в то же время работу 32-х разрядных приложений и совместимость системы, производители процессоров добавили в свои продукты функцию "расширение адреса". Результатом является то, что во многих случаях даже 32-х разрядные процессоры могут адресовать более 4 ГБ физической памяти. Однако, ограничение на объём памяти, которая может быть непосредственно связана с логическими адресами, остаётся. Только самая нижняя часть памяти (до 1 или 2 ГБ, в зависимости от оборудования и конфигурации ядра) имеет логические

адреса; (* Ядро версии 2.6 (с дополнительным патчем) может поддерживать на архитектуре x86 режим "4G/4G", который разрешает большие виртуальные адресные пространства ядра и пользовательского пространства при умеренных затратах производительности.) остальная (верхняя память) не имеет. Перед доступом к заданной странице верхней памяти ядро должно установить явное виртуальное соответствие, чтобы сделать эту страницу доступной в адресном пространстве ядра. Таким образом, многие структуры данных ядра должны быть размещены в нижней памяти; верхняя память имеет тенденцию быть зарезервированной для страниц процесса пространства пользователя.

Термин "верхняя память" может ввести некоторых в заблуждение, особенно поскольку он имеет другое значение в мире персональных компьютеров. Таким образом, чтобы внести ясность, мы определим здесь эти термины:

Нижняя память
: Память, для которой существуют логические адреса в пространстве ядра. Почти на каждой системе, с которой вы скорее всего встретитесь, вся память является нижней памятью.

Верхняя память
: Память, для которой логические адреса не существуют, потому что она выходит за рамки диапазона адресов, отведённых для виртуальных адресов ядра.

На системах i386 граница между нижней и верхней памятью обычно установлена на уровне только до 1 Гб, хотя эта граница может быть изменена во время конфигурации ядра. Эта граница не связана никаким образом со старым ограничением 640 Кб, имеющимся на оригинальном ПК, и её размещение не продиктовано оборудованием. Напротив, это предел, установленный в самом ядре, так как он разделяет 32-х разрядное адресное пространство между ядром и пространством пользователя.

Мы укажем на ограничения на использование верхней памяти, когда мы подойдём к ним в этой главе.

Карта памяти и структура page

Исторически сложилось, что для обращения к страницам физической памяти ядро использует логические адреса. Однако, добавление поддержки верхней памяти выявило очевидную проблему с таким подходом - логические адреса не доступны для верхней памяти.

Таким образом, функции ядра, которые имеют дело с памятью, вместо этого всё чаще используют указатели на **struct page** (определённой в ***<linux/mm.h>***). Эта структура данных используется для хранения информации практически обо всём, что ядро должно знать о физической памяти; для каждой физической страницы в системе существует одна **struct page**. Некоторые из полей этой структуры включают следующее:

atomic_t count;
: Число существующих ссылок на эту страницу. Когда количество падает до 0, страница возвращается в список свободных страниц.

void *virtual;
: Виртуальный адрес страницы ядра, если она отображена; в противном случае NULL. Страницы нижней памяти отображаются всегда; страницы верхней памяти - обычно нет.

Это поле появляется не на всех архитектурах; оно обычно компилируется только тогда, когда виртуальный адрес страницы ядра не может быть легко вычислен. Если вы хотите посмотреть на это поле, правильный метод заключается в использовании макроса ***page_address***, описанного ниже.

unsigned long flags;
Набор битовых флагов, характеризующих состояние этой странице. К ним относятся **PG_locked**, который указывает, что страница была заблокирована в памяти, и **PG_reserved**, который препятствует тому, чтобы система управления памятью вообще работала с этой страницей.

Внутри **struct page** существует гораздо больше информации, но она является частью более глубокой чёрной магии управления памятью и не представляет интерес для авторов драйверов.

Ядро поддерживает один или несколько массивов записей **struct page**, которые позволяют отслеживать всю физическую память системы. На некоторых системах имеется единственный массив, называемый **mem_map**. На других системах, однако, ситуация более сложная. Системы с неоднородным доступом к памяти (nonuniform memory access, NUMA) и другие с сильно разделённой физической памятью могут иметь более одного массива карты памяти, поэтому код, который предназначен для переносимости, должен избегать прямого доступа к массиву, когда это возможно. К счастью, как правило, довольно легко просто работать с указателями **struct page** не беспокоясь о том, откуда они берутся.

Некоторые функции и макросы, определённые для перевода между указателями **struct page** и виртуальными адресами:

struct page *virt_to_page(void *kaddr);
Этот макрос, определённый в ***<asm/page.h>***, принимает логический адрес ядра и возвращает связанный с ним указатель **struct page**. Так как он требует логического адрес, он не работает с памятью от ***vmalloc*** или верхней памятью.
struct page *pfn_to_page(int pfn);
Возвращает указатель **struct page** для заданного номера страничного блока. При необходимости он проверяет номер страничного блока на корректность с помощью ***pfn_valid*** перед его передачей в ***pfn_to_page***.
void *page_address(struct page *page);
Возвращает виртуальный адрес ядра этой страницы, если такой адрес существует. Для верхней памяти этот адрес существует, только если страница была отображена. Эта функция определена в ***<linux/mm.h>***. В большинстве случаев вы захотите использовать версию ***kmap***, а не ***page_address***.

#include <linux/highmem.h>
void *kmap(struct page *page);
void kunmap(struct page *page);
kmap возвращает виртуальный адрес ядра для любой страницы в системе. Для страниц нижней памяти она просто возвращает логический адрес страницы; для страниц верхней памяти ***kmap*** создаёт специальное отображение в предназначенной для этого части адресного пространства ядра. Отображения, созданные ***kmap***, всегда должны быть освобождены с помощью ***kunmap***; доступно ограниченное число таких отображений, так что лучше не удерживать их слишком долго. Вызовы ***kmap*** поддерживают счётчик, так что если две или более функции вызывают ***kmap*** на той же странице, всё работает

правильно. Отметим также, что ***kmap*** может заснуть, если отображение недоступно.

```
#include <linux/highmem.h>
#include <asm/kmap_types.h>
void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
```

kmap_atomic является высокопроизводительной формой ***kmap***. Каждая архитектура поддерживает небольшой список слотов (специализированные записи таблицы страниц) для атомарных kmap-ов; вызывающий ***kmap_atomic*** должен сообщить системе в аргументе **type**, какой из этих слотов использовать. Единственными слотами, которые имеют смысл для драйверов, являются **KM_USER0** и **KM_USER1** (для кода, работающего непосредственно из вызова из пользовательского пространства), и **KM_IRQ0** и **KM_IRQ1** (для обработчиков прерываний). Обратите внимание, что атомарные kmap-ы должны быть обработаны атомарно; ваш код не может спать, удерживая её. Отметим также, что ничто в ядре не предохраняет две функции от попытки использовать тот же слот и помешать друг другу (хотя для каждого процессора имеется уникальный набор слотов). На практике, конкуренция для атомарных слотов kmap, кажется, не будет проблемой.

Мы увидим варианты использования этих функций, когда перейдём к примеру кода далее в этой главе, и в последующих главах.

Таблицы страниц

В любой современной системе процессор должен иметь механизм для трансляции виртуальных адресов в соответствующие физические адреса. Этот механизм называется ***таблицей страниц***; по существу, это многоуровневый древовидный массив, содержащий отображения виртуального к физическому и несколько связанных флагов. Ядро Linux поддерживает набор таблиц страниц даже на архитектурах, которые не используют такие таблицы напрямую.

Многие операции, обычно выполняемые драйверами устройств, могут включать манипуляции таблицами страниц. К счастью для автора драйвера, ядро версии 2.6 ликвидировало всю необходимость непосредственно работать с таблицами страниц. В результате, мы не описываем их детально; любопытные читатели могут захотеть взглянуть на ***Understanding The Linux Kernel*** от Daniel P. Bovet и Marco Cesati (O'Reilly) для полной информации.

Области виртуальной памяти

Область виртуальной памяти (virtual memory area, VMA) представляет собой структуру данных ядра, используемую для управления различными регионами адресного пространства процесса. VMA представляет собой однородный регион в виртуальной памяти процесса: непрерывный диапазон виртуальных адресов, которые имеют одинаковые флаги разрешения и созданы одним и тем же объектом (скажем, файлом, или пространством для своппинга). Она примерно соответствует концепции "сегмента", хотя это лучше описывается как "объект памяти со своими свойствами". Карта памяти процесса состоит (по крайней мере) из следующих областей:

- Область для исполняемого кода программы (часто называемого текстом).
- Несколько областей для данных, в том числе проинициализированные данные (те,

которые имеют явно заданные значения в начале исполнения), неинициализированные данных (BSS), (* имя *BSS* является историческим пережитком старого ассемблерного оператора, означающего "блок, начатый символом" (“block started by symbol”). Сегмент BSS исполняемых файлов не сохраняется на диске и ядро отображает нулевую страницу в диапазоне адресов BSS.) и программный стек.
- По одной области для каждого активного отображения памяти.

Области памяти процесса можно увидеть, посмотрев в ***/proc/<pid>/maps*** (где ***pid***, конечно, заменяется на ID процесса). ***/proc/self*** является особым случаем ***/proc/pid***, потому что он всегда обращается к текущему процессу. Вот, например, несколько карт памяти (к которым мы курсивом добавили краткие комментарии):

```
# cat /proc/1/maps look at init
08048000-0804e000 r-xp 00000000 03:01 64652    /sbin/init  text
0804e000-0804f000 rw-p 00006000 03:01 64652    /sbin/init  data
0804f000-08053000 rwxp 00000000 00:00 0        zero-mapped BSS
40000000-40015000 r-xp 00000000 03:01 96278    /lib/ld-2.3.2.so   text
40015000-40016000 rw-p 00014000 03:01 96278    /lib/ld-2.3.2.so   data
40016000-40017000 rw-p 00000000 00:00 0        BSS for ld.so
42000000-4212e000 r-xp 00000000 03:01 80290    /lib/tls/libc-2.3.2.so   text
4212e000-42131000 rw-p 0012e000 03:01 80290    /lib/tls/libc-2.3.2.so   data
42131000-42133000 rw-p 00000000 00:00 0        BSS for libc
bffff000-c0000000 rwxp 00000000 00:00 0        Stack segment
ffffe000-fffff000 ---p 00000000 00:00 0        vsyscall page

# rsh wolf cat /proc/self/maps  #### x86-64 (trimmed)
00400000-00405000 r-xp 00000000 03:01 1596291     /bin/cat    text
00504000-00505000 rw-p 00004000 03:01 1596291     /bin/cat    data
00505000-00526000 rwxp 00505000 00:00 0 bss
3252200000-3252214000 r-xp 00000000 03:01 1237890 /lib64/ld-2.3.3.so
3252300000-3252301000 r--p 00100000 03:01 1237890 /lib64/ld-2.3.3.so
3252301000-3252302000 rw-p 00101000 03:01 1237890 /lib64/ld-2.3.3.so
7fbfffe000-7fc0000000 rw-p 7fbfffe000 00:00 0                  stack
ffffffffff600000-ffffffffffe00000 ---p 00000000 00:00 0        vsyscall
```

Поля в каждой строке:

```
start-end perm offset major:minor inode image
```

Каждое поле в ***/proc/*/maps*** (за исключением имени отображения) соответствует полю в структуре **vm_area_struct**:

start
end
Начало и окончание виртуальных адресов для этой области памяти.

perm
Битовая маска с разрешениями для области памяти на чтение, запись и исполнение. Это поле описывает, что процессу разрешено делать со страницами, которые принадлежат этой области. Последний символ в поле - это либо **p** для "закрытых" (“private”), или **s** для "общих" (“shared”).

offset
Где начинается область памяти в файле, с которым она связана. Смещение 0 означает, что начало области памяти соответствует началу файла.

major
minor
Старший и младший номера устройства удерживающего файл, который был на отображён. Как ни странно, при отображении устройства, старший и младший номера ссылаются на дисковый раздел, содержащий специальный файл устройства, который был открыт пользователем, а не самого устройства.

inode
Номер inode отображённого файла.

image
Имя файла (обычно это исполняемый файл), который был отображён.

Структура vm_area_struct

Когда процесс пользовательского пространства вызывает ***mmap***, чтобы отобразить память устройства в его адресное пространство, система реагирует, создавая новую VMA для предоставления этого отображения. Драйвер, который поддерживает ***mmap*** (и, таким образом, который реализует метод ***mmap***), должен помочь такому процессу завершая инициализацию этой VMA. Автор драйвера должен, следовательно, иметь по крайней мере минимальное понимание VMA для поддержки ***mmap***.

Давайте посмотрим на наиболее важные поля в **struct vm_area_struct** (определённой в ***<linux/mm.h>***). Эти поля могут быть использованы драйверами устройств в их реализации ***mmap***. Заметим, что ядро поддерживает списки и деревья VMA для оптимизации области поиска и некоторые поля **vm_area_struct** используются для поддержки такой организации. Поэтому VMA не могут быть созданы по желанию драйвера, или эти структуры нарушатся. Основными полями VMA являются следующие (обратите внимание на сходство между этими полями и выводом ***/proc***, который мы только что видели):

unsigned long vm_start;
unsigned long vm_end;
Диапазон виртуальных адресов, охватываемый этой VMA. Эти поля являются первыми двумя полями, показываемыми в ***/proc/*/maps***.

struct file *vm_file;
Указатель на структуру **struct file**, связанную с этой областью (если таковая имеется).

unsigned long vm_pgoff;
Смещение области в файле, в страницах. Когда отображается файл или устройство, это позиция в файле первой страницы, отображённой в эту область.

unsigned long vm_flags;
Набор флагов, описывающих эту область. Флагами, представляющими наибольший интерес для автора драйвера устройства, являются **VM_IO** и **VM_RESERVED**. **VM_IO** отмечает VMA как являющийся отображённым на память регион ввода/вывода. Среди прочего, флаг **VM_IO** мешает региону быть включенным в дампы процессов ядра.

VM_RESERVED сообщает системе управления памятью не пытаться выгрузить эту VMA; он должен быть установлен в большинстве отображений устройства.

struct vm_operations_struct *vm_ops;
Набор функций, которые ядро может вызывать для работы в этой области памяти. Его присутствие свидетельствует о том, что область памяти является "объектом" ядра, как и **struct file**, которую мы используем везде в этой книге.

void *vm_private_data;
Области, которые могут быть использованы драйвером для сохранения своей собственной информации.

Как и **struct vm_area_struct**, **vm_operations_struct** определена в ***<linux/mm.h>***; она включает операции, перечисленные ниже. Эти операции являются единственными необходимыми для обработки потребностей процесса в памяти и они перечислены в том порядке, как они объявлены. Далее в этой главе реализованы некоторые из этих функций.

void (*open)(struct vm_area_struct *vma);
Метод ***open*** вызывается ядром, чтобы разрешить подсистеме реализации VMA проинициализировать эту область. Этот метод вызывается в любое время, когда создаётся новая ссылка на эту VMA (например, когда процесс разветвляется). Единственное исключение происходит, когда VMA создана впервые через ***mmap***; в этом случае вместо этого вызывается метод драйвера ***mmap***.

void (*close)(struct vm_area_struct *vma);
При удалении области ядро вызывает эту операцию ***close***. Обратите внимание, что нет счётчика использований, связанного с VMA; область открывается и закрывается ровно один раз каждым процессом, который её использует.

struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int *type);
Когда процесс пытается получить доступ к странице, которая относится к действительной VMA, но которая в настоящее время не в памяти, для соответствующей области вызывается метод ***nopage*** (если он определён). Метод возвращает указатель **struct page** для физической страницы после того как, может быть, прочитал его из вторичного хранилища. Если для этой области метод ***nopage*** не определён, ядром выделяется пустая страница.

int (*populate)(struct vm_area_struct *vm, unsigned long address, unsigned long len,
pgprot_t prot, unsigned long pgoff, int nonblock);
Этот метод позволяет ядру "повредить" страницы в памяти, прежде чем они будут доступны пользовательскому пространству. Вообще-то, драйверу нет необходимости реализовывать метод ***populate***.

Карта памяти процесса

Последней частью головоломки управления памятью является структура карты памяти процесса, которая удерживает все другие структуры данных вместе. Каждый процесс в системе (за исключением нескольких вспомогательных потоков пространства ядра) имеет **struct mm_struct** (определённую в ***<linux/sched.h>***), которая содержит список виртуальных

областей памяти процесса, таблицы страниц и другие разные биты информации управления домашним хозяйством памяти вместе с семафором (**mmap_sem**) и спин-блокировкой (**page_table_lock**). Указатель на эту структуру можно найти в структуре задачи; в редких случаях, когда драйверу необходим к ней доступ, обычным способом является использование **current->mm**. Обратите внимание, что структура управления памятью может быть разделяемой между процессами; к примеру, таким образом работает реализация потоков в Linux.

На том мы завершаем обзор структур данных управления памятью в Linux. Покончив с этим, мы можем теперь приступить к реализации системного вызова ***mmap***.

Операция устройства mmap

Отображение памяти является одним из наиболее интересных особенностей современных систем UNIX. Что касается драйверов, отображение памяти может быть реализовано для предоставления пользовательским программам прямого доступа к памяти устройства.

Характерный пример использования ***mmap*** можно увидеть, посмотрев на подмножество виртуальных областей памяти для сервера X Window System:

```
cat /proc/731/maps
000a0000-000c0000 rwxs 000a0000 03:01 282652      /dev/mem
000f0000-00100000 r-xs 000f0000 03:01 282652      /dev/mem
00400000-005c0000 r-xp 00000000 03:01 1366927     /usr/X11R6/bin/Xorg
006bf000-006f7000 rw-p 001bf000 03:01 1366927     /usr/X11R6/bin/Xorg
2a95828000-2a958a8000 rw-s fcc00000 03:01 282652  /dev/mem
2a958a8000-2a9d8a8000 rw-s e8000000 03:01 282652  /dev/mem
...
```

Полный список областей VMA X сервера является длинным, но большинство из записей не представляют здесь интереса. Мы видим, однако, четыре отдельных отображения ***/dev/mem***, которые дают некоторое понимание, как X сервер работает с видеокартой. Первым отображением является a0000, который является стандартным местом для видеопамяти в 640-Кб дыре ISA. Далее вниз мы видим большое отображение в e8000000, адрес которого выше самой высокого адреса ОЗУ в системе. Это является прямым отображением видео памяти на адаптер.

Эти регионы можно также увидеть в ***/proc/iomem***:

```
000a0000-000bffff : Video RAM area
000c0000-000cffff : Video ROM
000d1000-000d1fff : Adapter ROM
000f0000-000fffff : System ROM
d7f00000-f7efffff : PCI Bus #01
  e8000000-efffffff : 0000:01:00.0
fc700000-fcffffff : PCI Bus #01
  fcc00000-fcc0ffff : 0000:01:00.0
```

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. В данном примере X сервера, использование ***mmap*** позволяет быстро и легко обращаться к памяти видеокарты. Для критичных к производительности приложений, таких как это, прямой доступ имеет большое

значение.

Как вы могли бы подозревать, не каждое устройство поддаётся абстракции ***mmap***; это не имеет смысла, например, для последовательных портов и других поточно-ориентированных устройств. Другим ограничением ***mmap*** является то, что отображение разделено на **PAGE_SIZE**. Ядро может управлять виртуальными адресами только на уровне таблиц страниц; таким образом, отображённая область должна быть кратной **PAGE_SIZE** и должна находиться в физической памяти начиная с адреса, который кратен **PAGE_SIZE**. Ядро управляет размером разбиения делая регион немного больше, если его размер не является кратным размеру страницы.

Эти ограничения не являются большим препятствием для драйверов, потому что программа в любом случае обращается к устройству зависящим от устройства способом. Поскольку программа должна знать о том, как работает устройство, программист не слишком обеспокоен необходимостью следить за деталями выравнивания страниц. Существует большое ограничение, когда на некоторых не-x86 платформах используются ISA устройства, потому что их аппаратное представление ISA может не быть непрерывным. Например, некоторые компьютеры Alpha видят ISA память как разбросанный набор 8-ми, 16-ти, или 32-х разрядных объектов без прямого отображения. В таких случаях вы не можете использовать ***mmap*** совсем. Неспособность выполнять прямое отображение адресов ISA в адреса Alpha связано с несовместимыми спецификациями передачи данных этих двух систем. В то время, как ранние процессоры Alpha могли выдавать только 32-х разрядные и 64-х разрядные обращения к памяти, ISA может делать только 8-ми разрядные и 16-ти разрядные передачи, и нет никакого способа для прозрачной связи одного протокола с другим.

Существуют веские преимущества использования ***mmap***, когда это возможно сделать. Например, мы уже видели X сервер, который передаёт большой объём данных в и из видеопамяти; отображение графического дисплея в пространство пользователя значительно улучшает пропускную способность, в противоположность реализации ***lseek/write***. Ещё одним типичным примером является программа управления PCI устройством. Большинство PCI периферии отображает их управляющие регистры на адреса памяти и высокопроизводительные приложения, возможно, предпочтут иметь прямой доступ к регистрам, вместо того, чтобы постоянно вызывать ***ioctl*** для выполнения этой работы.

Метод ***mmap*** является частью структуры **file_operations** и вызывается, когда происходит системный вызов ***mmap***. В случае с ***mmap***, ядро выполняет много работы перед фактическим вызовом метода и, следовательно, прототип метода сильно отличается от системного вызова. Это является отличием от таких вызовов, как ***ioctl*** и ***poll***, где до вызова метода ядро не делает ничего.

Системный вызов объявлен следующим образом (как описано на странице руководства ***mmap(2)***):

```
mmap (caddr_t addr, size_t len, int prot, int flags, int fd, off_t offset)
```

С другой стороны, файловая операция объявлена следующим образом:

```
int (*mmap) (struct file *filp, struct vm_area_struct *vma);
```

Аргумент **filp** в методе такой же, как представленный в Главе 3 39, а **vma** содержит информацию о диапазоне виртуальных адресов, которые используются для доступа к устройству. Таким образом, большая часть работы выполнена ядром; для реализации ***mmap***

драйверу необходимо только построить подходящие таблицы страниц для диапазона адресов и, если необходимо, заменить **vma->vm_ops** новым набором операций.

Есть два способа построения таблиц страниц: сделать всё это единожды с помощью функции, названной ***remap_pfn_range***, или делать по странице за раз, через метод VMA ***nopage***. Каждый метод имеет свои преимущества и недостатки. Начнём с подхода "все сразу", который является более простым. После этого мы добавим осложнения, необходимые для настоящей реализации.

Использование remap_pfn_range

Работа по созданию новых таблиц страниц для отображения диапазона физических адресов выполняется ***remap_pfn_range*** и ***io_remap_page_range***, которые имеют следующие прототипы:

```
int remap_pfn_range(struct vm_area_struct *vma,
                    unsigned long virt_addr, unsigned long pfn,
                    unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma,
                        unsigned long virt_addr, unsigned long phys_addr,
                        unsigned long size, pgprot_t prot);
```

Значение, возвращаемое функцией, является обычным 0 или отрицательным кодом ошибки. Давайте посмотрим на точное значение аргументов функций:

vma
Область виртуальной памяти, в которую сейчас отображается диапазон страниц.

virt_addr
Пользовательский виртуальный адрес, по которому должно начаться переназначение. Функция строит таблицы страниц для виртуального диапазона адресов между **virt_addr** и **virt_addr+size**.

pfn
Номер страничного блока, соответствующий физическому адресу, с которым должен быть связан виртуальный адрес. Номер страничного блока - это просто физический адрес, сдвинутый вправо на **PAGE_SHIFT** бит. Для большинства применений поле **vm_pgoff** структуры VMA содержит в точности необходимое вам значение. Функция оказывает влияние на физические адреса от **(pfn<<PAGE_SHIFT)** до **(pfn<<PAGE_SHIFT)+size**.

size
Размер в байтах области для переназначения.

prot
"Защита", требуемая для новой VMA. Драйвер может (и должен) использовать значение, находящееся в **vma->vm_page_prot**.

Аргументы для ***remap_pfn_range*** довольно просты и большинство из них уже вам предоставлены в VMA при вызове вашего метода ***mmap***. Однако, вам может быть интересно, почему есть две функции. Первая (***remap_pfn_range***) предназначена для ситуаций, когда **pfn**

ссылается на фактическое ОЗУ системы, а ***io_remap_page_range*** следует использовать, когда **phys_addr** указывает на память ввода/вывода. На практике эти две функции идентичны на всех архитектурах, кроме SPARC, и в большинстве ситуаций вы увидите использование ***remap_pfn_range***. Однако, в интересах написания переносимых драйверов, вы должны использовать тот вариант ***remap_pfn_range***, который подходит для вашей индивидуальной ситуации.

Другая сложность связана с кэшированием: как правило, ссылки на память устройства не следует кэшировать процессором. Часто системная BIOS настраивает всё должным образом, но также возможно запретить кэширование определённых VMA через поле защиты. К сожалению, отключение кэширования на этом уровне весьма зависимо от процессора. Возможно, любопытный читатель пожелает взглянуть на функцию ***pgprot_noncached*** из ***drivers/char/mem.c***, чтобы увидеть как это происходит. Мы больше не будем здесь обсуждать эту тему.

Простая реализация

Если вашему драйверу необходимо сделать простое, линейное отображение памяти устройства в адресное пространство пользователя, ***remap_pfn_range*** является почти всем, что вам действительно необходимо, чтобы сделать эту работу. Следующий код взят из ***drivers/char/mem.c*** и показывает, как выполняется эта задача в типичном модуле, названном ***simple*** (Simple Implementation Mapping Pages with Little Enthusiasm, простая реализация отображения страниц без особого энтузиазма):

```
static int simple_remap_mmap(struct file *filp, struct vm_area_struct *vma)
{
    if (remap_pfn_range(vma, vma->vm_start, vm->vm_pgoff,
                vma->vm_end - vma->vm_start,
                vma->vm_page_prot))
        return -EAGAIN;
    vma->vm_ops = &simple_remap_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

Как вы можете видеть, переназначение памяти - это просто вопрос вызова ***remap_pfn_range*** для создания необходимых таблиц страниц.

Добавление операций VMA

Как мы уже видели, структура **vm_area_struct** содержит набор операций, которые могут быть применены к VMA. Теперь мы рассмотрим простой способ обеспечения этих операций. В частности, мы предоставим для нашей VMA операции ***open*** и ***close***. Эти операции вызываются, когда процесс открывает или закрывает VMA; в частности, метод ***open*** вызывается при ветвлении процесса и создаёт новую ссылку на VMA. Методы VMA ***open*** и ***close*** вызываются в дополнение к обработке, выполняемой ядром, поэтому нет необходимости заново реализовывать любую работу, проделанную им. Они существуют как способ для драйверов сделать любую дополнительную обработку, которая может им потребоваться.

Как выясняется, простому драйверу, такому, в частности, как ***simple***, не требуется делать никакой дополнительной обработки. Так что мы создали методы ***open*** и ***close***, которые

печатают сообщение в системный журнал, информируя всех, что они были вызваны. Не особенно полезно, но это позволит нам показать, как эти методы могут быть предоставлены, и увидеть, когда они вызываются.

С этой целью мы переопределяем **vma->vm_ops** по умолчанию на операции, которые вызывают ***printk***:

```
void simple_vma_open(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA open, virt %lx, phys %lx\n",
    vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
    printk(KERN_NOTICE "Simple VMA close.\n");
}

static struct vm_operations_struct simple_remap_vm_ops = {
    .open = simple_vma_open,
    .close = simple_vma_close,
};
```

Чтобы сделать эти операции активными для заданного отображения, необходимо сохранить указатель на **simple_remap_vm_ops** в поле **vm_ops** соответствующей VMA. Это обычно выполняется в методе ***mmap***. Если вернуться назад к примеру ***simple_remap_mmap***, вы увидите такие строки кода:

```
vma->vm_ops = &simple_remap_vm_ops;
simple_vma_open(vma);
```

Обратите внимание на явный вызов ***simple_vma_open***. Поскольку метод ***open*** не вызывается при начале ***mmap***, мы должны вызвать его явно, если мы хотим, чтобы это заработало.

Отображение памяти с помощью nopage

Хотя ***remap_pfn_range*** работает хорошо во многих, если не в большинстве, реализаций в драйвере ***mmap***, иногда бывает необходимо быть немного более гибким. В таких ситуациях может быть востребована реализация с использованием метода VMA ***nopage***.

Ситуация, в которой подход ***nopage*** является полезным, может быть создана системным вызовом ***mremap***, который используется приложениями для изменения границ адресов отображаемого региона. Случается, что ядро не уведомляет драйверы непосредственно, когда отображённая VMA изменяется с помощью ***mremap***. Если VMA уменьшается в размерах, ядро может тихо избавиться от ненужных страниц не уведомляя драйвер. Если, наоборот, VMA расширяется, драйвер в конце концов узнает об этом через вызовы ***nopage***, когда для новых страниц потребуется создать отображение, поэтому нет необходимости выполнять отдельное уведомление. Поэтому, если вы хотите поддерживать системный вызов ***mremap***, должен быть реализован метод ***nopage***. Здесь мы покажем простую реализацию ***nopage*** для устройства ***simple***.

Напомним, что метод ***nopage*** имеет следующий прототип:

```
struct page *(*nopage)(struct vm_area_struct *vma, unsigned long address, int
*type);
```

Когда пользовательский процесс пытается получить доступ к странице в VMA, которая не присутствует в памяти, вызывается связанная с ней функция ***nopage***. Параметр **address** содержит виртуальный адрес, который вызвал ошибку, округлённый вниз к началу страницы. Функция ***nopage*** должна найти и вернуть указатель **struct page**, который относится к той странице, которую хочет пользователь. Эта функция также должна заботиться об увеличении счётчика использования для страницы, которую она возвращает, вызывая макрос ***get_page***:

```
get_page(struct page *pageptr);
```

Этот шаг необходим, чтобы на отображённых страницах сохранить счётчики ссылок верными. Ядро поддерживает этот счётчик для каждой страницы; когда счётчик уменьшается до 0, ядро знает, что страница может быть помещена в список свободных страниц. Когда VMA становится неотображённой, ядро уменьшает счётчик использования для каждой страницы в этой области. Если драйвер не увеличил счётчик при добавлении страницы в эту область, счётчик использования станет 0 преждевременно и целостность системы нарушится.

Методу ***nopage*** следует также хранить тип ошибки в месте, указываемом аргументом **type**, но только если этот аргумент не NULL. В драйверах устройств подходящим значением для **type** неизменно будет **VM_FAULT_MINOR**.

Обычно, если вы используете ***nopage***, предстоит сделать очень мало работы, когда вызывается ***mmap***; наша версия выглядит следующим образом:

```
static int simple_nopage_mmap(struct file *filp, struct vm_area_struct *vma)
{
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;

    if (offset >= __pa(high_memory) || (filp->f_flags & O_SYNC))
        vma->vm_flags |= VM_IO;
    vma->vm_flags |= VM_RESERVED;

    vma->vm_ops = &simple_nopage_vm_ops;
    simple_vma_open(vma);
    return 0;
}
```

Основной вещью, которую ***mmap*** должна сделать, это заменить значение по умолчанию (NULL) указателя **vm_ops** своими собственными операциями. Метод ***nopage*** затем заботится о "переназначении" одной страницы за раз и возвращает адрес его структуры **struct page**. Поскольку мы просто реализуем здесь окно на физическую память, шаг переназначения прост: необходимо только найти и вернуть указатель на **struct page** для требуемого адреса. Наша метод ***nopage*** выглядит следующим образом:

```
struct page *simple_vma_nopage(struct vm_area_struct *vma, unsigned long
address, int *type)
{
    struct page *pageptr;
    unsigned long offset = vma->vm_pgoff << PAGE_SHIFT;
    unsigned long physaddr = address - vma->vm_start + offset;
```

```
    unsigned long pageframe = physaddr >> PAGE_SHIFT;

    if (!pfn_valid(pageframe))
        return NOPAGE_SIGBUS;
    pageptr = pfn_to_page(pageframe);
    get_page(pageptr);
    if (type)
        *type = VM_FAULT_MINOR;
    return pageptr;
}
```

Хотя, опять же, мы просто отображаем здесь основную память, функции ***nopage*** необходимо только найти корректную **struct page** для ошибочного адреса и увеличить её счётчик ссылок. Таким образом, необходимой последовательностью событий является расчёт требуемого физического адреса и преобразование его в номер блока страницы, сдвигая его вправо на **PAGE_SHIFT** бит. Поскольку пространство пользователя может дать нам любой понравившийся ему адрес, мы должны гарантировать, что мы имеем верный блок страницы; для нас это делает функция ***pfn_valid***. Если адрес вне диапазона, мы возвращаем **NOPAGE_SIGBUS**, который заставляет сигнал шины быть доставленным вызывающему процессу. В противном случае, ***pfn_to_page*** получает необходимый указатель **struct page**, мы можем увеличить его счётчик ссылок (вызовом ***get_page***) и возвратить его.

Метод ***nopage*** обычно возвращает указатель на **struct page**. Если по какой причине нормальная страница не может быть возвращена (например, запрошенный адрес находится за пределами области памяти устройства), для сигнализации об ошибке может быть возвращён **NOPAGE_SIGBUS**; это и есть то, что делает выше код ***simple***. ***nopage*** также может вернуть **NOPAGE_OOM** для указания ошибок, вызываемых ограниченными ресурсами.

Обратите внимание, что эта реализация работает для областей памяти ISA, но не для них на PCI шине. Память PCI отображается выше самой высокой системной памяти и для таких адресов не существует записей в карте системной памяти. Поскольку не существует **struct page**, чтобы вернуть на неё указатель, ***nopage*** не может быть использована в таких ситуациях; вы должны использовать вместо неё ***remap_pfn_range***.

Если метод ***nopage*** оставлен NULL, код ядра, который обрабатывает ошибки страницы, с ошибочным виртуальным адресом связывает нулевую страницу. ***Нулевая страница*** является страницей с механизмом копирования при записи (copy-on-write page), которая читается как 0 и которая используется, например, для отображения сегмента BSS. Любой процесс, ссылающийся на нулевую страницу, видит именно это: страница заполнена нулями. Если процесс пишет на страницу, это заканчивается изменением частной копии. Поэтому, если процесс расширяет отображаемую область, вызывая ***mremap***, и драйвер не имеет реализованной ***nopage***, процесс заканчивается с заполненной нулями памятью вместо ошибки сегментации.

Переназначение заданных областей ввода/вывода

Все примеры, которые мы видели до сих пор, являются заново выполненными реализациями ***/dev/mem***; они переназначают физические адреса в пространстве пользователя. Однако, типичный драйвер хочет отображать только небольшой диапазон адресов, который относится к его периферийному устройству, а не всю память. Для отображения в пространство пользователя только части всего диапазона памяти, драйверу необходимо лишь поиграть со смещениями. Следующий пример добивается цели для

драйвера, отображая область в **simple_region_size** байт, начиная с физического адреса **simple_region_start** (который должен быть выровнен по странице):

```
unsigned long off = vma->vm_pgoff << PAGE_SHIFT;
unsigned long pfn = page_to_pfn(simple_region_start + off);
unsigned long vsize = vma->vm_end - vma->vm_start;
unsigned long psize = simple_region_size - off;

if (vsize > psize)
    return -EINVAL; /* диапазон слишком велик */
remap_pfn_range(vma, vma_>vm_start, pfn, vsize, vma->vm_page_prot);
```

В дополнение к расчёту смещения, этот код представляет проверку, которая сообщает об ошибке, когда программа пытается отобразить больше памяти, чем имеется в области ввода/вывода целевого устройства. В этом коде **psize** является физическим размером ввода/вывода, который остаётся после указания смещения, и **vsize** является запрошенным размером виртуальной памяти; функция отказывается отображать адреса, которые выходят за рамки разрешённого диапазона памяти.

Обратите внимание, что пользовательский процесс всегда может использовать ***mremap*** для расширения своего отображения, возможно, за пределом физической области устройства. Если ваш драйвер не определил метод ***nopage***, он никогда не уведомится о таком расширении и дополнительная область отобразиться на нулевую страницу. Как автор драйвера, вы можете также захотеть предотвратить поведение такого рода; отображение нулевой страницы в конце вашего региона не является совсем уж плохой вещью, но очень маловероятно, что программист хочет, чтобы это произошло.

Простейшим способом предотвращения расширения отображения является реализация простого метода ***nopage***, который всегда вызывает сигнал шины, который будет отправлен к ошибающемуся процессу. Такой метод будет выглядеть следующим образом:

```
struct page *simple_nopage(struct vm_area_struct *vma, unsigned long address,
int *type)
{ return NOPAGE_SIGBUS; /* послать SIGBUS */}
```

Как мы уже видели, метод ***nopage*** вызывается только тогда, когда процесс разыменовывает адрес, который находится в пределах известной VMA, но для которых в настоящее время нет действительной записи в таблице страниц. Если мы использовали ***remap_pfn_range*** для отображения всего региона устройства, метод ***nopage***, показанный здесь, вызывается только для ссылок за пределами этого региона. Таким образом, он может безопасно вернуть **NOPAGE_SIGBUS**, чтобы просигнализировать об ошибке. Конечно, более тщательная реализация ***nopage*** может проверить, чтобы увидеть, находится ли ошибочный адрес в области устройства, и выполнить переназначение, если дело обстоит именно так. Однако, опять же, ***nopage*** не работает с областями памяти PCI, так что расширение PCI отображений невозможно.

Перераспределение ОЗУ

Интересным ограничением ***remap_pfn_range*** является то, что она даёт доступ только к зарезервированным страницам и физическим адресам выше вершины физической памяти. В Linux страницы физических адресов помечены как "зарезервированные" в карте памяти, чтобы указать, что они недоступны для управления памятью. На ПК, например, диапазон между 640

Кб и 1 Мб помечен как зарезервированный, как и страницы, которые содержат в себе код ядра. Зарезервированные страницы заблокированы в памяти и являются единственными, которые могут быть безопасно отображены в пользовательское пространство; это ограничение является основным требованием для системной стабильности.

Таким образом, ***remap_pfn_range*** не позволяет переназначить обычные адреса, в том числе те, которые вы получите, вызвав ***get_free_page***. Вместо этого, она свяжет их с нулевой страницей. Всё представляется работающим, с тем исключением, что этот процесс видит свои собственные, заполненные нулями страницы, а не переназначенную оперативную память, как он надеялся. Тем не менее, функция выполняет всё, что необходимо делать большинству драйверов оборудования, потому что она может переназначить верхние буферы PCI и ISA память.

Ограничения ***remap_pfn_range*** можно увидеть, запустив ***mapper***, один из примеров программ в ***misc-progs*** в файлах, находящихся на FTP сайте O'Reilly. ***mapper*** является простым инструментом, который можно использовать для быстрой проверки системного вызова ***mmap***; он отображает части файла, указанного параметрами командной строки, в режиме "только для чтения" и выводит отображённый регион на стандартный вывод. Следующая сессия, например, показывает, что ***/dev/mem*** не делает отображения физической страницы, расположенной по адресу 64 Кб, вместо этого мы видим страницу, заполненную нулями (в данном примере компьютером является ПК, но результат будет таким же и на других платформах):

```
morgana.root# ./mapper /dev/mem 0x10000 0x1000 | od -Ax -t x1
mapped "/dev/mem" from 65536 to 69632
000000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
*
001000
```

Неспособность ***remap_pfn_range*** иметь дело с ОЗУ предполагает, что основанные на памяти устройства, такие как ***scull***, не могут легко реализовать ***mmap***, потому что его память устройства является обычным ОЗУ, а не памятью ввода/вывода. К счастью, доступно довольно простое решения для обхода этой проблемы для любого драйвера, которому необходимо отобразить ОЗУ в пользовательское пространство; используется метод ***nopage***, с которым мы познакомились ранее.

Перераспределение ОЗУ с помощью метода nopage

Способом отображения реальной оперативной памяти в пользовательское пространство является использование ***vm_ops->nopage***, чтобы иметь дело с ошибками страниц по одной. Пример реализации является частью модуля ***scullp***, представленного в Главе 8 203.

scullp - это странично-ориентированное символьное устройство. Поскольку он странично-ориентированный, он может реализовать на своей памяти ***mmap***. Код, реализующий отображение памяти, использует некоторые понятия, введённые в разделе "Управление памятью в Linux" 395.

Перед изучением кода давайте посмотрим на дизайнерские решения, который влияют на реализацию ***mmap*** в ***scullp***:

- ***scullp*** не освобождает память устройства, пока устройство отображается. Это вопрос политики, а не требование, и это отличие от поведения ***scull*** и аналогичных устройств,

которые обрезаются до длины 0, когда открываются для записи. Отказ от освобождения отображаемого устройства ***scullp*** позволяет процессу перезаписывать области активно используемые другим процессом, так что вы можете протестировать и посмотреть, как взаимодействуют процессы и память устройства. Чтобы избежать отключения подключенного устройства, драйвер должен хранить количество активных отображений; для этой цели в структуре устройства используется поле **vmas**.

- Отображение памяти выполняется только тогда, когда параметр ***scullp* order** (установленный во время загрузки модуля) равен 0. Параметр определяет, как вызывается ***__get_free_pages*** (смотрите раздел "get_free_page и друзья" 211 в Главе 8 221). Ограничение нулевого порядка (которое заставляет страницы выделяться по одной за раз, а не большими группами) диктуется внутренностями ***__get_free_pages***, функцией выделения, используемой в ***scullp***. Чтобы увеличить эффективность выделения, ядро Linux поддерживает список свободных страниц для каждого порядка выделения и только счётчик ссылок на первой странице в кластере увеличивается с помощью ***get_free_pages*** и уменьшается с помощью ***free_pages***. Метод ***mmap*** в устройстве ***scullp*** запрещается, если порядок больше нуля, потому что ***nopage*** имеет дело с одной страницей, а не кластерами страниц. ***scullp*** просто не знает, как правильно управлять счётчиками ссылок для страниц, которые являются частью выделений более высокого порядка. (Вернитесь к разделу "scull, использующий целые страницы: scullp" 212 в Главе 8 203, если вам необходимо освежить в памяти ***scullp*** и понятие порядка выделения памяти.)

Ограничение нулевого порядка в основном предназначено, чтобы сохранить код простым. Можно правильно реализовать ***mmap*** для многостраничных выделений, играя со счётчиком использования на страницах, но это бы только добавило сложности примеру, не давая какой-либо интересной информации.

Коду, который предназначен для отображения ОЗУ в соответствии с только что изложенными правилами, необходимо реализовать VMA методы ***open***, ***close*** и ***nopage***; он также нуждается в доступе к карте памяти, чтобы настроить счётчики использования страниц.

Эта реализация ***scullp_mmap*** очень короткая, потому что опирается на функцию ***nopage***, чтобы сделать всю интересную работу:

```
int scullp_mmap(struct file *filp, struct vm_area_struct *vma)
{
    struct inode *inode = filp->f_dentry->d_inode;

    /* отказать в отображении, если порядок не 0 */
    if (scullp_devices[iminor(inode)].order)
        return -ENODEV;

    /* не делаем здесь ничего: "nopage" будет заполнять дыры */
    vma->vm_ops = &scullp_vm_ops;
    vma->vm_flags |= VM_RESERVED;
    vma->vm_private_data = filp->private_data;
    scullp_vma_open(vma);
    return 0;
}
```

Целью оператора **if** является избежать отображения устройств, для которых порядок выделения не равен 0. Операции ***scullp*** хранятся в поле **vm_ops** и указатель на структуру устройства спрятан в поле **vm_private_data**. В конце концов, вызывается **vm_ops->open**

для обновления числа активных отображений для данного устройства.

open и ***close*** просто отслеживают счётчик отображений и определены следующим образом:

```
void scullp_vma_open(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas++;
}

void scullp_vma_close(struct vm_area_struct *vma)
{
    struct scullp_dev *dev = vma->vm_private_data;

    dev->vmas--;
}
```

Большинство работы затем выполняется в ***nopage***. В реализации ***scullp***, параметр **address** для ***nopage*** используется, чтобы вычислить смещение в устройстве; смещение затем используется для поиска нужной страницы в дереве памяти ***scullp***:

```
struct page *scullp_vma_nopage(struct vm_area_struct *vma, unsigned long
address, int *type)
{
    unsigned long offset;
    struct scullp_dev *ptr, *dev = vma->vm_private_data;
    struct page *page = NOPAGE_SIGBUS;
    void *pageptr = NULL; /* по умолчанию "не указан" */

    down(&dev->sem);
    offset = (address - vma->vm_start) + (vma->vm_pgoff << PAGE_SHIFT);
    if (offset >= dev->size) goto out; /* вне диапазона */

    /*
     * Теперь получим устройство scullp из списка, затем страницу.
     * Если устройство имеет дыру, процесс получает SIGBUS при
     * доступе к дыре.
     */
    offset >>= PAGE_SHIFT; /* смещение является числом страниц */
    for (ptr = dev; ptr && offset >= dev->qset;) {
        ptr = ptr->next;
        offset -= dev->qset;
    }
    if (ptr && ptr->data) pageptr = ptr->data[offset];
    if (!pageptr) goto out; /* дыра или конец файла */
    page = virt_to_page(pageptr);

    /* получили её, теперь увеличиваем счётчик */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
out:
    up(&dev->sem);
    return page;
```

```
}
```

scullp использует память, полученную с помощью ***get_free_pages***. Такая память адресуется используя логические адреса, так что всё, что требуется сделать ***scullp_nopage***, чтобы получить указатель **struct page**, это вызвать ***virt_to_page***.

Устройство ***scullp*** теперь работает как ожидается, как вы можете видеть в этом примере выходных данных из утилиты ***mapper***. Здесь мы выполняем распечатку каталога ***/dev*** (который является большим) в устройство ***scullp*** и затем используем утилиту ***mapper***, чтобы посмотреть на кусочки этого листинга с помощью ***mmap***:

```
morgana% ls -l /dev > /dev/scullp
morgana% ./mapper /dev/scullp 0 140
mapped "/dev/scullp" from 0 (0x00000000) to 140 (0x0000008c)
total 232
crw-------    1 root     root      10,  10 Sep 15 07:40 adbmouse
crw-r--r--    1 root     root      10, 175 Sep 15 07:40 agpgart
morgana% ./mapper /dev/scullp 8192 200
mapped "/dev/scullp" from 8192 (0x00002000) to 8392 (0x000020c8)
d0h1494
brw-rw----    1 root     floppy     2,  92 Sep 15 07:40 fd0h1660
brw-rw----    1 root     floppy     2,  20 Sep 15 07:40 fd0h360
brw-rw----    1 root     floppy     2,  12 Sep 15 07:40 fd0H360
```

Перераспределение виртуальных адресов ядра

Хотя это требуется редко, интересно увидеть, как драйвер может используя ***mmap*** отобразить виртуальный адрес ядра в пространство пользователя. Напомним, что верный виртуальный адрес ядра является адресом, возвращаемым такой функцией, как ***vmalloc***, то есть виртуальный адрес отображается в таблицах страниц ядра. Код в этом разделе взят из ***scullv***, который является модулем, работающим подобно ***scullp***, но выделяющим своё хранилище через ***vmalloc***.

Большая часть реализации ***scullv*** подобна той, что мы только что видели для ***scullp***, за исключением того, что нет необходимости проверять параметр **order**, который управляет выделением памяти. Причиной этого является то, что ***vmalloc*** выделяет свои страницы по одной за раз, потому что одностраничные выделения имеют гораздо больше шансов на успех, чем многостраничные. Поэтому проблема порядка выделения не относится к пространству ***vmalloc***.

Помимо этого, есть только одна разница между реализациями ***nopage***, используемыми ***scullp*** и ***scullv***. Напомним, что ***scullp***, как только он нашёл интересующую страницу, получает соответствующий указатель **struct page** с помощью ***virt_to_page***. Эта функция, однако, не работает с виртуальными адресами ядра. Взамен вы должны использовать ***vmalloc_to_page***. Таким образом, заключительная часть версии ***nopage*** в ***scullv*** выглядит следующим образом:

```
    /*
     * После поиска scullv, "page" является теперь адресом страницы,
     * необходимой текущему процессу. Так как это адрес vmalloc,
     * преобразуем его в struct page.
     */
    page = vmalloc_to_page(pageptr);
```

