Модуль 3. Модули ядра, пространства и устройства

Модули ядра и прикладные программы

Прежде чем продолжать изучение программирования драйверов в Linux, следует подчеркнуть некоторые отличия между структурой и особенностями функционирования модулей ядра и прикладными программами.