```
    /* получили её, теперь увеличиваем счётчик */
    get_page(page);
    if (type)
        *type = VM_FAULT_MINOR;
out:
    up(&dev->sem);
    return page;
```

На основе этого обсуждения вы также могли бы захотеть отобразить в пространство пользователя адреса, возвращаемые ***ioremap***. Однако, это было бы ошибкой; адреса от ***ioremap*** являются специальными и не могут рассматриваться как обычные виртуальные адреса ядра. Вместо этого для переназначения областей памяти ввода/вывода в пространство пользователя вы должны использовать ***remap_pfn_range***.

Выполнение прямого ввода/вывода

Большинство операций ввода/вывода буферизируются через ядро. Использование буфер пространства ядра обеспечивает уровень разделения между пользовательским пространством и реальным устройством; это разделение может сделать программирования проще и во многих ситуациях может также принести выгоды в производительности. Однако, есть случаи, где может быть полезно выполнять ввод/вывод непосредственно в или из буфера пользовательского пространства. Если число передаваемых данных большое, передача данных напрямую, без дополнительного копирования через пространство ядра, может ускорить процесс.

Одним из примеров прямого ввода/вывода, используемого в ядре версии 2.6, является драйвер ленточного SCSI накопителя. Ленточные накопители (стримеры) могут передавать много данных через систему и передача данных, как правило, ориентирована на запись, так что от буферизации данных в ядре пользы мало. Таким образом, при выполнении соответствующих условий (например, буфер пользовательского пространства выровнен постранично), драйвер SCSI накопителя выполняет свой ввод/вывод без копирования данных.

Тем не менее, важно понимать, что прямой ввод/вывод не всегда обеспечивают повышение производительности, как можно было ожидать. Накладные расходы инициализации прямого ввода/вывода (которая включает в себя обработку ошибок и точное определение соответствующих пользовательских страниц) могут быть значительными и выгоды от прямого ввода/вывода теряются. Например, использование прямого ввода/вывода требует, чтобы системный вызов ***write*** выполнялся синхронно; в противном случае, приложение не узнает, когда оно может повторно использовать свой буфер ввода/вывода. Остановка приложения до завершения каждой записи может замедлить ход событий, вот почему приложения, использующие прямой ввод/вывод часто также используют асинхронные операции ввода/вывода.

Настоящей моралью этой истории, в любом случае, является то, что реализация прямого ввода/вывода в символьном драйвере, как правило, не нужна и может быть вредной. Вы должны предпринять этот шаг, только если вы уверены, что накладные расходы буферированного ввода/вывода действительно замедляют работу. Обратите также внимание, что блочным и сетевым драйверам о реализации прямого ввода/вывода необходимости беспокоиться нет совсем; в обоих случаях высокоуровневый код ядра подготовит и выполнит использование прямого ввода/вывода когда это указано, и коду на уровне драйвера даже не требуется знать, что выполняется прямой ввод/вывод.

Ключом к реализации прямого ввода/вывода в ядре версии 2.6 является функция,

названная ***get_user_pages***, которая объявлена в ***<linux/mm.h>*** со следующим прототипом:

```
int get_user_pages(struct task_struct *tsk,
                   struct mm_struct *mm,
                   unsigned long start,
                   int len,
                   int write,
                   int force,
                   struct page **pages,
                   struct vm_area_struct **vmas);
```

Эта функция имеет несколько аргументов:

tsk
: Указатель на задачу, выполняющую ввод/вывод; его основной целью является сообщить ядру, кто должен отвечать за любые ошибки страницы, происходящие при создании буфера. Этот аргумент почти всегда передается как **current**.

mm
: Указатель на структуру управления памятью, описывающую адресное пространство для отображения. Структура **mm_struct** - часть, которая связывает воедино все части (области VMA) виртуального адресного пространства процесса. Для использования в драйвере этот аргумент всегда должен быть **current->mm**.

start
len
: **start** является (странично-выровненным) адресом в буфере пользовательского пространства, а **len** является длиной буфера в страницах.

write
force
: Если **write** не равен нулю, страницы отображаются для доступа на запись (подразумевая, конечно, что пространство пользователя выполняет операцию чтения). Флаг **force** приказывает ***get_user_pages*** отменить защиту на данных страницах для обеспечения запрашиваемого доступа; драйверы должны всегда передавать здесь 0.

pages
vmas
: Выходные параметры. После успешного завершения **pages** содержат список указателей на структуры **struct page**, описывающие буфер пользовательского пространства и **vmas** содержит указатели на соответствующие области VMA. Очевидно, параметры должны указывать на массивы, способные содержать по меньшей мере **len** указателей. Любой параметр может быть NULL, но для действительной работы с буфером вам необходимы, по крайней мере, указатели **struct page**.

get_user_pages является низкоуровневой функцией управления памятью с достаточно сложным интерфейсом. Она также требует, чтобы перед вызовом для адресного пространства были получены в режиме чтения семафоры чтения/записи ***mmap***. В результате, вызов ***get_user_pages*** обычно выглядит примерно так:

```
down_read(&current->mm->mmap_sem);
```

```
result = get_user_pages(current, current->mm, ...);
up_read(&current->mm->mmap_sem);
```

Возвращаемое значение является числом действительно отображённых страниц, которое может быть меньше, чем запрошенное число (но больше нуля).

После успешного завершения вызывающий имеет массив **pages**, указывающий на буфер пользовательского пространства, который заблокирован в памяти. Для работы непосредственно с буфером, на код пространства ядра должен преобразовать каждый указатель **struct page** в виртуальный адрес ядра с помощью ***kmap*** или ***kmap_atomic***. Обычно, однако, устройства, для которых прямой ввод/вывод является оправданным, используют операции DMA, так что ваш драйвер, вероятно, захочет создать из массива указателей **struct page** список разборки/сборки. Мы обсудим, как это сделать в разделе "Преобразования разборки/сборки" 432.

После завершения вашей операции прямого ввода/вывода вы должны освободить пользовательские страницы. Однако, прежде, чем сделать это, вы должны проинформировать ядро, если вы изменили содержание этих страниц. В противном случае, ядро может думать, что страницы являются "чистыми", что означает, что они соответствуют копии, находящейся на устройстве подкачки, и освободить их без записи в резервное хранилище. Итак, если вы изменили страницы (в ответ на запрос чтения пользовательского пространства), вы должны отметить каждую затронутую страницу как изменённую следующим вызовом:

```
void SetPageDirty(struct page *page);
```

(Этот макрос определён в ***<linux/page-flags.h>***). Большинство кода, который выполняет эту операцию сначала выполняет проверку, чтобы убедиться, что эта страница не в зарезервированной части карты памяти, которая никогда не выгружается. Таким образом, код обычно выглядит следующим образом:

```
if (! PageReserved(page))
    SetPageDirty(page);
```

Так как память пользовательского пространства, как правило, не отмечена как зарезервированная, эта проверка не должна быть строгой необходимостью, но если вы залезли своими грязными ручонками глубоко внутрь подсистемы управления памятью, лучше быть тщательным и осторожным.

Независимо от того, были ли изменены страницы, они должны быть освобождены из страничного кэша, или они останутся там навсегда. Вызовом для использования является:

```
void page_cache_release(struct page *page);
```

Этот вызов должен, конечно, делаться ***после*** того, как страница была отмечена изменённой, если это необходимо.

Асинхронный ввод/вывод

Одной из новых функций, добавленных в ядро версии 2.6, была возможность асинхронного ввода/вывода. Асинхронный ввод/вывод позволяет пользовательскому пространству инициировать операции, не ожидая их завершения; таким образом, во время выполнения ввода/вывода приложение может выполнять другую обработку. Сложные,

высокопроизводительные приложения могут также использовать асинхронный ввод/вывод для выполнения множества операций, происходящих в одно и то же время.

Реализация асинхронного ввода/вывода не является обязательной и очень немногие авторы драйверов беспокоятся об этом; большинство устройств не пользуются этой возможностью. Как мы увидим в ближайших главах, блочные и сетевые драйверы всегда полностью асинхронны, так что кандидатами на явную поддержку асинхронного ввода/вывода являются только символьные драйверы. Символьное устройство может получить выгоду от этой поддержки, если есть веские причины для выполнения в любой момент времени более одной операции ввода/вывода. Хорошим примером являются потоковые ленточные накопители, где привод может остановиться и значительно замедлиться, если операции ввода/вывода не приходят достаточно быстро. Приложение, пытающееся получить максимальную производительность от потокового накопителя, могло бы использовать асинхронный ввод/вывод, чтобы в любой момент времени иметь множество операций готовых к выполнению.

Мы представляем краткий обзор того, как он работает, для того редкого автора драйвера, которому необходимо реализовать асинхронный ввод/вывод. Мы освещаем асинхронный ввод/вывод в этой главе, потому что его реализация также почти всегда включает в себя операции прямого ввода/вывода (если данные буферируются в ядре, обычно можно реализовать асинхронное поведение без введения дополнительной сложности в пространстве пользователя).

Драйверы, поддерживающие асинхронный ввод/вывод должна подключать ***<linux/aio.h>***. Для реализации асинхронного ввода/вывода имеются три метода ***file_operations*** :

```
ssize_t (*aio_read) (struct kiocb *iocb, char *buffer, size_t count, loff_t
offset);
ssize_t (*aio_write) (struct kiocb *iocb, const char *buffer, size_t count,
loff_t offset);
int (*aio_fsync) (struct kiocb *iocb, int datasync);
```

Операция ***aio_fsync*** представляет интерес только для кода файловой системы, поэтому мы далее не будем обсуждать её здесь. Два других, ***aio_read*** и ***aio_write***, выглядят очень похожими на обычные методы ***read*** и ***write***, но с несколькими исключениями. Одно из них то, что параметр **offset** передается по значению; асинхронные операции никогда не изменяют позицию в файле, поэтому передавать указатель на неё нет никаких причин. Эти методы также принимают параметр **iocb** (“I/O control block”, "блок управления ввода/вывода"), до которого мы доберёмся через мгновение.

Целью методов ***aio_read*** и ***aio_write*** является начать операции чтения или записи, которые могут или не могут быть завершёнными на момент их возвращения. Если возможно немедленно завершить операцию, метод должен сделать это и вернуть обычный статус: число переданных байт или отрицательный код ошибки. Таким образом, если ваш драйвер имеет метод ***read***, названный ***my_read***, следующий метод ***aio_read*** совершенно правильный (хотя и довольно бессмысленный):

```
static ssize_t my_aio_read(struct kiocb *iocb, char *buffer, ssize_t count,
loff_t offset)
{
    return my_read(iocb->ki_filp, buffer, count, &offset);
}
```

Обратите внимание, что указатель **struct file** можно найти в поле **ki_filp** структуры **kiocb**.

Если вы поддерживаете асинхронный ввод/вывод, вы должны осознавать тот факт, что ядро может при необходимости создать "синхронные IOCB". Это, по существу, асинхронные операции, которые должны быть выполнены на самом деле синхронно. Впору задаться вопросом, почему это делается так, но лучше просто сделать то, что запрашивает ядро. Синхронные операции помечены в IOCB; ваш драйвер должен запросить этот статус следующим образом:

```
int is_sync_kiocb(struct kiocb *iocb);
```

Если эта функция возвращает отличное от нуля значение, ваш драйвер должен выполнить операцию синхронно.

Однако, в конце концов, смыслом всей этой структуры является обеспечение асинхронных операций. Если ваш драйвер сможет приступить к работе (или просто встать в очередь до какого-нибудь будущего времени, когда она может быть выполнена), он должен сделать две вещи: запомнить всё, что необходимо знать об операции и вернуть вызвавшему **-EIOCBQUEUED**. Запоминание информации об операции включает в себя организацию доступа к буферу пользовательского пространства; после возврата вы не получите снова возможность доступа к этому буферу пока не будете работать в контексте вызывающего процесса. В общем, это означает, что вы, вероятно, должны создать прямое отображение в ядре (с помощью ***get_user_pages***) или отображение DMA. Код ошибки **-EIOCBQUEUED** показывает, что операция ещё не завершена, и о её окончательный статус будет сообщено позже.

Когда "потом" приходит, ваш драйвер должен сообщить ядру, что операция завершена. Это делается с помощью вызова ***aio_complete***:

```
int aio_complete(struct kiocb *iocb, long res, long res2);
```

Здесь **iocb** является тем же IOCB, который была первоначально вам передан, и **res** является обычным статусом результата для этой операции. **res2** является вторым кодом результата, который будет возвращён в пользовательское пространство; большинство реализаций асинхронного ввода/вывода передают **res2** как 0. После того, как вы вызвали ***aio_complete***, вы не должны больше прикасаться к IOCB или пользовательскому буферу.

Пример асинхронного ввода/вывода

Асинхронный ввод/вывод в исходниках примеров реализует странично-ориентированный драйвер ***scullp***. Реализация проста, но вполне достаточна, чтобы показать, как должны быть устроены асинхронные операции.

Методы ***aio_read*** и ***aio_write*** на самом деле делают немногое:

```
static ssize_t scullp_aio_read(struct kiocb *iocb, char *buf, size_t count,
loff_t pos)
{
    return scullp_defer_op(0, iocb, buf, count, pos);
}

static ssize_t scullp_aio_write(struct kiocb *iocb, const char *buf, size_t
count, loff_t pos)
```

```
{
    return scullp_defer_op(1, iocb, (char *) buf, count, pos);
}
```

Эти методы просто вызывают общие функции:

```
struct async_work {
    struct kiocb *iocb;
    int result;
    struct work_struct work;
};

static int scullp_defer_op(int write, struct kiocb *iocb, char *buf, size_t
count, loff_t pos)
{
    struct async_work *stuff;
    int result;

    /* Копируем сейчас, пока мы может иметь доступ к буферу */
    if (write)
        result = scullp_write(iocb->ki_filp, buf, count, &pos);
    else
        result = scullp_read(iocb->ki_filp, buf, count, &pos);

    /* Если это синхронная IOCB, мы возвращаем наш статус сейчас. */
    if (is_sync_kiocb(iocb))
        return result;

    /* В противном случае отложим завершение на несколько миллисекунд. */
    stuff = kmalloc (sizeof (*stuff), GFP_KERNEL);
    if (stuff == NULL)
        return result; /* Памяти нет, так что просто завершаем */
    stuff->iocb = iocb;
    stuff->result = result;
    INIT_WORK(&stuff->work, scullp_do_deferred_op, stuff);
    schedule_delayed_work(&stuff->work, HZ/100);
    return -EIOCBQUEUED;
}
```

Более полная реализация использовала бы ***get_user_pages***, чтобы отобразить пользовательский буфер в пространство ядра. Мы решили сохранить жизнь простой просто копируя данные вначале. Затем делается вызов ***is_sync_kiocb***, чтобы увидеть, должна ли эта операция быть выполнена синхронно; если да, то возвращается статус результата, и мы завершаем работу. В противном случае, мы запоминаем соответствующую информацию в небольшой структуре, организуем "завершение" через очередь задач и возвращаем **-EIOCBQUEUED**. На данный момент управление возвращается к пользовательскому пространству.

Позднее, очередь задач выполняет нашу функцию завершения работы:

```
static void scullp_do_deferred_op(void *p)
{
    struct async_work *stuff = (struct async_work *) p;
    aio_complete(stuff->iocb, stuff->result, 0);
```

```
    kfree(stuff);
}
```

Здесь это просто вопрос вызова ***aio_complete*** с сохранённой нами информацией. Конечно, настоящая реализация асинхронного ввода/вывода в драйвере является несколько более сложной, но следует структуре такого вида.

Прямой доступ к памяти

Прямой доступ к памяти или DMA (direct memory access), это сложная тема, которая завершает наш обзор вопросов работы с памятью. DMA - это аппаратный механизм, позволяющий компонентам периферии передавать их данные ввода/вывода непосредственно в и из основной памяти, без необходимости привлечения системного процессора. Использование этого механизма может существенно повысить пропускную способность в и из устройства, так как позволяет устранить большие вычислительные накладные расходы.

Обзор передачи данных с прямым доступом к памяти

До введения подробностей программирования, давайте рассмотрим, как выполняются передачи DMA, для упрощения обсуждения учитывая только входные передачи.

Передача данных может быть вызвана двумя способами: либо программа запрашивает данные (через такую функцию, как ***read***) или оборудование асинхронно помещают данные в систему. В первом случае вовлечённые этапы могут быть подытожены следующим образом:

1. Когда процесс вызывает ***read***, метод драйвера выделяет буфер DMA и поручает оборудованию передавать свои данные в этот буфер. Процесс помещается в сон.
2. Оборудование записывает данные в буфер DMA и вызывает прерывание, когда это выполнено.
3. Обработчик прерывания получает входные данные, подтверждает прерывание и пробуждает процесс, который теперь имеет возможность прочитать данные.

Второй случай наступает, когда DMA используется асинхронно. Так происходит, например, с устройством сбора данных, которое выполняет выдаёт данные, даже если никто их не читает. В этом случае драйвер должен содержать буфер, чтобы последующий вызов ***read*** вернул все накопленные данные в пространство пользователя. Этапы, связанные с такого рода передачей, немного отличаются:

1. Оборудование вызывает прерывание, чтобы объявить, что поступили новые данные.
2. Обработчик прерывания выделяет буфер и сообщает оборудованию, куда передать данные.
3. Периферийное устройство записывает данные в буфер и по завершении вызывает другое прерывание.
4. Обработчик отправляет новые данные, будит любой соответствующий процесс, и заботится о ведении хозяйства.

Вариант асинхронного подхода часто виден с сетевыми картами. Эти карты часто ожидают увидеть круговой буфер (часто называемый DMA-шным буфером), находящийся в памяти, используемой совместно с процессором; каждый входящий пакет помещается в следующий доступный буфер в круге и сигнализируется прерыванием. Затем драйвер передаёт сетевые пакеты остальной части ядра и помещает новый буфер DMA в круг.

Обработка шаги во всех этих случаях подчеркнуть, что эффективная обработка DMA опирается на прерывание отчетности. Хотя можно реализовать DMA с опрашивающим драйвером, это не будет иметь смысла, потому что опрашивающий драйвер сведёт на нет преимущества производительности, которые предлагает DMA вместо более простого ввода/вывода, управляемого процессором. (* Конечно, из всего есть исключения; смотрите раздел "Уменьшение числа прерываний" 505 в Главе 17 521 для демонстрации того, как высокопроизводительные сетевые драйверы лучше реализуются с использованием опроса.)

Другим важным объектом, представленным здесь, является буфер DMA. DMA требует от драйверов устройств выделить один или несколько специальных буферов, подходящих для DMA. Обратите внимание, что многие драйверы выделяют свои буфера во время инициализации и используют их до выключения - слово ***выделить*** в предыдущем тексте, следовательно, означает "заполучить выделенный ранее буфер".

Выделение DMA буфера

Этот раздел описывает выделение DMA буферов на низком уровне; в ближайшее время мы введём высокоуровневый интерфейс, но ещё неплохо понимать материал, представленный здесь.

Основным вопросом, который остро стоит с DMA буферами является тот, что когда они больше, чем одна страница, они должны занимать смежные страницы в физической памяти, так как устройство передаёт данные с помощью системной шины ISA или PCI, обе из которых имеют физические адреса. Интересно отметить, что это ограничение не распространяется на SBus (смотрите раздел "SBus" 308 в Главе 12 288), которая использует на периферийной шине виртуальные адреса. Некоторые архитектуры могут также использовать виртуальные адреса на шине PCI, но переносимый драйвер не может рассчитывать на такую возможность.

Хотя DMA буферы могут быть выделены либо при загрузке системы или во время выполнения программы, модули могут выделить их буферы только во время выполнения. (эти методы представила Глава 8 203; раздел "Получение больших буферов" 219 описывает выделение при загрузке системы, а "Как работает kmalloc" 203 и "get_free_page и друзья" 211 описали выделение во время выполнения.) Авторы драйверов должны позаботиться, чтобы выделить верный тип памяти, когда она используется для операций DM; не все зоны памяти подходят для этого. В частности, с DMA на некоторых системах и с некоторыми устройствами может не работать верхняя память - периферийные устройства просто не могут работать с такими большими адресами.

Большинство устройств на современных шинах могут обрабатывать 32-х разрядные адреса, это означает, что для них прекрасно работает обычное выделение памяти. Однако, некоторые PCI устройства, не следуют полному стандарту PCI и не могут работать с 32-х разрядными адресами. И конечно, устройства ISA ограничены только 24-х разрядными адресами.

Для устройств с такого рода ограничением память должна быть выделена из зоны DMA, добавлением флага **GFP_DMA** при вызове ***kmalloc*** или ***get_free_pages***. Если этот флаг присутствует, выделяется только та память, которая может быть адресована 24-мя разрядами. В качестве альтернативы, вы можете использовать общий слой DMA (который мы обсудим в ближайшее время), чтобы выделить буферы для обхода ограничений вашего устройства.

Самостоятельное выделение

Мы уже видели, как ***get_free_pages*** может выделить до нескольких мегабайт (так как порядок может иметь диапазон до **MAX_ORDER**, в настоящее время 11), но запросы высокого порядка подвержены неудаче даже когда запрошенный буфер гораздо меньше, чем 128 Кб, потому что системная память становится с течением времени фрагментированной. (* Слово *фрагментация* обычно применяется к дискам, чтобы выразить идею о том, что файлы не хранятся последовательно на магнитном носителе. То же самое относится и к памяти, где каждое виртуальное адресное пространства становится разбросанным по физической памяти и становится трудно получить последовательные свободные страницы при запросе буфера DMA.)

Когда ядро не может вернуть запрашиваемый объём памяти или когда необходимо более чем 128 Кб (общее требование, например, для устройств видеозахвата PCI), альтернативой возвращению **-ENOMEM** является выделение памяти во время загрузки или резервирование для буфера вершины физической памяти. Мы описали выделение во время загрузки в разделе "Получение больших буферов" 219 в Главе 8 203, но для модулей это недоступно. Резервирование верхней части ОЗУ осуществляется передачей для ядра во время загрузки аргумента **mem=**. Например, если у вас есть 256 Мб, аргумент **mem=255M** предохраняет ядро от использования верхнего мегабайта. Чтобы получить доступ к такой памяти, ваш модуль впоследствии может использовать следующий код:

```
dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);
```

allocator (распределитель), часть кодов примеров, сопровождающих книгу, предлагает простой API для проверки и управления такой защищённой оперативной памятью и успешно используется на нескольких архитектурах. Однако этот трюк не работает, когда у вас есть система с большим объёмом памяти (то есть если физической памяти больше, чем может вместить в себя адресное пространство процессора). Другим вариантом, конечно, является выделение буфера с флагом выделения **GFP_NOFAIL**. Однако, этот подход сильно нагружает подсистему управления памятью и он рискует вообще заблокировать систему; его лучше избегать до тех пор, пока действительно нет другого пути.

Однако, если вы заходите так далеко, что выделяете большой буфер DMA, стоит допустить мысли об альтернативах. Если ваше устройство может выполнять ввод/вывод с разборкой/сборкой, вы можете выделить ваш буфер небольшими кусочками и пусть устройство сделает всё остальное. Ввод/вывод с разборкой/сборкой также может быть использован при выполнении прямого ввода/вывода в пользовательском пространстве, что может быть лучшим решением, когда требуется действительно огромный буфер.

Шинные адреса

Драйвер устройства, использующий DMA, должен общаться с оборудованием, подключенным к интерфейсной шине, которая использует физические адреса, в то время как программный код использует виртуальные адреса.

На самом деле, ситуация несколько более сложная, чем эта. Оборудование на основе DMA использует ***шинные***, а не ***физические*** адреса. Хотя шинные адреса ISA и PCI на ПК являются просто физическими адресами, это не верно для каждой платформы. Иногда интерфейсная шина подключена через схему моста, который отображает адреса ввода/вывода на другие физические адреса. Некоторые системы даже имеют странично-отображающую схему, которая может сделать так, чтобы произвольные страницы выглядели для периферийной шины непрерывными.

На самом низком уровне (опять же, в ближайшее время мы будем рассматривать высокоуровневое решение), ядро Linux обеспечивает переносимое решение экспортируя следующие функции, определённые в ***<asm/io.h>***. Использование этих функций настоятельно не рекомендуется, потому что они работают эффективно только на системах с очень простой архитектурой ввода/вывода; тем не менее, вы можете столкнутся с ними при работе с кодом ядра.

```
unsigned long virt_to_bus(volatile void *address);
void *bus_to_virt(unsigned long address);
```

Эти функции выполняют простые преобразования между логическими адресами ядра и шинными адресами. Они не будут работать в любой ситуации, где устройство управления памятью ввода/вывода должно быть запрограммировано или где должны быть использованы возвратные буферы. Правильным способом выполнения этого преобразования является использование универсального уровня DMA, так что мы теперь перейдём к этой теме.

Универсальный уровень DMA

DMA операции, в конечном счёте, сводятся к выделению буфера и передачи вашему устройству шинных адресов. Тем не менее, задача написания переносимых драйверов, которые выполняют DMA безопасно и корректно на всех архитектурах значительно труднее, чем можно подумать. Разные системы имеют разные представления о том, как должно работать согласование кэша; если вы не решите этот вопрос правильно, ваш драйвер может повредить память. Некоторые системы имеют сложное шинное оборудование, которое может сделать задачу DMA легче или труднее. И не все системы могут выполнять DMA из всех частей памяти. К счастью, ядро предоставляет шинно- и архитектурно-независимый уровень DMA, который скрывает большинство из этих проблем от автора драйвера. Мы настоятельно рекомендуем вам использовать этот уровень для DMA операций в любой драйвере, который вы пишете.

Многие из приведённых ниже функций требуют указатель на **struct device**. Эта структура является низкоуровневым представлением устройства внутри модели устройства Linux. Это не то, с чем драйверам часто приходится работать напрямую, но вам это необходимо, когда используется универсальный уровень DMA. Обычно вы можете найти эту структуру похороненной внутри шинной специфики, описывающей ваше устройство. Например, он может быть найден как поле **dev** в **struct pci_device** или **struct usb_device**. Структура **device** подробно описана в Главе 14 347.

Драйверы, которые используют следующие функции, должны подключать ***<linux/dma-mapping.h>***.

Работа с проблемным оборудованием

Первый вопрос, на который необходимо ответить, прежде чем пытаться выполнять DMA, является ли данное устройство способным на такие операции на текущем оборудовании. Многие устройства ограничены в диапазоне памяти, который они могут адресовать, по ряду причин. По умолчанию ядро предполагает, что устройство может выполнять DMA на любой 32-х разрядный адрес. Если это не так, вы должны сообщить ядру об этом факте сделав вызов:

```
int dma_set_mask(struct device *dev, u64 mask);
```

mask должна указать биты, которые ваше устройство может адресовать; например, если

оно ограничено 24-мя битами, вам бы передали **mask** как 0x0FFFFFF. Возвращаемое значение отлично от нуля, если с заданной маской DMA возможен; если ***dma_set_mask*** возвращает 0, вы не можете использовать DMA операции с этим устройством. Таким образом, код инициализации драйвера для устройства, ограниченного 24-х разрядными DMA операциями, может выглядеть так:

```
if (dma_set_mask (dev, 0xffffff))
    card->use_dma = 1;
else {
    card->use_dma = 0; /* Нам придётся жить без DMA */
    printk (KERN_WARN, "mydev: DMA not supported\n");
}
```

Опять же, если устройство поддерживает нормальные, 32-х разрядные операции DMA, вызывать ***dma_set_mask*** необходимости нет.

Отображения DMA

Отображения DMA являются комбинацией выделения буфера DMA и генерации адреса для этого буфера, который доступен устройству. Заманчиво получить такой адрес просто вызвав ***virt_to_bus***, но есть веские основания для избегания такого подхода. Первым из них является то, что соответствующее оборудование поставляется с IOMMU, что обеспечивает для шины набор регистров отображения. IOMMU может организовать появление любой физической памяти в диапазоне адресов, доступных устройству, и это может сделать физически разделённые буферы выглядящими последовательно для устройства. Работа с IOMMU требует использования универсального уровня DMA; ***virt_to_bus*** не подходит для этой задачи.

Заметим, что не все архитектуры имеют IOMMU; в частности, популярная платформа x86 не имеет поддержки IOMMU. Тем не менее, должным образом написанному драйверу нет необходимости знать о поддержке ввода/вывода оборудованием, на котором он выполняется.

Настройка пригодных для устройства адресов в некоторых случаях также может потребовать создания ***возвратного буфера***. Возвратные буферы создаются, когда драйвер пытается выполнить DMA на адрес недоступный для периферийного устройства, например, адрес верхней памяти. Затем данные копируются в и из возвратного буфера, как необходимо. Разумеется, использование возвратных буферов может замедлить ход событий, но иногда альтернативы нет.

Отображения DMA также должны обратиться к вопросу о согласовании кэша. Помните, что современные процессоры хранят копии наиболее часто используемых областей памяти в быстром, локальном кэше; без этого кэша приемлемая производительность невозможна. Если ваше устройство изменяет область основной памяти, крайне важно, чтобы любые процессорные кэши, охватывающие эту область, стали недействительными; в противном случае, процессор может работать с неправильным образом основной памяти и в результатом будет повреждение данных. Аналогично, когда ваше устройство использует DMA для чтения данных из основной памяти, любые изменения в этой памяти, находящиеся в кэшах процессора, должны быть прочитаны в первую очередь. Эти вопросы ***согласованности кэша*** могут создать бесконечные непонятные и труднообнаруживаемые ошибки, если программист не осторожен. Некоторые архитектуры управляют согласованием кэша аппаратно, но другие требуют поддержки от программного обеспечения. Универсальный уровень DMA делает многое, чтобы обеспечить, чтобы всё работало корректно на всех архитектурах, но, как

мы увидим, надлежащее поведение требует соблюдения небольшого набора правил.

Для представления шинных адресов отображение DMA устанавливает новый тип, **dma_addr_t**. Переменные типа **dma_addr_t** должны рассматриваться драйвером как непрозрачные; единственными допустимыми операциями являются передачи их подпрограммам поддержки DMA и в само устройство. Как и шинный адрес, **dma_addr_t** может привести к неожиданным проблемам, если использовать его непосредственно центральным процессором.

Код PCI различает два типа отображений DMA, в зависимости от ожидания того, как долго будет существовать буфер DMA:

Согласованные отображения DMA

Эти отображения обычно существуют во время жизни драйвера. Согласованный буфер должен быть одновременно доступным для процессора и периферии (другие типы отображения, как мы увидим позднее, могут быть доступны в любой момент времени только одному или другому). В результате согласованные отображения должны находиться в согласованной с кэшем памяти. Согласованные отображения могут быть дорогими в настройке и использовании.

Потоковые отображения DMA

Потоковые отображения, как правило, создаются на одну операцию. Как мы увидим, некоторые архитектуры позволяют значительную оптимизацию, когда используются потоковые отображения, но эти отображения также управляются строгим набором правил в том, как они могут быть доступны. Когда это возможно, разработчики ядра рекомендуют использовать потоковые отображения вместо согласованных отображений. Есть две причины для такой рекомендации. Первой является то, что в системах, которые поддерживают регистры отображения, каждое отображение DMA использует на шине один или более из них. Согласованные отображения, которые имеют долгую жизнь, могут монополизировать эти регистры в течение длительного времени, даже когда они не используются. Другая причина состоит в том, что на некотором оборудовании потоковые отображения могут быть оптимизированы способами, недоступными для согласованных отображений.

Эти два типа отображения должны управляться разными способами; настало время взглянуть на подробности.

Создание согласованных отображений DMA

Драйвер может создать согласованное отображение вызовом ***dma_alloc_coherent***:

```
void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t
*dma_handle, int flag);
```

Эта функция обрабатывает как выделение, так и отображение буфера. Первыми двумя аргументами являются структура устройства и размер необходимого буфера. Функция возвращает результат отображения DMA в двух местах. Возвращаемым значением функции является виртуальный адрес ядра для буфера, который может быть использован драйвером; между тем, адрес соответствующей шины возвращается в **dma_handle**. Выделение выполняется в этой функции, так что буфер находится в месте, которое работает с DMA; обычно память выделяется только с помощью ***get_free_pages*** (но заметьте, что размер в

байтах, а не в значении порядка). Аргумент **flag** является обычным значением **GFP_**, описывающим как будет выделена память; как правило, он должен быть **GFP_KERNEL** (обычно) или **GFP_ATOMIC** (при работе в атомарном контексте).

Когда буфер больше не нужен (обычно во время выгрузки модуля), он должен быть возвращён системе с помощью ***dma_free_coherent***:

```
void dma_free_coherent(struct device *dev, size_t size, void *vaddr,
dma_addr_t dma_handle);
```

Обратите внимание, что эта функция, как и многие универсальные функции DMA, требует, чтобы были указаны все аргументы: размер, процессорный адрес и шинный адрес.

Пулы DMA

Пул DMA представляет собой механизм выделения для небольших, согласованных отображений DMA. Отображения, полученные от ***dma_alloc_coherent*** могут иметь минимальный размер в одну страницу. Если вашему устройству необходимы небольшие области DMA, чем это, лучше всего использовать пул DMA. Пулы DMA также полезны в ситуациях, когда может возникнуть соблазн выполнять DMA для небольших областей, встроенных в более крупные структуры. Некоторые весьма непонятные ошибки драйверов были прослежены до проблем согласования кэша с полями структур, находящимися рядом с небольшими участками DMA. Чтобы избежать этой проблемы, вы всегда должны выделять области для операций DMA явно, в стороне от других, не-DMA структур данных.

Функции пула DMA определены в ***<linux/dmapool.h>***.

Пул DMA должен быть создан до использования вызовом:

```
struct dma_pool *dma_pool_create(const char *name, struct device *dev,
                                 size_t size, size_t align,
                                 size_t allocation);
```

Тут, **name** - это имя для пула, **dev** - ваше структура устройства, **size** - размер буферов, которые будут выделены из этого пула, **align** - необходимое аппаратное выравнивание для выделений из пула (выраженное в байтах) и **allocation**, если не равно нулю, предел памяти, которую выделения не должны превышать. Если, например, **allocation** передана как 4096, буферы, выделенные из этого пула не пересекают границы в 4 Кб.

Когда вы завершили работу с пулом, он может быть освобождён с помощью:

```
void dma_pool_destroy(struct dma_pool *pool);
```

Вы должны вернуть все выделения в пул до его уничтожения.

Выделения обрабатываются с помощью ***dma_pool_alloc***:

```
void *dma_pool_alloc(struct dma_pool *pool, int mem_flags, dma_addr_t
*handle);
```

Для этого вызова, **mem_flags** является обычным набором флагов выделения **GFP_**. Если всё идёт хорошо, область памяти (размера, указанного при создании пула) выделяется и

происходит возврат. Как и с ***dma_alloc_coherent***, адрес результирующего буфера DMA возвращается в виде виртуального адреса ядра и хранится в **handle** как шинный адрес. Ненужные буферы должны быть возвращены в пул с помощью:

```
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t addr);
```

Создание потоковых отображений DMA

Потоковые отображения по ряду причин имеют более сложный интерфейс, чем согласованный вариант. Эти отображения рассчитывают на работу с буфером, который уже был выделен драйвером, и, следовательно, иметь дело с адресами, которые они не выбирают. На некоторых архитектурах потоковые отображения могут также иметь несколько несоприкасающихся страниц и составные буферы "разборки/сборки". По всем этим причинам потоковые отображения имеют свой собственный набор функций отображения.

При создании потокового отображения вы должны сообщить ядру, в каком направлении движутся данные. Для этой цели были определены некоторые символы (с типом **enum dma_data_direction**):

DMA_TO_DEVICE
DMA_FROM_DEVICE
Эти два символа должна быть достаточно очевидны. Если данные отправляются в устройство (возможно, в ответ на системный вызов ***write***), следует использовать **DMA_TO_DEVICE**; данные, идущие в процессор, наоборот, обозначаются **DMA_FROM_DEVICE**.

DMA_BIDIRECTIONAL
Если данные могут двигаться в любом направлении, используйте **DMA_BIDIRECTIONAL**.

DMA_NONE
Этот символ предоставляется только как помощь при отладке. Попытки использовать буферы с этим "направлением" вызовут панику ядра.

Может возникнуть желание всегда просто указывать **DMA_BIDIRECTIONAL**, но авторам драйверов не следует поддаваться этому соблазну. На некоторых архитектурах за такой выбор придётся заплатить падением производительности.

Если у вас для передачи есть единственный буфер, отображайте его с помощью ***dma_map_single***:

```
dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size,
                          enum dma_data_direction direction);
```

Возвращаемым значением является шинный адрес, который вы можете передать в устройство или NULL, если что-то пойдёт не так.

После завершения передачи отображение должно быть удалено с помощью ***dma_unmap_single***:

```
void dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                      enum dma_data_direction direction);
```

Здесь, аргументы **size** и **direction** должны соответствовать тем, которые использовались для отображения буфера.

Для потокового отображения DMA применяются некоторые важные правила:

- Буфер должен быть использован только для передачи, которая соответствует значению заданного при отображении направления.
- После того, как буфер был отображён, он принадлежит устройству, а не процессору. До тех пор, пока буфер не отключён, драйвер не должен касаться его содержимого любым способом. Драйверу безопасно обращаться к содержимому буфера только после того, как была вызвана ***dma_unmap_single*** (с одним исключением, которое мы увидим в ближайшее время). Среди прочего, это правило предполагает, что буфер записи для устройство не может быть отображён, пока он не содержит все данные для записи.
- Буфер не должен быть отключён во время активности DMA, либо гарантирована серьёзная нестабильность системы.

Вы можете быть удивлены, почему драйвер не может работать с буфером, как только он был отображён. Фактически, есть две причины, почему это правило имеет смысл. Во-первых, когда буфер отображается для DMA, ядро должно гарантировать, что все данные в этом буфере были записаны в память на самом деле. Вполне вероятно, что когда вызывается ***dma_map_single***, некоторые данные находятся в кэше процессора и должны быть явно сброшены в память. Данные, записанные в буфер процессором после сброса, могут быть не видны для устройства.

Во-вторых, рассмотрим, что произойдёт, если буфер отображён в область памяти, которая не доступна для устройства. Некоторые архитектуры в этом случае просто дают ошибку, но другие создают возвратный буфер. Возвратный буфер - это просто отдельная область памяти, доступная для устройства. Если буфер отображается с направлением **DMA_TO_DEVICE** и требуется возвратный буфер, содержимое оригинального буфера копируется как часть операции отображения. Ясно, что изменения в оригинальном буфере после копирования устройству не видны. Аналогичным образом при **DMA_FROM_DEVICE** возвратные буферы копируются обратно в исходный буфер функцией ***dma_unmap_single***; данные из устройства не имеют смысла, пока не выполнено это копирование.

Кстати, возвратные буферы являются одной из причин, почему важно правильно задавать направление. Возвратные буферы при **DMA_BIDIRECTIONAL** копируются до и после операции, что часто является ненужной тратой циклов процессора.

Иногда драйверу необходимо получить доступ к содержимому потокового буфера DMA без его отключения. Это делает возможным следующий вызов:

```
void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr,
                             size_t size, enum dma_data_direction direction);
```

Эта функция должна быть вызвана перед тем, как процессор обращается к потоковому буферу DMA. После того, как вызов был сделан, процессор "владеет" буфером DMA и может работать с ним, как это требуется. Однако, перед доступом в буфер устройства, право собственности должно быть ему возвращено:

```
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr,
```

```
                                    size_t size, enum dma_data_direction
direction);
```

Процессор, опять же, не должен обращаться к буферу DMA после того, как был сделан этот вызов.

Одностраничные потоковые отображения

Иногда вы можете захотеть настроить отображение на буфер для которого у вас есть указатель **struct page**; например, это может произойти с буферами пространства пользователя, отображаемыми с помощью ***get_user_pages***. Чтобы создать и снести потоковые отображения с использованием указателей **struct page**, используйте следующее:

```
dma_addr_t dma_map_page(struct device *dev, struct page *page,
                        unsigned long offset, size_t size,
                        enum dma_data_direction direction);

void dma_unmap_page(struct device *dev, dma_addr_t dma_address,
                    size_t size, enum dma_data_direction direction);
```

Для отображения части страницы могут быть использованы аргументы **offset** и **size**. Однако, рекомендуется избегать отображения неполных страниц, пока вы действительно не уверены в том, что делаете. Отображаемая часть страницы может привести к возникновению проблем с согласованием кэшей, если такое выделение охватывает только часть строки кэша; это, в свою очередь, может привести к повреждению памяти и ошибкам, чрезвычайно трудным для отладки.

Преобразования разборки/сборки

Отображения разборки/сборки представляют собой особый тип потокового отображения DMA. Предположим, что у вас есть несколько буферов и все они должны быть переданы в или из устройства. Эта ситуация может произойти несколькими способами, включающими системные вызовы ***readv*** или ***writev***, кластерный дисковый запрос ввода/вывода или печать страниц в отображённом буфере ввода/вывода ядра. Можно просто по очереди отображать каждый буфер и выполнять необходимые операции, но в отображении всего списка сразу есть свои преимущества.

Многие устройства могут принимать список разборки (scatterlist) из указателей массивов и длин, и передавать их все за одну операцию DMA; например, сетевое "нулевое копирование" (“zero-copy”) становится проще, если пакеты могут быть собраны из множества кусков. Другой причиной отображать списки разборки как целое является использование преимуществ систем, которые имеют в шинном оборудовании регистры отображения. На таких системах с точки зрения устройства физически разрозненные страницы могут быть собраны в единый непрерывный массив. Этот метод работает только когда записи в списке разборки равны по длине размеру страницы (за исключением первого и последнего), но когда он работает, это может обернуться несколькими операциями в одном DMA и, соответственно, ускорить процесс.

Наконец, если должен быть использован возвратный буфер, имеет смысл объединять весь список в один буфер (поскольку он всё равно копируется).

Итак, теперь вы убеждены, что в некоторых ситуациях отображение списков разборки

является стоящим. Первый шаг в отображении списка разборки заключается в создании и заполнении массива **struct scatterlist**, описывающего буферы для передачи. Эта структура архитектурно-зависима и описана в ***<asm/scatterlist.h>***. Тем не менее, она всегда содержит три поля:

struct page *page;
Указатель **struct page**, соответствующий буферу, используемому в операции разборки/сборки.

unsigned int length;
unsigned int offset;
Длина этого буфера и его смещение на странице.

Чтобы отобразить операцию разборки/сборки DMA, для каждого буфера, подлежащего передаче, ваш драйвер должен задать в записи **struct scatterlist** поля **page**, **offset** и **length**. Затем вызовите:

```
int dma_map_sg(struct device *dev, struct scatterlist *sg, int nents,
               enum dma_data_direction direction)
```

где **nents** является количеством передаваемых записей списка разборки. Возвращаемое значение является количеством буферов DMA для передачи; оно может быть меньше, чем **nents**.

Для каждого буфера для передачи в устройство во входном списке разборки ***dma_map_sg*** определяет собственный шинный адрес. В рамках этой задачи, она также объединяет буферы, которые находятся в памяти рядом друг с другом. Если система, на которой работает ваш драйвер, имеет блок управления памятью ввода/вывода, ***dma_map_sg*** также программирует регистры устройства отображения, с возможным результатом, что с точки зрения вашего устройства вы сможете передать единый непрерывный буфер. Однако, вы никогда не узнаете, как будет выглядеть результирующая передача, пока не сделаете вызов.

Ваш драйвер должен передавать каждый буфер, возвращаемый ***dma_map_sg***. Шинный адрес и длина каждого буфера хранятся в записях **struct scatterlist**, но их расположение в записях структуры меняется от одной архитектуры к другой. Чтобы можно было писать переносимый код, были определены два макроса:

dma_addr_t sg_dma_address(struct scatterlist *sg);
Возвращает шинный (DMA) адрес из этой записи списка разборки.

unsigned int sg_dma_len(struct scatterlist *sg);
Возвращает длину этого буфера.

Опять же, помните, что адреса и длины буферов передачи могут отличаться от того, что были переданы в ***dma_map_sg***.

После завершения передачи, отображение разборки/сборки отключается с помощью вызова ***dma_unmap_sg***:

```
void dma_unmap_sg(struct device *dev, struct scatterlist *list,
                  int nents, enum dma_data_direction direction);
```

Обратите внимание, что **nents** должно быть количеством записей, которые вы передавали перед этим в ***dma_map_sg***, а не количеством DMA буферов, которые были вам возвращены функцией.

Отображения разборки/сборки являются потоковыми отображениями DMA и к ним, как к одной из разновидностей, применяются те же правила доступа. Если необходимо получить доступ к отображённому списку разборки/сборки, сначала необходимо его синхронизировать:

```
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                         int nents, enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
                            int nents, enum dma_data_direction direction);
```

Двухадресный цикл отображения PCI

Обычно, уровень поддержки DMA работает с 32-х разрядными шинными адресами, возможно, ограниченными маской DMA определённого устройства. Шина PCI, однако, также поддерживает 64-х разрядный режим адресации с циклом двойной адресации (double-address cycle, DAC). Универсальный уровень DMA не поддерживает этот режим по ряду причин, первая из которых в том, что это возможность специфична для PCI. Кроме того, во многих реализациях DAC в лучшем случае глючит и поскольку DAC медленнее, чем обычный, 32-х разрядный DMA, это может быть накладным расходом. Тем не менее, существуют приложения, где использование DAC может быть тем, что делать правильно; если вы имеете устройство, которое может работать с очень большими буферами, размещёнными в верхней памяти, вы можете рассмотреть вопрос о реализации поддержки DAC. Эта поддержка доступна только для шины PCI, так что должны быть использованы процедуры, предназначенные для PCI.

Для использования DAC ваш драйвер должен подключить ***<linux/pci.h>***. Вы должны установить отдельную маску DMA:

```
int pci_dac_set_dma_mask(struct pci_dev *pdev, u64 mask);
```

Вы можете использовать DAC адресацию только если эта функция возвращает 0.

Для DAC отображений используется специальный тип (**dma64_addr_t**). Чтобы установить одно из таких отображений, вызовите ***pci_dac_page_to_dma***:

```
dma64_addr_t pci_dac_page_to_dma(struct pci_dev *pdev, struct page *page,
                                 unsigned long offset, int direction);
```

Вы заметите, что DAC отображения могут быть сделаны только из указателей **struct page** (в конце концов, они должны находиться в верхней памяти, или в их использовании нет смысла); они должны быть созданы одной страницей за раз. Аргумент **direction** является эквивалентом PCI для **enum dma_data_direction**, используемого в универсальном уровне DMA; он должен быть **PCI_DMA_TODEVICE**, **PCI_DMA_FROMDEVICE** или **PCI_DMA_BIDIRECTIONAL**.

Отображения DAC не требуют внешних ресурсов, поэтому нет необходимости явно освобождать их после использования. Следует, однако, относиться к DAC отображениям как и другим потоковым отображениям и соблюдать правила, касающиеся владения буфером. Существует набор функций для синхронизации буферов DMA, которые аналогичны универсальному варианту:

```
void pci_dac_dma_sync_single_for_cpu(struct pci_dev *pdev,
                                     dma64_addr_t dma_addr,
                                     size_t len,
                                     int direction);

void pci_dac_dma_sync_single_for_device(struct pci_dev *pdev,
                                        dma64_addr_t dma_addr,
                                        size_t len,
                                        int direction);
```

Простой пример DMA для PCI

В качестве примера того, как могут быть использованы отображения DMA, приведём простой пример кодирования DMA для PCI устройства. Фактический вид операций DMA на шине PCI сильно зависит от управляемого устройства. Таким образом, этот пример не будет применим к какому-то реальному устройству; вместо этого, он является частью гипотетического драйвера, названного **dad** (DMA Acquisition Device, Устройство, получающее DMA). Драйвер для этого устройства может определить передающую функцию следующим образом:

```
int dad_transfer(struct dad_dev *dev, int write, void *buffer, size_t count)
{
    dma_addr_t bus_addr;

    /* Отобразить буфер для DMA */
    dev->dma_dir = (write ? DMA_TO_DEVICE : DMA_FROM_DEVICE);
    dev->dma_size = count;
    bus_addr = dma_map_single(&dev->pci_dev->dev, buffer, count, dev-
>dma_dir);
    dev->dma_addr = bus_addr;

    /* Проинициализировать устройство */

    writeb(dev->registers.command, DAD_CMD_DISABLEDMA);
    writeb(dev->registers.command, write ? DAD_CMD_WR : DAD_CMD_RD);
    writel(dev->registers.addr, cpu_to_le32(bus_addr));
    writel(dev->registers.len, cpu_to_le32(count));

    /* Начать операцию */
    writeb(dev->registers.command, DAD_CMD_ENABLEDMA);
    return 0;
}
```

Эта функция отображает буфер для передачи и начинает операцию с устройством. Другая половина работы должна быть сделана процедурой обслуживания прерывания, которая выглядит следующим образом:

```
void dad_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct dad_dev *dev = (struct dad_dev *) dev_id;

    /* Убедиться, что это прерывание действительно от нашего устройства */

```

```
    /* Отключить буфер DMA */
    dma_unmap_single(dev->pci_dev->dev, dev->dma_addr, dev->dma_size, dev-
>dma_dir);

    /* Только теперь обращаться к буферу безопасно, копируем пользователю, и
так далее. */
    ...
}
```

Очевидно, остальные подробности были исключены из этого примера, включая все действия, которые могут быть необходимы для предотвращения попытки запустить несколько одновременных операций DMA.

DMA для устройств ISA

Шина ISA допускает два вида передачи DMA: простой (native) DMA и управление DMA по шине ISA (ISA bus master DMA). Простой DMA использует для управления сигнальными линиями на шине ISA стандартную схему контроллера DMA на материнской плате. С другой стороны, управление DMA по шине ISA обрабатывается полностью за счёт периферийного устройства. Последний тип DMA используется редко и не требует обсуждения здесь, потому что он похож на DMA для устройств PCI, по крайней мере с точки зрения драйвера. Примером управления по шине ISA является SCSI контроллер 1542, драйвером которого в исходных текстах ядра является ***drivers/scsi/aha1542.c***.

Что касается простого DMA, есть три объекта, участвующие в передаче данных DMA по шине ISA:

DMA контроллер 8237 (DMAC)

Контроллер хранит информацию о передаче DMA, такую как направление, адрес в памяти и размер передачи. Он также содержит счётчик, который отслеживает состояние текущих передач. Когда контроллер получает сигнал запроса DMA, он получает управление шиной и управляет сигнальными линиями, так что устройство может читать или записывать свои данные.

Периферийное устройство

Устройство должно активировать сигнал запроса DMA, когда оно готово к передаче данных. Фактическая передача управляется DMAC; аппаратное устройство последовательно читает и записывает данные по шине, когда контроллер стробирует устройство. Когда передача закончилась, устройство обычно вызывает прерывание.

Драйвер устройства

Драйвер делает немного; он предоставляет контроллер DMA направление, шинный адрес и размер передачи. Он также сообщает своей периферии подготовиться для передачи данных и отвечает на прерывание, когда DMA завершается.

Оригинальный контроллер DMA, используемый в ПК мог управлять четырьмя "каналами", каждый из которых связан с одним набором регистров DMA. В один момент времени в контроллере DMA могли хранить свою информацию четыре устройства. Новые ПК содержат эквивалент двух устройств DMAC: (* Эти схемы теперь являются частью чипсета материнской платы, но несколько лет назад они были двумя отдельными чипами 8237.) второй контроллер (ведущий) подключен к системному процессору, а первый (ведомый) подключается к каналу 0

второго контроллера. (* Оригинальный ПК имел только один контроллер; второй был добавлен в платформах на основе 286. Тем не менее, второй контроллер подключен как ведущий, поскольку он обрабатывает 16-ти разрядные передачи; первый передаёт только восемь бит за раз и существует для обеспечения обратной совместимости.)

Каналы нумеруются как 0 - 7: 4 канал не доступен для ISA периферии, поскольку он используется для внутреннего каскадирования ведомого и ведущего контроллера. Таким образом, доступными каналами являются 0 - 3 на ведомом (8-ми разрядные каналы) и 5 - 7 на ведущем (16-ти разрядные каналы). Размер любой передачи DMA, хранящийся в контроллере, представляет собой 16-ти разрядное число, представляющее собой количество шинных циклов. Таким образом, максимальный размер передачи - 64 Кб для ведомого контроллера (поскольку он передаёт восемь бит за один цикл) и 128 Кб для ведущего (который выполняет 16-ти разрядные передачи).

Поскольку контроллер DMA является общесистемным ресурсом, ядро помогает работе с ним. Оно использует регистрацию DMA для предоставления механизма запроса и освобождения для каналов DMA и набор функций для настройки канальной информации в контроллере DMA.

Регистрация использования DMA

Вы должны привыкнуть к регистрациям в ядре - мы уже видели их для портов ввода/вывода и линий прерывания. Регистрация канала DMA похожа на другие. После того, как был подключен ***<asm/dma.h>***, могут быть использованы следующие функции для получения и освобождения собственности для канала DMA:

```
int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);
```

Аргумент **channel** является числом от 0 до 7, или, точнее, положительным числом, меньше **MAX_DMA_CHANNELS**. На ПК **MAX_DMA_CHANNELS** определён как 8 для соответствия оборудованию. Аргументом **name** является строка, идентифицирующая устройство. Указанное имя появляется в файле ***/proc/dma***, который может быть прочитан пользовательской программой.

Возвращаемое значение из ***request_dma*** равно 0 для успешного выполнения и **-EINVAL** или **-EBUSY**, если была ошибка. Первая означает, что запрашиваемый канал не соответствует диапазону, а последняя означает, что канал занят другим устройством.

Мы рекомендуем вам быть такими же внимательными с каналами DMA, как и с портами ввода/вывода и линиями прерывания; гораздо лучше запрашивать канал во время ***открытия***, чем запрашивать его в функции инициализации модуля. Задержка запроса позволяет некоторое совместное использование драйверами; например, ваша звуковая карта и ваш аналоговый интерфейс ввода/вывода может делить канал DMA, пока они не будут использоваться одновременно.

Мы также рекомендуем вам запрашивать канал DMA ***после*** того, как вы запросили линию прерывания и чтобы вы освободили его ***перед*** прерыванием. Это является обычным порядком запроса этих двух ресурсов; следующее соглашение позволяет избежать возможных взаимоблокировок. Обратите внимание, что каждому устройству, использующему DMA, необходима также и линия IRQ; в противном случае оно бы не смогло просигнализировать о завершении передачи данных.

В типичном случае код для ***open*** выглядит следующим образом, который относится к нашему гипотетическому модулю ***dad***. Устройство ***dad*** использует, как показано, быстрый обработчик прерывания без поддержки разделяемых линий прерываний.

```
int dad_open (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;
    /* ... */
    if ( (error = request_irq(my_device.irq, dad_interrupt, SA_INTERRUPT,
"dad", NULL)) )
        return error; /* или реализовать блокирующее открытие */

    if ( (error = request_dma(my_device.dma, "dad")) ) {
        free_irq(my_device.irq, NULL);
        return error; /* или реализовать блокирующее открытие */
    }
    /* ... */
    return 0;
}
```

Реализация ***close***, которая соответствует только что показанному ***open***, выглядит следующим образом:

```
void dad_close (struct inode *inode, struct file *filp)
{
    struct dad_device *my_device;

    /* ... */
    free_dma(my_device.dma);
    free_irq(my_device.irq, NULL);
    /* ... */
}
```

Вот как выглядит на системе с установленной звуковой картой файл ***/proc/dma***:

```
merlino% cat /proc/dma
 1: Sound Blaster8
 4: cascade
```

Интересно отметить, что в драйвере звука по умолчанию получает канал DMA при загрузка системы и не освобождает его. Запись **cascade** является заполнителем, показывающим, что канал 4 не доступен для драйверов, как было объяснено ранее.

Общение с контроллером DMA

После регистрации основная часть работы драйвера состоит в настройке для надлежащего функционирования контроллера DMA. Эта задача не тривиальна, но, к счастью, ядро экспортирует все функции, необходимые для типичного драйвера.

Драйверу необходимо настроить контроллер DMA либо когда вызываются ***read*** или ***write***, или при подготовке к асинхронным передачам. Эта последняя задача выполняется либо во время ***открытия***, или в ответ на команду ***ioctl***, в зависимости от драйвера и политики,

которую он реализует. Код, приведенный здесь, представляет собой код, который обычно вызывается методами устройства ***read*** или ***write***.

В этом разделе приводится краткий обзор внутренностей контроллера DMA, чтобы вы поняли приведённый здесь код. Если вы хотите узнать больше, мы бы настоятельно призвали вас прочитать ***<asm/dma.h>*** и некоторые руководства к оборудованию с описанием архитектуры ПК. В частности, мы не имеем дело с вопросом о 8-ти разрядных передач данных взамен 16-ти разрядных. Если вы пишете драйверы устройств для плат ISA устройств, для таких устройств вам следует найти соответствующую информацию в руководстве по оборудованию.

Контроллер DMA является разделяемым ресурсом и могла бы возникнуть путаница, если бы его попытался запрограммировать одновременно более чем один процессор. По этой причине контроллер защищён спин-блокировкой, называемой ***dma_spin_lock***. Драйверам не следует манипулировать с этой блокировкой напрямую, однако, чтобы сделать это, для вас предоставлены две функции:

unsigned long claim_dma_lock( );
Приобретает спин-блокировку DMA. Эта функция также блокирует прерывания на местном процессоре; поэтому возвращаемое значение представляет собой набор флагов, описывающих предыдущее состояние прерывания; он должен быть передан следующей функции для восстановления состояния прерывания, когда вы завершили работу с блокировкой.

void release_dma_lock(unsigned long flags);
Возвращает спин-блокировкой DMA и восстанавливает предыдущий статус прерывания.

При использовании функций, описанных далее, должна удерживаться спин-блокировка. Однако, не следует её удерживать в ходе фактического ввода/вывода. При удержании спин-блокировки драйвер никогда не должен засыпать.

Информация, которая должна быть загружена в контроллер, состоит из трёх элементов: адрес ОЗУ, число атомарных объектов, которые должны быть переданы (в байтах или словах), и направление передачи. С этой целью ***<asm/dma.h>*** экспортирует следующие функции:

void set_dma_mode(unsigned int channel, char mode);
Показывает, какой канал должен читать из устройства (**DMA_MODE_READ**) или записывать в него (**DMA_MODE_WRITE**). Существует третий режим, **DMA_MODE_CASCADE**, который используется для освобождения управления шиной. Каскадирование является способом подключения первого контроллера ко второму, но может также использоваться как настоящие устройства управления шиной ISA. Мы не будем обсуждать здесь управление шиной.

void set_dma_addr(unsigned int channel, unsigned int addr);
Присваивает адрес буферу DMA. Функция сохраняет 24 младших значащих бит **addr** в контроллере. Аргумент **addr** должен быть шинным адресом (смотрите раздел "Шинные адреса" 425 ранее в этой главе).

void set_dma_count(unsigned int channel, unsigned int count);
Задаёт количество байт для передачи. Аргумент **count** представляет число байтов также и для 16-ти разрядных каналов; в этом случае число должно быть четным.

В дополнение к этим функциям существует ряд возможностей для ведения домашнего хозяйства, которые должны использоваться при работе с устройствами DMA:

void disable_dma(unsigned int channel);
Канал DMA может быть отключен в контроллере. Для предотвращения сбоев в работе, до того, как настроен контроллер, канал должны быть отключён. (В противном случае, может возникнуть повреждение, поскольку контроллер программируется с помощью 8-ми разрядных передач данных и, следовательно, ни одна из предыдущих функций не выполняется атомарно).

void enable_dma(unsigned int channel);
Данная функция сообщает контроллеру, что канал DMA содержит достоверные данные.

int get_dma_residue(unsigned int channel);
Иногда драйверу необходимо знать, когда передача DMA завершена. Эта функция возвращает количество байтов, которые ещё не переданы. Возвращаемое значение 0 после успешной передачи и непредсказуемо (но не 0), пока контроллер работает. Непредсказуемость исходит из необходимости получения 16-ти разрядного остатка через две 8-ми разрядные операции ввода.

void clear_dma_ff(unsigned int channel);
Эта функция очищает триггер (flip-flop, регистр данных) DMA. Триггер используется для управления доступом к 16-ти разрядным регистрам. Регистры доступны через две последовательные 8-ми разрядные операции, а для выбора младшего байта (когда он очищается), или старшего байта (если он устанавливается) используется триггер. После передачи восьми бит триггер автоматически меняет состояние; перед обращением к регистрам DMA программист должен очистить триггер (чтобы установить его в известное состояние).

Используя эти функции, для подготовке к передаче DMA драйвер может реализовать следующую функцию:

```
int dad_dma_prepare(int channel, int mode, unsigned int buf, unsigned int
count)
{
    unsigned long flags;

    flags = claim_dma_lock( );
    disable_dma(channel);
    clear_dma_ff(channel);
    set_dma_mode(channel, mode);
    set_dma_addr(channel, virt_to_bus(buf));
    set_dma_count(channel, count);
    enable_dma(channel);
    release_dma_lock(flags);

    return 0;
}
```

Затем для проверки для успешного завершения DMA используется следующая функция:

```
int dad_dma_isdone(int channel)
```

```
{
    int residue;
    unsigned long flags = claim_dma_lock ( );
    residue = get_dma_residue(channel);
    release_dma_lock(flags);
    return (residue == 0);
}
```

Единственное, что остаётся сделать, это настроить плату устройства. Эта задача зависит от устройства и обычно состоит в чтении или записи нескольких портов ввода/вывода. Устройства отличаются очень сильно. Например, некоторые устройства ожидают, что программист сообщит оборудованию, насколько велик буфер DMA, а иногда драйвер должен прочитать значения, которые находятся в устройстве. Для настройки платы вашим единственным другом будет руководство по оборудованию.

Краткая справка

Эта глава ввела следующие символы, связанные с обработкой памяти.

Вводный материал

#include <linux/mm.h>
#include <asm/page.h>
Большинство функций и структур, связанных с управлением памятью, являются прототипизированными и определёнными в этих заголовочных файлах.

void *__va(unsigned long physaddr);
unsigned long __pa(void *kaddr);
Макросы, которые выполняют преобразования между логическими адресами ядра и физическими адресами.

PAGE_SIZE
PAGE_SHIFT
Константы, которые дают размер (в байтах) страницы используемого оборудования и число битов, на которое номер блока страницы должен быть сдвинут для преобразования его в физический адрес.

struct page
Структура, которая представляет собой аппаратную страницу в карте памяти системы.

struct page *virt_to_page(void *kaddr);
void *page_address(struct page *page);
struct page *pfn_to_page(int pfn);
Макрос, который выполняет преобразование между логическими адресами ядра и связанными с ними записями в карте памяти. ***page_address*** работает только для страниц нижней памяти или страниц верхней памяти, которые были отображены явным образом. ***pfn_to_page*** преобразует номера страничного блока в связанный с ним указатель **struct page**.

unsigned long kmap(struct page *page);
void kunmap(struct page *page);
kmap возвращает виртуальный адрес ядра, который отображается на данную страницу, создавая отображение, если это будет необходимо. ***kunmap*** удаляет отображение для данной страницы.

#include <linux/highmem.h>
#include <asm/kmap_types.h>

void *kmap_atomic(struct page *page, enum km_type type);
void kunmap_atomic(void *addr, enum km_type type);
Высокопроизводительная версия ***kmap***; результирующие отображения могут быть проведены только атомарным кодом. Для драйверов типы должны быть **KM_USER0**, **KM_USER1**, **KM_IRQ0** или **KM_IRQ1**.
struct vm_area_struct;
Структура, описывающая VMA.

Реализация mmap

int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_add, unsigned long pfn, unsigned long size, pgprot_t prot);
int io_remap_page_range(struct vm_area_struct *vma, unsigned long virt_add, unsigned long phys_add, unsigned long size, pgprot_t prot);
Функции, которые находятся в центре ***mmap***. Они отображают **size** байт физических адресов, начиная с номера страницы, указанного **pfn** для виртуального адреса **virt_add**. Защитные биты, связанные с виртуальным пространством, указаны в **prot**. Когда целевой адрес находится в пространстве памяти ввода/вывода, следует использовать ***io_remap_page_range***.
struct page *vmalloc_to_page(void *vmaddr);
Преобразует виртуальный адреса, полученный от ***vmalloc***, в соответствующий указатель **struct page**.

Выполнение прямого ввода/вывода

int get_user_pages(struct task_struct *tsk, struct mm_struct *mm, unsigned long start, int len, int write, int force, struct page **pages, struct vm_area_struct **vmas);
Функция, которая блокирует буфер пользовательского пространства в памяти и возвращает соответствующие указатели **struct page**. Вызывающий должен иметь **mm->mmap_sem**.
SetPageDirty(struct page *page);
Макрос, который отмечает данную страницу как "грязную" (изменённую) и нуждающуюся в записи в резервное хранилище прежде, чем она может быть освобождена.
void page_cache_release(struct page *page);
Освобождает данную страницу из страничного кэша.
int is_sync_kiocb(struct kiocb *iocb);
Макрос, который возвращает ненулевое значение, если данное IOCB требует синхронного выполнения.
int aio_complete(struct kiocb *iocb, long res, long res2);
Функция, которая свидетельствует о завершении операции асинхронного ввода/вывода.

Прямой доступ к памяти

#include <asm/io.h>
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
Устаревшие и осуждаемые для использования функции, которые выполняют преобразования между адресами ядра, виртуальными и шинными. Для общения с периферийными устройствами должны быть использованы шинные адреса.

#include <linux/dma-mapping.h>

Заголовочный файл, необходимый для определения универсальных функций DMA.

int dma_set_mask(struct device *dev, u64 mask);

Для периферийных устройств, которые не могут решить адресовать полностью 32-х разрядный диапазон, эта функция сообщает ядру адресуемый диапазон и возвращает ненулевое значение, если DMA является возможным.

void *dma_alloc_coherent(struct device *dev, size_t size, dma_addr_t *bus_addr, int flag)
void dma_free_coherent(struct device *dev, size_t size, void *cpuaddr, dma_handle_t bus_addr);

Выделяют и освобождают согласованные отображения DMA для буфера, который будет существовать в течении жизни драйвера.

#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, int mem_flags, dma_addr_t *handle);
void dma_pool_free(struct dma_pool *pool, void *vaddr, dma_addr_t handle);

Функции, которые создают, уничтожают и используют пулы DMA для управления небольшими областями DMA.

enum dma_data_direction;
DMA_TO_DEVICE
DMA_FROM_DEVICE
DMA_BIDIRECTIONAL
DMA_NONE

Символы, используемые, чтобы указать функциям потокового отображения направление, в котором передаются данные, в или из буфера.

dma_addr_t dma_map_single(struct device *dev, void *buffer, size_t size, enum dma_data_direction direction);
void dma_unmap_single(struct device *dev, dma_addr_t bus_addr, size_t size, enum dma_data_direction direction);

Создают и уничтожают одноразовое потоковое отображение DMA.

void dma_sync_single_for_cpu(struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);
void dma_sync_single_for_device(struct device *dev, dma_handle_t bus_addr, size_t size, enum dma_data_direction direction);

Синхронизация буфера, который имеет потоковое отображение. Эти функции должны быть использованы, если процессор должен получить доступ к буферу в то время, как потоковое отображение имеет место (то есть, в то время, когда буфером владеет устройство).

#include <asm/scatterlist.h>
struct scatterlist { /* ... */ };
dma_addr_t sg_dma_address(struct scatterlist *sg);
unsigned int sg_dma_len(struct scatterlist *sg);

Структура **scatterlist**, описывающая операцию ввода/вывода, которая включает в себя более одного буфера. Для получения шинных адресов и размеров буферов для передачи устройству при реализации операций разборки/сборки могут быть использованы

макросы ***sg_dma_address*** и ***sg_dma_len***.

dma_map_sg(struct device *dev, struct scatterlist *list, int nents,
enum dma_data_direction direction);
dma_unmap_sg(struct device *dev, struct scatterlist *list, int nents,
enum dma_data_direction direction);
void dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg, int nents,
enum dma_data_direction direction);
void dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg, int nents,
enum dma_data_direction direction);

dma_map_sg отображает операцию разборки/сборки, а ***dma_unmap_sg*** отменяет это отображение. Если буферы должны быть доступны, когда отображение активно, для синхронизации могут быть использованы ***dma_sync_sg_****.

/proc/dma

Файл, который содержит текстовый снимок выделенных каналов в контроллерах DMA. DMA, базирующиеся на PCI не показываются, потому что каждая плата работает самостоятельно, без необходимости выделения каналов в контроллере DMA.

#include <asm/dma.h>

Заголовок, который определяет или прототипизирует все функции и макросы, относящиеся к DMA. Он должен быть подключен для использования любого из следующих символов.

int request_dma(unsigned int channel, const char *name);
void free_dma(unsigned int channel);

Выполняют регистрацию DMA. Регистрация должна быть выполнена перед использованием каналов DMA на ISA.

unsigned long claim_dma_lock( );
void release_dma_lock(unsigned long flags);

Запрашивают и освобождают спин-блокировку DMA, которая должна удерживаться до вызова других функций DMA для ISA, описанных далее в этом списке. Они также отключают или снова включают прерывания на местном процессоре.

void set_dma_mode(unsigned int channel, char mode);
void set_dma_addr(unsigned int channel, unsigned int addr);
void set_dma_count(unsigned int channel, unsigned int count);

Программирование информации DMA в контроллер DMA. **addr** является шинным адресом.

void disable_dma(unsigned int channel);
void enable_dma(unsigned int channel);

Во время настройки канал DMA должен быть отключён. Эти функции изменяют состояние канала DMA.

int get_dma_residue(unsigned int channel);

Если драйверу необходимо узнать, как выполняется передача DMA, он может вызвать эту функцию, которая возвращает количество передаваемых данных, которые ещё не переданы. После успешного завершения DMA функция возвращает 0; во время передачи данных значение непредсказуемо.

void clear_dma_ff(unsigned int channel);

Для передачи 16-ти разрядных значений с помощью двух 8-ми разрядных операций контроллером используется DMA триггер. Перед отправкой любых данных в контроллер он должно быть очищен.

Глава 16, Блочные драйверы

Пока наше обсуждение ограничивалось символьными драйверами. Однако, в системах Linux существуют и другие типы драйверов и для нас настало время несколько расширить наши знания. Поэтому в этой главе рассматриваются блочные драйверы.

Блочный драйвер обеспечивает доступ к устройствам, которые передают доступные для произвольного выбора данные в блоках фиксированного размера - в первую очередь, это дисковые драйверы. Видение ядром Linux блочных устройств в корне отличается от символьных устройств; в результате чего блочные драйверы имеют особый интерфейс и свои особые проблемы.

Эффективные блочные драйверы имеют решающее значение для производительности - и не только для явного чтения и записи в пользовательских приложениях. Современные системы с виртуальной памятью работают передвигая (будем надеяться) ненужные данные во вторичное хранилище, которое является обычно дисковым накопителем. Блочные драйверы являются связующим звеном между основной памятью и вторичным хранилищем; таким образом, они могут рассматриваться как представляющие часть подсистемы виртуальной памяти. Хотя возможно написать блочный драйвер, не зная о **struct page** и других важных концепций памяти, тот, кому требуется написать высокопроизводительный драйвер, должен опираться на материал, охваченный в Главе 15 395.

Большая часть разработки блочного уровня сосредоточена на производительности. Многие символьные устройства могут работать ниже своей максимальной скорости и производительность системы в целом не страдает. Однако, система не может работать хорошо, если её блочная подсистема ввода/вывода не является хорошо отлаженной. Интерфейс блочного драйвера в Linux позволяет получить максимальную отдачу от блочного устройства, но неизбежно навязывает уровень сложности, с которой вы должны иметь дело. К счастью, в версии 2.6 блочный интерфейса значительно улучшен по сравнению с тем, что было в старых ядрах.

В этой главе, как и следовало ожидать, в центре внимания пример драйвера, который реализует блочно-ориентированное, базирующееся в памяти устройство. Это, по существу, электронный диск. Ядро уже содержит гораздо превосходящую реализацию электронного диска, но наш драйвер (названный ***sbull***) позволяет нам продемонстрировать создание блочного драйвера при сведении к минимуму необходимой сложности.

Прежде чем углубляться в подробности, давайте точно определим несколько терминов. ***Блок*** представляет собой блок данных фиксированного размера, размер которого определяется ядром. Блоки часто 4096 байт, но это значение может изменяться в зависимости от архитектуры и требования используемой файловой системы. ***Сектор***, напротив, представляет собой небольшой блок, размер которого обычно определяется базовым оборудованием. Ядро предполагает иметь дело с устройствами, которые реализуют секторы по 512 байт. Если ваше устройство использует другой размер, ядра адаптируется и избегает генерации запросов ввода/вывода, которые оборудование обработать не может. Однако, следует помнить, что всякий раз, когда ядро предоставляет вам номер сектора, оно работает в мире секторов по 512 байт. Если вы используете другой размер аппаратного сектора, вы должны номер сектора ядра соответственно отмасштабировать. В драйвере ***sbull*** мы увидим, как это делается.

Регистрация

Блочным драйверам, как и символьным драйверам, необходимо использовать набор интерфейсов регистрации, чтобы их устройства стали для ядра доступными. Концепции являются похожими, но все детали регистрации блочного устройства отличаются. У вас есть для изучения целый ряд новых структур данных и операций устройства.

Регистрация блочного драйвера

Первым шагом, выполняемым большинством блочных драйверов является регистрация себя в ядре. Функцией для выполнения этой задачи является ***register_blkdev*** (которая объявлена в ***<linux/fs.h>***):

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

Аргументами являются старший номер, который будет использовать ваше устройство, и связанное с ним имя (которое ядро будет показывать в ***/proc/devices***). Если **major** передаётся как 0, ядро выделяет новый старший номер и возвращает его вызвавшему. Как всегда, отрицательное возвращаемое значение из ***register_blkdev*** указывает, что произошла ошибка.

Соответствующая функция для отмены регистрации блочного драйвера:

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

Здесь аргументы должны совпадать с переданными в ***register_blkdev*** или эта функция вернёт **-EINVAL** и ничего не отменит.

В ядре версии 2.6 вызов ***register_blkdev*** совершенно не обязателен. Функции, выполняемые ***register_blkdev***, уменьшаются с течением времени; единственными задачами, выполняемыми этим вызовом на данном этапе являются: (1) выделение, если требуется, динамического старшего номера; (2) создание записи в ***/proc/devices***. В будущих ядрах ***register_blkdev*** может быть полностью удалена. Между тем, однако, большинство драйверов до сих пор её вызывают; это традиция.

Регистрация диска

Хотя ***register_blkdev*** и может быть использована для получения основного номер, она не делает доступным системе никакого дискового накопителя. Существует отдельный интерфейс

регистрации, который необходимо использовать для управления отдельными дисками. Использование этого интерфейса требует знания пары новых структур, поэтому с них мы и начинаем.

Операции блочного устройства

Символьные устройства делают свои операции доступными для системы с помощью структуры **file_operations**. Аналогичная структура используется и с блочными устройствами; это **struct block_device_operations**, которая объявлена в ***<linux/fs.h>***. Ниже приводится краткий обзор полей, находящихся в этой структуре; мы рассмотрим их более подробно снова, когда углубимся в детали драйвера ***sbull***:

int (*open)(struct inode *inode, struct file *filp);
int (*release)(struct inode *inode, struct file *filp);
Функции, которые работают как и их эквиваленты символьного драйвера; они вызываются, когда устройство открываться и закрываться. Блочный драйвер мог бы реагировать на вызов открытия увеличением оборотов устройства, блокировкой дверцы (для съёмных носителей) и так далее. Если вы блокируете в устройстве носитель, вы, безусловно, должны разблокировать его методом ***release***.

int (*ioctl)(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
Метод, реализующий системный вызов ***ioctl***. Однако, блочный уровень сначала перехватывает большое количество стандартных запросов; таким образом, большинство методов ***ioctl*** блочного драйвера довольно короткие.

int (*media_changed) (struct gendisk *gd);
Метод, вызываемый ядром, чтобы проверить, поменял ли пользователь накопитель в приводе, возвращает ненулевое значение, если это так. Очевидно, что этот метод применим только к приводам, которые поддерживают сменные носители (и достаточно умны, чтобы сделать флаг "накопитель быть заменён" доступным для драйвера); в других случаях он может быть опущен.
Аргумент **struct gendisk** это то, как ядро представляет себе один диск; мы будем рассматривать эту структуру в следующем разделе.

int (*revalidate_disk) (struct gendisk *gd);
Метод ***revalidate_disk*** вызывается в ответ на замену носителя; это даёт драйверу шанс выполнить любую работу, необходимую для того, чтобы подготовить к использованию новый накопитель. Функция возвращает значение **int**, но это значение ядром игнорируется.

struct module *owner;
Указатель на модуль, который владеет этой структурой; он должен, как правило, быть проинициализированным как **THIS_MODULE**.

Внимательные читатели, возможно, заметили интересное упущение из этого списка: нет функций, которые фактически читают или записывают данные. В блочной подсистеме ввода/вывода эти операции обрабатываются функцией ***request***, которая заслуживает своего собственного большого раздела и будет рассмотрена далее в этой главе. Прежде чем мы сможем поговорить об обслуживании запросов, мы должны завершить наше обсуждение регистрации диска.

Структура gendisk

struct gendisk (объявленная в ***<linux/genhd.h>***) является представлением в ядре отдельного дискового устройства. На самом деле ядро так же использует структуры **gendisk** для представления разделов, но авторы драйверов не должны об этом знать. В **struct gendisk** есть несколько полей, которые должны быть проинициализированы блочным драйвером:

int major;
int first_minor;
int minors;
Поля, которые описывают номер(а) устройства, используемого диском. Как минимум, накопитель должен использовать по крайней мере один младший номер. Однако (и в большинстве случаев так должно быть), если ваш накопитель должен допускать разбиение, необходимо также выделить один младший номер для каждого возможного раздела. Общим значением для **minors** является 16, которое учитывает "полнодисковое" устройство и 15 разделов. Некоторые дисковые приводы используют 64 младших номеров для каждого устройства.

char disk_name[32];
Поле, которое должно содержать имя дискового устройства. Оно появляется в ***/proc/partitions*** и sysfs.

struct block_device_operations *fops;
Содержит операции устройства из предыдущего раздела.

struct request_queue *queue;
Структура, используемая ядром для управления запросами ввода/вывода для этого устройства; мы изучим её в разделе "Обработка запроса" 455.

int flags;
(Редко используемый) набор флагов, описывающий состояние привода. Если ваше устройство имеет съёмный носитель, следует установить **GENHD_FL_REMOVABLE**. Приводы CD-ROM могут установить **GENHD_FL_CD**. Если по каким-то причинам вы не хотите, чтобы информация о разделах показывалась в ***/proc/partitions***, установите **GENHD_FL_SUPPRESS_PARTITION_INFO**.

sector_t capacity;
Ёмкость этого диска, в секторах по 512 байт. Тип **sector_t** может быть размером в 64 бит. Драйвер не должен устанавливать это поле напрямую; вместо этого число секторов передаётся в ***set_capacity***.

void *private_data;
Блочные драйверы могут использовать это поле для указания на свои собственные внутренние данные.

Ядро предоставляет небольшой набор функций для работы со структурами **gendisk**. Мы представить их здесь, а затем рассмотрим, как ***sbull*** использует их, чтобы сделать свои диски доступными системе.

struct gendisk представляет собой динамически создаваемую структуру, которая требует специальной манипуляции в ядре для инициализации; драйверы не могут создавать эту структуру сами. Вместо этого, вы должны вызвать:

```
struct gendisk *alloc_disk(int minors);
```

Аргумент **minors** должен быть числом младших номеров, используемых для этого диска; заметьте, что вы не сможете позже изменить поле minors и ожидать, что всё будет нормально работать.

Когда диск больше не нужен, он должен быть освобождён с помощью:

```
void del_gendisk(struct gendisk *gd);
```

gendisk является структурой со счётчиком ссылкой (она содержит **kobject**). Для манипулирования счётчиком ссылок доступны функции ***get_disk*** и ***put_disk***, но драйверы никогда не должны этого делать. Как правило, вызов ***del_gendisk*** удаляет последнюю ссылку на **gendisk**, но это не гарантируется. Таким образом, вполне возможно, что структура могла бы продолжать свое существование (и ваши методы могли бы быть вызванными) после вызова ***del_gendisk***. Однако, если вы удаляете структуру, когда нет пользователей (то есть после окончательного освобождения или в функции очистки вашего модуля), вы можете быть уверены, что вы не услышите о ней снова.

Выделенная структура **gendisk** не делает диск доступным системе. Чтобы это сделать, вы должны проинициализировать структуру и вызвать ***add_disk***:

```
void add_disk(struct gendisk *gd);
```

Имейте в виду одну важную вещь: как только вы вызываете ***add_disk***, диск становится "живым" и его методы могут быть вызваны в любое время. Фактически, первые такие вызовы произойдут, вероятно, ещё до того, как произойдёт возврат из ***add_disk***; ядро будет читать первые несколько блоков в попытке найти таблицу разделов. Таким образом, вы не должны вызывать ***add_disk***, пока ваш драйвер полностью не проинициализирован и не готов откликаться на запросы об этом диске.

Инициализация в sbull

Пора взяться за несколько примеров. Драйвер ***sbull*** (доступный на FTP сайте O'Reilly вместе с остальными примерами исходного кода) реализует расположенные в памяти виртуальные диски. Для каждого диска ***sbull*** выделяет (для простоты с помощью ***vmalloc***) массив памяти; затем он делает массив доступным через блочные операции. Драйвер ***sbull*** может быть протестирован путем разбиения виртуального устройства, построения на нём файловых систем и монтирования его в иерархию системы.

Как и наши другие драйверы из примеров, ***sbull*** позволяет, чтобы основной номер указывался во время компиляции или во время загрузки модуля. Если номер не указан, он выделяется динамически. Так как для динамического выделения необходим вызов ***register_blkdev***, ***sbull*** делает это:

```
sbull_major = register_blkdev(sbull_major, "sbull");
if (sbull_major <= 0) {
    printk(KERN_WARNING "sbull: unable to get major number\n");
```

```
    return -EBUSY;
}
```

Также как и другие виртуальные устройства, представленные в этой книге, устройство ***sbull*** описывается внутренней структурой:

```
struct sbull_dev {
    int size;                       /* Размер устройства в секторах */
    u8 *data;                       /* Массив данных */
    short users;                    /* Как много пользователей */
    short media_change;             /* Флаг был ли заменён носитель? */
    spinlock_t lock;                /* Для взаимного исключения */
    struct request_queue *queue;    /* Очередь запросов устройства */
    struct gendisk *gd;             /* Структура gendisk */
    struct timer_list timer;        /* Для имитации замены носителя */
};
```

Чтобы проинициализировать эту структуру и сделать связанные устройства доступными системе требуются несколько шагов. Начнём с основной инициализации и выделения основной памяти:

```
memset (dev, 0, sizeof (struct sbull_dev));
dev->size = nsectors*hardsect_size;
dev->data = vmalloc(dev->size);
if (dev->data == NULL) {
    printk (KERN_NOTICE "vmalloc failure.\n");
    return;
}
spin_lock_init(&dev->lock);
```

Сначала:
```
 struct sbull_dev {
    ...
    int size;     /* Размер устройства в секторах */
    ...
```

далее он инициализирован в байтах??!!
```
 dev->size = nsectors * hardsect_size;
```

что он означает на самом деле?

Ответ автора:

Да, комментарий ошибочен; где-то по ходу программы это поле стало размером в байтах. Так что среди прочего это означает, что тип 'int' может не быть лучшим выбором - хотя вряд ли он переполнится с размерами, обычно используемыми с простыми RAM-дисками.

Важно создать и проинициализировать спин-блокировку до следующего шага, который является созданием очереди запросов. Мы рассмотрим этот процесс более подробно, когда доберёмся до обработки запросов; на данный момент достаточно сказать, что необходимым вызовом является:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

Здесь ***sbull_request*** является нашей функцией ***request*** - функцией, которая фактически выполняет запросы блочного чтения и записи. Когда мы создаём очередь запросов, мы должны предусмотреть спин-блокировку, которая контролирует доступ к этой очереди. Блокировка осуществляется драйвером, а не общими частями ядра, потому что зачастую очереди запросов и другие структуры данных драйвера относятся к той же критической секции; как правило, они адресуются совместно. Как и любая функция, которая выделяет память, ***blk_init_queue*** может потерпеть неудачу, так что вы должны проверять возвращаемое значение, прежде чем продолжить.

После того как мы получили память для нашего устройства и очередь запросов, мы можем создать, проинициализировать и установить соответствующую структуру **gendisk**. Код, который выполняет эту работу:

```
dev->gd = alloc_disk(SBULL_MINORS);
if (! dev->gd) {
    printk (KERN_NOTICE "alloc_disk failure\n");
    goto out_vfree;
}
dev->gd->major = sbull_major;
dev->gd->first_minor = which*SBULL_MINORS;
dev->gd->fops = &sbull_ops;
dev->gd->queue = dev->queue;
dev->gd->private_data = dev;
snprintf (dev->gd->disk_name, 32, "sbull%c", which + 'a');
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
add_disk(dev->gd);
```

Здесь **SBULL_MINORS** - это число младших номеров, которые поддерживает каждое устройство ***sbull***. Когда мы устанавливаем первый младший номер для каждого устройства, мы должны принимать во внимание все числа, полученные предыдущими устройствами. Имена дисков, таким образом, устанавливаются так, что первый называется ***sbulla***, второй ***sbullb*** и так далее. Пользовательское пространство может затем добавить номера разделов, так что третий раздел на втором устройстве мог бы быть ***/dev/sbullb3***.

Когда всё будет создано, мы завершаем вызовом ***add_disk***. Весьма вероятно, что некоторые наши методы будут вызваны до возвращения из ***add_disk***, поэтому мы позаботились, чтобы этот вызов был самым последним шагом в инициализации нашего устройства.

Замечание о размерах секторов

Как мы уже отмечали ранее, ядро обрабатывает каждый диск, как линейный массив из секторов по 512 байт. Однако, не каждое оборудование использует такой размер сектора. Сделать так, чтобы устройство с другим размером сектора работало, не особенно трудно; это просто вопрос заботы о нескольких деталях. Устройство ***sbull*** экспортирует параметр **hardsect_size**, который может быть использован для изменения "аппаратного" размера сектора устройства; глядя на эту реализацию, вы сможете увидеть, как добавить такую поддержку в собственные драйверы.

Первая из этих деталей - сообщить ядру размер поддерживаемого вашим устройством сектора. Размер аппаратного сектора является одним из параметров в очереди запросов, а не

структуры **gendisk**. Этот размер устанавливается вызовом ***blk_queue_hardsect_size*** сразу после того, как создана очередь:

```
blk_queue_hardsect_size(dev->queue, hardsect_size);
```

Как только это будет сделано, ядро придерживается аппаратного размера сектора вашего устройства. Все запросы ввода/вывода выровнены должным образом начале аппаратного сектора и длина каждого запроса является целым числом секторов. Вы должны помнить, однако, что ядро всегда внутри себя работает с секторами по 512 байт; таким образом, соответственно необходимо перевести все номера секторов. Так, например, когда ***sbull*** устанавливает ёмкость устройства в своей структуре **gendisk**, вызов выглядит так:

```
set_capacity(dev->gd, nsectors*(hardsect_size/KERNEL_SECTOR_SIZE));
```

KERNEL_SECTOR_SIZE представляет собой локально-заданную константу, которую мы используем для масштабирования между секторами ядра по 512 байт и независимого размера, который мы выбрали для использования. Такой вид расчёта появляется часто, когда мы смотрим на логику обработки запроса в ***sbull***.

Операции блочного устройства

В предыдущем разделе мы кратко рассмотрели структуру ***block_device_operations***. Теперь нам потребуется некоторое время, чтобы взглянуть на эти операции немного более подробно, прежде чем переходить к обработке запроса. Теперь настало время отметить ещё одну особенностью драйвера ***sbull***: он претендует быть сменным устройством. Всякий раз, когда последний пользователь закрывает устройство, запускается 30-ти секундный таймер; если устройство не открыли в течение этого времени, содержимое устройства очищается и ядру будет сообщено, накопитель сменили. 30-ти секундная задержка даёт пользователю время, например, смонтировать устройство ***sbull*** после создания на нём файловой системы.

Методы open и release

Для реализации моделирования смены носителя информации, ***sbull*** должен знать, когда устройство закроет последний пользователь. Драйвером поддерживается учёт пользователей. Работа по сохранению текущего значения счётчика выполняется методами ***open*** и ***close***.

Метод ***open*** очень похож на свой эквивалент для символьного драйвера; он принимает в качестве аргументов соответствующие указатели на структуры **inode** и **file**. Когда **inode** ссылается на блочное устройство, поле **i_bdev->bd_disk** содержит указатель на соответствующую структуру **gendisk**; этот указатель может быть использован для получения внутренних структур данных драйвера устройства. Это, фактически, первое, что делает ***sbull*** в методе ***open***:

```
static int sbull_open(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    del_timer_sync(&dev->timer);
    filp->private_data = dev;
    spin_lock(&dev->lock);
    if (! dev->users)
        check_disk_change(inode->i_bdev);
    dev->users++;
```

```
    spin_unlock(&dev->lock);
    return 0;
}
```

После того, как ***sbull_open*** получила указатель на свою структуру устройства, она вызывает ***del_timer_sync*** для удаления таймера "носитель заменён", если таковой является активным. Обратите внимание, что мы не выполняем спин-блокировку устройства, пока таймер не был удалён; если поступить иначе, возможна взаимоблокировка, если таймерная функция запустится, прежде чем мы сможем её удалить. При заблокированном устройстве мы вызываем функцию ядра, названную ***check_disk_change***, чтобы проверить не произошла ли смена носителя. Можно возразить, что этот вызов должно сделать ядро, но стандартным шаблоном для драйвера является его обработка во время работы ***open***.

Последний шагом является увеличение счётчика пользователей и возвращение.

Задача метода ***release***, наоборот, уменьшить счётчик пользователей и, если указано, запустить таймер смены носителя:

```
static int sbull_release(struct inode *inode, struct file *filp)
{
    struct sbull_dev *dev = inode->i_bdev->bd_disk->private_data;

    spin_lock(&dev->lock);
    dev->users--;
    if (!dev->users) {
        dev->timer.expires = jiffies + INVALIDATE_DELAY;
        add_timer(&dev->timer);
    }
    spin_unlock(&dev->lock);

    return 0;
}
```

В драйвере, который обрабатывает реальные аппаратные устройства, методы ***open*** и ***release*** должны бы были установить состояние драйвера и оборудования соответственно. Эта работа может включать ускорение или замедление вращения диска, блокировку дверцы съёмного устройства, выделение буферов DMA и так далее.

Вы можете быть удивлены, кто фактически открывает блочное устройство. Есть несколько операций, которые являются причиной открытия блочного устройства непосредственно из пространства пользователя; к ним относятся разбиение диска, создание файловой системы на разделе или работа контроля файловой системы. Кроме того, блочный драйвер получает вызов ***open*** при монтировании раздела. В этом случае нет никакого процесса пространства пользователя, содержащего дескриптор открытого на устройстве файла; открытый файл, вместо того, удерживается самим ядром. Блочный драйвер не может определить разницу между операцией ***mount*** (которая открывает устройство из пространства ядра) и вызов из такой утилиты, как ***mkfs*** (которая открывает его из пространства пользователя).

Поддержка сменных носителей

Структура ***block_device_operations*** включает в себя два метода для поддержки съёмных носителей. Если вы пишете драйвер для неудаляемого устройства, вы можете безболезненно пропустить эти методы. Их реализация относительно проста.

Чтобы увидеть, был ли заменён носитель, вызывается метод ***media_changed*** (из ***check_disk_change***); если это произошло, он должен вернуть ненулевое значение. Реализация в ***sbull*** проста; она запрашивает флаг, который был установлен, если истёк таймер смены носителя:

```
int sbull_media_changed(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;

    return dev->media_change;
}
```

После замены носителя вызывается метод ***revalidate***; его работой является сделать всё, что требуется для подготовки драйвера для операций с новым носителем, если это необходимо. После вызова ***revalidate*** ядро пытается перечитать таблицу разделов и начать заново работу с устройством. Реализация в ***sbull*** просто сбрасывает флаг **media_change** и обнуляет память устройства, чтобы имитировать подключение пустого диска.

```
int sbull_revalidate(struct gendisk *gd)
{
    struct sbull_dev *dev = gd->private_data;

    if (dev->media_change) {
        dev->media_change = 0;
        memset (dev->data, 0, dev->size);
    }
    return 0;
}
```

Метод ioctl

Блочные устройства могут предоставить для выполнения функций управления устройством метод ***ioctl***. Высокоуровневый код блочной подсистемы перехватывает ряд команд ***ioctl*** всегда прежде чем драйвер их получит, однако (смотрите для полного набора ***drivers/block/ioctl.c*** в исходных кодах ядра). По сути, современный блочный драйвер может совсем не иметь реализации для очень многих команд ***ioctl***.

Метод ***ioctl*** в ***sbull*** обрабатывает только одну команду - запрос о конфигурации устройства:

```
int sbull_ioctl (struct inode *inode, struct file *filp,
                 unsigned int cmd, unsigned long arg)
{
    long size;
    struct hd_geometry geo;
    struct sbull_dev *dev = filp->private_data;

    switch(cmd) {
        case HDIO_GETGEO:
        /*
         * Получить конфигурацию: поскольку мы являемся виртуальным
устройством, мы должны сделать
         * что-то правдоподобное. Итак, мы заявляем о 16 секторах, четырёх
головках,
```

```
             * и рассчитываем соответствующее число цилиндров. Мы устанавливаем
             * начало данных на четвёртом секторе.
             */
            size = dev->size*(hardsect_size/KERNEL_SECTOR_SIZE);
            geo.cylinders = (size & ~0x3f) >> 6;
            geo.heads = 4;
            geo.sectors = 16;
            geo.start = 4;
            if (copy_to_user((void __user *) arg, &geo, sizeof(geo)))
                return -EFAULT;
            return 0;
        }

        return -ENOTTY; /* команда неизвестна */
    }
```

Предоставление информации о конфигурации может показаться любопытной задачей, так как наше устройство чисто виртуальное и не имеет ничего общего с дорожками и цилиндрами. Даже самое настоящее блочное оборудование за многие годы было снабжено гораздо более сложными структурами. Ядро не связано с конфигурацией блочного устройства; оно рассматривает его просто как линейный массив секторов. Однако, есть определённые утилиты пространства пользователя, которые всё ещё ожидают возможности для запроса конфигурации диска. В частности, утилита ***fdisk***, которая редактирует таблицы разделов, зависит от информации о цилиндрах и не функционирует должным образом, если такая информация недоступна.

Мы хотели бы, чтобы устройство ***sbull*** было допускающим разбиение даже устаревшими простодушными инструментами. Таким образом, мы предоставили метод ***ioctl***, который поставляется с правдоподобной выдуманной конфигурацией, которая могла бы соответствовать ёмкости нашего устройства. Большинство дисковый драйверов делают что-то подобное. Обратите внимание, что, как обычно, если необходимо, счётчик секторов преобразуется, чтобы соответствовать соглашению о 512 байтах, используемому в ядре.

Обработка запроса

Сердцем каждого блочного драйвера является его функция ***request***. Эта функция то, где выполняется настоящая работа, или, по крайней мере, начинается; всё остальное является накладными расходами. Следовательно, мы потратим изрядное количество времени на рассмотрение обработки запроса в блочных драйверах.

Производительность дисковых драйверов может быть важной частью производительности системы в целом. Таким образом, блочная подсистема ядра была написана очень много имея ввиду о производительности; она делает всё возможное, чтобы позволить вашему драйверу получить максимум от устройства, которым он управляет. Это хорошо, поскольку это позволяет молниеносно быстрый ввод/вывода. С другой стороны, блочная подсистема выставляет излишне много сложностей в драйверном API. Можно написать очень простую функцию ***request*** (мы увидеть её в ближайшее время), но если ваш драйвер должен общаться на высоком уровне со сложным оборудованием, это будет далеко не просто.

Введение в метод request

Метод ***request*** блочного драйвера имеет следующий прототип:

```
void request(request_queue_t *queue);
```

Эта функция вызывается всякий раз, когда ядро считает, что пришло время для вашего драйвера выполнить чтения, записи или другие операции с устройством. Функция ***request*** не требует полного завершения всех запросов из очередь до своего возвращения; в самом деле, в большинстве реальных устройств вероятно незавершение любого из них. Она должна, тем не менее, начать выполнение этих запросов и убедиться, что все они, в конечном счёте, драйвером обработаны.

Каждое устройство имеет очередь запросов. Так происходит потому, что фактические передачи на и с диска могут иметь место далеко от времени, когда ядро их запросило и потому, что ядру необходима гибкость для планирования каждой передаче в наиболее подходящий момент (например, группировка вместе запросов, которые воздействуют на секторы, расположенные близко друг к другу на диске). И функция ***request***, если вы помните, связана с очередью запросов при создании этой очереди. Давайте оглянемся обратно на то, как ***sbull*** создаёт свою очередь:

```
dev->queue = blk_init_queue(sbull_request, &dev->lock);
```

Таким образом, при создании очереди функция ***request*** связывается с ней. Мы также обеспечили спин-блокировку, как часть процесса создания очереди. Всякий раз, когда вызывается наша функция ***request***, эта блокировка удерживается ядром. В результате функция ***request*** является работающей в атомарном контексте; она должна следовать всем обычным правилам для атомарного кода, обсуждённых в Главе 5 101.

Блокировка для очереди также предохраняет ядро от очередей любых других запросов для вашего устройства, пока ваша функция ***request*** удерживает блокировку. При некоторых условиях вы можете захотеть рассмотреть отказ от этой блокировки во время выполнения функции ***request***. Однако, если вы это сделаете, вы должны быть уверены, что не обращаетесь к очереди запросов, или что любая другая структура данных защищена блокировкой, если блокировка не удерживается. Вы также должны повторно запросить блокировку до возвращения функции ***request***.

И наконец, вызов функции ***request*** является (как правило) полностью асинхронным по отношению к действиям любого процесса пользовательского пространства. Вы не можете предполагать, что ядро работает в контексте процесса, который инициировал текущий запрос. Вам неизвестно, находится ли буфер ввода/вывода, предоставленный запросом, в ядре или в пространстве пользователя. Таким образом, любой вид операции, которая явно обращается в пользовательское пространство, является ошибкой, и, безусловно, приведёт к проблеме. Как вы увидите, всё, что вашему драйверу необходимо знать о запросе, содержится в структурах, переданных через очередь запросов.

Простой метод request

Драйвер примера ***sbull*** предоставляет несколько разных методов для обработки запроса. По умолчанию ***sbull*** использует метод, названный ***sbull_request***, который призван быть примером простейшего из возможных методов запроса. Вот он без дальнейших церемоний:

```
static void sbull_request(request_queue_t *q)
{
    struct request *req;

```

```
    while ((req = elv_next_request(q)) != NULL) {
        struct sbull_dev *dev = req->rq_disk->private_data;
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
        }
        sbull_transfer(dev, req->sector, req->nr_sectors,
                       req->buffer, rq_data_dir(req));
        end_request(req, 1);
    }
}
```

Эта функция представляет структуру **struct request**. Мы будем детально рассматривать **struct request** позже; сейчас же достаточно сказать, что это она представляет для выполнения нами блочный запрос ввода/вывода.

Для получения первого незавершённого запроса в очереди ядро предоставляет функцию ***elv_next_request***; эта функция возвращает NULL, если запросов для обработки нет. Обратите внимание, что ***elv_next_request*** не удаляет запрос из очереди. Если вы вызываете её дважды без каких-либо промежуточных операций, оба раза она возвращает ту же структуру **request**. В этой простой режим работы, запросы снимаются с очереди только тогда, когда они завершены.

Блочная очередь запросов может содержать запросы, которые на самом деле не перемещают блоки на или с диска. Такие запросы могут включать зависящие от производителя низкоуровневые диагностические операции или инструкции, касающиеся режимов специализированных устройств, таких как режим пакетной записи для записываемого носителя. Большинство блочных драйверов не знают, как обрабатывать такие запросы и просто их не выполняют; ***sbull*** работает таким же образом. Вызов ***blk_fs_request*** сообщает нам, видим ли мы запрос файловой системы, который перемещает блоки данных. Если запрос не является запросом файловой системы, мы передаём его в ***end_request***:

```
void end_request(struct request *req, int succeeded);
```

Когда мы определяем, что запросы не файловой системы, мы передаём **succeeded** как 0, чтобы указать, что мы завершили запрос не успешно. В противном случае, для фактического перемещения данных мы вызываем ***sbull_transfer***, используя набор полей, предусмотренных в структуре **request**:

sector_t sector;
: Указатель начального сектора на нашем устройстве. Запомните, что этот номер сектора, как и все такие номера, передаваемые между ядром и драйвером, выражается в секторах по 512 байт. Если ваше оборудование использует другой размер сектора, необходимо **sector** соответственно отмасштабировать. Например, если оборудование использует секторы по 2048 байт, необходимо разделить номер начального сектора на четыре перед помещением его в запрос для оборудования.

unsigned long nr_sectors;
: Количество (512 байтных) секторов, которые будут переданы.

char *buffer;
: Указатель на буфер в или из которого эти данные должны быть переданы. Этот

указатель является виртуальным адресом ядра и может быть в случае необходимости непосредственно разыменован драйвером.

rq_data_dir(struct request *req);
Этот макрос извлекает из запроса направление передачи; нулевое возвращаемое значение означает чтение из устройства и ненулевое возвращаемое значение означает запись в устройство.

Используя эту информацию, драйвер ***sbull*** может осуществлять фактическую передачу данных простым вызовом ***memcpy*** - в конце концов, наши данные уже находятся в памяти. Функция, которая выполняет эту операцию копирования (***sbull_transfer***), также выполняет масштабирование размеров секторов и гарантирует, что мы не пытаемся копировать за предел нашего виртуального устройства:

```
static void sbull_transfer(struct sbull_dev *dev, unsigned long sector,
        unsigned long nsect, char *buffer, int write)
{
    unsigned long offset = sector*KERNEL_SECTOR_SIZE;
    unsigned long nbytes = nsect*KERNEL_SECTOR_SIZE;

    if ((offset + nbytes) > dev->size) {
        printk (KERN_NOTICE "Beyond-end write (%ld %ld)\n", offset, nbytes);
        return;
    }
    if (write)
        memcpy(dev->data + offset, buffer, nbytes);
    else
        memcpy(buffer, dev->data + offset, nbytes);
}
```

С помощью этого кода ***sbull*** реализует завершённое, простое, находящееся в ОЗУ дисковое устройство. Это, однако, по нескольким причинам не реалистичный драйвер для многих типов устройств.

Первой из этих причин является то, что ***sbull*** выполняет запросы синхронно, по одному за раз. Высокопроизводительные дисковые устройства способны в один момент времени иметь множество невыполненных запросов; встроенный в плату дисковый контроллер может выбирать их для выполнения в оптимальном порядке (надеемся). Пока мы обрабатываем только первый запрос в очередь, мы никогда не сможем иметь множество запросов, выполняемых в данный момент времени. Способность работать с более чем одним запросом требует более глубокого понимания очередей запросов и структуры **request**; следующие несколько разделов помогут выстроить такое понимание.

Однако, существует для рассмотрения и другой вопрос. При работе с дисковыми устройствами лучшая производительность достигается тогда, когда система выполняет большие передачи с участием нескольких секторов, которые находятся на диске вместе. Самыми высокими накладными расходами на дисковые операции всегда является позиционирование читающих и пишущих головок; после того, как это выполнено, время, затрачиваемое на фактическое чтение или запись данных, почти незначительно. Разработчики, которые создают и реализуют файловые системы и подсистемы виртуальной памяти, понимают это, поэтому они делают всё возможное, чтобы расположить соответствующие данные на диске смежно и передать в одном запросе так много секторов, как возможно. Блочная подсистема также помогает в этом отношении; очереди запросов содержат

много логики, направленной на обнаружение смежных запросов и их объединение в более крупные операции.

Однако, драйвер ***sbull*** получает всю эту работу и просто её игнорирует. За раз передаётся только один буфер, это означает, что наибольшая одиночная передача почти никогда не превышает размера одной страницы. Блочный драйвер может работать гораздо лучше, по сравнению с этим, но это требует более глубокого понимания структур **request** и структур **bio**, на которых построены запросы.

Следующие несколько разделов копаются немного поглубже в том, как блочный уровень делает свою работу и в структурах данных, которые являются результатом этой работы.

Очереди запросов

В самом простом смысле, блочная очередь запросов это именно это: очередь блочных запросов ввода/вывода. Если посмотреть под крышку, очередь запросов оказывается удивительно сложной структурой данных. К счастью, драйверам не требуется беспокоиться о большинстве из этих сложностей.

Очереди запросов следят за невыполненными запросами блочного ввода/ вывода. Но они также играют решающую роль в создании этих запросов. Очередь запросов хранит параметры, которые описывают, какие виды запросов устройство способно обслужить: их максимальный размер, сколько отдельных сегментов может быть в запросе, размер аппаратного сектора, требования выравнивания и так далее. Если ваша очередь запросов настроена правильно, она никогда не должна подарить вам запрос, который ваше устройство не сможет обработать.

Очереди запросов также реализуют интерфейс подключения модулей, который позволяет использовать несколько ***планировщиков*** (scheduler) ввода/вывода (или ***транспортёров*** (elevator)). Работой планировщика ввода/вывода является предоставление запросов ввода/ вывода вашему драйверу таким образом, чтобы максимизировать производительность. С этой целью большинство планировщиков ввода/вывода накапливают пакеты запросов, сортируют их в порядке увеличения (или уменьшения) значения блочного указателя, и предоставляют драйверу запросы в таком порядке. Головка диска, когда предоставляется упорядоченный список запросов, проходит свой путь от одного конца диска к другому, подобно полному транспортёру, движущемуся в одном направлении, пока все его "запросы" (люди, ожидающие, чтобы выйти) не выполнены. Ядро версии 2.6 имеет "планировщик со сроком завершения", который прикладывает усилия к тому, чтобы каждый запрос выполнялся в течение заданного максимального времени, и "упреждающий планировщик", который фактически ненадолго задерживает устройство после запроса на чтение в ожидании того, что будет получено почти сразу другое смежное чтение. На момент написания, планировщиком по умолчанию является упреждающей планировщик, который, кажется, даёт лучшую производительность интерактивной системе.

Планировщик ввода/вывода также отвечает за объединение прилегающих запросов. Когда в планировщик ввода/вывода передан новый запрос, он ищет в очереди запросы, относящиеся к прилегающим секторам; если нашёл и если запрос в результате не будет слишком большим, такие два запроса объединяются.

Очереди запросов имеют тип **struct request_queue** или **request_queue_t**. Этот тип и многие функции, которые с ним работают, определены в ***<linux/blkdev.h>***. Если вы заинтересованы в реализации очередей запросов, вы можете найти большую часть кода в ***drivers/block/ll_rw_block.c*** и ***elevator.c***.

Создание и удаление очереди

Как мы видели в нашем коде примера, очередь запросов является динамической структурой данных, которая должна быть создан подсистемой блочного ввода/вывода. Функцией для создания и инициализации очереди запросов является:

```
request_queue_t *blk_init_queue(request_fn_proc *request, spinlock_t *lock);
```

Аргументы, конечно, функция ***request*** для этой очереди и спин-блокировка, которая управляет доступом к этой очереди. Эта функция выделяет память (совсем немного памяти, на самом деле) и может из-за этого закончится неудачно; вам следует всегда проверять возвращаемое значение, прежде чем пытаться использовать эту очередь.

В рамках инициализации очереди запросов вы можете установить поле **queuedata** (которое является указателем **void ***) в любое значение, которое вам нравится. Это поле эквивалентно в очереди запросов полю **private_data**, которое мы уже видели в других структурах.

Чтобы вернуть очередь запросов системе (как правило, при выгрузке модуля), вызовите ***blk_cleanup_queue***:

```
void blk_cleanup_queue(request_queue_t *);
```

После этого вызова ваш драйвер больше не увидит запросов от данной очереди и не должен не ссылаться на неё снова.

Функции для очереди

Для манипулирования запросами в очереди существует очень небольшой набор функций - по крайней мере в том, что касается драйверов. Вы должны удерживать блокировку очереди, прежде чем вызывать эти функции.

Функцией, которая возвращает процессу следующий запрос, является ***elv_next_request***:

```
struct request *elv_next_request(request_queue_t *queue);
```

Мы уже видели эту функцию в простом примере ***sbull***. Она возвращает процессу указатель на следующий запрос (определённый планировщиком ввода/вывода) или **NULL**, если больше не осталось запросов для обработки. ***elv_next_request*** оставляет запрос в очереди, но помечает его как активный; эта метка предотвращает попытку планировщика ввода/вывода объединить с ним другие запросы, когда вы начинаете его выполнять.

Чтобы действительно удалить запрос из очереди, используйте ***blkdev_dequeue_request***:

```
void blkdev_dequeue_request(struct request *req);
```

Если ваш драйвер одновременно обрабатывает множество запросов из одной и той же очереди, он должен убрать их из очереди таким же образом.

Если вам необходимо по какой-то причине поместить убранный из очереди запрос обратно в очереди вы можете вызвать:

```
void elv_requeue_request(request_queue_t *queue, struct request *req);
```

Функции управления очередью

Блочный уровень экспортирует набор функций, которые могут быть использованы драйвером для управления того, как работает очередь запросов. Эти функции включают в себя:

void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
Если ваше устройство достигло состояния, в котором он не может больше обрабатывать невыполненные команды, вы можете вызвать ***blk_stop_queue***, чтобы сообщить об этом блочному уровню. После этого вызова ваша функция ***request*** не будет вызываться, пока вы не вызовите ***blk_start_queue***. Разумеется, вы не должны забыть перезапустить очередь, когда ваше устройство сможет обрабатывать больше запросов. Блокировка очереди должна удерживаться при вызове любой из этих функций.

void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
Функция, которая сообщает ядру наивысший физический адрес, по которому устройство может выполнять DMA. Если приходит запрос, содержащий ссылку на памяти выше этого предела, для этой операции будет использоваться возвратный буфер; это, конечно, дорогой способ выполнения блочного ввода/вывода и его следует по возможности избегать. Вы можете предоставить в этом аргументе любые разумные физические адреса или использовать заранее определённые символы **BLK_BOUNCE_HIGH** (использовать возвратные буферы для страниц верхней памяти), **BLK_BOUNCE_ISA** (драйвер может выполнять DMA только в 16 Мб зоне ISA), или **BLK_BOUNCE_ANY** (драйвер может выполнять DMA по любому адресу). Значением по умолчанию является **BLK_BOUNCE_HIGH**.

void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
Функции, устанавливающие параметры, описывающие запросы, которые могут быть выполнены этим устройством. ***blk_queue_max_sectors*** может быть использована для установки максимального размера любого запроса в секторах (по 512 байт); по умолчанию 255. ***blk_queue_max_phys_segments*** и ***blk_queue_max_hw_segments*** управляют тем, как много физических сегментов (не смежных областей в системной памяти), могут содержаться в одном запросе. Используйте ***blk_queue_max_phys_segments***, чтобы сообщить, с каким количеством сегментов ваш драйвер подготовлен справиться; это может быть, например, размер статически созданного списка разборки. ***blk_queue_max_hw_segments***, напротив, является максимальным количеством сегментов, которые может обработать само устройство. Оба этих параметров по умолчанию равны 128. Наконец, ***blk_queue_max_segment_size*** сообщает ядру, насколько большим в байтах может быть любой отдельный сегмент запроса; по умолчанию 65.536 байт.

blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
Некоторые устройства не могут обрабатывать запросы, которые пересекают границы определённого размера памяти; если ваше устройство является одним из таких, используйте эту функцию, чтобы сообщить о такой границе ядру. Например, если ваше

устройство имеет проблемы с запросами, которые пересекают границу 4 Мб, передайте в маске 0x3fffff. Маска по умолчанию 0xffffffff.

void blk_queue_dma_alignment(request_queue_t *queue, int mask);
Функция, которая сообщает ядру об ограничении на выравнивание памяти, накладываемом устройством на передачи DMA. Все запросы создаются с заданным выравниванием и размер запроса также совпадает с выравниванием. Маска по умолчанию 0x1ff, которая приводит к тому, что все запросы выровнены по границам в 512 байт.

void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
Сообщает ядру о размере аппаратного сектора вашего устройства. Все запросы, генерируемые ядром являются кратными этому размеру и выровнены должным образом. Однако, всё взаимодействие между блочным уровнем и драйвером продолжает быть выраженным в секторах по 512 байт.

Анатомия запроса

В нашем простом примере мы столкнулись со структурой **request**. Однако, мы едва поцарапали поверхность этой сложной структуры данных. В этом разделе мы посмотрим некоторые детали того, как блочные запросы ввода/вывода представлены в ядре Linux.

Каждая структура **request** представляет собой один блочный запрос ввода/вывода, хотя она могла быть образована в результате слияния на более высоком уровне нескольких самостоятельных запросов. Секторы, передаваемые на любой конкретный запрос, могут быть распределены по всей основной памяти, хотя они всегда соответствуют на блочном устройстве набору последовательных секторов. Запрос представлен в виде набора сегментов, каждый из которых соответствует одному буферу в памяти. Ядра может объединить несколько запросов, связанных со смежными секторами на диске, но оно никогда не объединяет внутри одной структуры **request** операции чтения и записи. Ядро также удостоверяется, что запросы не объединены, если результат будет нарушать любое из ограничений очереди запросов, описанных в предыдущем разделе.

По сути, реализация структуры **request** представляет собой связный список структур **bio**, скомбинированных с некоторой служебной информацией, позволяющей драйверу следить за своей позицией, поскольку это работает через запрос. Структура **bio** является низкоуровневым описанием части запроса блочного ввода/вывода, сейчас мы её рассмотрим.

Структура bio

Когда ядро, в виде файловой системы, подсистемы виртуальной памяти, или системного вызова решает, что набор блоков должен быть передан в или от устройства блочного ввода/вывода, оно для описания этой операции помещает их вместе в структуру bio. Затем эта структура передаётся коду блочного ввода/вывода, который объединяет её с существующей структурой **request** или, в случае необходимости, создаёт новую. Структура **bio** содержит в себе всё, что необходимо блочному драйверу для выполнения запроса без ссылки на процесс пользовательского пространства, который был причиной начала этого запроса.

Структура **bio**, которая определена в ***<linux/bio.h>***, содержит ряд полей, которые могут быть использованы авторами драйверов:

sector_t bi_sector;
Первый (512 байтный) сектор, передаваемый для этой **bio**.

unsigned int bi_size;
Размер данных, подлежащих передаче, в байтах. Вместо этого, часто проще использовать ***bio_sectors(bio)***, макрос, предоставляющий размер в секторах.

unsigned long bi_flags;
Набор флагов, описывающих эту **bio**; наименьший значащий бит установлен, если это запрос на запись (хотя вместо просмотра этих флагов напрямую должен быть использован макрос ***bio_data_dir(bio)***).

unsigned short bio_phys_segments;
unsigned short bio_hw_segments;
Число физических сегментов, содержащихся в этой **bio** и количество сегментов, видимых оборудованием после выполнения DMA отображения, соответственно.

Ядро **bio**, однако, представляет собой массив, названный **bi_io_vec**, который состоит из следующей структуры:

```
struct bio_vec {
    struct page *bv_page;
    unsigned int bv_len;
    unsigned int bv_offset;
};
```

Рисунок 16-1 показывает, как все эти структуры связать вместе. Как вы можете видеть, к тому времени, как блок запроса ввода/вывода превращается в структуру **bio**, он разбит на отдельные страницы физической памяти. Всё, что необходимо сделать драйверу, это достигнуть данных через этот массив структур (в них есть **bi_vcnt**) и передать данные внутри каждой страницы (но только **len** байт, начиная с **offset**).

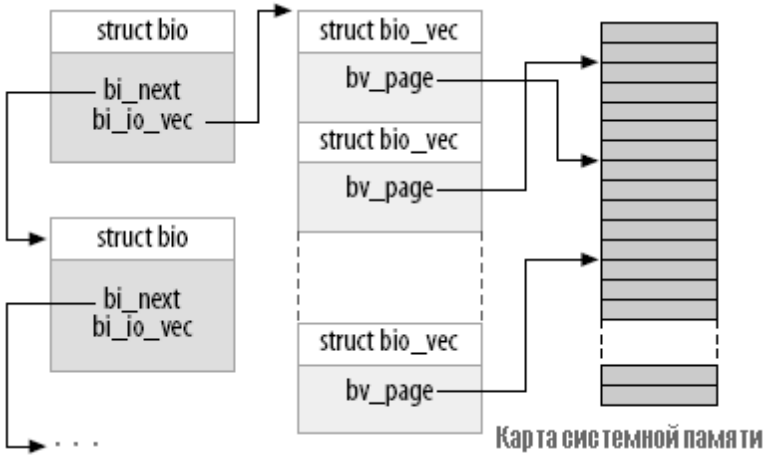


Рисунок 16-1. Структура bio

Работа непосредственно с массивом **bi_io_vec** не приветствуется в интересах того, чтобы разработчики ядра могли в будущем изменить структуру **bio**, не нарушая чего-либо. Чтобы выполнить это, для облегчения процесса работы со структурой **bio** предоставлен набор

макросов. Всё начинается с ***bio_for_each_segment***, который просто просматривает каждую необработанную запись в массиве **bi_io_vec**. Этот макрос следует использовать следующим образом:

```
int segno;
struct bio_vec *bvec;

bio_for_each_segment(bvec, bio, segno) {
    /* Делаем что-нибудь с этим сегментом */
}
```

Внутри этого цикла **bvec** указывает на текущую запись **bio_vec** и **segno** является текущим номером сегмента. Эти значения могут быть использованы для создания передач DMA (альтернативный способ с использованием ***blk_rq_map_sg*** описан в разделе "Блочные запросы и DMA" 470). Если вам необходимо получить доступ к страницам напрямую, вы должны сначала убедиться, что надлежащий виртуальный адрес ядра существует; для этого вы можете использовать:

```
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
```

Эта низкоуровневая функция позволяет вам напрямую отобразить буфер, находящийся в данной **bio_vec**, как указано индексом **i**. Создаётся атомарная ***kmap***; вызывающий должен предоставить для использования соответствующий слот (как описано в разделе "Карта памяти и структура page" 399 в Главе 15 395).

Для отслеживания текущего состояния обработки запроса блочный уровень также поддерживает в структуре **bio** набор указателей. Чтобы обеспечить доступ к этому состоянию, существуют несколько макросов:

struct page *bio_page(struct bio *bio);
Возвращает указатель на структуру **page**, представляющую страницу, которая будет передана следующей.

int bio_offset(struct bio *bio);
Возвращает смещение внутри страницы для передаваемых данных.

int bio_cur_sectors(struct bio *bio);
Возвращает число секторов, которые будут переданы из текущей страницы.

char *bio_data(struct bio *bio);
Возвращает логический адрес ядра, указывающий на данные для передачи. Обратите внимание, что этот адрес доступен только, если страница в запросе не находится в верхней памяти; вызов в других ситуациях является ошибкой. По умолчанию, блочная подсистема не передаёт в ваш драйвер буферы верхней памяти, но если вы изменили этот параметр с помощью ***blk_queue_bounce_limit***, вам, вероятно, не следует использовать **bio_data**.

char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
bio_kmap_irq возвращает виртуальный адрес ядра для любого буфера, независимо от того, находится ли он в верхней или нижней памяти. Используется атомарная ***kmap***,

поэтому драйвер не может спать, пока это отображение активно. Для отключения буфера используйте ***bio_kunmap_irq***. Обратите внимание, что аргумент **flags** передаётся здесь через указатель. Отметим также, что поскольку используется атомарная ***kmap***, вы не можете отобразить за раз более одного сегмента.

Все эти функции описаны только для доступа в "текущий" буфер - первый буфер, который, как ядро знает, не был передан. Драйверы часто хотят работать через несколько буферов в **bio** до завершения сигнализации о завершении для любого из них (с помощью ***end_that_request_first***, которая будет описана в ближайшее время), так что часто эти функции бесполезны. Для работы с внутренностями структуры **bio** существуют несколько других макросов (смотрите для подробностей ***<linux/bio.h>***).

Поля структуры запроса

Теперь, когда мы имеем представление о том, как работает структура **bio**, мы можем углубиться в **struct request** и посмотреть, как работает обработка запроса. Поля этой структуры включают в себя:

sector_t hard_sector;
unsigned long hard_nr_sectors;
unsigned int hard_cur_sectors;
Поля, которые отслеживают секторы, которые драйвер до сих пор не завершил. Первый сектор, который не был передан, хранится в **hard_sector**, общим числом секторов для передачи является **hard_nr_sectors**, а числом секторов, оставшихся в текущей **bio**, является **hard_cur_sectors**. Эти поля предназначены для использования только внутри блочной подсистемы; драйверам не следует их использовать.

struct bio *bio;
bio является связным списком структур **bio** для этого запроса. Вы не должны обращаться напрямую к этому полю; вместо того используйте ***rq_for_each_bio*** (описанную ниже).

char *buffer;
Простой пример драйвера ранее в этой главе использовал это поле, чтобы найти буфер для передачи. Благодаря нашему более глубокому пониманию, теперь мы можем видеть, что это поле есть просто результат вызова ***bio_data*** для текущей **bio**.

unsigned short nr_phys_segments;
Число отдельных сегментов, занятых этим запросом в физической памяти после того, как были объединены соседние страницы.

struct list_head queuelist;
Структура связного списка (описанного в разделе "Связные списки" 282 в Главе 11 275), которая связывает запрос с очередью запросов. Если (и только если) вы удаляете запрос из очереди с помощью ***blkdev_dequeue_request***, вы можете использовать голову этого списка для отслеживания запроса в внутреннем списке, поддерживаемом вашим драйвером.

Рисунок 16-2 показывает, как собраны вместе структура запроса и его компоненты структур **bio**. На рисунке, этот запрос был частично выполнен; поля **cbio** и **buffer** указывают на первую **bio**, которая ещё не была передана.

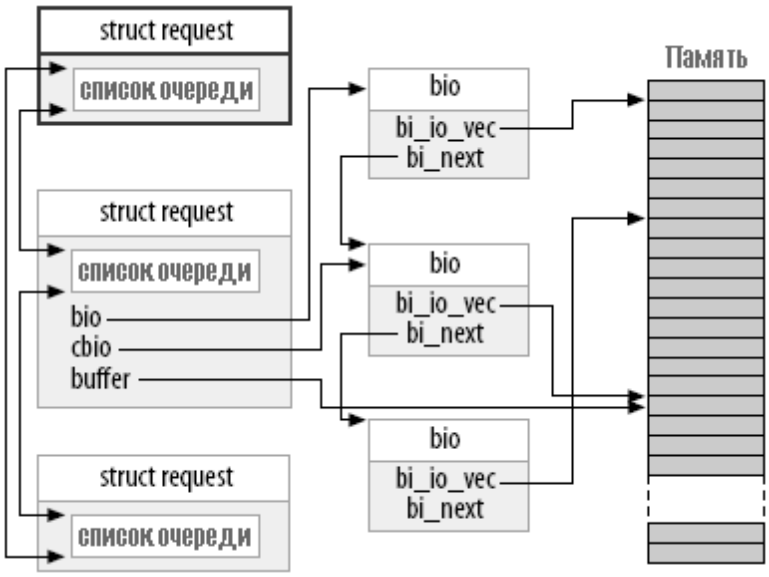


Рисунок 16-2. Очередь запросов с частично выполненным запросом

Внутри структуры **request** есть много других полей, но для большинства авторов драйверов списка в этом разделе должно быть достаточно.

Барьерные запросы

Чтобы улучшить производительность ввода/вывода, блочный уровень переупорядочивает запросы перед тем, как ваш драйвер их увидит. Ваш драйвер тоже может изменить порядок запросов, если для этого есть основания. Часто это переупорядочивание происходит путём передачи нескольких запросов к диску, позволяя оборудованию выяснить оптимальный порядок. Однако, существует проблема с неограниченным переупорядочиванием запросов: некоторые приложения требуют гарантий того, что определённые операции будут завершены до того, как начались другие. Менеджер реляционной база данных, например, должен быть абсолютно уверен, что их информация журналирования была скинута на диск до выполнения транзакции с содержимым базы данных. Журналирующие файловые системы, которые сейчас используются в большинстве систем Linux, имеют очень похожие ограничения на порядок выполнения. Если некорректные операции переупорядочены, результатом может быть серьёзное необнаруживаемое повреждение данных.

Блочный уровень версии 2.6 решает эту проблему с помощью концепции ***барьерного запроса***. Если запрос отмечен флагом **REQ_HARDBARRER**, он должен быть записан на диск прежде, чем инициируется любой следующий запрос. Под "записаны на диск" мы понимаем, что данные должны перемещены фактически и находится на физическом носителе. Многие накопители выполняют кэширование запросов на запись; такое кеширование увеличивает производительность, но это может противоречить целям барьерных запросов. Если случится сбой питания, пока критические данные всё ещё находятся в кэше накопителя, данные всё же потеряются, даже если накопитель сообщил о выполнении. Таким образом, драйвер, который реализует барьерные запросы должен предпринять шаги, чтобы заставить накопитель на самом деле записать данные на носитель информации.

Если ваш драйвер выполняет барьерные запросы, первый шаг заключается в

информировании блочного слоя об этом факт. Обработка барьера является ещё одной очередью запросов; она инициализируется с помощью:

```
void blk_queue_ordered(request_queue_t *queue, int flag);
```

Чтобы указать, что ваш драйвер реализует барьерные запросы, установите параметр **flag** в ненулевое значение.

Фактическая реализация барьерных запросов - это просто вопрос проверки на соответствующий флаг в структуре **request**. Для выполнения такой проверки была предоставлен макрос:

```
int blk_barrier_rq(struct request *req);
```

Если этот макрос возвращает ненулевое значение, запрос является барьерным запросом. В зависимости от того, как работает оборудование, возможно, придётся прекратить приём запросов из очереди, пока барьерный запрос не будет завершён. Другие накопители могут сами понимать барьерный запрос; в этом случае, всё, что должен сделать ваш драйвер, это сформировать для таких накопителей соответствующие операции.

Неповторяемые запросы

Блочные драйверы часто пытаются повторить запросы, которые в первый раз окончились неудачно. Такое поведение может привести к более надежной системе и помочь избежать потери данных. Ядро, однако, иногда помечает запросы как не требующие повтора. Такие запросы должны просто закончится неудачно так быстро, как это возможно, если они не могут быть выполнены с первого раза.

Если ваш драйвер принимает во внимание повтор неудавшегося запроса, сначала он должен сделать вызов:

```
int blk_noretry_request(struct request *req);
```

Если этот макрос возвращает ненулевое значение, ваш драйвер должен просто прервать запрос с кодом ошибки, вместо его повтора.

Функции завершения запроса

Есть, как мы увидим, несколько разных способов работы с помощью структуры **request**. Все они, однако, используют пару общих функций, которые выполняют обработку завершения запроса ввода/вывода или частей запроса. Обе эти функции атомарные и могут быть безопасно вызваны из атомарного контекста.

Когда ваше устройство завершило передачу некоторых или всех секторов в запросе ввода/вывода, оно должно проинформировать блочную подсистему с помощью:

```
int end_that_request_first(struct request *req, int success, int count);
```

Данная функция сообщает блочному коду, что ваш драйвер закончил передачу **count** секторов, начиная с того, где вы последний раз были прерваны. Если этот ввод/вывод был успешным, **success** передаётся как 1, иначе передаётся 0. Заметьте, что вы должны просигнализировать о завершении в порядке от первого секторе до последнего; если ваш

драйвер и устройство и как-то скрытно выполняют запросы не по порядку, вам придётся хранить нестандартное состояние выполнения до завершения передачи промежуточных секторов.

Возвращаемое значение ***end_that_request_first*** показывает, были ли переданы все секторы в этом запросе или нет. Возвращаемое значение 0 означает, что были переданы все секторы и что запрос выполнен. В этот момент вы должны удалить запрос из очереди с помощью ***blkdev_dequeue_request*** (если вы ещё этого не сделали) и передать его в:

```
void end_that_request_last(struct request *req);
```

end_that_request_last информирует ожидающих запрос, что он завершён и удаляет структуру **request**; она должна быть вызвана при удержании блокировки очереди.

В нашем простом примере ***sbull***, мы не использовали какую-либо из перечисленных выше функций. Этот пример, вместо этого, вызывает ***end_request***. Чтобы показать последствия этого вызова, вот вся функция ***end_request***, как она выглядит в ядре версии 2.6.10:

```
void end_request(struct request *req, int uptodate)
{
    if (!end_that_request_first(req, uptodate, req->hard_cur_sectors)) {
        add_disk_randomness(req->rq_disk);
        blkdev_dequeue_request(req);
        end_that_request_last(req);
    }
}
```

Функция ***add_disk_randomness*** использует время блочных запросов ввода/вывода, чтобы способствовать энтропии системного пула случайных чисел; она должна вызываться только если время обращения к диску действительно случайное. Это верно для большинства механических устройств, но это не относится к виртуальным устройствам, находящимся в памяти, таким как ***sbull***. По этой причине, более сложная версия ***sbull***, показанная в следующем разделе, не вызывает ***add_disk_randomness***.

Работа с bios

Теперь вы знаете достаточно, чтобы написать блочный драйвер, который работает непосредственно со структурами **bio**, которые составляют запрос. Однако, пример может помочь. Если драйвер ***sbull*** загружен с параметром **request_mode**, установленным в 1, он регистрирует функцию **request**, осведомлённую о **bio**, вместо простой функции, которую мы видели выше. Эта функция выглядит следующим образом:

```
static void sbull_full_request(request_queue_t *q)
{
    struct request *req;
    int sectors_xferred;
    struct sbull_dev *dev = q->queuedata;

    while ((req = elv_next_request(q)) != NULL) {
        if (! blk_fs_request(req)) {
            printk (KERN_NOTICE "Skip non-fs request\n");
            end_request(req, 0);
            continue;
```

```
        }
        sectors_xferred = sbull_xfer_request(dev, req);
        if (! end_that_request_first(req, 1, sectors_xferred)) {
            blkdev_dequeue_request(req);
            end_that_request_last(req);
        }
    }
}
```

Эта функция просто принимает каждый запрос, передаёт его в ***sbull_xfer_request***, затем завершает его с помощью ***end_that_request_first*** и, если необходимо, ***end_that_request_last***. Таким образом, эта функция обрабатывает высокоуровневую очередь и проблему части управления запроса. Однако, на самом деле работа по выполнению запроса приходится на ***sbull_xfer_request***:

```
static int sbull_xfer_request(struct sbull_dev *dev, struct request *req)
{
    struct bio *bio;
    int nsect = 0;

    rq_for_each_bio(bio, req) {
        sbull_xfer_bio(dev, bio);
        nsect += bio->bi_size/KERNEL_SECTOR_SIZE;
    }
    return nsect;
}
```

Здесь мы вводим ещё один макрос: ***rq_for_each_bio***. Как и следовало ожидать, этот макрос просто проходит через каждую структуру **bio** в запросе, давая нам указатель, который мы можем передать в ***sbull_xfer_bio*** для передачи. Эта функция выглядит следующим образом:

```
static int sbull_xfer_bio(struct sbull_dev *dev, struct bio *bio)
{
    int i;
    struct bio_vec *bvec;
    sector_t sector = bio->bi_sector;

    /* Работаем с каждым сегментом независимо. */
    bio_for_each_segment(bvec, bio, i) {
        char *buffer = __bio_kmap_atomic(bio, i, KM_USER0);
        sbull_transfer(dev, sector, bio_cur_sectors(bio),
                    buffer, bio_data_dir(bio) == WRITE);
        sector += bio_cur_sectors(bio);
        __bio_kunmap_atomic(bio, KM_USER0);
    }
    return 0; /* Всегда "успешно" */
}
```

Эта функция просто проходит через каждый сегмент в структуре **bio**, получает виртуальный адрес ядра для доступа в буфер, затем вызывает ту же функцию ***sbull_transfer***, которую мы видел раньше, для копирования данных.

Каждое устройство имеет свои собственные потребности, но, как правило, только что показанный код должен служить образцом для многих ситуаций, где есть необходимость

копаться в структурах **bio**.

Блочные запросы и DMA

Если вы работаете над высокопроизводительным блочным драйвером, скорее всего вы будете использовать для фактической передачи данных DMA. Блочный драйвер может, безусловно, пройти через структуры **bio**, как описано выше, создать отображения DMA для каждого из них, и передать результат в устройство. Существует, однако, более простой способ, если ваше устройство может выполнять ввод/вывод с разборкой/сборкой. Функция:

```
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct
scatterlist *list);
```

заполняет данный **list** полным набором сегментов данного запроса. Сегменты, которые являются соседними в памяти, объединяются до включения в лист разборки, так что вам не требуется пытаться находить их самостоятельно. Возвращаемое значение является числом записей в списке. Эта функция также передает обратно, в своём третьем аргументе, список разборки, подходящий для ***dma_map_sg***. (Смотрите раздел "Преобразования разборки/сборки" 432 в Главе 15 395 для более подробной информации о ***dma_map_sg***.)

Ваш драйвер должен выделить место для хранения списка разборки перед вызовом ***blk_rq_map_sg***. Список должен иметь возможность удерживать по крайней мере, так много записей, сколько имеет физических сегментов запрос; поле **nr_phys_segments** в **struct request** хранит такой счётчик, который не будет превышает максимальное число физических сегментов, заданных с помощью **blk_queue_max_phys_segments**.

Если вы не хотите, чтобы ***blk_rq_map_sg*** объединяла смежные сегменты, вы можете изменить заданное по умолчанию поведение таким вызовом:

```
clear_bit(QUEUE_FLAG_CLUSTER, &queue->queue_flags);
```

Некоторые дисковые SCSI драйверы отмечают таким образом свою очередь запросов, поскольку они не получают выгоды от объединения запросов.

Работа без очереди запросов

Ранее мы обсуждали работу, выполняемую ядром для оптимизации порядка запросов в очереди; эта работа включает в себя сортировку запросов и, возможно, даже остановку очереди, чтобы позволить поступить ожидаемым запросам. Эти методы помогают системной производительности при работе с реальным вращающимся дисковым накопителем. Однако, они полностью бесполезны с устройством, подобным ***sbull***. Многие блочно-ориентированные устройства, такие как массивы флэш-памяти, считыватели мультимедийных карт, используемых в цифровых камерах и диски в ОЗУ имеют настоящую производительность при произвольном доступе и не получают пользы от расширенной логики очередей запросов. Другие устройства, такие, как программные RAID массивы или виртуальные диски, созданные менеджерами логических разделов, не имеют характеристик производительности, для которых оптимизированы очереди запросов блочного уровня. Для такого рода устройства было бы лучше принимать запросы непосредственно от блочного уровня, и совсем не возиться с очередью запросов.

Для таких ситуаций, блочный уровень поддерживает режим работы "без очереди". Для использования данного режима ваш драйвер должен предоставить функцию "выполнить

запрос", а не функцию ***request***. Функция ***make_request*** имеет следующий прототип:

```
typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);
```

Обратите внимание, что очередь запросов по-прежнему присутствует, хотя она фактически никогда не будет принимать какие-либо запросы. Функция ***make_request*** принимает в качестве своего основного параметра структуру **bio**, которая представляет собой один или несколько буферов, подлежащих передаче. Функция ***make_request*** может выполнить одно из двух: оно может либо непосредственно выполнить передачу, либо может перенаправить запрос на другое устройство.

Выполнение прямой передачи является лишь вопросом работы через **bio** с помощью методов доступа, которые мы описали выше. Однако, поскольку не существует структуры request для работы с ней, ваша функция должна сообщить о завершении непосредственно создателю структуры **bio** вызовом ***bio_endio***:

```
void bio_endio(struct bio *bio, unsigned int bytes, int error);
```

Здесь **bytes** является количеством байтов, которое вы передали до этого. Она может быть меньше, чем количество байт, предоставленных **bio** в целом; таким образом, вы можете сообщить о частичном завершении и обновить внутренние указатели "текущего буфера" в **bio**. Вы должны либо вызвать ***bio_endio*** снова, как только ваше устройство выполнит дальнейший процесс, или просигнализировать об ошибке, если вам не удаётся выполнить запрос. Ошибки указываются передачей ненулевого значения параметра **error**; эта значение, как правило, код ошибки, такой как **-EIO**. ***make_request*** должна вернуть 0, независимо от того, успешен ли ввод/вывод.

Если ***sbull*** загружается с **request_mode=2**, он работает с функцией ***make_request***. Так как ***sbull*** уже имеет функцию, которая может передать одну **bio**, функция ***make_request*** очень проста:

```
static int sbull_make_request(request_queue_t *q, struct bio *bio)
{
    struct sbull_dev *dev = q->queuedata;
    int status;

    status = sbull_xfer_bio(dev, bio);
    bio_endio(bio, bio->bi_size, status);
    return 0;
}
```

Обратите внимание, что вы никогда не должны вызывать ***bio_endio*** из обычной функции ***request***; вместо этого работа выполняется функцией ***end_that_request_first***.

Некоторым блочным драйверам, таким как реализующим менеджеры разделов и программные RAID массивы, действительно необходимо перенаправить запрос другому устройству, которое выполняет фактический ввода/вывод. Написание такого драйвера выходит за рамки этой книги. Заметим, однако, что если функция ***make_request*** возвращает ненулевое значение, **bio** передаётся снова. "Эшелонированный" драйвер может, следовательно, изменить поле **bi_bdev**, чтобы указать на другое устройство, изменить начальное значение сектора, а затем вернуться; затем блочная система передаёт **bio** новому устройству. Существует также вызов ***bio_split***, которые может быть использован для

разделения **bio** на несколько кусков, для передачи более чем одному устройству. Хотя, если параметры очереди установлены должным образом, в расщеплении **bio** таким образом почти никогда нет необходимости.

В любом случае, вы должны сообщить блочной подсистеме, что ваш драйвер использует собственную функцию ***make_request***. Чтобы это сделать, вы должны создать очередь запросов с помощью:

```
request_queue_t *blk_alloc_queue(int flags);
```

Эта функция отличается от ***blk_init_queue*** в том, что она фактически не создаёт очередь для хранения запросов. Аргумент **flags** представляет собой набор флагов создания, которые будут использоваться при выделении для очереди памяти; обычно верным значением является **GFP_KERNEL**. После того, как вы имеете очередь, передайте её в вашу функцию ***make_request*** для ***blk_queue_make_request***:

```
void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);
```

Код ***sbull***, устанавливающий функцию ***make_request***, выглядит следующим образом:

```
dev->queue = blk_alloc_queue(GFP_KERNEL);
if (dev->queue == NULL)
    goto out_vfree;
blk_queue_make_request(dev->queue, sbull_make_request);
```

Для любопытных, некоторое время, проведённое в копании в ***drivers/block/ll_rw_block.c*** показывает, что все очереди имеют функцию ***make_request***. Версия по умолчанию, ***generic_make_request***, обрабатывает объединение **bio** в структуру **request**. Предоставляя собственную функцию ***make_request***, драйвер в действительности только подменяет данный метод ***очереди запросов*** и выполняет многое из этой работы.

Некоторые другие подробности

Этот раздел охватывает несколько других аспектов блочного уровня, которые могут представлять интерес для усовершенствованных драйверов. Чтобы написать корректный драйвер, нет необходимости использовать ни одну из следующих возможностей, но в некоторых ситуациях они могут быть полезны.

Предварительная подготовка команд

Блочный уровень представляет механизм для драйверов для изучения и предварительной обработки запросов, прежде чем они вернутся из ***elv_next_request***. Этот механизм позволяет драйверам установить фактические команды для привода заранее во времени, решить, можно ли вообще выполнить запрос, или выполнять другие виды служебных действий.

Если вы хотите использовать эту возможность, создайте функцию подготовки команд, которая соответствует этому прототипу:

```
typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);
```

Структура **request** включает в себя поле, называемое **cmd**, которое представляет собой

массив из **BLK_MAX_CDB** байт; этот массив может быть использован функцией подготовки для хранения команды реального оборудования (или любой другой полезной информации). Эта функция должна вернуть одно из следующих значений:

BLKPREP_OK
Подготовка команды прошла нормально и запрос может быть обработан функцией ***request*** вашего драйвера.

BLKPREP_KILL
Эта запрос не может быть завершён; он не удался с каким-то кодом ошибки.

BLKPREP_DEFER
Этот запрос не может быть завершён в это время. Он находится в начале очереди, но не передан в функцию ***request***.

Функция подготовки вызывается ***elv_next_request*** непосредственно перед тем, как запрос возвращается в ваш драйвер. Если эта функция возвращает **BLKPREP_DEFER**, возвращаемым значение из ***elv_next_request*** в ваш драйвер является NULL. Этот режим операции может быть полезным, если, например, ваше устройство достигло максимального числа запросов, которое может иметь невыполненными.

Чтобы блочный уровень мог вызвать вашу функцию подготовки, передайте её в:

```
void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);
```

По умолчанию очереди запросов не имеют функции подготовки.

Очереди помеченных команд

Оборудование, которое может иметь несколько активных запросов одновременно обычно поддерживает ту или иную форму очереди помеченных команд (tagged command queueing, TCQ). TCQ - просто техника подключения целочисленной "метки" к каждому запросу, так что когда накопитель завершает один из этих запросов, он может сообщить драйверу, какой из них. В предыдущих версиях ядра блочные драйверы, который реализовывали TCQ, делали всю работу самостоятельно; в версии 2.6, в блочный уровень была добавлена поддержка инфраструктуры TCQ для использования всеми драйверами.

Если ваш накопитель обрабатывает очереди помеченных команд, вы должны сообщить ядру этот факт во время инициализации вызовом:

```
int blk_queue_init_tags(request_queue_t *queue, int depth, struct
blk_queue_tag *tags);
```

Здесь, **queue** является вашей очередью запросов, а **depth** является числом помеченных запросов, которые ваше устройство может иметь невыполненными в любой момент времени. **tags** является необязательным указателем на массив структур **struct blk_queue_tag**; в нём должно быть **depth** их. Как правило, **tags** может быть передан как NULL и ***blk_queue_init_tags*** создаст массив. Если, однако, необходимо совместно использовать одни и те же метки несколькими устройствами, вы можете передать указатель на массив меток (хранящийся в поле **queue_tags**) от другой очереди запросов. Вы никогда не должны на самом деле самостоятельно выделять массив меток; блочный уровень должен проинициализировать

массив и не экспортирует для модулей функцию инициализации.

Так как ***blk_queue_init_tags*** выделяет память, она может потерпеть неудачу; в этом случае она возвращает вызывающему отрицательный код ошибки.

Если количество меток, которое ваше устройство может обрабатывать изменилось, вы можете проинформировать об этом ядру с помощью:

```
int blk_queue_resize_tags(request_queue_t *queue, int new_depth);
```

Во время этого вызова должна удерживаться блокировка очереди. Этот вызов может оказаться неудачным, в этом случае он возвращает отрицательный код ошибки.

Связь метки со структурой **request** осуществляется с помощью ***blk_queue_start_tag***, которая должна быть вызвана при удержании блокировки очереди:

```
int blk_queue_start_tag(request_queue_t *queue, struct request *req);
```

Если метка доступна, эта функция выделяет её по этому запросу, сохраняет номер метки в **req->tag** и возвращает 0. Она также извлекает запрос из очереди и связывает его со своей собственной структурой для отслеживания меток, поэтому если драйвер использует метки, необходимо позаботиться о том, чтобы не выполнять удаление запроса из очереди самостоятельно. Если доступных меток больше нет, ***blk_queue_start_tag*** оставляет запрос в очереди и возвращает ненулевое значение.

Когда все передачи для данного запроса завершены, ваш драйвер должен вернуть эту метку с помощью:

```
void blk_queue_end_tag(request_queue_t *queue, struct request *req);
```

Ещё раз, вы должны удерживать блокировку очереди перед вызовом этой функции. Этот вызов должен быть сделан после того, как ***end_that_request_first*** возвращает 0 (что означает, что запрос завершён), но перед вызовом ***end_that_request_last***. Помните, что запрос уже удалён из очереди, поэтому было бы ошибкой для вашего драйвера делать это на данном этапе.

Если вам необходимо найти запрос, связанный с данной меткой (когда накопитель сообщает о завершение, к примеру), используйте ***blk_queue_find_tag***:

```
struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);
```

Возвращаемое значение является ассоциированной структурой **request**, если что-то не пошло по настоящему не так.

Если дело действительно идёт не так, ваш драйвер может иметь необходимость сбросить или выполнять какие-то другие насильственные действия в отношении одного из своих устройств. В этом случае любые невыполненные помеченные команды не будут завершены. Блочный уровень предоставляет функцию, которая может помочь в усилиях по восстановлению в таких ситуациях:

```
void blk_queue_invalidate_tags(request_queue_t *queue);
```

Эта функция возвращает все невыполненные метки пулу и помещает соответствующие запросы обратно в очередь запросов. Когда вы вызываете эту функцию, должна удерживаться блокировка очереди.

Краткая справка

#include <linux/fs.h>
int register_blkdev(unsigned int major, const char *name);
int unregister_blkdev(unsigned int major, const char *name);
register_blkdev регистрирует блочный драйвер в ядре и, при необходимости, получает старший номер. Драйвер может быть разрегистрирован с помощью *unregister_blkdev*.
struct block_device_operations
Структура, которая содержит большинство методов для блочных драйверов.

#include <linux/genhd.h>
struct gendisk;
Структура, которая описывает единственное блочное устройство в ядре.
struct gendisk *alloc_disk(int minors);
void add_disk(struct gendisk *gd);
Функции, которые выделяют структуры **gendisk** и возвращают их в систему.
void set_capacity(struct gendisk *gd, sector_t sectors);
Сохраняет объём устройства (в секторах по 512 байт) в структуре **gendisk**.
void add_disk(struct gendisk *gd);
Добавляет в ядро диск. Как только эта функция вызвана, ваши методы диска могут быть использованы ядром.
int check_disk_change(struct block_device *bdev);
Функция ядра, которая проверяет смену носителя в данном дисковом накопителе и принимает необходимые меры по очистке при обнаружении таких изменений.

#include <linux/blkdev.h>
request_queue_t blk_init_queue(request_fn_proc *request, spinlock_t *lock);
void blk_cleanup_queue(request_queue_t *);
Функции, которые занимаются созданием и удалением очередей блочного запроса.
struct request *elv_next_request(request_queue_t *queue);
void end_request(struct request *req, int success);
elv_next_request получает следующий запрос от очереди запросов; *end_request* может быть использована в очень простых драйверах, чтобы отметить полное завершение (или частичное) запроса.
void blkdev_dequeue_request(struct request *req);
void elv_requeue_request(request_queue_t *queue, struct request *req);
Функции, которые удаляют запрос из очереди и помещают его обратно в случае необходимости.
void blk_stop_queue(request_queue_t *queue);
void blk_start_queue(request_queue_t *queue);
Если вам необходимо предотвратить дальнейшие вызовы вашего метода *request*, вызов *blk_stop_queue* выполняет этот трюк. Вызов *blk_start_queue* необходим, чтобы снова вызвать ваш метод *request*.
void blk_queue_bounce_limit(request_queue_t *queue, u64 dma_addr);
void blk_queue_max_sectors(request_queue_t *queue, unsigned short max);
void blk_queue_max_phys_segments(request_queue_t *queue, unsigned short max);
void blk_queue_max_hw_segments(request_queue_t *queue, unsigned short max);

void blk_queue_max_segment_size(request_queue_t *queue, unsigned int max);
blk_queue_segment_boundary(request_queue_t *queue, unsigned long mask);
void blk_queue_dma_alignment(request_queue_t *queue, int mask);
void blk_queue_hardsect_size(request_queue_t *queue, unsigned short max);
Функции, которые устанавливают разные параметры очереди, управляющие тем, как создаются запросы для данного устройства; параметры описаны в разделе "Функции управления очередью" 461.

#include <linux/bio.h>
struct bio;
Низкоуровневая структура, представляющая собой часть запроса блочного ввода/вывода.
bio_sectors(struct bio *bio);
bio_data_dir(struct bio *bio);
Два макроса, которые дают размер и направление передачи, описываемой структурой **bio**.
bio_for_each_segment(bvec, bio, segno);
Псевдоуправляющая структура, использующая для прохода по сегментам, которые составляют структуру **bio**.
char *__bio_kmap_atomic(struct bio *bio, int i, enum km_type type);
void __bio_kunmap_atomic(char *buffer, enum km_type type);
__bio_kmap_atomic может быть использована для создания виртуального адреса ядра для данного сегмента в структуре **bio**. Отображение должно быть отменено с помощью ***__bio_kunmap_atomic***.
struct page *bio_page(struct bio *bio);
int bio_offset(struct bio *bio);
int bio_cur_sectors(struct bio *bio);
char *bio_data(struct bio *bio);
char *bio_kmap_irq(struct bio *bio, unsigned long *flags);
void bio_kunmap_irq(char *buffer, unsigned long *flags);
Набор макросов доступа, которые предоставляют доступ к "текущему" сегменту в структуре **bio**.
void blk_queue_ordered(request_queue_t *queue, int flag);
int blk_barrier_rq(struct request *req);
Вызывайте ***blk_queue_ordered***, если ваш драйвер реализует барьерные запросы, как это необходимо. Макрос ***blk_barrier_rq*** возвращает ненулевое значение, если текущий запрос является барьерным запросом.
int blk_noretry_request(struct request *req);
Этот макрос возвращает ненулевое значение, если данный запрос не должен быть повторен в случае ошибок.
int end_that_request_first(struct request *req, int success, int count);
void end_that_request_last(struct request *req);
Используйте ***end_that_request_first*** для сообщения о завершении части запроса блочного ввода/вывода. Когда эта функция возвращает 0, запрос является завершённым и должен быть передан в ***end_that_request_last***.
rq_for_each_bio(bio, request)
Другое реализованное в виде макроса управление структурой; он проходит по всем **bio**, которые составляют запрос.
int blk_rq_map_sg(request_queue_t *queue, struct request *req, struct scatterlist *list);

Заполняет данный список разборки информацией, необходимой для отображения буферов данного запроса для DMA передачи.

typedef int (make_request_fn) (request_queue_t *q, struct bio *bio);

Прототип функции ***make_request***.

void bio_endio(struct bio *bio, unsigned int bytes, int error);

Сигнал завершения для данной **bio**. Эта функция должна использоваться только если ваш драйвер получил **bio** непосредственно из блочного уровня через функцию ***make_request***.

request_queue_t *blk_alloc_queue(int flags);

void blk_queue_make_request(request_queue_t *queue, make_request_fn *func);

Используйте ***blk_alloc_queue*** для создания очереди запросов, которая используется со своей функцией ***make_request***. Эта функция должна быть установлена с помощью ***blk_queue_make_request***.

typedef int (prep_rq_fn) (request_queue_t *queue, struct request *req);

void blk_queue_prep_rq(request_queue_t *queue, prep_rq_fn *func);

Прототип и установка функции для функции подготовки команды, которая может быть использована для подготовки необходимой аппаратной команды, прежде чем запрос передаётся в вашу функцию ***request***.

int blk_queue_init_tags(request_queue_t *queue, int depth, struct blk_queue_tag *tags);

int blk_queue_resize_tags(request_queue_t *queue, int new_depth);

int blk_queue_start_tag(request_queue_t *queue, struct request *req);

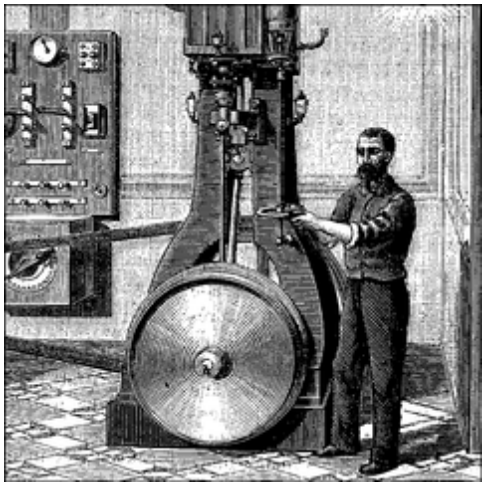
void blk_queue_end_tag(request_queue_t *queue, struct request *req);

struct request *blk_queue_find_tag(request_queue_t *qeue, int tag);

void blk_queue_invalidate_tags(request_queue_t *queue);

Функции поддержки для драйверов, использующих очереди помеченных команд.

Глава 17, Сетевые драйверы



Обсудив символьные и блочные драйверы, мы теперь готовы перейти в мир сетей. Сетевые интерфейсы представляют собой третий стандартный класс устройств Linux и в этой главе описывается, каким образом они взаимодействуют с остальной частью ядра.

Роль сетевого интерфейса в системе аналогична таковой для смонтированного блочного устройства. Блочное устройство регистрирует в ядре свои диски и методы и затем "передаёт" и "получает" блоки по запросу, означающему вызов его функции ***request***. Аналогичным образом сетевой интерфейс должен зарегистрировать себя в определённых структурах данных ядра, чтобы быть вызванным, когда происходит обмен пакетами с внешним миром.

Есть несколько важных различий между смонтированными дисками и интерфейсами доставки пакеты. Начнём с того, что диск существует в виде специального файла в каталоге */**dev***, в то время как сетевой интерфейс не имеет такой точки входа. Нормальные операции с файлами (чтение, запись и так далее) не имеют смысла применительно к сетевым интерфейсам, так что невозможно применять к ним подход Unix "всё есть файл". Таким образом, сетевые интерфейсы существуют в их собственном пространстве имён и экспортируют другой набор операций.

Хотя вы можете возразить,что при использовании сокетов приложения используют системные вызовы ***read*** и ***write***, эти вызовы на программное обеспечение объекта, который отличается от интерфейса. На одном физическом интерфейсе может быть мультиплексировано несколько сотен сокетов.

Но самое важное различие между ними в том, что блочные драйверы работают только отвечая на запросы из ядра, тогда как сетевые драйверы получают пакеты асинхронно извне. Таким образом, если блочный драйвер ***запрашивается***, чтобы послать буфер в ядро, сетевое устройство ***просит*** поместить входящие пакеты в ядро. Интерфейс ядра для сетевых драйверов разработан для этого другого режима управления.

Сетевые драйверы также должны быть готовы поддержать ряд административных задач, таких как настройка адресов, изменение параметров передачи и поддержание статистики трафика и ошибок. API для сетевых драйверов отражает эту необходимость и, таким образом, выглядит несколько отличающимся от интерфейсов, которые мы видели до сих пор.

Сетевая подсистема ядра Linux разработана так, чтобы быть полностью независимой от протокола. Это относится как к сетевым протоколам (Интернет-протокол [IP] по сравнению с IPX и другими протоколами) и аппаратным протоколам (Ethernet по сравнению с Token Ring и другими). Взаимодействие между сетевым драйвером и ядром имеет дело должным образом с одним сетевым пакетом за раз; это позволяет проблемам протокола быть аккуратно скрытыми от драйвера и физической передаче быть скрытой от протокола.

В этой главе описывается, каким образом сетевые интерфейсы вписываются в остальное ядро Linux и предоставляет примеры в виде базирующегося на памяти модульного сетевого интерфейса, который называется (как вы догадались) ***snull***. Для упрощения обсуждения интерфейс использует аппаратный протокол Ethernet и передаёт IP пакеты. Знания, которые вы приобретаете, рассматривая ***snull***, могут без труда применяться к протоколам, отличным от IP, и написание не-Ethernet драйвера отличается только в мелких деталях, относящихся с необходимому сетевому протоколу.

Эта глава не говорит о схемах нумерации IP, сетевых протоколах или других общих концепциях сетей. Такие темы (как правило) не представляют интерес для автора драйвера и невозможно предложить удовлетворительный обзор сетевой технологии менее чем в несколько сот страниц. Заинтересованному читателю настоятельно рекомендуется обратиться к другим книгам, описывающим сетевые проблемы.

Прежде, чем перейти к сетевым устройствам, сделаем одно замечание по терминологии. Сетевой мир использует термин ***октет***, чтобы сослаться на группу из восьми битов, которая, как правило, наименьшая единица, понимаемая сетевыми устройствами и протоколами. Термин байт почти никогда не встречается в этом контексте. В соответствии со стандартным использованием, мы будем использовать октет, когда речь идёт о сетевых устройствах.

Термин "заголовок" также заслуживает быстрого упоминания. Заголовок представляет собой набор байтов (ошибка, октетов) предшествующих пакету, так как он передаётся через различные уровни сетевой подсистемы. Когда приложение отправляет блок данных через TCP сокет, сетевая подсистема разбивает данные на пакеты и помещает TCP заголовок, описывающий, где каждый пакет находится в потоке, в начало. Нижние уровни затем размещают IP заголовок, используемый для маршрутизации пакетов по назначению, перед TCP заголовком. Если пакет движется по среде, подобной Ethernet, заголовок Ethernet, интерпретируемый оборудованием, идёт впереди остального. Сетевым драйверам не требуется самим заботится о высокоуровневых заголовках (как правило), но зачастую они должны быть вовлечены в создание заголовка на аппаратном уровне.

Каким разработан snull

В этом разделе обсуждаются концепции дизайна, которые привели сетевому интерфейсу ***snull***. Хотя эта информация может показаться несущественной, непонимание может привести к проблемам при игре с кодом примера.

Первым и самым главным дизайнерским решением было то, что интерфейсы примера должны оставаться независимыми от реального оборудования, так же как и большая часть кода примеров, используемого в этой книге. Это ограничение привело к чему-то, похожему на переделку интерфейса обратной связи. ***snull*** не является интерфейсом обратной связи; однако, он имитирует сеансы связи с настоящими удалёнными сетевыми устройствами, чтобы лучше продемонстрировать задачу написать сетевого драйвера. Драйвер обратной связи Linux на самом деле довольно прост; он может быть найден в ***drivers/net/loopback.c***.

Ещё одной особенностью ***snull*** является то, что он поддерживает только IP трафик. Это является следствием внутренние работы интерфейса - ***snull*** требуется смотреть внутрь и интерпретировать пакеты, чтобы правильно эмулировать пару аппаратных интерфейсов. Настоящие интерфейсы не зависят от протокола передачи и это ограничение ***snull*** не влияет на фрагменты кода, показанные в этой главе.

Назначение IP адресов

Модуль ***snull*** создаёт два интерфейса. Эти интерфейсы отличаются от простой обратной связи, где всё, что вы передаёте через один интерфейс возвращается обратно через другой, а не к самому себе. Это выглядит так, как будто у вас есть два внешних подключения, но на самом деле компьютер отвечая самому себе.

К сожалению, этот эффект не может быть достигнут только путём назначения IP адреса, потому что ядро не будет посылать пакеты через интерфейс А, которые были направлены на его собственный интерфейс Б. Вместо этого, оно будет использовать канал обратной связи без передачи через ***snull***. Чтобы установить связь через интерфейсы ***snull***, адреса источника и места назначения при передачи данных должны быть изменены. Другими словами, пакеты, отправленные через один из интерфейсов должны быть получены другим, но получатель исходящих пакетов не должен быть распознан, как локальный компьютер. То же самое относится к адресу источника полученных пакетов.

Для достижения такого рода "скрытой обратной связи", интерфейс ***snull*** переключает наименее значащий бит третьего октета обоих адресов источника и назначения; то есть, изменяет адрес сети и адрес компьютера IP адресов класса С. Результирующим эффектом является то, что пакеты, отправленные в сеть А (подключенной к **sn0**, первому интерфейсу) появляются на интерфейсе **sn1** как пакеты, принадлежащие сети Б.

Чтобы избежать иметь дело со слишком большим числом адресов, давайте назначим символические имена IP адресам, включающим:

- **snullnet0** - это сеть, которая подключена к интерфейсу **sn0**. Аналогично, **snullnet1** - сеть, подключенная к **sn1**. Адреса этих сетей должны отличаться только младшим битом третьего октета. Эти сети должны иметь 24-х разрядные сетевые маски.
- **local0** - это IP адрес, присвоенный интерфейсу **sn0**; он принадлежит **snullnet0**. Адресом, связанным с **sn1** является **local1**. **local0** и **local1** должны различаться младшим битом третьего октета и четвертым октетом.
- **remote0** является сетевым устройством в **snullnet0** и его четвертый октет такой же, что и у **local1**. Любой пакет, отправленный на **remote0** достигает **local1** после того, как его сетевой адрес был изменён кодом интерфейса. Сетевое устройство **remote1** принадлежит **snullnet1** и его четвёртый октет такой же, как у **local0**.

Функционирование интерфейса ***snull*** изображено на Рисунке 17-1, на котором имя сетевого устройства, связанного с каждым интерфейсом, напечатано вблизи имени интерфейса.

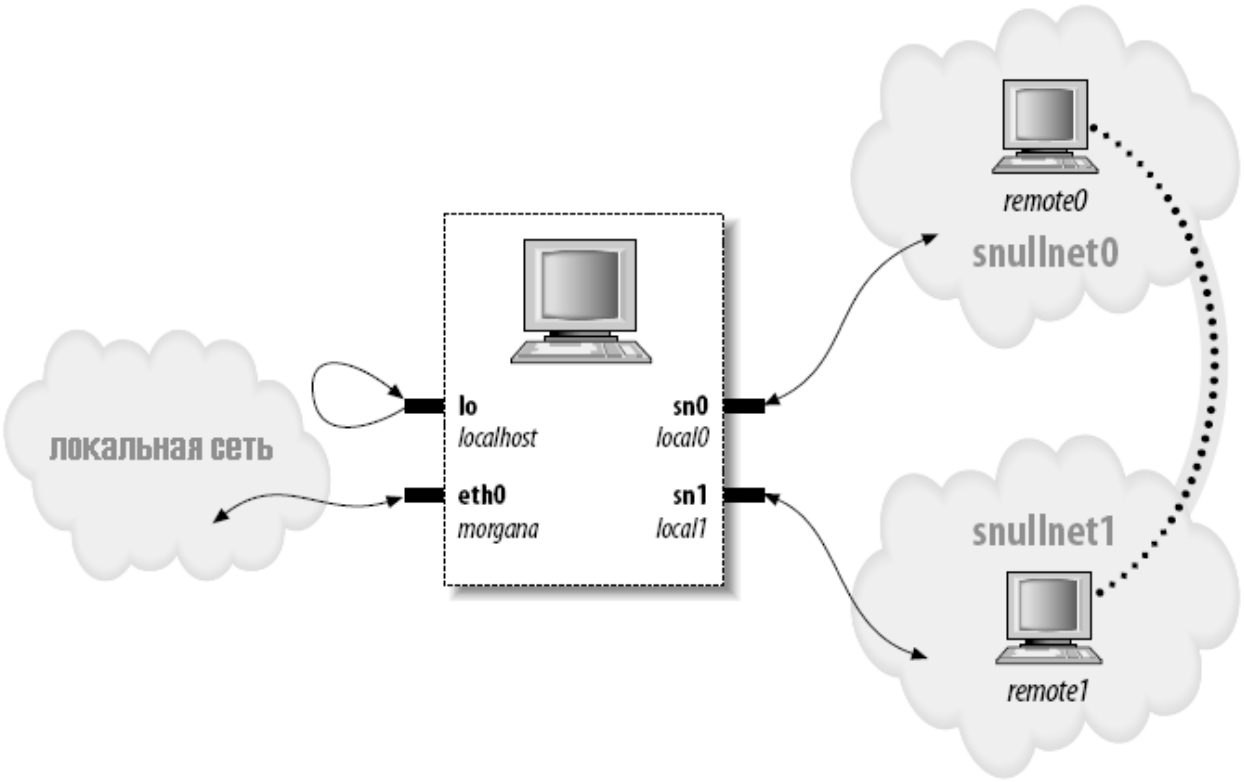


Рисунок 17-1. Как сетевое устройство видит свои интерфейсы

Вот возможные значения для сетевых адресов. Как только вы поместили эти строки в */etc/networks*, вы можете обратиться к вашим сетям по имени. Эти значения были выбраны из диапазона адресов, отведённых для частного использования.

```
snullnet0 192.168.0.0
snullnet1 192.168.1.0
```

Ниже приведены возможные адреса сетевых устройств, чтобы поместить их в */etc/networks*:

```
192.168.0.1 local0
192.168.0.2 remote0
192.168.1.2 local1
192.168.1.1 remote1
```

Важная особенность этих адресов в том, что часть для адреса устройства **local0** такая же, как и у **remote1**, и часть для адреса устройства **local1** та же самая, что и у **remote0**. Вы можете использовать совершенно разные номеров до тех пор, пока выполняется это отношение.

Однако, будьте осторожны, если ваш компьютер уже подключен к сети. Адреса, которые вы выбираете, могут быть реальными адресами Интернет или внутренней сети, и их назначение вашим интерфейсам прервёт связь с настоящими сетевыми устройствами. Например, хотя только что показанные адреса не являются маршрутизируемыми Интернет-адресами, они уже могут использоваться в вашей частной сети.

Независимо от адресов, которые вы выберете, вы можете правильно настроить

интерфейсы для работы, вводя следующие команды:

```
ifconfig sn0 local0
ifconfig sn1 local1
```

Вам может понадобиться добавить параметр **netmask 255.255.255.0**, если выбранный адресный диапазона ее является диапазоном класса C.

На данный момент может быть достигнут "удалённый" конец интерфейса. Следующая распечатка экрана показывает, как компьютер через интерфейс ***snull*** достигает **remote0** и **remote1**:

```
morgana% ping -c 2 remote0
64 bytes from 192.168.0.99: icmp_seq=0 ttl=64 time=1.6 ms
64 bytes from 192.168.0.99: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss

morgana% ping -c 2 remote1
64 bytes from 192.168.1.88: icmp_seq=0 ttl=64 time=1.8 ms
64 bytes from 192.168.1.88: icmp_seq=1 ttl=64 time=0.9 ms
2 packets transmitted, 2 packets received, 0% packet loss
```

Заметим, что вам не удастся достичь какого-либо другого "компьютера", принадлежащего этим двум сетям, потому что пакеты отбрасываются вашим компьютером после того, как адрес был изменён и пакет был получен. Например, пакет, направленный на 192.168.0.32 будет выходить через **sn0** и вновь появится на **sn1** с адресом назначения 192.168.1.32, который не является локальным адресом для данного компьютера.

Физический транспорт пакетов

Что касается передачи данных, интерфейсы ***snull*** принадлежат к классу Ethernet.

snull эмулирует Ethernet, поскольку подавляющее большинство существующих сетей, по крайней мере сегменты, которые соединяют рабочие станции, базируются на технологии Ethernet, будь то 10Base-T, 100Base-T или Gigabit. Кроме того, ядро предлагает некоторую обобщённую поддержку для устройств Ethernet и нет никаких оснований не использовать её. Преимущество быть устройством Ethernet настолько сильно, что даже интерфейс ***plip*** (интерфейс, который используют принтерные порты) заявляет о себе, как о устройстве Ethernet.

Последним преимуществом использования установки Ethernet для ***snull*** является то, что вы можете запустить на интерфейсе ***tcpdump***, чтобы увидеть проходящие через него пакеты. Наблюдение за интерфейсом с помощью ***tcpdump*** может оказаться полезным способом увидеть, как работают эти два интерфейса.

Эта быстрая и грязная модификация данных уничтожает не-IP пакеты. Если вы хотите доставить через ***snull*** другие протоколы, вы должны изменить исходный код модуля.

Как уже упоминалось ранее, ***snull*** работает только с IP пакетами. Это ограничение - результат того факта, что ***snull*** роется в пакетах и даже изменяет их, чтобы код работал. Код изменяет источник, место назначения и контрольную сумму в IP заголовке каждого пакета, не проверяя, передаётся ли на самом деле IP информация.

Подключение к ядру

Мы начинаем просмотр структуры сетевых драйверов анализируя исходный код ***snull***. Сохранение под рукой исходного кода нескольких драйверов может помочь вам следить за обсуждением и увидеть, как работают реальные сетевые драйверы Linux. В качестве места для начала, мы предлагаем ***loopback.c***, ***plip.c*** и ***e100.c***, в порядке возрастания сложности. Все эти файлы находятся в ***drivers/net*** в дереве исходных текстов ядра.

Регистрация устройства

Когда модуль драйвера загружается в работающее ядро, он запрашивает ресурсы и предлагает возможности; в этом нет ничего нового. И также нет ничего нового в том, как запрашиваются ресурсы. Драйверу необходимо проверить его устройство и размещение оборудования (порты ввода/вывода и линию прерывания), но не регистрировать их, как описано в разделе "Установка обработчика прерываний" 247 в Главе 10 246. Способ, которым сетевой драйвер регистрируется своей функцией инициализации модуля, отличается от символьных и блочных драйверов. Поскольку для сетевых интерфейсов не существует эквивалента старшего и младшего номеров, сетевой драйвер не запрашивает такой номер. Вместо этого, драйвер помещает структуру данных для каждого вновь обнаруженного интерфейса в глобальный список сетевых устройств.

Каждый интерфейс описывается объектом **struct net_device**, которая определена в ***<linux/netdevice.h>***. Драйвер ***snull*** хранит указатели на две такие структуры (**sn0** и **sn1**) в простом массиве:

```
struct net_device *snull_devs[2];
```

Структура **net_device**, как и многие другие структуры ядра, содержит **kobject** и является, таким образом, учитываемой по ссылке и экспортируемой через sysfs. Как и в случае с другими такими структурами, она должна быть создана динамически. Функцией ядра, предоставленной для выполнения такого выделения памяти, является ***alloc_netdev***, которая имеет следующий прототип:

```
struct net_device *alloc_netdev(int sizeof_priv,
                                const char *name,
                                void (*setup)(struct net_device *));
```

Здесь, **sizeof_priv** является размером области "личных данных" драйвера; в случае сетевых устройств, эта область выделяется вместе со структурой **net_device**. На самом деле, они обе создаются вместе в одном большом участке памяти, но авторы драйверов должны притворяться, что они не знают этого. **name** является именем этого интерфейса, как оно видится пользовательским пространством; это имя может содержать **%d** в стиле ***printf***. Ядро заменяет **%d** следующим доступным номером интерфейса. Наконец, **setup** является указателем на функцию инициализации, которая вызывается для настройки остальной части структуры **net_device**. Мы скоро доберёмся до функции инициализации, но на данный момент достаточно сказать, что ***snull*** выделяет память для своих двух структур устройства таким образом:

```
snull_devs[0] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
snull_devs[1] = alloc_netdev(sizeof(struct snull_priv), "sn%d", snull_init);
```

```
if (snull_devs[0] == NULL || snull_devs[1] == NULL)
    goto out;
```

Как всегда, мы должны проверить возвращаемое значение, чтобы быть уверенными, что выделение памяти успешно.

Сетевая подсистема предоставляет ряд вспомогательных интерфейсных функций вокруг ***alloc_netdev*** для различных типов интерфейсов. Наиболее распространённой является ***alloc_etherdev***, которая определена в ***<linux/etherdevice.h>***:

```
struct net_device *alloc_etherdev(int sizeof_priv);
```

Эта функция выделяет память для сетевого устройства используя для аргумента имени **eth %d**. Он обеспечивает свою функцию инициализации (***ether_setup***), которая устанавливает несколько полей в **net_device** в соответствующие значения для устройств Ethernet. Таким образом, для функции ***alloc_etherdev*** нет поставляемой драйвером функции инициализации, драйвер должен просто выполнять требуемую инициализацию непосредственно после успешного выделения памяти. Авторы драйверов для других типов устройств могут захотеть воспользоваться одной из других вспомогательных функций, такой как ***alloc_fcdev*** (определена в ***<linux/fcdevice.h>***) для волоконно-оптических устройств, ***alloc_fddidev*** (***<linux/fddidevice.h>***) для FDDI устройств или ***alloc_trdev*** (***<linux/trdevice.h>***) для устройств Token Ring.

snull мог бы без проблем использовать ***alloc_etherdev***; мы решили использовать вместо этого ***alloc_netdev***, как способ демонстрации низкоуровневого интерфейса и получения контроля над именем, назначаемым интерфейсу.

После того как структура ***net_device*** была проинициализирована, завершение этого процесса - просто вопрос передачи структуры в ***register_netdev***. В ***snull*** вызов выглядит следующим образом:

```
for (i = 0; i < 2; i++)
    if ((result = register_netdev(snull_devs[i])))
        printk("snull: error %i registering device \"%s\"\n",
                result, snull_devs[i]->name);
```

Здесь применимо обычное предостережение: как только вы вызвали ***register_netdev***, ваш драйвер может быть вызван для операций с устройством. Таким образом, вы не должны регистрировать устройство, пока всё не было полностью проинициализировано.

Инициализация каждого устройства

Мы посмотрели на создание и регистрацию структур **net_device**, но мы прошло мимо промежуточного шага полной инициализации такой структуры. Обратите внимание, что **struct net_device** всегда собирается воедино во время работы, она не может быть создана во время компиляции тем же способом, как структуры **file_operations** или **block_device_operations**. Эта инициализация должна быть завершена перед вызовом ***register_netdev***. Структура **net_device** большая и сложная; к счастью, ядро заботится о некоторых общих значениях по умолчанию для Ethernet через функцию ***ether_setup*** (которую вызывает ***alloc_etherdev***).

Так как ***snull*** использует ***alloc_netdev***, он имеет отдельную функцию инициализации. Суть

этой функции (***snull_init***) выглядит следующим образом:

```
ether_setup(dev); /* assign some of the fields */

dev->open            = snull_open;
dev->stop            = snull_release;
dev->set_config      = snull_config;
dev->hard_start_xmit = snull_tx;
dev->do_ioctl        = snull_ioctl;
dev->get_stats       = snull_stats;
dev->rebuild_header  = snull_rebuild_header;
dev->hard_header     = snull_header;
dev->tx_timeout      = snull_tx_timeout;
dev->watchdog_timeo  = timeout;
/* сохраняем флаги по умолчанию, добавляем только NOARP */
dev->flags           |= IFF_NOARP;
dev->features        |= NETIF_F_NO_CSUM;
dev->hard_header_cache = NULL; /* Запрет кэширования */
```

Вышеприведённый код является довольно обычной инициализацией структуры **net_device**; это, главным образом, вопрос сохранения указателей на наши разнообразные функции драйвера. Одно необычной особенностью этого кода является установка **IFF_NOARP** в флагах. Это указывает, что данный интерфейс не может использовать протокол разрешения адреса (Address Resolution Protocol, ARP). ARP является низкоуровневым протоколом Ethernet; его работа заключается в превращении IP адресов в Ethernet адреса протокола управления доступом к среде (medium access control, MAC). Поскольку "удалённые" системы, моделируемые ***snull***, в действительности не существуют, никто не способен ответить на ARP запросы для них. Вместо того, чтобы усложнить ***snull*** добавлением реализации ARP, мы решили пометить интерфейс как неспособный обработать этот протокол. Установка **hard_header_cache** делается по той же причине: она отключает кэширование (несуществующих) ARP ответов на этом интерфейсе. Эта тема подробно обсуждается в разделе "Разрешение MAC адреса" 512 далее в этой главе.

Код инициализации также устанавливает пару полей (**tx_timeout** и **watchdog_timeo**), которые связаны с обработкой таймаутов передачи. Мы тщательно освещаем эту тему в разделе "Таймауты при передаче" 499.

Сейчас мы рассмотрим ещё одно поле **struct net_device**, **priv**. Его роль аналогична указателю **private_data**, который мы использовали для символьных драйверов. В отличие от **fops->private_data**, этот указатель **priv** создаётся вместе со структурой **net_device**. Прямой доступ к полю **priv** также не рекомендуется по причинам производительности и гибкости. Когда драйверу необходимо получить доступ к закрытому указателю данных, он должен использовать функцию ***netdev_priv***. Таким образом, драйвер ***snull*** полон деклараций, таких как:

```
struct snull_priv *priv = netdev_priv(dev);
```

Модуль ***snull*** объявляет структуру данных **snull_priv**, которая будет использоваться для **priv**:

```
struct snull_priv {
    struct net_device_stats stats;
    int status;
```

```
    struct snull_packet *ppool;
    struct snull_packet *rx_queue; /* Список входящих пакетов */
    int rx_int_enabled;
    int tx_packetlen;
    u8 *tx_packetdata;
    struct sk_buff *skb;
    spinlock_t lock;
};
```

Структура включает, среди прочего, экземпляр **struct net_device_stats**, который является стандартным местом для хранения статистики интерфейса. Следующие строки в ***snull_init*** создают и инициализируют **dev->priv**:

```
priv = netdev_priv(dev);
memset(priv, 0, sizeof(struct snull_priv));
spin_lock_init(&priv->lock);
snull_rx_ints(dev, 1);      /* разрешаем приём прерываний */
```

Выгрузка модуля

При выгрузке модуля не происходит ничего особенного. Функция очистки модуля просто отменяет регистрацию интерфейсов, выполняет все необходимые внутренние очистки и возвращает структуру **net_device** обратно системе:

```
void snull_cleanup(void)
{
    int i;

    for (i = 0; i < 2; i++) {
        if (snull_devs[i]) {
            unregister_netdev(snull_devs[i]);
            snull_teardown_pool(snull_devs[i]);
            free_netdev(snull_devs[i]);
        }
    }
    return;
}
```

Вызов ***unregister_netdev*** удаляет интерфейс из системы; ***free_netdev*** возвращает структуру **net_device** ядру. Если где-то на эту структуру существует ссылка, она может продолжить существовать, но драйверу не надо заботиться об этом. После того, как вы отменили регистрацию интерфейса, ядро уже не вызывает его методы.

Обратите внимание, что наша внутренняя очистка (выполненная в ***snull_teardown_pool***) не может произойти, пока устройство не разрегистрировано. Однако, это должно произойти прежде, чем мы вернём структуру **net_device** системе; после того, как мы вызвали ***free_netdev***, мы не сможем делать любые дальнейшие ссылки на устройство или нашу закрытую область.

Структура net_device в деталях

Структура **net_device** находится в самом центре уровня сетевого драйвера и заслуживает полного описания. Этот список описывает все поля, но больше предоставлен как справочная

информация, чем как материал для запоминания. Оставшаяся часть этой главы кратко описывает каждое поле, как только оно будет использовано в коде примера, так что вам не потребуется возвращаться к этому разделу.

Общая информация

Первая часть **struct net_device** состоит из следующих полей:

char name[IFNAMSIZ];
Имя этого устройства. Если имя устанавливаемое драйвером содержит строку форматирования **%d**, ***register_netdev*** заменяет её числом, чтобы сделать имя уникальным; присваиваемые номера начинаются с 0.

unsigned long state;
Состояние устройства. Поле включает в себя несколько флагов. Драйверы обычно не манипулируют этими флагами напрямую; вместо того, предоставлен набор вспомогательных функций. Эти функции будут вкратце обсуждаться, когда мы перейдём к операциям драйвера.

struct net_device *next;
Указатель на следующее устройство в глобальном связном списке. Это поле не должно быть затронуто драйвером.

int (*init)(struct net_device *dev);
Функция инициализации. Если этот указатель установлен, эта функция вызывается ***register_netdev*** для завершения инициализации структуры **net_device**. Большинство современных сетевых драйверов больше не используют эту функцию; вместо этого инициализация выполняется перед регистрацией интерфейса.

Информация об оборудовании

Следующие поля содержат низкоуровневую информацию об оборудовании для относительно простых устройств. Они являются пережитком первых дней сетевой работы в Linux; самые современные драйверы их не используют (с возможным исключением **if_port**). Мы перечисляем их здесь для полноты.

unsigned long rmem_end;
unsigned long rmem_start;
unsigned long mem_end;
unsigned long mem_start;
Информация о памяти устройства. Эти поля хранят адреса начала и окончания разделяемой памяти, используемой устройством. Если устройство имеет разную память для приёма и передачи, поля **mem** используются для памяти передачи и поля **rmem** для памяти приёма. Поля **rmem** никогда не ссылаются за пределы самого драйвера. По соглашению, в поля **end** настроены так, что **end - start** является объёмом доступной на борту памяти.

unsigned long base_addr;
Базовый адрес ввода/вывода сетевого интерфейса. Это поле, как и предыдущие, задаётся драйвером в течение зондирования устройства. Для просмотра и изменения текущего значения может быть использована команда ***ifconfig***. **base_addr** может быть

задан явно командной строкой ядра при загрузке системы (с помощью параметра **netdev=**) или во время загрузки модуля. Это поле, как и поля памяти, описанные выше, ядром не используется.

unsigned char irq;
Назначенный номер прерывания. Значение **dev->irq** печатается ***ifconfig*** при перечислении интерфейсов. Эта значение обычно можно установить во время старта системы или загрузки модуля и изменить позже с помощью ***ifconfig***.

unsigned char if_port;
Порт, используемый в мультипортовых устройствах. Это поле используется, например, устройствами, которые поддерживают оба Ethernet соединения: как коаксиальную (**IF_PORT_10BASE2**), так и витую пару (**IF_PORT_100BASET**). Полный набор известных типов портов определён в ***<linux/netdevice.h>***.

unsigned char dma;
Канал DMA, выделенный устройством. Поле имеет смысл лишь для некоторых периферийных шин, таких как ISA. Оно не используется за пределами самого драйвера устройства, только для информационных целей (в ***ifconfig***).

Информация об интерфейсе

Большая часть информации об интерфейсе устанавливается должным образом функцией ***ether_setup*** (или любой другой функцией установки, соответствующей данному типу оборудования). Сетевые карты могут рассчитывать на эту функцию общего назначения для большинства из этих полей, но поля **flags** и **dev_addr** являются зависимыми от устройства и должны быть заданы явно во время инициализации.

Некоторые не-Ethernet интерфейсы могут использовать вспомогательные функции, аналогичные ***ether_setup***. ***drivers/net/net_init.c*** экспортирует ряд таких функций, включая следующие:

void ltalk_setup(struct net_device *dev);
Устанавливает поля для устройства LocalTalk.

void fc_setup(struct net_device *dev);
Инициализирует поля для волоконно-оптических устройств.

void fddi_setup(struct net_device *dev);
Конфигурирует интерфейс для сети с Fiber Distributed Data Interface (распределенным интерфейсом передачи данных по волоконно-оптическим каналам, FDDI).

void hippi_setup(struct net_device *dev);
Готовит полей для High-Performance Parallel Interface (высокопроизводительного параллельного интерфейса, HIPPI) драйвера высокоскоростного соединения.

void tr_setup(struct net_device *dev);
Выполняет настройку для сетевых интерфейсов token ring (маркерное кольцо).

Большинство устройств охватываются одним из этих классов. Однако, если у вас что-то радикально новое и отличающееся, необходимо определить следующие поля вручную:

unsigned short hard_header_len;
Длина аппаратного заголовка, то есть число октетов, которые предваряют передаваемый пакет до заголовка IP или другой протокольной информации. Для Ethernet интерфейсов значением **hard_header_len** является **14** (**ETH_HLEN**).

unsigned mtu;
Максимальный блок передачи (maximum transfer unit, MTU). Это поле используется сетевым уровнем чтобы управлять передачей пакета. Ethernet имеет MTU из 1500 октетов (**ETH_DATA_LEN**). Это значение может быть изменено с помощью ***ifconfig***.

unsigned long tx_queue_len;
Максимальное количество кадров, которые могут быть поставлены в очередь в очередь передачи устройства. Это значение устанавливается ***ether_setup*** в 1000, но вы можете его изменить. Например, ***plip*** использует 10, чтобы избежать потери памяти (***plip*** имеет меньшую пропускную способность, чем настоящий Ethernet интерфейс).

unsigned short type;
Аппаратный тип интерфейса. Поле **type** используется ARP для определения, какой тип аппаратного адреса поддерживает интерфейс. Верным значением для Ethernet интерфейсов является **ARPHRD_ETHER** и это значение устанавливается ***ether_setup***. Распознаваемые типы определены в ***<linux/if_arp.h>***.

unsigned char addr_len;
unsigned char broadcast[MAX_ADDR_LEN];
unsigned char dev_addr[MAX_ADDR_LEN];
Длина аппаратного (MAC) адреса и адреса оборудования устройства. Длиной Ethernet адреса является шесть октетов (мы имеем в виду аппаратный идентификатор интерфейсной платы), а широковещательный (broadcast) адрес состоит из шести октетов **0xff**; ***ether_setup*** принимает меры, чтобы эти значения были корректными. Адрес устройства, с другой стороны, должен быть прочитан из интерфейсной платы зависимым от устройства способом и драйвер должен скопировать его в **dev_addr**. Аппаратный адрес используется для генерации корректных заголовков Ethernet, до того, как пакет передан драйверу для передачи. Устройство ***snull*** не использует физический интерфейс и он изобретает свой собственный аппаратный адрес.

unsigned short flags;
int features;
Флаги интерфейса (подробности ниже).

Поле **flags** является битовой маской, включающей следующие битовые значения. Префикс **IFF_** означает "интерфейсные флаги". Некоторыми флагами управляет ядро, а некоторые устанавливаются интерфейсом во время инициализации, чтобы заявить о различных возможностях и других особенностях интерфейса. Действительными флагами, которые определены в ***<linux/if.h>***, являются:

IFF_UP
Драйвер может только читать этот флаг. Ядро устанавливает его, когда интерфейс активен и готов к передаче пакетов.

IFF_BROADCAST

Этот флаг (поддерживаемый сетевым кодом) сообщает, что интерфейс позволяет широковещательные (broadcast) запросы. Ethernet платы делают это.

IFF_DEBUG
Он отмечает режим отладки. Этот флаг может быть использован для управления многословностью ваших вызовов ***printk*** или для других целей отладки. Хотя ни один драйвер в дереве в настоящее время не использует этот флаг, он может быть установлен и сброшен пользовательскими программами через ***ioctl*** и ваш драйвер может использовать его. Для включения и выключения этого флага может быть использована программа ***misc-progs/netifdebug***.

IFF_LOOPBACK
Этот флаг должен быть установлен только в интерфейсе обратной связи. Ядро проверяет **IFF_LOOPBACK** вместо аппаратного имени **lo** как отдельный интерфейс.

IFF_POINTOPOINT
Этот флаг сигнализирует, что интерфейс подключен к соединению "точка-точка". Он устанавливается драйвером или, иногда, ***ifconfig***. Например, его устанавливают ***plip*** и драйвер ***PPP***.

IFF_NOARP
Это означает, что интерфейс не может выполнять ARP. Например, интерфейсам точка-точка не требуется запускать ARP, который лишь создаст дополнительный трафик без извлечения полезной информации. ***snull*** работает без возможностей ARP, поэтому он устанавливает флаг.

IFF_PROMISC
Этот флаг устанавливается (сетевым кодом), чтобы активировать неразборчивое функционирование (promiscuous operation). По умолчанию, Ethernet интерфейсы используют аппаратный фильтр, для уверенности, что они получают широковещательные пакеты и пакеты, направляемые только на этот аппаратный адрес интерфейса. Пакетные снифферы, такие как ***tcpdump***, устанавливают на интерфейсе режим неразборчивости, с тем, чтобы получать все пакеты, которые проходят по среде передачи интерфейса.

IFF_MULTICAST
Этот флаг устанавливается драйвером, чтобы отметить интерфейсы, которые поддерживают многоадресную (multicast) передачу. ***ether_setup*** устанавливает **IFF_MULTICAST** по умолчанию, поэтому если ваш драйвер не поддерживает многоадресность, он должен во время инициализации очистить этот флаг.

IFF_ALLMULTI
Этот флаг говорит интерфейсу получать все многоадресные (multicast) пакеты. Ядро устанавливает его, когда сетевое устройство выполняет многоадресную маршрутизацию, только если установлен **IFF_MULTICAST**. Драйвера может только читать **IFF_ALLMULTI**. Флаги многоадресности используются в разделе "Многоадресность" 517 далее в этой главе.

IFF_MASTER
IFF_SLAVE
Эти флаги используются кодом выравнивания нагрузки. Драйверу интерфейса не требуется знать о них.

IFF_PORTSEL
IFF_AUTOMEDIA
Эти флаги сигнализируют о том, что устройство способен переключаться между несколькими типами носителей; например, между неэкранированной витой парой (unshielded twisted pair, UTP) и коаксиальными кабелями Ethernet. Если **IFF_AUTOMEDIA** установлен, устройство выбирает необходимый тип носителя автоматически. На практике ядро не использует ни один из этих флагов.

IFF_DYNAMIC
Этот флаг, установленный драйвером, свидетельствует о том, что адрес этого интерфейса можно изменить. В настоящее время ядром не используется.

IFF_RUNNING
Этот флаг показывает, что этот интерфейс подключен и работает. Он присутствует в основном для совместимости с BSD; ядро мало использует его. Большинству сетевых драйверов нет необходимости беспокоиться о **IFF_RUNNING**.

IFF_NOTRAILERS
Этот флаг не используется в Linux, но он существует для совместимости с BSD.

Когда программа изменяет **IFF_UP**, вызывается метод устройства ***open*** или ***stop***. Кроме того, при изменении **IFF_UP** или любого другого флага вызывается метод ***set_multicast_list***. Если драйвер должен выполнять какие-то действия в ответ на модификацию этих флагов, он должен выполнить это действие в ***set_multicast_list***. Например, когда установлен или сброшен **IFF_PROMISC**, ***set_multicast_list*** должен уведомить об этом фильтр оборудования на плате. Обязанности этого метода устройства изложены в разделе "Многоадресность" 517.

Поле **features** в структуре **net_device** устанавливается драйвером, чтобы сообщить ядру о любых специальных аппаратных возможностях, которые имеет этот интерфейс. Мы обсудим некоторые из этих возможностей; другие выходят за рамки этой книги. Полный набор состоит из:

NETIF_F_SG
NETIF_F_FRAGLIST
Оба этих флага управляют использованием ввода/вывода с разборкой/сборкой. Если ваш интерфейс может передать пакет, который был разбит на несколько отдельных сегментов памяти, вам следует установить **NETIF_F_SG**. Конечно, необходимо реально реализовать ввод/вывод с разборкой/сборкой (мы расскажем, как это делается в разделе "Ввод/вывод с разборкой/сборкой" 500). **NETIF_F_FRAGLIST** заявляет, что ваш интерфейс может справиться с пакетами, которые были фрагментированы; в версии 2.6 это делает только драйвер обратной связи (loopback).
Заметим, что ядро не выполняет ввод/вывод с разборкой/сборкой для вашего устройства, если оно не обеспечивает также в той или иной форме подсчёт контрольной суммы. Причина в том, что если ядро должно сделать проход по фрагментированному ("нелинейному") пакету для расчёта контрольной суммы, оно может также в одно и тоже время копировать данные и объединять пакет.

NETIF_F_IP_CSUM
NETIF_F_NO_CSUM
NETIF_F_HW_CSUM

Эти флаги всеми способами говорят ядру, что нет необходимости применять контрольные суммы к некоторым или всем пакетам, покидающим систему через этот интерфейс. Установите **NETIF_F_IP_CSUM**, если ваш интерфейс может рассчитывать контрольную сумму IP пакетов, но не других. Если для этого интерфейса даже не требуется расчёт контрольных сумм, установите **NETIF_F_NO_CSUM**. Драйвер обратной связи (loopback) устанавливает этот флаг и ***snull*** это тоже делает; как только пакеты переданы через системную память, нет (надо надеяться!) возможности для их повреждения и нет необходимости в их проверке. Если ваше оборудование само рассчитывает контрольную сумму, установите **NETIF_F_HW_CSUM**.

NETIF_F_HIGHDMA
Установите этот флаг, если ваше устройство может выполнять DMA в верхней памяти. В отсутствие этого флага, все пакетные буферы, предоставляемые вашему драйверу, выделяются в нижней памяти.

NETIF_F_HW_VLAN_TX
NETIF_F_HW_VLAN_RX
NETIF_F_HW_VLAN_FILTER
NETIF_F_VLAN_CHALLENGED
Эти параметры описывают, что ваше оборудование поддерживает пакеты 802.1q VLAN. Поддержка VLAN выходит за границы того, что мы можем охватить в этой главе. Если пакеты VLAN сбивают с толка ваше устройство (что они в действительности делать не должны), установите флаг **NETIF_F_VLAN_CHALLENGED**.

NETIF_F_TSO
Установите этот флаг, если ваше устройство может выполнять избавление от сегментации TCP (TCP segmentation offloading). TSO является дополнительной функцией, которую мы не можем здесь описать.

Методы устройства

Как и в случае с символьными и блочными драйверами, каждое сетевое устройство заявляет о функциях, которые на нём работают. В этом разделе перечислены операции, которые могут быть выполнены на сетевых интерфейсах. Некоторые из этих операций могут быть оставлены NULL, а другие, как правило, не тронуты, поскольку для них подходящие методы назначает ***ether_setup***.

Методы устройства для сетевого интерфейса можно разделить на две группы: основные и необязательные. Основные методы включают в себя те, которые необходимы, чтобы было возможно использовать интерфейс; необязательные методы реализуют более расширенную функциональность, которая не требуется в обязательном порядке. Ниже приведены основные методы:

int (*open)(struct net_device *dev);
Открывает интерфейс. Интерфейс открывается всякий раз, когда его активирует ***ifconfig***. Метод ***open*** должен зарегистрировать все необходимые системные ресурсы (порты ввода/вывода, IRQ, DMA и так далее), включить оборудование и выполнить все другие настройки, требующиеся вашему устройству.

int (*stop)(struct net_device *dev);
Останавливает интерфейс. Интерфейс остановлен, когда он приведён в выключенное

состояние. Эта функция должна отменить операции, выполненные во время открытия.

int (*hard_start_xmit) (struct sk_buff *skb, struct net_device *dev);
Метод, который инициирует передачу пакета. Полный пакет (заголовки протокола и всё остальное) содержится в структуре буфера сокета (**sk_buff**). Буферы сокетов описываются позже в этой главе.

int (*hard_header) (struct sk_buff *skb, struct net_device *dev, unsigned short type, void *daddr, void *saddr, unsigned len);
Функция (вызываемая перед ***hard_start_xmit***), которая строит аппаратный заголовок из аппаратных адресов источника и назначения, которые были получены ранее; её работа заключается в организации информации, переданной ей в качестве аргументов в соответствующем зависимом от устройства аппаратном заголовке. ***eth_header*** является функцией по умолчанию для интерфейсов, подобных Ethernet, и ***ether_setup*** устанавливает это поле соответственно.

int (*rebuild_header)(struct sk_buff *skb);
Функция, используемая для перестройки аппаратного заголовка после завершения разрешения (определения адреса) протоколом ARP, но перед передачей пакета. Функция по умолчанию, используемая устройствами Ethernet, для заполнения пакета недостающей информацией использует код поддержки ARP.

void (*tx_timeout)(struct net_device *dev);
Метод, вызываемый сетевым кодом, когда передача пакета не была завершена в течение разумного периода, исходя из предположения, что было пропущено прерывание или интерфейс заблокирован. Он должен справиться с этой проблемой и возобновить передачу пакетов.

struct net_device_stats *(*get_stats)(struct net_device *dev);
Всякий раз, когда приложению необходимо получить статистику для интерфейса, вызывается этот метод. Это происходит, например, когда запускается ***ifconfig*** или ***netstat -i***. Пример реализации для ***snull*** приводится в разделе "Статистическая информация" 516.

int (*set_config)(struct net_device *dev, struct ifmap *map);
Изменяет конфигурацию интерфейса. Этот метод является точкой входа для настройки драйвера. Во время выполнения с помощью ***set_config*** может быть изменён адрес ввода/вывода для устройства и его номер прерывания. Данная возможность может быть использована администратором системы, если интерфейс не может быть для этого прозондирован. Драйверам для современного оборудования обычно не требуется реализовывать этот метод.

Остальные операции устройства являются необязательными:

int weight;
int (*poll)(struct net_device *dev; int *quota);
Метод, предусмотренный NAPI-совместимыми драйверами, для работы интерфейса в режиме опроса, с запрещёнными прерываниями. NAPI (и его поле **weight**) рассматривается в разделе "Уменьшение числа прерываний" 505.

void (*poll_controller)(struct net_device *dev);

Функция, которая просит драйвер проверить события на интерфейсе в ситуациях, когда прерывания запрещены. Она используется особых сетевых задач внутри ядра, таких как удалённое управление и отладка ядра через сеть.

int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
Выполняет специфичные для интерфейса команды ***ioctl***. (Реализация этих команд описана в разделе "Дополнительные команды ioctl" 515.) Соответствующее поле в **struct net_device** можно оставить как NULL, если интерфейс не требует каких-либо команд, специфичных для интерфейса.

void (*set_multicast_list)(struct net_device *dev);
Метод вызывается, когда изменяется список многоадресного запроса для устройства и когда изменяются флаги. Смотрите раздел "Многоадресность" 517 для более подробной информации и примера реализации.

int (*set_mac_address)(struct net_device *dev, void *addr);
Функция, которая может быть реализована, если интерфейс поддерживает возможность изменить свой аппаратный адрес. Многие интерфейсы совсем не поддерживают эту возможность. Другие используют реализацию по умолчанию ***eth_mac_addr*** (из ***drivers/net/net_init.c***). ***eth_mac_addr*** только копирует новый адрес в **dev->dev_addr** и делает это только тогда, когда интерфейс не работает. Драйверам, которые используют ***eth_mac_addr***, следует устанавливать аппаратный MAC адрес из **dev->dev_addr** в их методе ***open***.

int (*change_mtu)(struct net_device *dev, int new_mtu);
Функция, которая выполняет действия, если произошло изменение максимального блока передачи (MTU) для интерфейса. Если драйверу необходимо сделать что-то особенное при изменении пользователем MTU, он должен объявить эту свою функцию; в противном случае, функция по умолчанию сделает это по-своему. ***snull*** имеет шаблон для данной функции, если вам это интересно.

int (*header_cache) (struct neighbour *neigh, struct hh_cache *hh);
header_cache вызывается для заполнения структуры **hh_cache** результатами запроса ARP. Почти все драйверы, подобные работающим с Ethernet, могут использовать для ***eth_header_cache*** реализацию по умолчанию .

int (*header_cache_update) (struct hh_cache *hh, struct net_device *dev, unsigned char *haddr);
Метод, который обновляет адрес назначения в структуре **hh_cache** в ответ на изменение. Ethernet устройства используют ***eth_header_cache_update***.

int (*hard_header_parse) (struct sk_buff *skb, unsigned char *haddr);
Метод ***hard_header_parse*** извлекает адрес источника из пакета, содержащегося в **skb**, скопирует его в буфер по **haddr**. Возвращаемым значением этой функции является длина этого адреса. Ethernet устройства обычно используют ***eth_header_parse***.

Вспомогательные поля

Остальные поля данных **struct net_device** используются интерфейсом для хранения полезной информации о состоянии. Некоторые из этих полей используются ***ifconfig*** и ***netstat*** для предоставления пользователю информации о текущей конфигурации. Таким образом,

интерфейс должен присвоить значения этим полям:

unsigned long trans_start;
unsigned long last_rx;
Поля, которые содержат число тиков (jiffies). Драйвер несёт ответственность за обновление этих значений при начале передачи и когда получен пакет, соответственно. Значение **trans_start** используется сетевой подсистемой для обнаружения блокировки передатчика. **last_rx** настоящее время не используется, однако драйвер должен поддерживать это поле в любом случае, чтобы быть готовым для его использования в будущем.

int watchdog_timeo;
Минимальное время (в тиках), которое должно пройти, прежде чем сетевой уровень примет решение о том, что при передаче произошёл таймаут, и вызовет функцию драйвера ***tx_timeout***.

void *priv;
Эквивалент **filp->private_data**. В современных драйверах это поле задается функцией ***alloc_netdev*** и не должно быть доступно непосредственно; используйте для этого ***netdev_priv***.

struct dev_mc_list *mc_list;
int mc_count;
Поля, которые управляют многоадресной передачей. **mc_count** является числом элементов в **mc_list**. Для более подробной информации смотрите раздел "Многоадресность" 517.

spinlock_t xmit_lock;
int xmit_lock_owner;
xmit_lock используется, чтобы избежать множества одновременных вызовов функции драйвера ***hard_start_xmit***. **xmit_lock_owner** является числом процессоров, которые получили **xmit_lock**. Драйвер не должен делать изменений в этих полях.

В **struct net_device** есть и другие поля, но они не используются сетевыми драйверами.

Открытие и закрытие

Наш драйвер может проверить интерфейс во время загрузки модуля или при загрузке ядра. Однако, до того, как интерфейс сможет передавать пакеты, ядро должно его открыть и присвоить ему адрес. Ядро открывает или закрывает интерфейс в ответ на команду ***ifconfig***.

Когда ***ifconfig*** используется, чтобы присвоить интерфейсу адрес, она выполняет две задачи. Во-первых, она присваивает адрес при помощи **ioctl(SIOCSIFADDR)** (Socket I/O Control Set Interface Address, установка адреса интерфейса для управления сокетом ввода/вывода). Затем она устанавливает бит **IFF_UP** в **dev->flag** с помощью **ioctl (SIOCSIFFLAGS)** (Socket I/O Control Set Interface Flags, установка флагов интерфейса для управления сокетом ввода/вывода), чтобы включить интерфейс.

Что касается устройства, **ioctl(SIOCSIFADDR)** не делает ничего. Функция драйвера не вызывается - задача является независимой от устройства и её выполняет ядро. Однако,

последняя команд (**ioctl(SIOCSIFFLAGS)**) вызывает метод ***open*** для данного устройства.

Аналогично, когда интерфейс выключается, ***ifconfig*** использует **ioctl(SIOCSIFFLAGS)** для очистки **IFF_UP** и вызывается метод ***stop***.

Оба метода устройства возвращают 0 в случае успеха и обычное отрицательное значение в случае ошибки.

Как же касается собственно кода, драйвер должен выполнять многие из тех же задач, что делают символьные и блочные драйверы. ***open*** запрашивает все необходимые системные ресурсы, необходимые и приказывает интерфейсу подняться; ***stop*** выключает интерфейс и освобождает системные ресурсы. Однако, сетевые драйверы во время ***открытия*** должны выполнять некоторые дополнительные действия.

Во-первых, до того, как интерфейс сможет общаться с внешним миром, из аппаратного устройства в **dev->dev_addr** должен быть скопирован аппаратный (MAC) адрес. Аппаратный адрес может быть скопирован в устройство во время открытия. Программный интерфейс ***snull*** присваивает его внутри ***open***; он просто подделывает аппаратный адрес, используя строку ASCII длиной **ETH_ALEN**, длины адресов Ethernet оборудования.

Метод ***open*** также должен запустить очередь передачи интерфейса (разрешая ему принимать пакеты для передачи), как только он готов начать передачу данных. Чтобы запустить очередь, ядро предоставляет функцию:

```
void netif_start_queue(struct net_device *dev);
```

Код ***open*** для ***snull*** выглядит следующим образом:

```
int snull_open(struct net_device *dev)
{
    /* request_region( ), request_irq( ), .... (подобно fops->open) */

    /*
     * Назначаем плате аппаратный адрес: используем "\0SNULx", где
     * x это 0 или 1. Первым байтом является '\0', чтобы не быть групповым
     * адресом (первый байт групповых адресов является нечётным).
     */
    memcpy(dev->dev_addr, "\0SNUL0", ETH_ALEN);
    if (dev == snull_devs[1])
        dev->dev_addr[ETH_ALEN-1]++; /* \0SNUL1 */
    netif_start_queue(dev);
    return 0;
}
```

Как вы можете видеть, в отсутствие реального оборудования, в методе ***open*** мало что можно сделать. То же самое относится и к методу ***stop***; он просто отменяет операции ***open***. По этой причине, функцию, реализующую ***stop***, часто называет ***close*** или ***release***.

```
int snull_release(struct net_device *dev)
{
    /* освобождаем порты, прерывание и остальное -- подобно fops->close */

    netif_stop_queue(dev); /* больше передавать не можем */
```

```
    return 0;
}
```

Функция:

```
void netif_stop_queue(struct net_device *dev);
```

является противоположностью ***netif_start_queue***; она отмечает, что устройство больше не может передавать пакеты. Функция должна вызываться при закрытии интерфейса (в методе ***stop***), но может также использоваться для временного прекращения передачи, как описано в следующем разделе.

Передача пакетов

Наиболее важными задачами, выполняемыми сетевыми интерфейсами, является передача и приём данных. Мы начнём с передачи, потому что это немного легче для понимания.

Передача подразумевает процесс посылки пакета через сетевую связь. Всякий раз, когда ядро должно передать пакет, оно вызывает метод драйвера ***hard_start_xmit***, чтобы поместить данные в исходящую очередь. Каждый пакет, обрабатываемым ядром, содержится в структуре буфера сокета (**struct sk_buff**), определение которой содержится в ***<linux/skbuff.h>***. Структура получила свое название от абстракции Unix, используемой для представления сетевого соединения, ***сокет***. Даже если интерфейс ничего с сокетами не делает, каждый сетевой пакет принадлежит сокету на более высоких сетевых уровнях и входные/выходные буферы любого сокета являются списками структур **struct sk_buff**. Та же структура **sk_buff** используется для размещения сетевых данных во всех сетевых подсистемах Linux, но что касается интерфейса, буфер сокета представляет собой просто пакет.

Указатель на **sk_buff** обычно называют **skb** и мы следуем этой практике и в коде примера и в тексте.

Буфер сокета представляет собой сложную структуру и для работы с ним ядро предлагает ряд функций. Функции описываются позже в разделе "Буферы сокетов" 509; сейчас же, чтобы написать рабочий драйвер, вполне достаточно нескольких основных фактов о **sk_buff**.

Буфер сокета, передаваемый в ***hard_start_xmit***, содержит физический пакет так, как он должен выглядеть на носителе информации, укомплектованный заголовками этого уровня передачи. Интерфейсу нет необходимости изменять передаваемые данные. **skb->data** указывает на передаваемый пакет, а **skb->len** является его длиной в октетах. Эта ситуация становится немного более сложной, если ваш драйвер может обработать ввод/вывод с разборкой/сборкой; мы опишем это в разделе "Ввод/вывод с разборкой/сборкой" 500.

Далее приведён код передачи пакетов в ***snull***; физические механизмы передачи были выделены в другую функцию, потому что каждый драйвер интерфейс должен реализовать его в зависимости от того специфического оборудования, которым он управляет:

```
int snull_tx(struct sk_buff *skb, struct net_device *dev)
{
    int len;
    char *data, shortpkt[ETH_ZLEN];
    struct snull_priv *priv = netdev_priv(dev);

```

```
    data = skb->data;
    len = skb->len;
    if (len < ETH_ZLEN) {
        memset(shortpkt, 0, ETH_ZLEN);
        memcpy(shortpkt, skb->data, skb->len);
        len = ETH_ZLEN;
        data = shortpkt;
    }
    dev->trans_start = jiffies; /* сохраняем текущее время */

    /* Запоминаем skb, так что мы сможем освободить его вовремя прерывания */
    priv->skb = skb;

    /* настоящая доставка данных зависит от устройства и здесь не показана */
    snull_hw_tx(data, len, dev);

    return 0; /* Наше простое устройство не может дать ошибку */
}
```

Таким образом, функция передачи просто выполняет некоторые проверки пакета и передаёт данные через функцию, связанную с оборудованием. Обратите внимание, однако, на те действия, которые выполняются, когда пакет для передачи меньше, чем минимальная длина, поддерживаемая используемым носителем информации (который, для ***snull***, наш виртуальный "Ethernet"). Во многих сетевых драйверах Linux (а также и других операционных систем) была обнаружена потеря данных в таких ситуациях. Вместо того, чтобы создать такого рода уязвимость безопасности, мы копируем короткие пакеты в отдельный массив, который мы можем явно дополнить нулями до полной длины, требуемой носителем информации. (Мы можем смело поместить данные в стек, поскольку минимальная длина, 60 байт, достаточно мала).

Возвращаемым значением из ***hard_start_xmit*** при успехе должен быть **0**; в этот момент ваш драйвер взял на себя ответственность за пакет, следует приложить все усилия для обеспечения успешности передачи и в конце необходимо освободить **skb**. Ненулевое возвращаемое значение показывает, что данный пакет не может быть передан в это время; ядро повторит попытку позже. В этой ситуации ваш драйвер должен остановить очередь, пока любая причиной отказа не будет разрешена.

"Зависящая от оборудования" функция передачи (***snull_hw_tx***) здесь пропущена, так как она полностью занята реализацией хитростей устройства ***snull***, включая манипулирование адресами источника и назначения, и представляет небольшой интерес для авторов настоящих сетевых драйверов. Конечно, она присутствует в исходном тексте примера для тех, кто хочет пойти и посмотреть, как она работает.

Управление конкуренцией при передаче

Функция ***hard_start_xmit*** защищена от одновременных вызовов спин-блокировкой (**xmit_lock**) в структуре **net_device**. Однако, как только функция возвращается, она может быть вызвана снова. Функция возвращается когда программа закончила инструктаж оборудования о передаче пакетов, но аппаратная передача к этому моменту будет, вероятно, не завершена. Это не проблема для ***snull***, который делает всю свою работу используя процессор, поэтому передача пакетов завершена до того, как функция передачи возвращается.

С другой стороны, настоящие аппаратные интерфейсы передают пакеты асинхронно и имеют ограниченный объём памяти, доступной для хранения исходящих пакетов. Когда это память исчерпана (что на некотором оборудовании происходит и с одним подготовленным для передачи пакетом), драйвер должен сообщить сетевой системе не начинать любую другую передачу, пока оборудование не готово к приёму новых данных.

Такое уведомление осуществляется вызовом ***netif_stop_queue***, эта функция приводилась ранее для остановки очереди. После того, как ваш драйвер остановил свою очередь, он должен организовать перезапуск очереди в какой-то момент в будущем, когда он снова сможет принимать пакеты для передачи. Чтобы сделать это, следует вызвать:

```
void netif_wake_queue(struct net_device *dev);
```

Эта функция выглядит похожей на ***netif_start_queue***, за исключением того, что она также толкает сетевую систему, чтобы она снова начала передачу пакетов.

Большинство современного сетевого оборудования поддерживает внутреннюю очередь для передачи нескольких пакетов; таким образом, оно может получить от сети лучшую производительность. Сетевые драйверы для таких устройств должны поддерживать в любой момент времени наличие нескольких исходящих передач, но память устройства можно заполнить независимо от того, поддерживает или нет оборудование несколько исходящих передач. Всякий раз, когда память устройства заполняется до той точки, где нет места для максимально большого пакета, драйвер должен остановить очередь, пока пространство снова не станет доступным.

Если вам необходимо отключить передачу пакетов из любой места, отличного от вашей функции ***hard_start_xmit*** (возможно, в ответ на запрос переконфигурации), функцией, которую вы захотите использовать является:

```
void netif_tx_disable(struct net_device *dev);
```

Эта функция ведёт себя так же, как ***netif_stop_queue***, но также гарантирует, что когда она возвращается, ваш метод ***hard_start_xmit*** не работает на другом процессоре. Как обычно, эта очередь может быть перезапущена с помощью ***netif_wake_queue***.

Таймауты при передаче

Большинство драйверов, которые имеют дело с реальным оборудованием, должны быть готовы к тому, что изредка оборудование не реагируют на запросы. Интерфейсы могут забыть, что они делают, или система может потерять прерывание. Такой вид проблемы часто возникает при использовании некоторых устройств, разработанных для работы на персональных компьютерах.

Многие драйверы решают эту проблему установкой таймеров; если операция ещё не завершена к тому времени, как таймер истечёт, что-то не так. Сетевая система, как это случается, по существу является сложным набором конечных автоматов, управляемых массой таймеров. Таким образом, сетевой код находится в хорошем положении, чтобы обнаружить таймауты передачи, как часть своей обычной работы.

Таким образом, сетевым драйверам не требуется самостоятельно беспокоиться о выявлении таких проблем. Вместо этого, они должны только установить период ожидания, который ведёт в поле **watchdog_timeo** структуры **net_device**. Этот период, который

выражен в тиках, должен быть достаточно продолжительным, чтобы учитывать нормальные задержки при передачи (например, коллизии, происходящие из-за перегруженности сетевой среды).

Если текущее системное время превышает время устройства **trans_start** по крайней мере на время периода ожидания, сетевой уровень в конечном счёте вызывает метод драйвера ***tx_timeout***. Работой этого метода является сделать все необходимое, чтобы выяснить проблему и обеспечить надлежащее завершение любых передач, которые уже находятся в процессе выполнения. Важно, в частности, чтобы драйвер не потерял любой из буферов сокетов, которые были доверены ему сетевым кодом.

snull обладает способностью имитировать блокировку передатчика, которая управляется двумя параметрами во время загрузки:

```
static int lockup = 0;
module_param(lockup, int, 0);

static int timeout = SNULL_TIMEOUT;
module_param(timeout, int, 0);
```

Если драйвер загружен с параметром **lockup=n**, блокировка моделируется только для каждого n-го переданного пакета и поле **watchdog_timeo** установлено в заданное значение ожидания. При моделировании блокировок для предотвращения возникновения других попыток передачи ***snull*** также вызывает ***netif_stop_queue***.

Обработчик таймаута передачи в ***snull*** выглядит следующим образом:

```
void snull_tx_timeout (struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    PDEBUG("Transmit timeout at %ld, latency %ld\n", jiffies,
                jiffies - dev->trans_start);
    /* Имитируем прерывание передачи, чтобы дать произойти событиям */
    priv->status = SNULL_TX_INTR;
    snull_interrupt(0, dev, NULL);
    priv->stats.tx_errors++;
    netif_wake_queue(dev);
    return;
}
```

Когда при передаче происходит таймаут, драйвер должен отметить ошибку в статистике интерфейса и принять меры, чтобы сбросить устройств в нормальное состояние, чтобы могли быть переданы новые пакеты. Когда таймаут происходит в ***snull***, драйвер вызывает ***snull_interrupt*** для заполнения "пропавшего" прерывания и перезапускает очередь передачи с помощью ***netif_wake_queue***.

Ввод/вывод с разборкой/сборкой

Процесс создания пакета для передачи по сети включает в себя сборку нескольких штук. Пакетные данные часто должны быть скопированы из пользовательского пространства и также должны быть добавлены заголовки, используемые разными уровнями сетевого стека. Эта сборка может требовать копирования немалого количества данных. Однако, если сетевой интерфейс, который назначен для передачи пакета, может выполнять ввод/вывод с разборкой/

сборкой, пакет не требуется собирать в единый кусок и большей части такого копирования можно избежать. Ввод/вывод с разборкой/сборкой позволяет также передачу с "нулевым копированием" сетевых данных непосредственно из буферов пользовательского пространства.

Ядро не пропускает разобранные пакеты в ваш метод ***hard_start_xmit***, пока не установлен бит **NETIF_F_SG** в поле **features** структуры устройства. Если вы установили этот флаг, вам необходимо посмотреть на специальное поле "информация для общего использования" внутри **skb**, чтобы увидеть, состоит ли пакет из одного или нескольких фрагментов и найти разбросанные фрагменты, если это необходимо. Для доступа к этой информации существует специальный макрос; он называется ***skb_shinfo***. Первый шаг при передаче потенциально фрагментированных пакетов обычно выглядит примерно так:

```
if (skb_shinfo(skb)->nr_frags == 0) {
    /* Просто как обычно используем skb->data и skb->len */
}
```

Поле **nr_frags** сообщает, как много фрагментов было использовано для построения пакета. Если оно равно **0**, пакет состоит из одного куска и может быть доступен как обычно через поле **data**. Однако, если оно отлично от нуля, ваш драйвер должен перебрать и упорядочить для передачи каждый отдельный фрагмент. Поле **data** структуры **skb** указывает просто на первым фрагмент (по сравнению с полным пакетом, как в нефрагментированном случае). Длина фрагмент должна рассчитываться вычитанием **skb->data_len** из **skb->len** (которое по-прежнему содержит длину полного пакета). Остальные фрагменты должны быть найдены в массиве, названном **frags**, в общей информационной структур; каждая запись в **frags** является структурой **skb_frag_struct**:

```
struct skb_frag_struct {
    struct page *page;
    __u16 page_offset;
    __u16 size;
};
```

Как вы можете видеть, мы снова имеем дело со структурами page, а не виртуальными адресами ядра. Ваш драйвер должен перебрать все фрагменты, отображая каждый для DMA передачи и не забывая при этом первый фрагмент, который указан напрямую через **skb**. Ваше оборудование, разумеется, должно собрать фрагменты и передать их как единый пакет. Обратите внимание, что если у вас установлен флаг функции **NETIF_F_HIGHDMA**, некоторые или все эти фрагменты могут быть расположены в верхней памяти.

Приём пакетов

Приём данных из сети сложнее, чем передача, потому что **sk_buff** должен быть выделен и передан в верхние уровни изнутри атомарного контекста. Существуют два режима приёма пакетов, которые могут быть реализованы сетевыми драйверами: управляемый прерыванием и опрос. Большинство драйверов реализуют технику управления прерыванием и именно её мы рассмотрим в первую очередь. Некоторые драйверы для адаптеров с высокой пропускной способностью могут также применять технику опроса; мы рассмотрим этот подход в разделе "Уменьшение числа прерываний" 505.

Реализация в ***snull*** отделяет "аппаратные" детали от аппаратно-независимых служебных действий. Таким образом, функция ***snull_rx*** вызывается из обработчика "прерываний" ***snull*** после того, как оборудование получило пакет и он уже находится в памяти компьютера.

snull_rx получает указатель на данные и длину пакета; его единственной обязанностью является отправить пакет и некоторую дополнительную информацию в вышележащие уровни сетевого кода. Этот код не зависит от способа получения указателя данных и длины.

```
void snull_rx(struct net_device *dev, struct snull_packet *pkt)
{
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);

    /*
     * Пакет вернулся из среды передачи.
     * Построим вокруг него skb, чтобы верхние уровни могли его обработать
     */
    skb = dev_alloc_skb(pkt->datalen + 2);
    if (!skb) {
        if (printk_ratelimit( ))
            printk(KERN_NOTICE "snull rx: low on mem - packet dropped\n");
        priv->stats.rx_dropped++;
        goto out;
    }
    memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);

    /* Запишем метаданные и затем передадим на уровень приёма */
    skb->dev = dev;
    skb->protocol = eth_type_trans(skb, dev);
    skb->ip_summed = CHECKSUM_UNNECESSARY; /* не проверяем это */
    priv->stats.rx_packets++;
    priv->stats.rx_bytes += pkt->datalen;
    netif_rx(skb);
out:
    return;
}
```

Функция носит достаточно общий характер, чтобы быть шаблоном для любого сетевого драйвера, но необходимы некоторые объяснения, прежде чем вы сможете с уверенностью повторно использовать этот фрагмент кода.

Первым шагом является выделение буфера для хранения пакета. Обратите внимание, что функции выделение буфера (***dev_alloc_skb***) необходимо знать длину данных. Информация используется функцией для выделения памяти для буфера. ***dev_alloc_skb*** вызывает ***kmalloc*** с атомарным приоритетом, поэтому она может безопасно использоваться во время прерывания. Ядро предлагает и другие интерфейсы для выделения буфера сокета, но они не стоят представления здесь; буферы сокетов подробно описываются в разделе "Буферы сокетов" 509.

Конечно, возвращаемое из ***dev_alloc_skb*** значение должно быть проверено и ***snull*** делает это. Однако, прежде чем жаловаться на неудачи, мы вызываем ***printk_ratelimit***. Генерация сотен или тысяч консольных сообщений в секунду является хорошим способом полностью затормозить систему и скрыть истинный источник проблем; ***printk_ratelimit*** помогает предотвратить эту проблему возвращая 0, когда на консоль идёт слишком много вывода, и вывод должен быть немного замедлен.

Если есть действительный указатель **skb**, пакетные данные копируются в буфер с помощью вызова ***memcpy***; функция ***skb_put*** обновляет указатель на конец данных в буфере и возвращает указатель на вновь созданное пространство.

Если вы пишете высокопроизводительный драйвер для интерфейса, который может выполнять полное управление шиной ввода/вывода, существует возможность оптимизации, которую стоит рассмотрения здесь. Некоторые драйверы выделяют буферы сокетов для входящих пакетов до их приёма, а затем поручают интерфейсу разместить пакетные данные прямо в пространство буфера сокета. Сетевой слой сотрудничает с этой стратегией, выделяя все буферы сокетов в DMA-совместимом пространстве (которое может быть в верхней памяти, если ваше устройство имеет установленным флаг функции **NETIF_F_HIGHDMA**). Делая вещи таким образом, отпадает необходимость в отдельной операции копирования для заполнения буфера сокета, но обязывает быть осторожными с размерами буферов, потому что вы не будете знать заранее, как велик входящий пакет. В этой ситуации также важна реализация метода ***change_mtu***, поскольку она позволяет драйверу реагировать на изменение максимального размера пакета.

Сетевой уровень должен иметь разъяснение некоторой информации, прежде чем он сможет понять смысл пакета. С этой целью до того, как буфер будет передан наверх должны быть присвоены значения полям **dev** и **protocol**. Код поддержки Ethernet экспортирует вспомогательную функцию (***eth_type_trans***), которая находит соответствующего значения для помещения в **protocol**. Затем необходимо указать, как должен быть выполнен подсчёт контрольной суммы или как она была посчитана для пакета (***snull*** не требуется выполнять подсчёт каких-либо контрольных сумм). Возможными политиками для **skb->ip_summed** являются:

CHECKSUM_HW
: Устройство уже выполнило подсчёт контрольных сумм на аппаратном уровне. Примером аппаратного подсчёта контрольной суммы является интерфейс SPARC HME.

CHECKSUM_NONE
: Контрольные суммы ещё не были проверены и задача должна быть выполнена системным программным обеспечением. Это значение по умолчанию во вновь выделенных буферах.

CHECKSUM_UNNECESSARY
: Не выполнять какой-либо подсчёт контрольных сумм. Это является политикой в ***snull*** и интерфейсе обратной связи.

Вы можете быть удивлены, почему здесь должен быть указан статус контрольной суммы, когда мы уже установил флаг в поле **features** нашей структуры **net_device**. Ответ в том, что флаг **features** сообщает ядру о том, как наше устройство обрабатывает исходящие пакеты. Он не используется для входящих пакетов, которые должны, напротив, быть помечены индивидуально.

Наконец, драйвер обновляет свой счётчик статистики, чтобы запротоколировать, что пакет был получен. Структура статистики состоит из нескольких полей; наиболее важными из них являются **rx_packets**, **rx_bytes**, **tx_packets** и **tx_bytes**, которые содержат количество пакетов, полученных и переданных, и общее количество переданных октетов. Все эти поля подробно описаны в разделе "Статистическая информация" 516.

Последний шаг в приёме пакета выполняется ***netif_rx***, которая вручает буфер сокета верхним уровням. ***netif_rx*** фактически возвращает целое число; **NET_RX_SUCCESS(0)** означает, что пакет был получен успешно; любое другое значение указывает на проблемы.

Существуют три возвращаемых значения (**NET_RX_CN_LOW**, **NET_RX_CN_MOD** и **NET_RX_CN_HIGH**), которые свидетельствуют об увеличении уровня перегрузки в сетевой подсистеме; **NET_RX_DROP** означает, что пакет был отброшен. Драйвер мог бы использовать эти значения для остановки снабжения пакетами ядра, когда перегруженность становится высокой, однако, на практике большинство драйверов игнорируют возвращаемое ***netif_rx*** значение. Если вы пишете драйвер для устройства с высокой пропускной способностью и хотите выполнять правильные действия в ответ на перегруженность, лучшим подходом является реализация NAPI, с которым мы познакомимся после быстрого обсуждения обработчиков прерываний.

Обработчик прерывания

Большинство аппаратных интерфейсов управляются посредством обработчика прерывания. Оборудование прерывает процессор, чтобы просигнализировать об одном из двух возможных событий: прибыл новый пакет или завершена передача исходящего пакета. Сетевые интерфейсы могут также генерировать прерывания для сигнализации об ошибках, изменениях состояния соединения и так далее.

Обычная процедура прерывания может определить отличие между прерыванием прибытия нового пакета и уведомлением о выполнении передачи, проверяя регистр статуса находящийся на физическом устройстве. Интерфейс ***snull*** работает аналогично, но его слово статуса реализовано в программном обеспечении и находится в **dev->priv**. Обработчик прерывания для сетевого интерфейса выглядит следующим образом:

```
static irqreturn_t snull_regular_interrupt(int irq, void *dev_id, struct
pt_regs *regs)
{
    int statusword;
    struct snull_priv *priv;
    struct snull_packet *pkt = NULL;
    /*
     * Как обычно, проверяем указатель "устройства" для уверенности,
     * что прервало действительно оно.
     * Затем присваиваем "struct device *dev"
     */
    struct net_device *dev = (struct net_device *)dev_id;
    /* ... и проверяем оборудование, если оно действительно наше */

    /* параноидальность */
    if (!dev)
        return;

    /* Блокируем устройство */
    priv = netdev_priv(dev);
    spin_lock(&priv->lock);

    /* получаем статусное слово: настоящее сетевое устройство использует
инструкции ввода/вывода */
    statusword = priv->status;
    priv->status = 0;
    if (statusword & SNULL_RX_INTR) {
        /* отправляем его в snull_rx для обработки */
        pkt = priv->rx_queue;
        if (pkt) {
```

```
            priv->rx_queue = pkt->next;
            snull_rx(dev, pkt);
        }
    }
    if (statusword & SNULL_TX_INTR) {
        /* передача завершена: освобождаем skb */
        priv->stats.tx_packets++;
        priv->stats.tx_bytes += priv->tx_packetlen;
        dev_kfree_skb(priv->skb);
    }

    /* Разблокируем устройство и заканчиваем */
    spin_unlock(&priv->lock);
    if (pkt) snull_release_buffer(pkt); /* Делаем это вне блокировки! */
    return IRQ_HANDLED;
}
```

Первой задачей обработчика является получение указателя на корректную **struct net_device**. Этот указатель обычно поступает от указателя **dev_id**, полученного в качестве аргумента.

С ситуацией "передача завершена" имеет дело интересная часть этого обработчика. В этом случае обновляется статистика и вызывается ***dev_kfree_skb***, чтобы вернуть системе (больше не требуемый) буфер сокет. На самом деле есть три варианта этой функции, которые могут быть вызваны:

dev_kfree_skb(struct sk_buff *skb);
: Эта версия может быть вызвана, когда вы знаете, что ваш код не будет работать в контексте прерывания. Так как ***snull*** не имеет фактического аппаратного прерывания, мы и используем эту версию.

dev_kfree_skb_irq(struct sk_buff *skb);
: Если вы знаете, что будете освобождать буфер в обработчике прерывания, используйте эту версию, которая оптимизирована для этого случая.

dev_kfree_skb_any(struct sk_buff *skb);
: Это версия используется, если соответствующий код может быть запущен или в контексте прерывания, или не в контексте прерывания.

Наконец, если ваш драйвер временно остановил очередь передачи, это обычное место для её перезапуска с помощью ***netif_wake_queue***.

Приём пакета, в отличие от передачи, не требует специальной обработки прерывания. Всё, что требуется, это вызвать ***snull_rx*** (который мы уже видели).

Уменьшение числа прерываний

Когда сетевой драйвер написан так, как мы описали выше, процессор прерывается для каждого пакета, полученного вашим интерфейсом. Во многих случаях это является желаемым режимом работы,и это не проблема. Однако, интерфейсы с высокой пропускной способностью могут получать тысячи пакетов в секунду. С такого рода нагрузкой по прерыванию может пострадать общая производительность системы.

Качестве одного из способов повышения эффективности Linux на высокопроизводительных системах, разработчики сетевой подсистемы создали альтернативный интерфейс (названный NAPI) (* NAPI означает "новый API"; исследователи сетей хороши при создании интерфейсов, а не их именовании.) на основе опроса. "Опрос" может быть грязным словом среди разработчиков драйверов, которые часто рассматривают техники опроса, как неэлегантные и неэффективные. Однако, опрос является неэффективным, только если интерфейс опрашивается, когда нет работы. Когда система имеет высокоскоростной интерфейс, обрабатывающий большой трафик, для обработки ***всегда*** есть много пакетов. В таких ситуациях нет необходимости прерывать процессор; достаточно часто забирать пакеты из интерфейса.

Остановка приёма прерываний может значительно разгрузить процессор. NAPI-совместимые драйверы может быть также сказано не передавать пакеты ядру, если эти пакеты просто отброшены в сетевом коде из-за перегрузки, что также может увеличить производительность, когда это требуется больше всего. По различным причинам также менее вероятно, что NAPI драйверам потребуется изменять порядок пакетов.

Однако, не все устройства могут работать в режиме NAPI. NAPI-совместимый интерфейс должен быть способен сохранять несколько пакетов (либо на самой карте, либо в памяти кольца DMA). Интерфейс должен быть способен отключить прерывания для принятых пакетов, продолжая генерировать прерывания для успешной передачи и других событий. Есть и другие тонкие вопросы, которые могут сделать написание NAPI-совместимого драйвера труднее; смотрите для подробностей ***Documentation/networking/NAPI_HOWTO.txt*** в дереве исходного кода ядра.

Сравнительно небольшое число драйверов реализуют NAPI интерфейс. Однако, если вы пишете драйвер для интерфейса, который может генерировать большое число прерываний, затраты времени на реализацию NAPI вполне могут оказаться полезными.

Драйвер ***snull***, когда он загружен с параметром **use_napi**, установленным в ненулевое значение, работает в режиме NAPI. Во время инициализации мы должны установить несколько дополнительных полей **struct net_device**:

```
if (use_napi) {
    dev->poll   = snull_poll;
    dev->weight = 2;
}
```

Поле **poll** должно быть указывать на функцию опроса вашего драйвера; в ближайшее время мы посмотри на ***snull_poll***. Поле **weight** описывает относительную важность этого интерфейса: насколько большой трафик должен быть принят с помощью интерфейса, когда ресурсов недостаточно. Нет строгих правил, как должен быть установлен параметр **weight**; по соглашению, 10 MBps Ethernet интерфейсы устанавливают **weight** в **16**, в то время как более быстрые интерфейсы используют **64**. Вы не должны устанавливать **weight** в значение большее, чем количество пакетов, которое может хранить ваш интерфейс. В ***snull*** мы установили **weight** в двойку в качестве способа продемонстрировать замедленный приём пакета.

Следующим шагом в создании NAPI-совместимого драйвера является изменение обработчика прерывания. Когда ваш интерфейс (который должен стартовать с разрешёнными прерываниями при приёме) сигнализирует о том, что пакет прибыл, обработчик прерывания не должен обрабатывать пакет. Наоборот, он должен отключить дальнейшие прерывания при

получении и сообщить ядру, что пришло время начать опрашивать этот интерфейс. В обработчике "прерывания" ***snull*** код, который отвечает на прерывания приёма пакетов, был изменён на следующий:

```
if (statusword & SNULL_RX_INTR) {
    snull_rx_ints(dev, 0); /* Запрещаем дальнейшие прерывания */
    netif_rx_schedule(dev);
}
```

Когда интерфейс сообщает нам, что пакет доступен, обработчик прерываний оставляет его в интерфейсе; всё, что должно произойти на данном этапе - это вызов ***netif_rx_schedule***, который распоряжается, чтобы наш метод ***poll*** был вызван в какой-то момент в будущем.

Метод ***poll*** имеет следующий прототип:

```
int (*poll)(struct net_device *dev, int *budget);
```

Реализация в ***snull*** метода ***poll*** выглядит следующим образом:

```
static int snull_poll(struct net_device *dev, int *budget)
{
    int npackets = 0, quota = min(dev->quota, *budget);
    struct sk_buff *skb;
    struct snull_priv *priv = netdev_priv(dev);
    struct snull_packet *pkt;

    while (npackets < quota && priv->rx_queue) {
        pkt = snull_dequeue_buf(dev);
        skb = dev_alloc_skb(pkt->datalen + 2);
        if (! skb) {
            if (printk_ratelimit( ))
                 printk(KERN_NOTICE "snull: packet dropped\n");
            priv->stats.rx_dropped++;
            snull_release_buffer(pkt);
            continue;
        }
        memcpy(skb_put(skb, pkt->datalen), pkt->data, pkt->datalen);
        skb->dev = dev;
        skb->protocol = eth_type_trans(skb, dev);
        skb->ip_summed = CHECKSUM_UNNECESSARY; /* не проверяем его */
        netif_receive_skb(skb);

        /* Ведение статистики */
        npackets++;
        priv->stats.rx_packets++;
        priv->stats.rx_bytes += pkt->datalen;
        snull_release_buffer(pkt);
    }
    /* Если мы обработали все пакеты, мы завершили; сообщаем ядру и снова
разрешаем прерывания */
    *budget -= npackets;
    dev->quota -= npackets;
    if (! priv->rx_queue) {
        netif_rx_complete(dev);
```

```
            snull_rx_ints(dev, 1);
            return 0;
        }
        /* Мы не смогли обработать всё. */
        return 1;
}
```

Центральная часть функция связана с созданием **skb**, удерживающего пакет; этот код такой же, что мы видели ранее в ***snull_rx***. Ряд вещей, однако, отличается:

- Параметр **budget** определяет максимальное количество пакетов, которое нам разрешено передавать в ядро. Внутри структуры устройства поле **quota** даёт другой максимум; метод ***poll*** должен соблюдать меньший из двух пределов. Следует также уменьшить и **dev->quota** и ***budget** на количество фактически полученных пакетов. Значение **budget** является максимальным количеством пакетов, которые текущий процессор может получать от всех интерфейсов, а **quota** является значением для каждого интерфейса, которое обычно начинается как **weight**, назначенный интерфейсу во время инициализации.
- Пакеты должны передаваться в ядро с помощью ***netif_receive_skb***, а не ***netif_rx***.
- Если метод ***poll*** в состоянии обработать все имеющиеся пакеты в пределах данных ему ограничений, он должна снова включить прерывания при приёме пакетов, вызвав ***netif_rx_complete*** для выключения опроса, и вернуть 0. Возвращаемое значение **1** указывает, что есть пакеты, которые еще предстоит обработать.

Сетевая подсистема гарантирует, что любой предоставленный устройством метод ***poll*** не будет вызываться одновременно на более, чем одном процессоре. Однако, вызов ***poll*** все же может произойти одновременно с вызовами других методов вашего устройства.

Изменение состояния соединения

Сетевые подключения, по определению, имеют дело с внешним миром за пределами локальной системы. Поэтому они часто подвержены внешним событиям и они могут быть временными обстоятельствами. Сетевая подсистема должна знать, когда сетевые соединения устанавливаются или пропадают, и она предоставляет несколько функций, которые драйвер может использовать, чтобы передать эту информацию.

Большинство сетевых технологий, связанных с настоящим, физическим соединением предоставляют состояние ***несущей*** (carrier); присутствии несущей означает, что оборудование присутствует и готово функционировать. Ethernet адаптеры, например, считывают несущий сигнал на проводе; когда пользователь спотыкается о кабель, эта несущая пропадает и связь пропадает. По умолчанию сетевыми устройствами предполагается, что несущий сигнал присутствует. Однако, драйвер может изменить это состояние явным образом с помощью этих функций:

```
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
```

Если ваш драйвер обнаруживает отсутствие несущей на одном из своих устройств, он должен вызвать ***netif_carrier_off***, чтобы проинформировать ядро об этом изменении. Когда несущая возвращается, должна быть вызвана ***netif_carrier_on***. Некоторые драйверы также вызывают ***netif_carrier_off*** при выполнении основных изменений конфигурации (таких, как тип носителя); как только адаптер закончил устанавливать себя в исходное состояние, новая

несущая обнаружена и трафик может возобновиться.

Также существует функция, возвращающая целочисленное значение:

```
int netif_carrier_ok(struct net_device *dev);
```

Она может быть использована для проверки текущего состояния несущей (как это отражено в структуре устройства).

Буферы сокетов

На данный момент мы охватили большинство вопросов, связанных с сетевыми интерфейсами. Но всё ещё отсутствует более подробное обсуждение структуры **sk_buff**. Эта структура находится в центре сетевой подсистемы ядра Linux и теперь мы представим как основные поля этой структуры, так и функции, используемые для работы с ними.

Хотя не существует строгой необходимости понимать внутренности **sk_buff**, умение посмотреть на её содержимое может быть полезным, когда вы ищете проблемы и когда вы пытаетесь оптимизировать свой код. Например, если вы посмотрите в ***loopback.c***, вы найдёте оптимизацию на основе знаний внутренностей **sk_buff**. Здесь применимо обычное предупреждение: если вы пишете код, который использует знание структуры **sk_buff**, вы должны быть готовы увидеть его поломку в будущих версиях ядра. Всё же, иногда преимущества производительности оправдывают дополнительные затраты на техническую поддержку.

Мы не будем описывать здесь всю структуру, только те поля, которые могут быть использованы в пределах драйвера. Если вы хотите увидеть больше, вы можете посмотреть на ***<linux/skbuff.h>***, где определяется эта структура и прототипы функций. Дополнительная информация о том, как использовать эти поля и функции может быть легко получена поиском в исходных текстах ядра.

Важные поля

Здесь представлены те поля, к которым драйверу мог бы потребоваться доступ. Они перечислены без какого-либо порядка.

struct net_device *dev;
Устройство принимает или отправляет этот буфер.

union { /* ... */ } h;
union { /* ... */ } nh;
union { /* ... */ } mac;
Указатели на заголовки различных уровней, содержащиеся в пакете. Каждое поле объединения является указателем на другой тип структуры данных. **h** содержит указатели на заголовки транспортного уровня (например, **struct tcphdr *th**); **nh** включает в себя заголовки сетевого уровня (такие, как **struct iphdr *iph)** и **mac** собирает указатели заголовков канального уровня (такие, как **struct ethdr *ethernet**).
Если вашему драйверу необходимо посмотреть на адреса источника и назначения TCP пакета, он может найти их в **skb->h.th**. Смотрите файл заголовка для полного набора заголовочных типов, которые могут быть доступны таким образом.
Отметим, что сетевые драйверы несут ответственность за установку указателя **mac** для

входящих пакетов. Этой задачей обычно занимается ***eth_type_trans***, но не-Ethernet драйверы вынуждены устанавливать **skb->mac.raw** напрямую, как показано в разделе "Не-Ethernet заголовки".

unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;

Указатели, используемые для адресации данных в пакете. **head** указывает на начало выделенного пространства, **data** является началом достоверных октетов (и обычно немного больше, чем **head**), **tail** является окончанием достоверных октетов и **end** указывает на максимальный адрес, который может достичь **tail**. Ещё один способ посмотреть на это заключается в том, что доступным пространством буфера является **skb->end - skb->head**, а в настоящее время используемым пространством данных является **skb->tail - skb->data**.

unsigned int len;
unsigned int data_len;

len является полной длиной данных в пакете, в то время как **data_len** является длиной части пакета, хранящейся в отдельных фрагментах. Поле **data_len** равно 0, если используется ввод/вывод с разборкой/сборкой.

unsigned char ip_summed;

Политика подсчёта контрольной суммы для этого пакета. Поле устанавливается драйвером для входящих пакетов, как описано в разделе "Приём пакетов" 501.

unsigned char pkt_type;

Классификация пакета, используемая при его доставке. Драйвер несёт ответственность за установку его в **PACKET_HOST** (этот пакет для меня), **PACKET_OTHERHOST** (нет, этот пакет не для меня), **PACKET_BROADCAST** или **PACKET_MULTICAST**. Ethernet драйверы не изменяют явно **pkt_type**, поскольку это делает за них ***eth_type_trans***.

shinfo(struct sk_buff *skb);
unsigned int shinfo(skb)->nr_frags;
skb_frag_t shinfo(skb)->frags;

По причинам производительности, некоторые информация **skb** хранится в отдельной структуре, которая появляется в памяти сразу после **skb**. К этой "общей информации" (называется так потому, что может использоваться совместно копиями **skb** в рамках сетевого кода) необходимо обращаться через макрос ***shinfo***. В этой структуре находятся несколько полей, но большинство из них выходят за рамки этой книги. В разделе "Ввод/вывод с разборкой/сборкой" 500 мы видели **nr_frags** и **frags**.

Остальные поля структуры не особенно интересны. Они используются для ведения списка буферов, учёта памяти, принадлежащей сокету, который владеет буфером, и так далее.

Функции, работающие с буферами сокетов

Сетевые устройства, которые используют структуру **sk_buff**, работают с ней с помощью официальных интерфейсных функций. С буферами сокетов работает много функций; здесь самые интересные из них:

struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);

Выделение буфера. Функция ***alloc_skb*** выделяет буфер и инициализирует **skb->data** и **skb->tail** в **skb->head**. Функция ***dev_alloc_skb*** является ярлыком, который вызывает ***alloc_skb*** с приоритетом **GFP_ATOMIC** и резервирует некоторое пространство между **skb->head** и **skb->data**. Эти пространство данных используется для оптимизаций внутри сетевого уровня и не должно быть затронуто драйвером.

void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);

Освобождение буфера. Вызов ***kfree_skb*** используется внутри ядра. Драйвер должен вместо этого использовать одну из форм ***dev_kfree_skb***: ***dev_kfree_skb*** для контекста без прерывания, ***dev_kfree_skb_irq*** для контекста прерывания или ***dev_kfree_skb_any*** для кода, который может работать в любом контексте.

unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);

Обновление полей **tail** и **len** структуры **sk_buff**; они используются для добавления данных в конец буфера. Возвращаемым значением каждой функции является предыдущее значение **skb->tail** (иными словами, оно указывает на только что созданное пространство данных). Драйверы могут использовать возвращаемое значение, чтобы скопировать данные, вызывая **memcpy skb_put(...), data, len)** или другой эквивалент. Разница между этими двумя функциями в том, что ***skb_put*** выполняет проверку, чтобы убедиться, что данные помещаются в буфер, а ***__skb_put*** проверку опускает.

unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);

Функции для уменьшения **skb->data** и увеличения **skb->len**. Они похожи на ***skb_put***, за исключением того, что данные добавляются в начало пакета, а не в конец. Возвращаемое значение указывает на только что созданное пространство данных. Функции используются для добавления аппаратного заголовка перед передачей пакета. Вновь, ***__skb_push*** отличается тем, что она не проверяет имеющееся пространство на достаточность.

int skb_tailroom(struct sk_buff *skb);

Возвращает размер пространства, доступного для размещения данных в буфере. Если драйвер помещает в буфер больше данных, чем он может содержать, система паникует. Хотя вы можете возразить, что будет достаточно ***printk***, чтобы отметить ошибку, повреждение памяти настолько вредно для системы, что разработчики решили предпринять решительные действия. На практике вам не требуется доступное место, если буфер был выделен корректно. Поскольку драйверы обычно получают размер пакета до выделения буфера, только драйвер с серьёзной ошибкой помещает в буфер слишком много данных и паника может рассматриваться как должное наказание.

int skb_headroom(struct sk_buff *skb);

Возвращает количество пространства, доступного перед данными, то есть, сколько октетов можно "затолкать" в буфер.

void skb_reserve(struct sk_buff *skb, int len);
Увеличивает как **data**, так и **tail**. Функция может быть использована для резервирования запаса до заполнения буфера. Большинство интерфейсов Ethernet резервируют два байта перед пакетом; таким образом, заголовок IP выравнивается по 16-ти байтной границе, после 14-го байта Ethernet заголовка. Так же это делает и ***snull***, хотя команда не была показана в разделе "Приём пакетов" 501, чтобы избежать введения дополнительных концепций в том месте.

unsigned char *skb_pull(struct sk_buff *skb, int len);
Удаляет данные из головы пакета. Драйвер нет необходимости использовать эту функцию, но она включена здесь для полноты. Она уменьшает s**kb->len** и увеличивает **skb->data**; это способ отделения аппаратного заголовка (Ethernet или эквивалентного) от начала входящих пакетов.

int skb_is_nonlinear(struct sk_buff *skb);
Возвращает истинное значение, если этот **skb** разделён на несколько фрагментов для ввода/вывода с разборкой/сборкой.

int skb_headlen(struct sk_buff *skb);
Возвращает длину первого сегмента **skb** (ту часть, на которую указывает **skb->data**).

void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
Если вы должны обратиться напрямую к фрагментам в нелинейном **skb** изнутри ядра, то эти функции отображают и отключают их отображение для вас. Используется атомарная kmap, поэтому вы не можете иметь одновременно более одного отображённого фрагмента.

Ядро определяет несколько других функций, которые работают с буферами сокетов, но они предназначен для использования верхними уровнями сетевого кода и драйвер в них не нуждается.

Разрешение MAC адреса

Интересным вопросом в Ethernet коммуникации является то, как связать MAC адрес (уникальный ID аппаратного интерфейса) с IP адресом. Большинство протоколов имеют аналогичную проблему, но мы сосредоточимся на здесь на случае, подобном Ethernet. Мы постараемся предложить полное описание вопроса, поэтому мы покажем три ситуации: ARP, Ethernet заголовки без ARP (такие, как у ***plip***) и не-Ethernet заголовки.

Использование ARP с Ethernet

Обычный способ иметь дело с разрешением адреса заключается в использовании протокола разрешения адреса (Address Resolution Protocol, ARP). К счастью, ARP управляется ядром и интерфейсу Ethernet нет необходимости делать ничего особенного для поддержки ARP. Пока **dev->addr** и **dev->addr_len** установлены корректно во время открытия, драйверу не требуется беспокоиться о разрешении IP адресов в MAC адреса; необходимые методы устройства для **dev->hard_header** и **dev->rebuild_header** назначает ***ether_setup***.

Хотя ядро нормально обрабатывает детали разрешения адреса (и кэширование

результатов), оно призывает интерфейс драйвера помочь в создании пакетов. В конце концов, драйвер знает о деталях заголовка физического уровня, в то время как авторы сетевого кода пытались оградить остальную часть ядра от такого знания. С этой целью ядро вызывает метод драйвера ***hard_header***, чтобы выложить пакет с результатами запроса ARP. Как правило, авторам Ethernet драйверов нет необходимости знать об этом процессе - обо всем позаботится общий код Ethernet.

Подмена ARP

Простые сетевые интерфейсы точка-точка, такие как ***plip***, могли бы извлечь выгоду от использования Ethernet заголовков, избегая накладных расходов на отправку ARP пакетов взад и вперёд. Код примера в ***snull*** также относится к этому классу сетевых устройств. ***snull*** не может использовать ARP, поскольку драйвер изменяет IP адреса в передаваемых пакетах, а ARP пакеты также обмениваются IP адресами. Хотя мы могли бы с небольшими проблемами реализовать простой генератор ответов ARP, более наглядно показать, как непосредственно обрабатывать заголовки физического уровня.

Если ваше устройство хочет использовать обычные аппаратные заголовки без работающего ARP, вы необходимо переопределить метод по умолчанию для **dev->hard_header**. Вот как это реализует ***snull***, очень короткая функция:

```
int snull_header(struct sk_buff *skb, struct net_device *dev,
                 unsigned short type, void *daddr, void *saddr,
                 unsigned int len)
{
    struct ethhdr *eth = (struct ethhdr *)skb_push(skb,ETH_HLEN);

    eth->h_proto = htons(type);
    memcpy(eth->h_source, saddr ? saddr : dev->dev_addr, dev->addr_len);
    memcpy(eth->h_dest, daddr ? daddr : dev->dev_addr, dev->addr_len);
    eth->h_dest[ETH_ALEN-1] ^= 0x01; /* назначение - это мы xor 1 */
    return (dev->hard_header_len);
}
```

Функция просто принимает информацию, предоставленную ядром и форматирует её в стандартный заголовок Ethernet. Она также переключает бит в адресе назначения Ethernet по причинам, описанным позже.

Когда интерфейсом принят пакет, аппаратный заголовок используется парой способов функцией ***eth_type_trans***. Мы уже видели этот вызов в ***snull_rx***:

```
skb->protocol = eth_type_trans(skb, dev);
```

Функция извлекает из Ethernet заголовка идентификатор протокола (в данном случае, **ETH_P_IP**), она также задаёт **skb->mac.raw**, удаляет аппаратный заголовок из данных пакета (с помощью ***skb_pull***) и устанавливает **skb->pkt_type**. Это последнее поле при создании **skb** по умолчанию установлено в **PACKET_HOST** (которое указывает, что пакет направлен в этот компьютер) и ***eth_type_trans*** изменяет его, чтобы изобразить адрес назначения Ethernet: если этот адрес не совпадает с адресом интерфейса, который его получил, поле **pkt_type** установлено в **PACKET_OTHERHOST**. Впоследствии, если интерфейс находится в режиме неразборчивости (promiscuous mode) или в ядре разрешена пересылка пакетов, ***netif_rx*** отбрасывает любой пакет типа **PACKET_OTHERHOST**. По этой причине,

snull_header делает всё возможное, чтобы сделать аппаратный адрес назначения соответствующим "принимающему" интерфейсу.

Если ваш интерфейс является соединением точка-точка, вы бы не хотели получать неожиданные многоадресные пакеты. Чтобы избежать этой проблемы, помните, что адрес назначения, который в первом октете имеет в младшем бите (LSB) 0, относится к одному сетевому устройству (то есть или **PACKET_HOST** или **PACKET_OTHERHOST**). Драйвер ***plip*** использует в качестве первого октета своего аппаратного адреса **0xfc**, а ***snull*** использует **0x00**. Оба адреса приводят к работающему соединению точка-точка, подобному Ethernet.

Не-Ethernet заголовки

Мы только что видели, что аппаратный заголовок содержит некоторую информацию, в дополнение к адресу назначения, наиболее важной из которой является коммуникационный протокол. Теперь мы опишем, как аппаратные заголовки могут быть использованы для инкапсуляции соответствующей информации. Если вам потребуется узнать детали, вы сможете извлечь их из исходных текстов ядра или технической документации для конкретной среды передачи. Большинство авторов драйверов могут проигнорировать это обсуждение и использовать только Ethernet реализацию.

Стоит отметить, что не вся информация будет предоставлена каждым протоколом. Соединение точка-точка, такое как ***plip*** или ***snull***, могло бы избежать передачи целого заголовка Ethernet без потери общности. Метод устройства ***hard_header***, показанный ранее как реализованный функцией ***snull_header***, получает информацию по доставке, на уровне протокола и аппаратных адресов, от ядра. Он также получает 16-ти разрядный номер протокола аргументе **type**; IP, например, идентифицируется как **ETH_P_IP**. Драйвер ожидает правильность доставки данных пакета и номера протокола к принимающему устройству. Соединение точка-точка может пропустить адреса своего аппаратного заголовка, передавая только номер протокола, так как доставка гарантируется независимо от адресов источника и назначения. Соединение только на основе IP могло бы даже совсем избежать передачи всех аппаратных заголовков.

Если пакет принят на другом конце соединения, принимающая функция в драйвере должно правильно установить поля **skb->protocol**, **skb->pkt_type** и **skb->mac.raw**.

skb->mac.raw является символьным указателем, используемым механизмом разрешения адреса, реализованном на верхних уровнях сетевого кода (например, ***net/ipv4/arp.c***). Он должен указывать на адрес машины, который соответствует **dev->type**. Возможные значения типа устройства определены в ***<linux/if_arp.h>***; Ethernet интерфейсы используют **ARPHRD_ETHER**. Например, вот как ***eth_type_trans*** имеет дело с заголовком Ethernet полученных пакетов:

```
skb->mac.raw = skb->data;
skb_pull(skb, dev->hard_header_len);
```

В простейшем случае (соединение точка-точка без каких-либо заголовков), **skb->mac.raw** может указывать на статический буфер, содержащий аппаратный адрес этого интерфейса, **protocol** может быть установлен в **ETH_P_IP** и **packet_type** может быть оставлен в его значении по умолчанию **PACKET_HOST**.

Поскольку каждый тип оборудования уникален, трудно дать более конкретные советы, чем

уже приведённые. Однако, ядро полно примеров. Смотрите, например, драйвер AppleTalk (***drivers/net/appletalk/cops.c***), драйверы для инфракрасной связи (такие как ***drivers/net/irda/smc_ircc.c***), или PPP драйвер (***drivers/net/ppp_generic.c***).

Дополнительные команды ioctl

Мы уже видели, что для сокетов реализован системный вызов ***ioctl***; **SIOCSIFADDR** и **SIOCSIFMAP** являются примерами "ioctl для сокетов". Теперь давайте посмотрим, как сетевым кодом используется третий аргумент системных вызовов.

Когда для сокета выполняется системный вызов ***ioctl***, номер команды является одним из символов, определенных в ***<linux/sockios.h>*** и функция ***sock_ioctl*** вызывает непосредственно зависимую от протокола функцию (где "протокол" относится к основным используемым сетевым протоколам, например, IP или AppleTalk).

Любая команда ***ioctl***, которая не распознана уровнем протокола, передаётся на уровень устройства. Эти относящиеся к устройству команды ioctl принимают третий аргумент от пользовательского пространства, **struct ifreq *** . Эта структура определена в ***<linux/if.h>***. Команды **SIOCSIFADDR** и **SIOCSIFMAP** работают фактически со структурой **ifreq**. Дополнительный аргумент для **SIOCSIFMAP**, хотя и определяется как **ifmap**, является лишь полем **ifreq**.

В дополнение к использованию стандартизированных вызовов, каждый интерфейс может определить свои собственные команды ***ioctl***. Интерфейс ***plip***, например, позволяет через ***ioctl*** изменять свои внутренние значения таймаутов. Реализация ***ioctl*** для сокетов распознаёт 16 команд, как предназначенные для интерфейса: от **SIOCDEVPRIVATE** до **SIOCDEVPRIVATE +15**. (* Обратите внимание, что согласно <*linux/sockios.h*> такие SIOCDEVPRIVATE команды являются устаревшими. Чем их следует заменить, однако, не ясно и многочисленные драйверы в дереве исходных кодов всё ещё их используют.)

Когда одна из этих команд распознана, в соответствующем драйвере интерфейса вызывается **dev->do_ioctl**. Функция получает тот же указатель **struct ifreq ***, который использует функция ***ioctl*** общего назначения:

```
int (*do_ioctl)(struct net_device *dev, struct ifreq *ifr, int cmd);
```

Указатель **ifr** сообщает пространству ядра адрес, который хранит копию структуры, переданной пользователем. После возвращения ***do_ioctl***, структура копируется обратно в пространство пользователя; таким образом, драйвер может использовать свои команды для получения и возвращения данных. Зависимые от устройства команды могут захотеть использовать поля в **struct ifreq**, но они уже содержат стандартизированные значения и маловероятно, что драйвер сможет адаптировать эту структуру под свои потребности. Поле **ifr_data** является объектом **caddr_t** (указателем), который предназначен для использования под потребности данного устройства. Драйвер и программа, используемая для вызова команд ***ioctl***, должны договориться об использовании **ifr_data**. Например, ***pppstats*** использует зависящие от устройства команды для получения информации от драйвера интерфейса ***ppp***.

Не стоит здесь показывать реализацию ***do_ioctl***, но с информацией этой главы и примерами ядра вы должны суметь её написать, когда вам это потребуется. Однако, следует отметить, что реализация в ***plip*** использует **ifr_data** некорректно и её не следует использовать в качестве примера для реализации ***ioctl***.

Статистическая информация

Последним необходимым методом драйвера является ***get_stats***. Этот метод возвращает указатель на статистику для данного устройства. Его реализация довольно проста; показанный метод работает даже когда несколько интерфейсов управляются одним и тем же драйвером, потому что статистика размещена внутри структуры данных устройства.

```
struct net_device_stats *snull_stats(struct net_device *dev)
{
    struct snull_priv *priv = netdev_priv(dev);
    return &priv->stats;
}
```

Реальная работа, необходимая для получения достоверной статистики, распределена по всему драйверу, где обновляются различные поля. Ниже приведён список наиболее интересных полей в **struct net_device_stats**:

unsigned long rx_packets;
unsigned long tx_packets;
Общее количество входящих и исходящих пакетов, успешно переданных интерфейсом.

unsigned long rx_bytes;
unsigned long tx_bytes;
Количество байтов, полученных и переданных интерфейсом.

unsigned long rx_errors;
unsigned long tx_errors;
Количество ошибочных приёмов и передач. Нет конца тому, что может пойти не так при передаче пакета и структура **net_device_stats** включает в себя шесть счётчиков для специфичных ошибок приёма и пяти ошибок для передачи. Для полного списка смотрите ***<linux/netdevice.h>***. Если возможно, ваш драйвер должен вести подробную статистику ошибок, потому что они могут быть наиболее полезны для системных администраторов, пытающихся обнаружить проблему.

unsigned long rx_dropped;
unsigned long tx_dropped;
Количество пакетов, отброшенных во время приёма и передачи. Пакеты отбрасываются, если нет памяти, необходимой для данных пакета. **tx_dropped** используется редко.

unsigned long collisions;
Число коллизий из-за перегрузки среды передачи.

unsigned long multicast;
Количество принятых многоадресных пакетов.

Стоит повторить, что метод ***get_stats*** может быть вызван в любое время, даже когда интерфейс выключен, поэтому драйвер должен хранить статистическую информацию так долго, как существует структура **net_device**.

Многоадресность

Многоадресный (multicast) пакет - это сетевой пакет предназначенный для получения более чем одним сетевым устройством, но не всеми. Эта функциональность получена присвоением специальных аппаратных адресов группам сетевых устройств. Пакеты, направляемые на один из специальных адресов должны быть получены всеми устройствами такой группы. В случае Ethernet, групповой адрес имеет установленный младший значащий бит первого октета адреса в адресе назначения, тогда как все платы устройств в своём аппаратном адресе имеют этот бит очищенным.

Хитрая часть общения с группами устройств и аппаратными адресами выполняется приложениями и ядром, и драйверу интерфейса не придётся иметь дело с этими проблемами.

Передача многоадресных пакетов является простой задачей, поскольку они выглядят так же, как любые другие пакеты. Интерфейс передаёт их через среду коммуникации не глядя на адрес назначения. Это ядро должно назначить корректный аппаратный адрес назначения; методу устройства ***hard_header***, если определён, не требуется смотреть в данные, им скомпонованные.

Ядро выполняет работу по отслеживанию групповых адресов, представляющих интерес в любой момент времени. Список может часто меняться, так как это является функцией приложений, которые работают в любое время, и пользовательского интереса. Работой драйвера является принять список интересующих групповых адресов и доставлять в ядро любые пакеты, посланные на эти адреса. Как драйвер реализует многоадресный список несколько зависит от того, как работает используемое оборудование. Как правило, оборудование принадлежит к одному из трёх классов в том, что касается многоадресности:

- Интерфейсы, которые не могут иметь дело с многоадресной рассылкой. Эти интерфейсы либо получают пакеты, направленные только на их аппаратный адрес (плюс широковещательные (broadcast) пакеты) или получать каждый пакет. Они могут получать многоадресные пакеты только принимая каждый пакет, таким образом, потенциально перегружая операционную систему огромным числом "неинтересных" пакетов. Обычно такие интерфейсы не считаются совместимыми с многоадресностью и драйвер не будет устанавливать **IFF_MULTICAST** в **dev->flags**. Интерфейсы точка-точка представляют собой особый случай, потому что они всегда принимают каждый пакет без выполнения какой-либо аппаратной фильтрации.
- Интерфейсы, которые могут отделять многоадресные пакеты от других пакетов (компьютер-компьютер или широковещательного). Эти интерфейсы могут быть проинструктированы для получения каждого многоадресного пакета и дать программному обеспечению определить, интересен ли адрес для данного сетевого устройства. Накладные расходы, добавляемые в данном случае, являются приемлемыми, поскольку число многоадресных пакетов в типичной сети является небольшим.
- Интерфейсы, которые могут выполнять аппаратное обнаружение групповых адресов. Этим интерфейсам может быть передан список групповых адресов для которых пакеты должны быть получены и они игнорируют другие многоадресные пакеты. Это оптимальный случай для ядра, потому что оно не тратит время процессора на отбрасывание "неинтересных" пакетов, полученных интерфейсом.

Ядро пытается использовать возможности высокоуровневых интерфейсов, поддерживая третий класс устройств, который наиболее универсален и наилучший. Таким образом, ядро уведомляет драйвер при каждом изменении в списке действительных групповых адресов и

передаёт новый список в драйвер, чтобы он мог обновить аппаратные фильтры в соответствии с новой информацией.

Поддержка многоадресности в ядре

Поддержка многоадресных пакетов состоит из нескольких элементов: метод устройства, структура данных и флаги устройства:

void (*dev->set_multicast_list)(struct net_device *dev);
Метод устройства, вызываемый всякий раз, когда список машинных адресов, связанных с устройством, изменяется. Он также вызывается при изменении **dev->flags**, поскольку некоторые флаги (например, **IFF_PROMISC**), также могут требовать, чтобы вы перепрограммировали аппаратный фильтр. Метод получает указатель на **struct net_device** в качестве аргумента и возвращает **void**. Драйвер, не заинтересованный в реализации этого метода, может оставить это поле установленным в NULL.

struct dev_mc_list *dev->mc_list;
Связный список всех групповых адресов, ассоциированный с этим устройством. Фактическое определение этой структуры приведено в конце этого раздела.

int dev->mc_count;
Количество объектов в связном списке. Эта информация несколько излишняя, но проверка **mc_count** на 0 является полезной для проверки списка.

IFF_MULTICAST
Пока драйвер не установит этот флаг в **dev->flags**, интерфейсу не будет предложено обрабатывать многоадресные пакеты. Тем не менее, ядро вызывает метод драйвера ***set_multicast_list***, когда **dev->flags** изменяется, потому что многоадресный список может быть изменён, пока интерфейс был неактивен.

IFF_ALLMULTI
Флаг устанавливается в **dev->flags** сетевым программным обеспечением, чтобы сообщить драйверу получать из сети все многоадресные пакеты. Это происходит при включении многоадресной маршрутизации. Если этот флаг установлен, **dev->mc_list** не должен использоваться для фильтрации многоадресных пакетов.

IFF_PROMISC
Флаг устанавливается в **dev->flags**, когда интерфейс переключен неразборчивый режим (promiscuous mode). Интерфейсом должен быть получен каждый пакет, независимо от **dev->mc_list**.

Последним кусочком информации, необходимой разработчику драйвера, является определение **struct dev_mc_list**, которое находится в ***<linux/netdevice.h>***:

```
struct dev_mc_list {
    struct        dev_mc_list *next;       /* Следующий адрес в списке */
    __u8          dmi_addr[MAX_ADDR_LEN]; /* Аппаратный адрес */
    unsigned char dmi_addrlen;             /* Длина адреса */
    int           dmi_users;               /* Число пользователей */
    int           dmi_gusers;              /* Число групп */
};
```

Поскольку групповые и аппаратные адреса не зависят от фактической передачи пакетов, эта структура является переносимой между сетевыми реализациями и каждый адрес определяется строкой октетов и длиной, как и в **dev->dev_addr**.

Типичная реализация

Лучшим способом описать разработку ***set_multicast_list*** является показать вам некоторый псевдокод.

Следующая функция представляет собой типичной реализацией функции в полнофункциональном (full-featured, ff) драйвере. Драйвер полнофункциональный в том, что он управляет интерфейсом, имеющим сложный аппаратный пакетный фильтр, который может содержать таблицу групповых адресов, которые будут приниматься этим устройством. Максимальным размером таблицы является **FF_TABLE_SIZE**.

Все функции с префиксом **ff_** являются местами для размещения аппаратно-зависимых операций:

```
void ff_set_multicast_list(struct net_device *dev)
{
    struct dev_mc_list *mcptr;

    if (dev->flags & IFF_PROMISC) {
        ff_get_all_packets( );
        return;
    }
    /* Если есть больше адресов, чем мы обрабатываем, получаем все
    многоадресные пакеты и сортируем затем их в программе. */
    if (dev->flags & IFF_ALLMULTI || dev->mc_count > FF_TABLE_SIZE) {
        ff_get_all_multicast_packets( );
        return;
    }
    /* Нет многоадресного списка? Просто выполняем свою работу */
    if (dev->mc_count == 0) {
        ff_get_only_own_packets( );
        return;
    }
    /* Запоминаем все групповые адреса в аппаратном фильтре */
    ff_clear_mc_list( );
    for (mc_ptr = dev->mc_list; mc_ptr; mc_ptr = mc_ptr->next)
        ff_store_mc_address(mc_ptr->dmi_addr);
    ff_get_packets_in_multicast_list( );
}
```

Эта реализация может быть упрощена, если интерфейс не может хранить многоадресную таблицу для входящих пакетов в аппаратном фильтре. В этом случае, **FF_TABLE_SIZE** уменьшается до 0 и последние четыре строки кода не требуются.

Как уже говорилось ранее, даже интерфейсам, которые не могут иметь дело с многоадресными пакетами, необходимо реализовать метод ***set_multicast_list*** для получения уведомлений об изменениях в **dev->flags**. Этот подход можно было бы назвать "неполной" (nonfeatured, nf) реализацией. Такая реализация очень проста, как показывает

следующий код:

```
void nf_set_multicast_list(struct net_device *dev)
{
    if (dev->flags & IFF_PROMISC)
        nf_get_all_packets( );
    else
        nf_get_only_own_packets( );
}
```

Реализация **IFF_PROMISC** имеет важное значение, поскольку в противном случае пользователи не смогут запустить ***tcpdump*** или любые другие сетевые анализаторы. С другой стороны, если интерфейс работает с соединением точка-точка, нет необходимости вовсе реализовывать ***set_multicast_list***, поскольку пользователи в любом случае получают каждый пакет.

Несколько других подробностей

Этот раздел охватывает несколько других вопросов, которые могут представлять интерес для авторов сетевых драйверов. В каждом случае мы просто стараемся направить вас в правильном направлении. Получение полной картины этого вопроса, вероятно, потребует затрат времени для рытья в исходниках ядра.

Поддержка интерфейса, не зависящего от среды передачи

Интерфейс, не зависящий от среды передачи (Media Independent Interface, или MII), является стандартом IEEE 802.3, описывающим, как трансивер Ethernet может взаимодействовать с сетевыми контроллерами; этому интерфейсу соответствуют многие продукты на рынке. Если вы пишете драйвер для MII-совместимого контроллера, ядро экспортирует универсальный уровень поддержки MII, который может сделать вашу жизнь легче.

Чтобы использовать универсальный уровень MII, вам следует подключить ***<linux/mii.h>***. Вам необходимо заполнить структуру **mii_if_info** информацией о физическом идентификаторе трансивера, поддерживается ли на самом деле полный дуплекс и так далее. Также для структуры **mii_if_info** требуются два метода:

```
int (*mdio_read) (struct net_device *dev, int phy_id, int location);
void (*mdio_write) (struct net_device *dev, int phy_id, int location, int
val);
```

Как и следовало ожидать, эти методы должны реализовывать связь с вашим специфичным интерфейсом MII.

Универсальный код MII обеспечивает набор функций для запроса и изменения режима режим работы трансивера, многие из них предназначены для работы с утилитой ***ethtool*** (описанной в следующем разделе). Для подробности посмотрите ***<linux/mii.h>*** и ***drivers/net/mii.c***.

Поддержка Ethtool

Ethtool является утилитой, разработанной, чтобы дать системным администраторам больше возможностей контроля по управлению сетевыми интерфейсами. С помощью ***ethtool***

можно управлять различными параметрами интерфейса, включая скорость, тип носителя, дуплексная работа, установка кольца DMA, аппаратный подсчёт контрольной суммы, операция пробуждения по сети и так далее, но только если ***ethtool*** поддерживается драйвером. ***Ethtool*** может быть загружена с http://sf.net/projects/gkernel/.

Соответствующие декларации поддержки ***ethtool*** могут быть найдены в ***<linux/ethtool.h>***. Основой является структура типа **ethtool_ops**, которая содержит полные 24 различных метода для поддержки ***ethtool***. Большинство из этих методов являются относительно простыми; для подробностей смотрите ***<linux/ethtool.h>***. Если ваш драйвер использует уровень MII, можно использовать ***mii_ethtool_gset*** и ***mii_ethtool_sset***, чтобы реализовать методы ***get_settings*** и ***set_settings***, соответственно.

Чтобы ***ethtool*** работала с вашим устройством, необходимо поместить указатель на вашу структуру **ethtool_ops** в структуру **net_device**. Для этой цели следует использовать макрос **SET_ETHTOOL_OPS** (определённый ***<linux/netdevice.h>***). Обратите внимание, что ваши методы для ***ethtool*** могут быть вызваны даже когда интерфейс выключен.

Netpoll

"Netpoll" является сравнительно поздним (2.6.5) дополнением к сетевому стеку; его целью является позволить ядру посылать и получать пакеты в тех случаях, когда полная сетевая подсистема и подсистема ввода/вывода могут быть недоступны. Он используется для функций, подобных удаленным сетевым консолям и удаленной отладки ядра. Поддержка ***netpoll*** в драйвере в любом случае не является необходимой, но это может сделать ваше устройство более полезным в некоторых ситуациях. В большинстве случаев поддержка ***netpoll*** также является относительно простой.

Драйверы, поддерживающие ***netpoll***, должны реализовать метод ***poll_controller***. Его задача состоит сохранять всё, что может происходить на контроллере в отсутствие прерываний от устройства. Почти все методы ***poll_controller*** принимают следующую форму:

```
void my_poll_controller(struct net_device *dev)
{
    disable_device_interrupts(dev);
    call_interrupt_handler(dev->irq, dev, NULL);
    reenable_device_interrupts(dev);
}
```

Метод ***poll_controller***, по сути, является просто моделированием прерываний от данного устройства.

Краткая справка

Этот раздел содержит ссылки на понятия, введённые в этой главе. Он также разъясняет роль каждого заголовочного файла, который должен подключать драйвер. Списки полей в структурах **net_device** и **sk_buff**, однако, здесь не повторяются.

#include <linux/netdevice.h>
: Заголовок, который содержит определения **struct net_device** и **struct net_device_stats** и включает в себя несколько других заголовков, которые необходимы сетевым драйверам.

struct net_device *alloc_netdev(int sizeof_priv, char *name, void (*setup)(struct

net_device *);
struct net_device *alloc_etherdev(int sizeof_priv);
void free_netdev(struct net_device *dev);
Функции для создания и освобождения структур **net_device**.
int register_netdev(struct net_device *dev);
void unregister_netdev(struct net_device *dev);
Регистрация и отмена регистрации сетевого устройства.
void *netdev_priv(struct net_device *dev);
Функция, которая возвращает указатель на используемую драйвером область структуры сетевого устройства.
struct net_device_stats;
Структура, которая содержит статистику устройства.
netif_start_queue(struct net_device *dev);
netif_stop_queue(struct net_device *dev);
netif_wake_queue(struct net_device *dev);
Функции, контролирующие прохождение пакетов в драйвере для передачи. Пакеты не передаются, пока не была вызвана ***netif_start_queue***. ***netif_stop_queue*** приостанавливает передачу и ***netif_wake_queue*** перезапускает очередь и подталкивает сетевой уровень для возобновления передачи пакетов.
skb_shinfo(struct sk_buff *skb);
Макрос, который обеспечивает доступ к части "общей информации" буфера пакета.
void netif_rx(struct sk_buff *skb);
Функция, которая может быть вызвана (в том числе во время прерывания), чтобы сообщить ядро о получении пакета и включении его в буфер сокета.
void netif_rx_schedule(dev);
Функция, которая сообщает ядру, что пакеты доступны и что надо начать опрос интерфейса; она используется только NAPI-совместимыми драйверами.
int netif_receive_skb(struct sk_buff *skb);
void netif_rx_complete(struct net_device *dev);
Функции, которые должны использоваться только NAPI-совместимыми драйверами. ***netif_receive_skb*** является NAPI эквивалентом для ***netif_rx***; она передаёт пакеты ядру. Когда NAPI-совместимый драйвер исчерпал поставку полученных пакетов, он должен снова включить прерывания и вызвать ***netif_rx_complete*** для остановки опроса.

#include <linux/if.h>
Подключается через ***netdevice.h***, этот файл объявляет интерфейсные флаги (макросы **IFF_**) и **struct ifmap**, которая играет важную роль в реализации ioctl для сетевых драйверов.
void netif_carrier_off(struct net_device *dev);
void netif_carrier_on(struct net_device *dev);
int netif_carrier_ok(struct net_device *dev);
Две первые из этих функций могут быть использованы, чтобы сообщить ядру, присутствует ли в настоящее время на данном интерфейсе несущий сигнал. ***netif_carrier_ok*** проверяет состояние несущей, как это отражено в структуре устройства.

#include <linux/if_ether.h>
ETH_ALEN
ETH_P_IP
struct ethhdr;
Подключаются через ***netdevice.h***, ***if_ether.h*** определяет все макросы **ETH_**, которые

используются для представления длин октетов (таких как длина адреса) и сетевых протоколов (таких как IP). Она также определяет структуру **ethhdr**.

#include <linux/skbuff.h>
Описание **struct sk_buff** и связанных с ней структур, а также нескольких встраиваемых функций для работы с буферами. Этот заголовок подключается через ***netdevice.h***.

struct sk_buff *alloc_skb(unsigned int len, int priority);
struct sk_buff *dev_alloc_skb(unsigned int len);
void kfree_skb(struct sk_buff *skb);
void dev_kfree_skb(struct sk_buff *skb);
void dev_kfree_skb_irq(struct sk_buff *skb);
void dev_kfree_skb_any(struct sk_buff *skb);
Функции, которые занимаются созданием и освобождением буферов сокетов. Драйверы обычно используют варианты **dev_**, которые предназначены для такой цели.

unsigned char *skb_put(struct sk_buff *skb, int len);
unsigned char *__skb_put(struct sk_buff *skb, int len);
unsigned char *skb_push(struct sk_buff *skb, int len);
unsigned char *__skb_push(struct sk_buff *skb, int len);
Функции, которые добавляют данные в **skb**; ***skb_put*** помещает данные в конец **skb**, а ***skb_push*** помещает их в начало. Обычные версии осуществляют проверку для уверенности, что в наличии имеется достаточное пространство; версии с двойным подчеркиванием не выполняют таких проверок.

int skb_headroom(struct sk_buff *skb);
int skb_tailroom(struct sk_buff *skb);
void skb_reserve(struct sk_buff *skb, int len);
Функции, которые выполняют управление пространством внутри **skb**. ***skb_headroom*** и ***skb_tailroom*** сообщают, сколько свободного места доступно на начале и конце **skb**, соответственно. ***skb_reserve*** может быть использована для резервирования места в начале **skb**, которое должно быть пустым.

unsigned char *skb_pull(struct sk_buff *skb, int len);
skb_pull "удаляет" данные из **skb** путём корректировки внутренних указателей.

int skb_is_nonlinear(struct sk_buff *skb);
Функция, которая возвращает значение "истина", если этот **skb** разделён на несколько фрагментов для ввода/вывода с разборкой/сборкой.

int skb_headlen(struct sk_buff *skb);
Возвращает длину первого сегмента **skb**, той части, на которую указывает **skb->data**.

void *kmap_skb_frag(skb_frag_t *frag);
void kunmap_skb_frag(void *vaddr);
Функции, которые обеспечивают прямой доступ к фрагментам в нелинейном **skb**.

#include <linux/etherdevice.h>
void ether_setup(struct net_device *dev);
Функция, которая устанавливает большинство методов устройства для реализации драйверов Ethernet общего назначения. Она также устанавливает **dev->flags** и присваивает следующее доступное **ethx** имя для **dev->name**, если первый символ в имени является пробелом или символом **NULL**.

unsigned short eth_type_trans(struct sk_buff *skb, struct net_device *dev);
Когда интерфейс Ethernet принимает пакет, эта функция может быть вызвана для установки **skb->pkt_type**. Возвращаемое значение является номером протокола, который обычно хранится в **skb->protocol**.

#include <linux/sockios.h>
SIOCDEVPRIVATE

Первая из 16-ти команд ***ioctl***, которые могут быть реализованы каждым драйвером для его личного использования. Все сетевые команды ***ioctl*** определены в ***sockios.h***.

#include <linux/mii.h>
struct mii_if_info;

Декларации и структура, поддерживающие драйверы устройств, реализующих стандарт MII.

#include <linux/ethtool.h>
struct ethtool_ops;

Декларации и структуры, которые позволяют устройствам работать с утилитой ***ethtool***.

Глава 18, TTY драйверы

Устройство tty получило своё название от очень старого сокращения, применяемого для телетайпа, и первоначально связанного только с физическим или виртуальным терминалом, подключенного к Unix машине. Со временем это название также стало означать любое устройство, похожее на последовательный порт, такое как терминальное соединение, которое могло бы также быть создано с использованием такого соединения. Примерами физических tty устройств являются последовательные порты, преобразователи из USB в последовательный порт, а также некоторые виды модемов, которым для правильной работы необходима специальная обработка (такие, как традиционные устройства в стиле Win-модемов). Виртуальные tty устройства поддерживают виртуальные консоли, которые используются в компьютере для логирования, работающие с помощью клавиатуры, через сетевое соединение, либо через сессию xterm.

Ядро tty драйвера Linux находится прямо под уровнем стандартного символьного драйвера и предоставляет набор возможностей, направленных на предоставление интерфейса для использования устройствами терминального стиля. Ядро отвечает одновременно за управление потоком данных через tty устройство и формат данных. Это позволяет tty драйверам сосредоточиться на обработке данных для и от оборудования, а не беспокоиться о том, как единообразным образом управлять взаимодействием с пользовательским пространством. Для управления потоком данных существует ряд различных дисциплин линии, которые могут быть виртуально "подключены" к любому tty устройству. Это выполняется разными драйверами дисциплины tty линии.

Как показывает Рисунок 18-1, ядро tty принимает данные от пользователя, который послал их tty устройство. Затем оно передает их в драйвер дисциплины tty линии, которая затем передает их в драйвер tty. tty драйвер преобразует данные в формат, который может быть отправлен в оборудование. Данные, принимаемые из tty оборудования, передаются обратно наверх через tty драйвер, в драйвер дисциплины tty линии и в tty ядро, откуда они могут быть получены пользователем. Иногда tty драйвер взаимодействует непосредственно с tty ядром и tty ядро отправляет данные непосредственно в tty драйвер, но обычно дисциплина tty линии имеет шанс изменить данные, которые передаются между ними.

Рисунок 18-1. Обзор ядра tty

Драйвер tty никогда не видит дисциплины tty линии. Драйвер не может ни взаимодействовать непосредственно с дисциплиной линии, ни даже осознавать её наличия. Работой драйвера является отформатировать данные, которые ему отправлены, таким образом, чтобы оборудование смогло их понять, и принимать данные от оборудования. Работой дисциплины tty линии является определённым образом форматировать данные, полученные от пользователя или оборудования. Данное форматирование обычно выполняется в форме преобразования протоколов, таких как PPP или Bluetooth.

Существуют три разных типа tty драйверов: консольный, последовательного порта и pty (всевдо-терминала). Драйверы консоли и pty уже написаны и, вероятно, являются единственно необходимыми из этих типов tty драйверов. Это позволяет любым новым драйверам использовать ядро tty для взаимодействия с пользователем и системой, подобно драйверам последовательного порта.

Чтобы определить, какие типы tty драйверов в данный момент загружены в ядро, и какие tty устройства присутствуют в настоящее время, посмотрите в файл ***/proc/tty/drivers***. Этот файл состоит из списка присутствующих в данный момент времени различных tty драйверов, показывая имя драйвера, имя узла по умолчанию, старший номер для драйвера, диапазон младших номеров, используемых драйвером, и тип tty драйвера. Ниже приведён пример такого файла:

```
/dev/tty        /dev/tty       5      0 system:/dev/tty
/dev/console    /dev/console   5      1 system:console
/dev/ptmx       /dev/ptmx      5      2 system
/dev/vc/0       /dev/vc/0      4      0 system:vtmaster
usbserial       /dev/ttyUSB  188  0-254 serial
serial          /dev/ttyS      4  64-67 serial
pty_slave       /dev/pts     136  0-255 pty:slave
pty_master      /dev/ptm     128  0-255 pty:master
```

```
pty_slave       /dev/ttyp      3  0-255 pty:slave
pty_master      /dev/pty       2  0-255 pty:master
unknown         /dev/tty       4   1-63 console
```

Каталог ***/proc/tty/driver/*** содержит отдельные файлы для некоторых из драйверов tty, если они реализуют такую функциональность. Драйвер по умолчанию для последовательного порта создаёт в этом каталоге файл, показывающий много специфичной информации об оборудовании последовательного порта. Информация о том, как создать файл в этом каталоге, представлена ниже.

Все tty устройства, зарегистрированные в настоящее время и присутствующие в ядре, имеют свой собственный подкаталог в ***/sys/class/tty***. Внутри этого подкаталога есть файл "dev", который содержит старший и младший номер, присвоенный этому tty устройству. Если драйвер сообщает ядру расположение физического устройства и драйвер, связанный с tty устройством, оно создаёт к ним символьные ссылки. Пример этого дерева:

```
/sys/class/tty/
|-- console
|   `-- dev
|-- ptmx
|   `-- dev
|-- tty
|   `-- dev
|-- tty0
|   `-- dev
   ...
|-- ttyS1
|   `-- dev
|-- ttyS2
|   `-- dev
|-- ttyS3
|   `-- dev
   ...
|-- ttyUSB0
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/
ttyUSB0
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB1
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/
ttyUSB1
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
|-- ttyUSB2
|   |-- dev
|   |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/
ttyUSB2
|   `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
`-- ttyUSB3
    |-- dev
    |-- device -> ../../../devices/pci0000:00/0000:00:09.0/usb3/3-1/3-1:1.0/
ttyUSB3
    `-- driver -> ../../../bus/usb-serial/drivers/keyspan_4
```

Небольшой TTY драйвер

Чтобы объяснить, как работает ядро tty, мы создаём небольшой tty драйвер, который может быть загружен, может записать и прочитать данные и быть выгружен. Основной структурой данных любого tty драйвера является **struct tty_driver**. Она используется для регистрации и отмены регистрации tty драйвера в ядре tty и описана в заголовочном файле ядра ***<linux/tty_driver.h>***.

Чтобы создать **struct tty_driver**, должна быть вызвана функция ***alloc_tty_driver*** с количеством поддерживаемых этим драйвером tty устройств в качестве параметра. Это может быть сделано следующим коротким кодом:

```
/* создаём tty драйвер */
tiny_tty_driver = alloc_tty_driver(TINY_TTY_MINORS);
if (!tiny_tty_driver)
    return -ENOMEM;
```

После успешного вызова функции ***alloc_tty_driver***, **struct tty_driver** должна быть проинициализирована соответствующей информацией на основе потребностей tty драйвера. Эта структура содержит множество разных полей, но не все из них должны быть проинициализированы, чтобы получить работающий tty драйвер. Вот пример, который показывает, как проинициализировать эту структуру, и устанавливает достаточное число полей, чтобы создать работающий tty драйвер. Чтобы помочь скопировать набор назначенных операций, которые определены в драйвере, он использует функцию ***tty_set_operations***:

```
static struct tty_operations serial_ops = {
    .open = tiny_open,
    .close = tiny_close,
    .write = tiny_write,
    .write_room = tiny_write_room,
    .set_termios = tiny_set_termios,
};

...

    /* инициализация tty драйвера */
    tiny_tty_driver->owner = THIS_MODULE;
    tiny_tty_driver->driver_name = "tiny_tty";
    tiny_tty_driver->name = "ttty";
    tiny_tty_driver->devfs_name = "tts/ttty%d";
    tiny_tty_driver->major = TINY_TTY_MAJOR,
    tiny_tty_driver->type = TTY_DRIVER_TYPE_SERIAL,
    tiny_tty_driver->subtype = SERIAL_TYPE_NORMAL,
    tiny_tty_driver->flags = TTY_DRIVER_REAL_RAW | TTY_DRIVER_NO_DEVFS,
    tiny_tty_driver->init_termios = tty_std_termios;
    tiny_tty_driver->init_termios.c_cflag = B9600 | CS8 | CREAD | HUPCL |
CLOCAL;
    tty_set_operations(tiny_tty_driver, &serial_ops);
```

Переменные и функции, перечисленные выше, и как использовать эту структуру объяснено в остальной части главы.

Чтобы зарегистрировать этот драйвер в ядре tty, ***struct tty_driver*** должна быть передана в

функцию ***tty_register_driver***:

```
/* регистрируем этот tty драйвер */
retval = tty_register_driver(tiny_tty_driver);
if (retval) {
    printk(KERN_ERR "failed to register tiny tty driver");
    put_tty_driver(tiny_tty_driver);
    return retval;
}
```

При вызове ***tty_register_driver*** ядро создаёт в sysfs все различные tty файлы на весь диапазон младших номеров tty устройств, которые может иметь этот tty драйвер. Если вы используете ***devfs*** (не описанную в этой книге) и если не указан флаг **TTY_DRIVER_NO_DEVFS**, также создаются файлы в ***devfs***. Этот флаг может быть указан, если вы хотите вызвать ***tty_register_device*** только для тех устройств, которые реально существуют в системе, чтобы пользователь всегда имел достоверный список устройств, присутствующих в ядре, что и ожидают пользователи ***devfs***.

После регистрации самого себя, драйвер через функцию ***tty_register_device*** регистрирует устройства, которыми он управляет. Эта функция имеет три аргумента:

- Указатель на **struct tty_driver**, к которой принадлежит это устройство.
- Младший номер этого устройства.
- Указатель на **struct device**, с которой связано это tty устройство. Если это tty устройства не связано ни с какой **struct device**, этот аргумент может быть установлен NULL.

Наш драйвер регистрирует все tty устройства одновременно, поскольку они являются виртуальными и не связаны с какими-либо физическими устройствами:

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_register_device(tiny_tty_driver, i, NULL);
```

Чтобы отменить регистрацию драйвера в ядре tty, все tty устройства, которые были зарегистрированы вызовом ***tty_register_device***, необходимо освободить вызовом ***tty_unregister_device***. Затем вызовом ***tty_unregister_driver*** должна быть отменена регистрация **struct tty_driver**:

```
for (i = 0; i < TINY_TTY_MINORS; ++i)
    tty_unregister_device(tiny_tty_driver, i);
tty_unregister_driver(tiny_tty_driver);
```

struct termios

Переменной **init_termios** в **struct tty_driver** является **struct termios**. Эта переменная используется, чтобы обеспечить разумный набор настроек линии, если порт используется до того, как он проинициализирован пользователем. Драйвер инициализирует переменную с помощью стандартного набора значений, которые скопированы из переменной **tty_std_termios**. **tty_std_termios** определена в ядре tty следующим образом:

```
struct termios tty_std_termios = {
    .c_iflag = ICRNL | IXON,
    .c_oflag = OPOST | ONLCR,
```

```
    .c_cflag = B38400 | CS8 | CREAD | HUPCL,
    .c_lflag = ISIG | ICANON | ECHO | ECHOE | ECHOK |
               ECHOCTL | ECHOKE | IEXTEN,
    .c_cc = INIT_C_CC
};
```

Структура **struct termios** используется для хранения всех текущих настроек линии для данного порта tty устройства. Эти параметры линии управляют текущей скоростью передачи, размером данных, настройками потока данных и многими другими значениями. Полями этой структуры являются:

tcflag_t c_iflag;
Флаги режима ввода.

tcflag_t c_oflag;
Флаги режима вывода.

tcflag_t c_cflag;
Флаги режима управления.

tcflag_t c_lflag;
Флаги местного режима.

cc_t c_line;
Тип дисциплины линии.

cc_t c_cc[NCCS];
Массив управляющих символов.

Все флаги режимов определены как большое битовое поле. Различные значения режимов и для чего они используются можно увидеть в руководстве по **termios**, доступного в любом дистрибутиве Linux. Для получения на различных битов ядро предоставляет набор полезных макросов. Эти макросы определены в заголовочном файле ***include/linux/tty.h***.

Все поля, которые были определены в переменной **tiny_tty_driver**, необходимы для получения рабочего tty драйвера. Поле **owner** необходимо для предотвращения выгрузки tty драйвера во время открытия tty порта. В предыдущих версиях ядра это выполнялось самим tty драйвером с помощью логики подсчёта числа ссылок на модуль. Но программисты ядра решили, что было бы сложно разрешить все различные возможные состояния гонок, и поэтому за tty драйвер всё это управление выполняет ядро tty.

Поля **driver_name** и **name** выглядят очень похожими, но всё же используются для различных целей. Переменная **driver_name** должна быть установлена во что-то краткое, наглядное и уникальные среди всех tty драйверов в ядре. Так происходит потому, что оно показывается в ***/proc/tty/drivers*** для описания драйвера для пользователя и в sysfs в каталоге для tty класса, куда в настоящее время загружаются драйверы. Поле **name** используется, чтобы задать имя отдельным tty узлам, назначенных этому tty драйверу в дереве ***/dev***. Эта строка используется для создания tty устройства путём добавления используемого номера tty устройства в конец строки. Оно также используется для создания имени устройства в каталоге sysfs ***/sys/class/tty/***. Если в ядре разрешена devfs, это имя должно включать любой подкаталог, в котором хочет быть помещён tty драйвер. Так, например, драйвер последовательного порта в

ядре устанавливает поле **name** в **tts/**, если devfs разрешена, и в **ttyS**, если это не так. Эта строка также отображается в файле */proc/tty/drivers*.

Как мы уже упоминали, файл */proc/tty/drivers* показывает все в настоящее время зарегистрированные tty драйверы. При использовании для регистрации в ядре драйвера *tiny_tty* и отсутствии *devfs* этот файл выглядит похожим на следующий:

```
$ cat /proc/tty/drivers
tiny_tty             /dev/ttty    240     0-3 serial
usbserial            /dev/ttyUSB  188   0-254 serial
serial               /dev/ttyS      4  64-107 serial
pty_slave            /dev/pts     136   0-255 pty:slave
pty_master           /dev/ptm     128   0-255 pty:master
pty_slave            /dev/ttyp      3   0-255 pty:slave
pty_master           /dev/pty       2   0-255 pty:master
unknown              /dev/vc/       4    1-63 console
/dev/vc/0            /dev/vc/0      4       0 system:vtmaster
/dev/ptmx            /dev/ptmx      5       2 system
/dev/console         /dev/console   5       1 system:console
/dev/tty             /dev/tty       5       0 system:/dev/tty
```

Кроме того, когда драйвер *tiny_tty* зарегистрирован в ядре tty, каталог */sys/class/tty* в sysfs выглядит следующим образом:

```
$ tree /sys/class/tty/ttty*
/sys/class/tty/ttty0
`-- dev
/sys/class/tty/ttty1
`-- dev
/sys/class/tty/ttty2
`-- dev
/sys/class/tty/ttty3
`-- dev

$ cat /sys/class/tty/ttty0/dev
240:0
```

Старшее значение показывает старший номер для этого драйвер. Переменные типа и подтипа показывают каким типом tty драйвера является этот драйвер. В нашем примере последовательный драйвер является последовательным драйвером "нормального" типа. Единственным другим подтипом tty драйвера будет тип "с выноской" (управления) (“callout”). Устройства с выноской (управления) традиционно использовались для управления настройками линии устройства. В этом случае данные бы отправлялись и получались через один узел устройства, а любые изменения настройки линии, отправлялись бы на другой узел устройства, который бы был вынесенным устройством. Это требует использования двух младших номеров для каждого tty устройства. К счастью, почти все драйверы обрабатывают и данные и настройки линии используя один и тот же узел устройства, и тип с выноской в новых драйверах используются редко.

Переменная **flags** используется как tty драйвером, так и tty ядром для указания текущего состояние драйвера и типа, к которому принадлежит tty драйвер. Определено несколько макросов битовых масок, которые вы должны использовать при проверке или манипулировании этими флагами. Драйвером могут быть установлены три бита в переменной

flags:

TTY_DRIVER_RESET_TERMIOS
Этот флаг сообщает, что ядро tty сбрасывает настройки **termios** всякий раз, когда последний процесс закрыл это устройство. Это полезно для консольных и pty драйверов. Например, предположим, что пользователь покидает терминал в странном состоянии. Когда этот флаг установлен, терминал сбрасывается в нормальное значение, когда пользователь выходит из системы, или процесс, который управлял этой сессией, "убит".

TTY_DRIVER_REAL_RAW
Этот флаг сообщает, что этот tty драйвер гарантирует отправку уведомлений о чётности или символов переноса дисциплины завершения линии. Это позволяет дисциплине линии обрабатывать полученные символы намного более быстрым образом, так как не надо проверять каждый символ, полученный от этого tty драйвера. Поскольку это увеличивает скорость, данное значение обычно устанавливается для всех tty драйверов.

TTY_DRIVER_NO_DEVFS
Этот флаг указывает, что когда производится вызов ***tty_register_driver***, для данного tty устройства ядро tty не создаёт никаких записей в devfs. Это полезно для любого драйвера, который динамически создаёт и уничтожает устройства с младшими номерами. Примерами драйверов, которые устанавливают его, являются являются драйверы преобразования USB в последовательный порт, драйвер USB модема, tty драйвер для USB Bluetooth, а также ряд стандартных драйверов последовательного порта.
Если позже tty драйвер захочет зарегистрировать данное tty устройство в ядре tty, он должен вызвать ***tty_register_device*** с указателем на tty драйвер и младший номер устройства, которое было создано. Если этого не сделать, ядро tty по-прежнему передаёт все вызовы в этот tty драйвер, но некоторой внутренней, относящейся к tty функциональности может и не быть. Это включает уведомление ***/sbin/hotplug*** о новых tty устройствах и представление tty устройства в sysfs. Когда зарегистрированное tty устройство удаляется из машины, tty драйвер должен вызвать ***tty_unregister_device***.

Последний оставшийся бит в этой переменной управляется ядром tty и называется **TTY_DRIVER_INSTALLED**. Этот флаг устанавливается ядром tty после того, как драйвер был зарегистрирован, и никогда не должен быть устанавливаться tty драйвером.

Указатели на функции в tty_driver

Наконец, драйвер ***tiny_tty*** декларирует четыре указателя на функции.

open и close

Функция ***open*** вызывается ядром tty, когда пользователь вызывает ***open*** для назначенного tty драйверу узла устройства. Ядро tty вызывает её с указателем на структуру **tty_struct**, присвоенной этому устройству, и указателем на файл. Для корректной работы tty драйвером должно быть установлено поле ***open***; в противном случае, при вызове открытия пользователю возвращается **-ENODEV**.

Когда вызывается эта функция ***open***, ожидается, что tty драйвер либо сохраняет некоторые данные в переменной **tty_struct**, которая передаётся ему, либо сохраняет данные в статическом массиве, на который можно ссылаться на основе младшего номера порта. Это необходимо, чтобы tty драйвер знал, на какое устройство в настоящее время ссылаются, когда

впоследствии будут вызваны ***close***, ***write*** и другие функции.

Драйвер ***tiny_tty*** сохраняет указатель в структуре **tty**, как можно видеть в следующем коде:

```
static int tiny_open(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny;
    struct timer_list *timer;
    int index;

    /* проинициализируем указатель на случай, если что-то пойдёт не так */
    tty->driver_data = NULL;

    /* получить последовательный объект, связанный с этим указателем tty */
    index = tty->index;
    tiny = tiny_table[index];
    if (tiny == NULL) {
        /* к этому устройству обращение делается в первый раз, давайте его
создадим */
        tiny = kmalloc(sizeof(*tiny), GFP_KERNEL);
        if (!tiny)
            return -ENOMEM;

        init_MUTEX(&tiny->sem);
        tiny->open_count = 0;
        tiny->timer = NULL;

        tiny_table[index] = tiny;
    }

    down(&tiny->sem);

    /* сохраним нашу структуру в структуре tty */
    tty->driver_data = tiny;
    tiny->tty = tty;
```

В этом коде структура **tiny_serial** сохраняется в структуре **tty**. Это позволяет функциям ***tiny_write***, ***tiny_write_room*** и ***tiny_close*** получать структуру **tiny_serial** и манипулировать ею должным образом.

Структура **tiny_serial** определена как:

```
struct tiny_serial {
    struct tty_struct *tty;  /* указатель на tty для этого устройства */
    int         open_count;  /* сколько раз был открыт этот порт */
    struct semaphore   sem;  /* блокировка этой структуры */
    struct timer_list *timer;
};
```

Как мы уже видели, переменная **open_count** инициализируется в 0 во время вызова ***open*** при первом открытии порта. Это типичный счётчик ссылок, необходимый потому, что функции ***open*** и ***close*** драйвера tty могут быть вызваны для одного устройства множество раз, чтобы позволить нескольким процессам читать и записывать данные. Чтобы всё обработать правильно, должен храниться счётчик, сколько раз порт был открыт или закрыт; при

использовании порта драйвер увеличивает и уменьшает этот счётчик. Когда порт открыт в первый раз, может быть выполнена любая необходимая инициализация оборудования и выделение памяти. Когда порт закрывается в последний раз, может быть выполнена всё необходимое для выключения оборудования и очистке памяти.

Остальная часть функции ***tiny_open*** показывает, как отслеживать число открытий устройства:

```
++tiny->open_count;
if (tiny->open_count == 1) {
    /* здесь порт открыт первый раз */
    /* выполняем здесь любую необходимую инициализацию оборудования */
```

Функция ***open*** должна возвращать либо отрицательный номер ошибки, если что-то случилось, чтобы предотвратить успешное открытие, или 0, что означает успех.

Указатель на функцию ***close*** вызывается ядром tty, когда пользователь вызвал ***close*** для дескриптора файла, который ранее был создан вызовом ***open***. Это означает, что в это время устройство должно быть закрыто. Однако, поскольку функция ***open*** может быть вызвана более одного раза, функция ***close*** также может быть вызвана более одного раза. Поэтому эта функция должна отслеживать, сколько раз она была вызвана, чтобы определить, действительно ли в это время должно быть закрыто оборудование. Драйвер ***tiny_tty*** делает это с помощью следующего кода:

```
static void do_close(struct tiny_serial *tiny)
{
    down(&tiny->sem);

    if (!tiny->open_count) {
        /* порт никогда не открывался */
        goto exit;
    }

    --tiny->open_count;
    if (tiny->open_count <= 0) {
        /* Этот порт был закрыт последним пользователем. */
        /* Выполняем здесь любые относящиеся к оборудованию действия */

        /* выключить наш таймер */
        del_timer(tiny->timer);
    }
exit:
    up(&tiny->sem);
}

static void tiny_close(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (tiny)
        do_close(tiny);
}
```

Чтобы сделать настоящую работу по закрытию устройства, функция ***tiny_close*** просто вызывает функцию ***do_close***. Это делается для того, чтобы не дублировать логику закрытия

здесь и при выгрузке драйверов и при открытии порта. Функция ***close*** не имеет возвращаемого значения, так как провал не предполагается.

Движение данных

Функция ***write*** вызывается пользователем, когда есть данные для отправки в оборудование. Сначала ядро tty получает вызов, а затем передаёт эти данные в функцию tty драйвера ***write***. Ядро tty также сообщает tty драйверу размер отправляемых данных.

Иногда из-за скорости и ёмкости буфера оборудования tty не все символы, запрошенные записывающей программой, могут быть отправлены в момент, когда вызвана функция ***write***. Функция ***write*** должна возвращать количество символов, которое смогла отправить в оборудование (или поместить в очередь, чтобы отправить позднее), так что пользовательская программа может проверить, все ли данные были записаны на самом деле. Гораздо проще выполнить такую проверку в пользовательском пространстве, чем для драйвера ядра стоять и спать, пока все запрошенные данные смогут быть отправлены. Если во время вызова ***write*** произошла ошибка, должно быть возвращено отрицательное значение ошибки, а не число записанных символов.

Функция ***write*** может быть вызвана и из контекста прерывания и из пользовательского контекста. Это важно знать, так как tty драйверы не должны вызывать каких-либо функций, которые могли бы заснуть, когда они находятся в контексте прерывания. К ним относятся любые функции, которые могли бы быть вызвать ***schedule***, такие как обычные функции ***copy_from_user***, ***kmalloc*** и ***printk***. Если вы действительно хотите поместить драйвер в сон, сначала убедитесь, находится ли драйвер в контексте прерывания, вызвав ***in_interrupt***.

Это пример крошечного tty драйвера не подключенного к какому-то реальному оборудованию, поэтому его функция записи просто выполняет запись в журнале отладки ядра, предполагая, что данные записаны. Это делается с помощью следующего кода:

```
static int tiny_write(struct tty_struct *tty, const unsigned char *buffer,
int count)
{
    struct tiny_serial *tiny = tty->driver_data;
    int i;
    int retval = -EINVAL;

    if (!tiny)
        return -ENODEV;

    down(&tiny->sem);
    if (!tiny->open_count)
    /* порт не был открыт */
        goto exit;

    /* подделываем отправку данных в аппаратный порт
     * записывая их в журнал отладки ядра.
     */
    printk(KERN_DEBUG "%s - ", __FUNCTION__);
    for (i = 0; i < count; ++i)
        printk("%02x ", buffer[i]);
    printk("\n");
```

```
exit:
    up(&tiny->sem);
    return retval;
}
```

Функция ***write*** может быть вызвана, когда tty подсистема сама нуждается отправить какие-то данные в tty устройство. Это может произойти, если tty драйвер не реализует функцию ***put_char*** в **tty_struct**. В этом случае ядро tty использует обратный вызов функции ***write*** с размером данных 1. Это обычно происходит, когда ядро tty хочет преобразовать символ новой строки в перевод строки плюс символ новой строки. Самая большая проблема , которая может здесь возникнуть в том, что функция tty драйвера ***write*** не должна возвращать 0 для вызова такого вида. Это означает, что драйвер должен записать такой байт данных в устройство, так как вызывающий (ядро tty) не буферирует данные и пробует ещё раз позже. Так как функция ***write*** не может определить, вызвана ли она вместо ***put_char***, даже если в настоящее время отправляется только один байт данных, попытаемся реализовать функцию ***write*** так, чтобы перед возвращением она всегда писала по крайней мере один байт. Ряд текущих tty драйверов USB-последовательный порт не следуют этому правилу и в силу этого при подключении к ним некоторые типы терминалов не работают должным образом.

Когда ядро tty хочет узнать, как много места доступно в буфере записи tty драйвера, вызывается функция ***write_room***. Это число со временем изменяется, так как символы уходят наружу, освобождая буфер записи, и происходят вызовы функции ***write***, добавляя символы в буфер.

```
static int tiny_write_room(struct tty_struct *tty)
{
    struct tiny_serial *tiny = tty->driver_data;
    int room = -EINVAL;

    if (!tiny)
        return -ENODEV;

    down(&tiny->sem);

    if (!tiny->open_count) {
        /* порт не был открыт */
        goto exit;
    }
    /* вычисляем, как много места осталось в устройстве */
    room = 255;

exit:
    up(&tiny->sem);
    return room;
}
```

Другие функции буферизации

Функция ***chars_in_buffer*** в структуре **tty_driver** не требуется, чтобы иметь рабочий tty драйвер, но это рекомендуется. Эта функция вызывается, когда ядро tty хочет знать, сколько символов для отправки всё ещё остаётся в буфер записи tty драйвера. Если драйвер может хранить символы перед отправкой их в оборудование, он должен реализовать эту функцию, чтобы ядро tty имело возможность определить, все ли данные утекли из драйвера.

Для сброса в поток всех оставшихся данных, удерживаемых драйвером, могут быть использованы три функции обратного вызова в структуре **tty_driver**. Реализация их не требуется, но рекомендуется, если tty драйвер может буферизировать данные перед их отправкой в оборудование. Первые две функции обратного вызова называются ***flush_chars*** и ***wait_until_sent***. Эти функции вызываются, когда ядро tty отправило в tty драйвер несколько символов с помощью функции обратного вызова ***put_char***. Функция обратного вызова ***flush_chars*** вызывается, когда ядро tty хочет, чтобы tty драйвер начал отправку этих символов в оборудование, если он ещё не начал это делать. Этой функции разрешено вернуться до того, как все данные отправлены в оборудование. Функция обратного вызова ***wait_until_sent*** работает во многом аналогично; но она должна подождать отправки всех символов до возвращения в ядро tty или истечения переданного ей времени ожидания, в зависимости от того, что произойдёт в первую очередь. В течение этой функции, для её завершения, tty драйверу разрешается спать. Если значение времени ожидания, переданное в функцию обратного вызова ***wait_until_sent***, установлено в 0, функция должна ждать, пока эта операция не завершится.

Оставшейся функцией обратного вызова для сброса данных является ***flush_buffer***. Она вызывается ядром tty, когда tty драйвер должен сбросить из памяти все данные, всё ещё хранящиеся в его буферах записи. Любые данные, остающиеся в буфере, теряются и не передаются на устройство. (видимо, не остающиеся в буфере, а данные, которые до этого не поступили в буфер.)

Нет функции read?

Только с этими функциями драйвер ***tiny_tty*** может быть зарегистрирован, узел устройства открыт, данные записаны в устройство, узел устройства закрыт и отменена регистрация драйвера и он выгружен из ядра. Но ядро tty и структура **tty_driver** не предоставляет функцию чтения; в других словах, не существует функции обратного вызова для получения данных из драйвера ядром tty.

Вместо обычной функции чтения, за отправку в ядро tty любых данных, полученных от оборудования, несёт ответственность tty драйвер, когда он их получает. Ядро tty буферизует эти данные, пока они не запрошены пользователем. Поскольку ядро tty предоставляет логику буферизации, каждому tty драйверу нет необходимости реализовывать свою собственную логику буферизации. Ядро tty уведомляет tty драйвер, когда пользователь хочет, чтобы драйвер остановил и начал передачу данных, но если внутренние буферы tty полны, такого уведомления не происходит.

Ядро tty буферизует данные, полученные tty драйверами в структуре, названной **struct tty_flip_buffer**. Переключаемый буфер является структурой, которая содержит два основных массива данных. Данные, полученные от tty устройства хранятся в первом массиве. Когда этот массив полон, любой пользователь, ожидающий данные, уведомляется, что данные доступны для чтения. Хотя пользователь читает данные из этого массива, любые новые входящие данные хранятся во втором массиве. Когда этот массив заполнен, данные вновь сбрасываются пользователю, а драйвер начинает заполнять первый массив. По сути, принятые данные "переключаются" с одного буфера в другой, в надежде не переполнить их обоих. Чтобы попытаться предотвратить потерю данных, tty драйвер может контролировать, насколько велик входной массив, и если он заполнится, приказать tty драйверу очистить переключаемый буфер в этот момент времени, а не ожидать следующего шанса.

Детали структуры **struct tty_flip_buffer** не имеют значения для tty драйвера за с одним

исключением, переменной **count**. Эта переменная содержит сколько байт в настоящее время остаётся в буфере, который используется для приёма данных. Если это значение равно значению **TTY_FLIPBUF_SIZE**, буфер необходимо сбросить пользователю вызовом ***tty_flip_buffer_push***. Это показано следующим фрагментом кода:

```
for (i = 0; i < data_size; ++i) {
    if (tty->flip.count >= TTY_FLIPBUF_SIZE)
        tty_flip_buffer_push(tty);
    tty_insert_flip_char(tty, data[i], TTY_NORMAL);
}
tty_flip_buffer_push(tty);
```

Символы, полученные от tty драйвера для отправки пользователю, добавляются в переключаемый буфер вызовом ***tty_insert_flip_char***. Первым параметром этой функции является **struct tty_struct**, где должны быть сохранены данные, вторым параметром является символ для сохранения и третьим параметром являются любые флаги, которые должны быть установлены для этого символа. Значение флагов должно быть установлено в **TTY_NORMAL**, если получен обычный символ. Если это особый тип символа, указывающий на ошибку передачу данных, оно должно быть установлено на **TTY_BREAK**, **TTY_FRAME**, **TTY_PARITY** или **TTY_OVERRUN**, в зависимости от ошибки.

Для того, чтобы "протолкнуть" данные пользователю, выполняется вызов ***tty_flip_buffer_push***. Этот функция должна быть также вызвана, если переключаемый буфера близок к переполнению, как показано в этом примере. Поэтому, когда данные добавляются в переключаемый буфер, или когда переключаемый буфер заполнен, tty драйвер должен вызвать ***tty_flip_buffer_push***. Если tty драйвер может принимать данные на очень высоких скоростях, должен быть установлен флаг **tty->low_latency**, что приводит при вызове к немедленному вызову ***tty_flip_buffer_push***. В противном случае, вызов ***tty_flip_buffer_push*** планирует себя для выталкивания данных из буфера когда-то позднее в ближайшем будущем.

Настройки TTY линии

Когда пользователь хочет изменить настройки строки tty устройства или получить текущие настройки строки, он выполняет один из многих разнообразных вызовов **termios** библиотечной функции пользовательского пространства или непосредственно сделать вызов ***ioctl*** для узла tty устройства. Ядро tty преобразует оба этих интерфейса в ряд различных функций обратного вызова tty драйвера и вызов ***ioctl***.

set_termios

Большинство функций **termios** пользовательского пространства транслируются библиотекой в вызов ***ioctl*** узла драйвера. Большое количество различных вызовов ***ioctl*** для tty затем переводятся ядром tty в один вызов функции ***set_termios*** tty драйвера. Обратному вызову ***set_termios*** необходимо определить, какие настройки строки его просят изменить, и затем выполнить эти изменения в tty устройстве. Драйвер tty должен быть в состоянии декодировать все разнообразные настройки в структуре **termios** и отреагировать на любые необходимые изменения. Это сложная задача, так как все настройки линии упакованы в структуру **termios** большим числом способов.

Первое, что должна сделать функция ***set_termios***, это определить надо ли что-то изменять на самом деле. Это можно сделать с помощью следующего кода:

```
unsigned int cflag;

cflag = tty->termios->c_cflag;

/* проверяем, что действительно хотят, чтобы мы изменили */
if (old_termios) {
    if ((cflag == old_termios->c_cflag) &&
        (RELEVANT_IFLAG(tty->termios->c_iflag) ==
        RELEVANT_IFLAG(old_termios->c_iflag))) {
        printk(KERN_DEBUG " - nothing to change...\n");
        return;
    }
}
```

Макрос **RELEVANT_IFLAG** определён как:

```
#define RELEVANT_IFLAG(iflag) ((iflag) & (IGNBRK|BRKINT|IGNPAR|PARMRK|INPCK))
```

и используется, чтобы маскировать важные биты в переменной **cflags**. Затем это сравнивается со старым значением и смотрится, есть ли изменения. Если нет, то ничего не должно быть изменено, поэтому мы возвращаемся. Обратите внимание, что сначала проверяется, что переменная **old_termios** указывает на действительную структуру, прежде чем к ней обратиться. Это необходимо, поскольку в некоторых случаях эта переменная имеет значение NULL. Попытка доступа к полю по указателю NULL приводит к панике ядра.

Чтобы посмотреть на запрошенный размер в байтах, для выделения требуемых битов из переменной **cflag** может быть использована битовая маска **CSIZE**. Если размер не может быть определён, то обычно по умолчанию он принимается как восемь битов данных. Это может быть реализовано следующим образом:

```
/* получаем размер в байтах */
switch (cflag & CSIZE) {
    case CS5:
        printk(KERN_DEBUG " - data bits = 5\n");
        break;
    case CS6:
        printk(KERN_DEBUG " - data bits = 6\n");
        break;
    case CS7:
        printk(KERN_DEBUG " - data bits = 7\n");
        break;
    default:
    case CS8:
        printk(KERN_DEBUG " - data bits = 8\n");
        break;
}
```

Чтобы определить запрошенное значение чётности, к переменной **cflag** может быть применена битовая маска **PARENB**, чтобы определить, должна ли быть чётность вообще установлена. Если это так, может быть использована битовая маска **PARODD**, чтобы определить какая чётность должна использоваться: чёт или нечет. Реализация этого:

```
/* определяем чётность */
```

```
if (cflag & PARENB)
    if (cflag & PARODD)
        printk(KERN_DEBUG " - parity = odd\n");
    else
        printk(KERN_DEBUG " - parity = even\n");
else
    printk(KERN_DEBUG " - parity = none\n");
```

Запрошенные стоп-биты также могут быть определены с помощью переменной **cflag**, используя маску **CSTOPB**. Реализация этого:

```
/* выясняем требуемые стоп-биты */
if (cflag & CSTOPB)
    printk(KERN_DEBUG " - stop bits = 2\n");
else
    printk(KERN_DEBUG " - stop bits = 1\n");
```

Существуют два основных типа управления потоком данных: аппаратный и программный. Чтобы определить, запрашивает ли пользователь аппаратное управление потоком, к переменной **cflag** может применяться битовая маска **CRTSCTS**. Пример этого:

```
/* выясняем настройки аппаратного контроля передачи */
if (cflag & CRTSCTS)
    printk(KERN_DEBUG " - RTS/CTS is enabled\n");
else
    printk(KERN_DEBUG " - RTS/CTS is disabled\n");
```

Определение различных видов программного управления потоком и различных стоповых и стартовых символов немного более сложное:

```
/* определяем программное управление потоком */
/* если мы реализуем XON/XOFF, устанавливаем стартовый и
 * стоповый символ в устройстве */
if (I_IXOFF(tty) || I_IXON(tty)) {
    unsigned char stop_char = STOP_CHAR(tty);
    unsigned char start_char = START_CHAR(tty);

    /* если мы реализуем ВХОДНОЙ XON/XOFF */
    if (I_IXOFF(tty))
        printk(KERN_DEBUG " - INBOUND XON/XOFF is enabled, "
                "XON = %2x, XOFF = %2x", start_char, stop_char);
    else
        printk(KERN_DEBUG" - INBOUND XON/XOFF is disabled");

    /* если мы реализуем ВЫХОДНОЙ XON/XOFF */
    if (I_IXON(tty))
        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is enabled, "
                "XON = %2x, XOFF = %2x", start_char, stop_char);
    else
        printk(KERN_DEBUG" - OUTBOUND XON/XOFF is disabled");
}
```

Наконец, должна быть определена скорость передачи данных. Чтобы помочь сделать это, ядро tty предоставляет основные функцию ***tty_get_baud_rate***. Функция возвращает целое

число, показывающее запрошенную скорость передачи данных для данного tty устройства:

```
/* получаем скорость передачи */
printk(KERN_DEBUG " - baud rate = %d", tty_get_baud_rate(tty));
```

Теперь, когда tty драйвер определили все разнообразные параметры линии, можно на основе этих значений должным образом проинициализировать оборудование.

tiocmget и tiocmset

В версии 2.4 и более старых ядрах, для получения и установки различных параметров управления линией использовалось несколько вызовов tty ***ioctl***. Они были обозначены константами **TIOCMGET**, **TIOCMBIS**, **TIOCMBIC** и **TIOCMSET**. **TIOCMGET** была использована для получения значений настройки строки в ядре, и начиная с версии 2.6, этот вызов ***ioctl*** был превращён в функцию обратного вызова tty драйвера, названную ***tiocmget***. Другие три ioctl-ы были упрощены и теперь представлены одной функцией обратного вызова tty драйвера, названной ***tiocmset***.

Функция ***tiocmget*** в tty драйвере вызывается ядром tty, когда ядро хочет узнать текущие физические значения линий управления определённого tty устройства. Это обычно делается для получения значений линий DTR и RTS последовательного порта. Если tty драйвер не может напрямую прочитать регистр MCR или MSR последовательного порта, потому что оборудование не позволяет этого, их копия должна сохраняться локально. Ряд драйверов USB-последовательный порт должны реализовать такого рода "теневую" переменную. Вот как может быть реализована эта функция, если сохраняются локальные копии этих переменных:

```
static int tiny_tiocmget(struct tty_struct *tty, struct file *file)
{
    struct tiny_serial *tiny = tty->driver_data;
    unsigned int result = 0;
    unsigned int msr = tiny->msr;
    unsigned int mcr = tiny->mcr;
    result = ((mcr & MCR_DTR) ? TIOCM_DTR  : 0) | /* DTR установлен */
             ((mcr & MCR_RTS)  ? TIOCM_RTS  : 0) | /* RTS установлен */
             ((mcr & MCR_LOOP) ? TIOCM_LOOP : 0) | /* LOOP установлен */
             ((msr & MSR_CTS)  ? TIOCM_CTS  : 0) | /* CTS установлен */
             ((msr & MSR_CD)   ? TIOCM_CAR  : 0) | /* Установлено обнаружение
несущей (Carrier detect) */
             ((msr & MSR_RI)   ? TIOCM_RI   : 0) | /* Установлен индикатор
вызова (Ring) */
             ((msr & MSR_DSR)  ? TIOCM_DSR  : 0);  /* DSR установлен */
    return result;
}
```

Когда ядро хочет установить значения управляющих линий определённого tty устройства, ядро tty вызывает в tty драйвере функцию ***tiocmset***. Ядро tty сообщает tty драйверу, какие значения установить и какие очистить, передавая их в двух переменных: **set** и **clear**. Эти переменные содержат битовую маску настроек линии, которые должны быть изменены. Вызов ***ioctl*** никогда не просит драйвер установить и сбросить один и тот же бит в одно время, поэтому не важно, какие операции происходят в первую очередь. Вот пример того, как эта функция может быть реализована tty драйвером:

```
static int tiny_tiocmset(struct tty_struct *tty, struct file *file,
```

```
                            unsigned int set, unsigned int clear)
{
    struct tiny_serial *tiny = tty->driver_data;
    unsigned int mcr = tiny->mcr;
    if (set & TIOCM_RTS)
        mcr |= MCR_RTS;
    if (set & TIOCM_DTR)
        mcr |= MCR_RTS;

    if (clear & TIOCM_RTS)
        mcr &= ~MCR_RTS;
    if (clear & TIOCM_DTR)
        mcr &= ~MCR_RTS;

    /* устанавливаем в устройстве новое значение для MCR */
    tiny->mcr = mcr;
    return 0;
}
```

ioctl-ы

Функция обратного вызова ***ioctl*** в **struct tty_driver** вызывается ядром tty, когда для узла устройства вызывается ***ioctl(2)***. Если tty драйвер не знает, как обработать переданное ему значение ioctl, он должен вернуть **-ENOIOCTLCMD**, чтобы попытаться дать ядру tty реализовать универсальную версию вызова.

Ядро версии 2.6 определяет около 70 различных tty ***ioctl***-ов, которые могут быть отправлены в tty драйвер. Большинство tty драйверов не обрабатывают их все, а только небольшую часть более общих команд. Вот список наиболее популярных tty ***ioctl***-ов, что они означают и как их реализовать:

TIOCSERGETLSR

Получает значение регистра статуса строки (LSR) этого tty устройства.

TIOCGSERIAL

Получает информацию последовательной линии. Вызывающий с помощью этого вызова может потенциально за раз получить много информации о последовательной линии tty устройства. Некоторые программы (такие, как ***setserial*** и ***dip***) вызывают эту функцию, чтобы убедиться, что скорость передачи была установлена правильно и получить общую информацию о том, каким типом устройства управляет tty драйвер. Вызывающий передаёт указатель на большую структуру типа **serial_struct**, которую tty драйвер должен заполнить соответствующими значениями.

Вот пример того, как это может быть реализовано:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int
cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCGSERIAL) {
        struct serial_struct tmp;
        if (!arg)
            return -EFAULT;
        memset(&tmp, 0, sizeof(tmp));
```

```
        tmp.type     = tiny->serial.type;
        tmp.line     = tiny->serial.line;
        tmp.port     = tiny->serial.port;
        tmp.irq      = tiny->serial.irq;
        tmp.flags    = ASYNC_SKIP_TEST | ASYNC_AUTO_IRQ;
        tmp.xmit_fifo_size = tiny->serial.xmit_fifo_size;
        tmp.baud_base      = tiny->serial.baud_base;
        tmp.close_delay    = 5*HZ;
        tmp.closing_wait   = 30*HZ;
        tmp.custom_divisor = tiny->serial.custom_divisor;
        tmp.hub6     = tiny->serial.hub6;
        tmp.io_type  = tiny->serial.io_type;
        if (copy_to_user((void __user *)arg, &tmp, sizeof(tmp)))
            return -EFAULT;
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

TIOCSSERIAL

Устанавливает информацию последовательной линии. Это является противоположностью **TIOCGSERIAL** и позволяет пользователю за раз установить состояние последовательной линии tty устройства. Указатель на **struct serial_struct**, передаваемый в этом вызове, заполнен данными, которые tty устройство должно сейчас установить. Если tty драйвер не реализует этот вызов, большинство программ по-прежнему работает правильно.

TIOCMIWAIT

Ожидание изменения MCP. Пользователь запрашивает этот ***ioctl*** в необычных обстоятельствах, при которых он хочет заснуть в ядре, пока что-то не случится в регистре MSR tty устройства. Параметр **arg** содержит тип события, которое ждёт пользователь. Это обычно используется для ожидания изменений линии состояния, сигнализирующих, что к отправке на это устройство готовы дополнительные данные. Будьте осторожны при реализации этого ***ioctl*** и не используйте вызов ***interruptible_sleep_on***, так как это небезопасно (есть много неприятных состояний гонок, создаваемых им). Вместо этого, чтобы избежать таких проблем, следует использовать ***wait_queue***. Вот пример того, как реализовать этот ***ioctl***:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int
cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCMIWAIT) {
        DECLARE_WAITQUEUE(wait, current);
        struct async_icount cnow;
        struct async_icount cprev;
        cprev = tiny->icount;
        while (1) {
            add_wait_queue(&tiny->wait, &wait);
            set_current_state(TASK_INTERRUPTIBLE);
            schedule( );
            remove_wait_queue(&tiny->wait, &wait);
            /* смотрим, не разбудил ли нас сигнал */
            if (signal_pending(current))
```

```
                return -ERESTARTSYS;
            cnow = tiny->icount;
            if (cnow.rng == cprev.rng && cnow.dsr == cprev.dsr &&
                cnow.dcd == cprev.dcd && cnow.cts == cprev.cts)
                return -EIO; /* нет изменения => ошибка */
            if (((arg & TIOCM_RNG) && (cnow.rng != cprev.rng)) ||
                ((arg & TIOCM_DSR) && (cnow.dsr != cprev.dsr)) ||
                ((arg & TIOCM_CD) && (cnow.dcd != cprev.dcd)) ||
                ((arg & TIOCM_CTS) && (cnow.cts != cprev.cts)) ) {
                return 0;
            }
            cprev = cnow;
        }
    }
    return -ENOIOCTLCMD;
}
```

Где-то в коде tty драйвера, который распознаёт изменения регистра MSR, для правильной работы этого кода должна быть вызвана следующая строка:

```
wake_up_interruptible(&tp->wait);
```

TIOCGICOUNT

Получает число прерываний. Это вызывается, когда пользователь хочет узнать, сколько прерываний случилось на последовательной линии. Если драйвер имеет обработчик прерываний, он должен определить внутреннюю структуру счетчиков, чтобы отслеживать такую статистику и увеличивать соответствующий счётчик каждый раз, когда функция выполняется ядром.

Этот вызов ***ioctl*** передаёт ядру указатель на структуру **serial_icounter_struct**, которая должна быть заполнена tty драйвером. Этот вызов часто выполняется в сочетании с предыдущим вызовом ***ioctl*** **TIOCMIWAIT**. Если tty драйвер во время работы драйвера сохраняет число всех этих прерываний, код для реализации этого вызова может быть очень простым:

```
static int tiny_ioctl(struct tty_struct *tty, struct file *file, unsigned int
cmd, unsigned long arg)
{
    struct tiny_serial *tiny = tty->driver_data;
    if (cmd == TIOCGICOUNT) {
        struct async_icount cnow = tiny->icount;
        struct serial_icounter_struct icount;
        icount.cts = cnow.cts;
        icount.dsr = cnow.dsr;
        icount.rng = cnow.rng;
        icount.dcd = cnow.dcd;
        icount.rx  = cnow.rx;
        icount.tx  = cnow.tx;
        icount.frame   = cnow.frame;
        icount.overrun = cnow.overrun;
        icount.parity  = cnow.parity;
        icount.brk = cnow.brk;
        icount.buf_overrun = cnow.buf_overrun;
        if (copy_to_user((void __user *)arg, &icount, sizeof(icount)))
            return -EFAULT;
```

```
        return 0;
    }
    return -ENOIOCTLCMD;
}
```

Обработка TTY устройствами proc и sysfs

Ядро tty предоставляет очень простой способ для любого tty драйвера обслуживать файл в каталоге */proc/tty/driver*. Этот файл создаётся, если драйвер определяет функции ***read_proc*** или ***write_proc***. Затем любой вызов чтения или записи этого файла отправляется в драйвер. Форматы эти функции точно такие же, у как стандартных функций обработки файлов в */proc*.

В качестве примера, простая реализация обратного вызова tty ***read_proc***, который просто выводит число портов, зарегистрированных в настоящее время:

```
static int tiny_read_proc(char *page, char **start, off_t off, int count, int
*eof, void *data)
{
    struct tiny_serial *tiny;
    off_t begin = 0;
    int length = 0;
    int i;

    length += sprintf(page, "tinyserinfo:1.0 driver:%s\n", DRIVER_VERSION);
    for (i = 0; i < TINY_TTY_MINORS && length < PAGE_SIZE; ++i) {
        tiny = tiny_table[i];
        if (tiny == NULL)
            continue;

        length += sprintf(page+length, "%d\n", i);
        if ((length + begin) > (off + count))
            goto done;
        if ((length + begin) < off) {
            begin += length;
            length = 0;
        }
    }
    *eof = 1;
done:
    if (off >= (length + begin))
        return 0;
    *start = page + (off-begin);
    return (count < begin+length-off) ? count : begin + length-off;
}
```

Ядро tty обрабатывает каталог sysfs и создание устройства, когда регистрируется tty драйвер, или когда созданы отдельные tty устройства, в зависимости от флага **TTY_DRIVER_NO_DEVFS** в **struct tty_driver**. Отдельный каталог всегда содержит файл ***dev***, который позволяет инструментам пользовательского пространства определять старший и младший номер, присвоенный устройству. Он также содержит символическую ссылку устройства и драйвера, если в вызове ***ty_register_device*** передан указатель на действительную **struct device**. Помимо этих трёх файлов, отдельные tty драйверы не могут создать в этом месте новые файлы в sysfs. Это, возможно, измениться в будущих версиях ядра.

Структура tty_driver в деталях

Структура ***tty_driver*** используется, чтобы зарегистрировать tty драйвер в ядре tty. Вот список всех различных полей в структуре и каким образом они используются в ядре tty:

struct module *owner;
Владелец модуля этого драйвера.

int magic;
"Магическое" (системное) значение для этой структуры. Всегда должно быть установлено в **TTY_DRIVER_MAGIC**. Инициализируется в функции ***alloc_tty_driver***.

const char *driver_name;
Имя драйвера, используемое в ***/proc/tty*** и sysfs.

const char *name;
Имя узла драйвера.

int name_base;
Начальный номер, используемый при создании имени для устройства. Это используется, когда ядро создаёт строковое представление данного tty устройства, связанного с этим tty драйвером.

short major;
Старший номер для драйвера.

short minor_start;
Начальный младший номер для драйвера. Он обычно установлен в то же значение, что **name_base**. Как правило, это значение равно 0.

short num;
Количество младших номеров, связанных с драйверов. Если драйвером используется весь диапазон старшего номера, это значение должно быть установлено в 255. Эта переменная инициализируется в функции ***alloc_tty_driver***.

short type;
short subtype;
Описывает, какой тип tty драйвера в настоящее время зарегистрирован в ядре tty. Значение **subtype** зависит от **type**. Поле **type** может быть:
TTY_DRIVER_TYPE_SYSTEM
Используется внутри подсистемы tty, чтобы запомнить, что она имеет дело с внутренним tty драйвером. subtype должен быть установлен в **SYSTEM_TYPE_TTY**, **SYSTEM_TYPE_CONSOLE**, **SYSTEM_TYPE_SYSCONS** или **SYSTEM_TYPE_SYSPTMX**. Этот тип не должен быть использован любым "обычным" tty драйвером.
TTY_DRIVER_TYPE_CONSOLE
Используется только консольным драйвером.
TTY_DRIVER_TYPE_SERIAL
Используется любым драйвер последовательного типа. **subtype** должен быть установлен в **SERIAL_TYPE_NORMAL** или **SERIAL_TYPE_CALLOUT**, в зависимости от типа вашего драйвера. Это один из наиболее часто используемых параметров для

поля **type**.

TTY_DRIVER_TYPE_PTY
Используется интерфейсом псевдо-терминала (pty). **subtype** должен быть установлен либо в **PTY_TYPE_MASTER**, либо в **PTY_TYPE_SLAVE**.

struct termios init_termios;
Первоначальные значения **struct termios** для данного устройства, когда оно создаётся.

int flags;
Флаги драйвера, как описывалось ранее в этой главе.

struct proc_dir_entry *proc_entry;
Структура записи драйвера в */proc*. Она создаётся ядром tty, если драйвер реализует функции *write_proc* или *read_proc*. Это поле не должно устанавливаться самим tty драйвером.

struct tty_driver *other;
Указатель на ведомый tty драйвер. Оно используется только pty драйвером и не должна использоваться любым другим tty драйвером.

void *driver_state;
Внутреннее состояние tty драйвера. Должно быть использовано только pty драйвером.

struct tty_driver *next;
struct tty_driver *prev;
Переменные связи. Эти переменные используются ядром tty для связывания в цепочку всех разных tty драйверов и не должна быть затронута любым tty драйвером.

Структура tty_operations в деталях

Структура *tty_operations* содержит все функции обратного вызова, который могут быть установлены tty драйверами и вызваны ядром tty. В настоящее время все указатели на функции, содержащиеся в этой структуре, также находятся в структуре **tty_driver**, но скоро это будет заменен экземпляром только этой структуры.

int (*open)(struct tty_struct * tty, struct file * filp);
Функция *open*.

void (*close)(struct tty_struct * tty, struct file * filp);
Функция *close*.

int (*write)(struct tty_struct * tty, const unsigned char *buf, int count);
Функция *write*.

void (*put_char)(struct tty_struct *tty, unsigned char ch);
Односимвольная функция *write*. Эта функция вызывается ядром tty, когда в устройство был записан один символ. Если tty драйвер не определил эту функцию, когда ядро tty хочет послать один символ, вместо неё вызывается функция *write*.

void (*flush_chars)(struct tty_struct *tty);
void (*wait_until_sent)(struct tty_struct *tty, int timeout);

Функция, которая сбрасывает данные в оборудование.

int (*write_room)(struct tty_struct *tty);
Функция, которая показывает, сколько в буфере свободного места.

int (*chars_in_buffer)(struct tty_struct *tty);
Функция, которая показывает, насколько буфер заполнен данными.

int (*ioctl)(struct tty_struct *tty, struct file * file, unsigned int cmd, unsigned long arg);
Функция ***ioctl***. Эта функция вызывается ядром tty, когда для этого узла устройства вызвана ***ioctl(2)***.

void (*set_termios)(struct tty_struct *tty, struct termios * old);
Функция ***set_termios***. Эта функция вызывается ядром tty, когда были изменены параметры termios устройства.

void (*throttle)(struct tty_struct * tty);
void (*unthrottle)(struct tty_struct * tty);
void (*stop)(struct tty_struct *tty);
void (*start)(struct tty_struct *tty);
Функции дросселирования данных. Эти функции используются, чтобы помочь управлению перерасходом входных буферов ядра tty. Функция ***throttle*** вызывается, когда входные буферы ядра tty стали полными. Драйвер tty должен попытаться просигнализировать устройству, что оно больше не должно посылать ему данные. Функция ***unthrottle*** вызывается, когда входные буферы ядра tty были опустошены и оно может теперь принимать данные. Затем tty драйвер должен просигнализировать устройству, что данные могут быть получены. Функции ***stop*** и ***start*** очень похожи на функции ***throttle*** и ***unthrottle***, но они показывают, что tty драйвер должен прекратить передачу данных в устройство и потом возобновить передачу данных.

void (*hangup)(struct tty_struct *tty);
Функция ***hangup***. Эта функция вызывается, когда tty драйвер должен дать отбой tty устройству. В это время должны произойти все необходимые для этого манипуляции оборудованием.

void (*break_ctl)(struct tty_struct *tty, int state);
Управляющая функция ***line break*** (разрыв строки). Эта функция вызывается, когда tty драйвер включает или выключает состояние линии BREAK для порта RS-232. Если состояние имеет значение -1, состояние BREAK должен быть "включено". Если состояние имеет значение 0, состояние BREAK должна быть "выключено". Если эта функция реализуется tty драйвером, ядро tty будет обрабатывать ioctl-ы **TCSBRK**, **TCSBRKP**, **TIOCSBRK** и **TIOCCBRK**. В противном случае, эти ioctl-ы отправляются в драйвер в функцию ***ioctl***.

void (*flush_buffer)(struct tty_struct *tty);
Сбрасывает буферы и теряет все оставшиеся данные.

void (*set_ldisc)(struct tty_struct *tty);
Функция ***set line discipline*** (установка дисциплины линии). Эта функция вызывается, когда ядро tty изменило дисциплину линии tty драйвера. Эта функция обычно не

используется и не должна быть определена драйвером.

void (*send_xchar)(struct tty_struct *tty, char ch);
Функция отправки ***символа X-типа***. Эта функция используется для передачи высокоприоритетного символа XON или XOFF в tty устройство. Символ для передачи указывается в переменной **ch**.

int (*read_proc)(char *page, char **start, off_t off, int count, int *eof, void *data);
int (*write_proc)(struct file *file, const char *buffer, unsigned long count, void *data);
Функции чтения и записи */proc*.

int (*tiocmget)(struct tty_struct *tty, struct file *file);
Получает текущие параметры линии заданного tty устройства. Если данные из tty устройства получены успешно, значение должно быть возвращено вызвавшему.

int (*tiocmset)(struct tty_struct *tty, struct file *file, unsigned int set, unsigned int clear);
Устанавливает текущие параметры линии заданного tty устройства. **set** и **clear** содержат различные настройки линии, которые должны быть либо установлены, либо сброшены.

Структура tty_struct в деталях

Переменная **tty_struct** используется ядром tty для сохранения текущего состояния заданного tty порта. Почти все из этих полей, будут использоваться только ядром tty, с несколькими исключениями. Поля, которые может использовать tty драйвер, описаны здесь:

unsigned long flags;
Текущее состояние tty устройства. Она является битовым полем и доступна через следующие макросы:

TTY_THROTTLED
Установлен, когда в драйвере вызвана функция ***throttle***. Не должен устанавливаться tty драйвером, только ядром tty.

TTY_IO_ERROR
Устанавливается драйвером, когда он не хочет, чтобы никакие данных не читались или не записывались в драйвер. Если пользовательская программа пытается сделать это, она получает от ядра ошибку **-IO**. Он обычно устанавливается, когда устройство выключается.

TTY_OTHER_CLOSED
Используется только pty драйвером, чтобы сообщить, когда был закрыт порт.

TTY_EXCLUSIVE
Устанавливается ядром tty, чтобы указать, что порт находится в монопольном режиме и может быть доступен только одному пользователю одновременно.

TTY_DEBUG
В ядре нигде не используется.

TTY_DO_WRITE_WAKEUP
Если он установлен, разрешён вызов функции ***write_wakeup*** дисциплины линии. Она обычно вызывается в то же время, когда tty драйвером вызывается функция ***wake_up_interruptible***.

TTY_PUSH
Используется только внутренне дисциплиной по умолчанию tty линии.

TTY_CLOSING
Используется ядром tty для отслеживания, находится ли порт в процессе закрытия в этот момент времени или нет.

TTY_DONT_FLIP
Используется дисциплиной по умолчанию tty линии для уведомления ядра tty, что оно не должно изменять переключаемый буфера при его установке.

TTY_HW_COOK_OUT
Если установлен tty драйвером, он уведомляет дисциплину линии, что будет "готовить" вывод для отправки в неё. Если он не установлен, то дисциплина линии копирует вывод драйвера по кусочкам; в противном случае, она должна оценить каждый посланный байт индивидуально для изменения линии. Этот флаг обычно не устанавливается tty драйвером.

TTY_HW_COOK_IN
Почти идентичен установке флага **TTY_DRIVER_REAL_RAW** в переменной драйвера **flags**. Этот флаг обычно, не устанавливается tty драйвером.

TTY_PTY_LOCK
Используется pty драйвером для блокировки и разблокировки порта.

TTY_NO_WRITE_SPLIT
Если установлен, ядро tty не разбивает запись в tty драйвер на куски обычных размеров. Это значение не должно быть использовано для предотвращения атак типа "отказ в обслуживании" на tty порты путём отправки в порт больших объёмов данных.

struct tty_flip_buffer flip;
Переключаемый буфер для tty устройства.

struct tty_ldisc ldisc;
Дисциплина линии для tty устройства.

wait_queue_head_t write_wait;
wait_queue для функции записи tty. Драйвер tty должен пробудить её, чтобы просигнализировать, когда он может получать больше данных.

struct termios *termios;
Указатель на текущие настройки **termios** для tty устройства.

unsigned char stopped:1;
Показывает, является ли tty устройство остановленным. Драйвер tty может установить это значение.

unsigned char hw_stopped:1;
Показывает, остановлено или нет оборудование tty устройства. Драйвер tty может установить это значение.

unsigned char low_latency:1;
Показывает, является ли это tty устройство устройством в небольшой задержкой, способным принимать данные с очень высокой скоростью. Драйвер tty может установить это значение.

unsigned char closing:1;
Показывает, находится ли tty устройство в середине закрытия порта. Драйвер tty может установить это значение.

struct tty_driver driver;
Текущая структура **tty_driver**, которая управляет этим tty устройством.

void *driver_data;
Указатель, который **tty_driver** может использовать для сохранения локальных данных для tty драйвера. Эта переменная не модифицируется ядром tty.

Краткая справка

Этот раздел содержит ссылки на понятия, введённые в этой главе. Он также разъясняет роль каждого заголовочного файла, который необходимо подключить tty драйверу. Однако, списки полей в структурах **tty_driver** и **tty_device** здесь не повторяются.

#include <linux/tty_driver.h>
Заголовочный файл, который содержит описание **struct tty_driver** и декларирует некоторые из различных флагов, используемых в этой структуре.

#include <linux/tty.h>
Заголовочный файл, который содержит описание **struct tty_struct** и ряд различных макросов для простого доступа к отдельным значениям полей **struct termios**. Он также содержит декларации функций драйверного ядра tty.

#include <linux/tty_flip.h>
Заголовочный файл, который содержит некоторые встраиваемые функции переключаемого буфера tty, которые упрощают манипулирование структурами переключаемого буфера.

#include <asm/termios.h>
Заголовочный файл, который содержит описание **struct termio** для заданной аппаратной платформы, для которой собирается ядро.

struct tty_driver *alloc_tty_driver(int lines);
Функция, которая создаёт **struct tty_driver**, которая может быть позже передана в функции ***tty_register_driver*** и ***tty_unregister_driver***.

void put_tty_driver(struct tty_driver *driver);
Функция, которая очищает структуру **struct tty_driver**, которая не была успешно зарегистрирована в ядре tty.

void tty_set_operations(struct tty_driver *driver, struct tty_operations *op);
Функция, которая инициализирует функции обратного вызова из **struct tty_driver**. Её необходимо вызвать до вызова ***tty_register_driver***.

int tty_register_driver(struct tty_driver *driver);
int tty_unregister_driver(struct tty_driver *driver);
Функции, которые регистрируют и отменяют регистрацию tty драйвера в ядре tty.

void tty_register_device(struct tty_driver *driver, unsigned minor, struct device *device);
void tty_unregister_device(struct tty_driver *driver, unsigned minor);
Функции, которые регистрируют и отменяют регистрацию одного tty устройства в ядре tty.

void tty_insert_flip_char(struct tty_struct *tty, unsigned char ch, char flag);
Функция, которая вставляет символы в переключаемый буфер tty устройства для чтения пользователем.

TTY_NORMAL
TTY_BREAK
TTY_FRAME
TTY_PARITY
TTY_OVERRUN
Различные значения параметра флагов, используемые в функции ***tty_insert_flip_char***.

int tty_get_baud_rate(struct tty_struct *tty);
Функция, которая получает установленную в настоящее время скорость передачи данных

для заданного tty устройства.

void tty_flip_buffer_push(struct tty_struct *tty);

Функция, которая заталкивает данные для пользователя в текущий переключаемый буфер.

tty_std_termios

Переменная, которая инициализирует структуру **termios** общим набором параметров по умолчанию для линии.

Индекс

В

Г

Д

З

И

К

Л

М

Н

О

П

P

C

T

У

Ф

Ш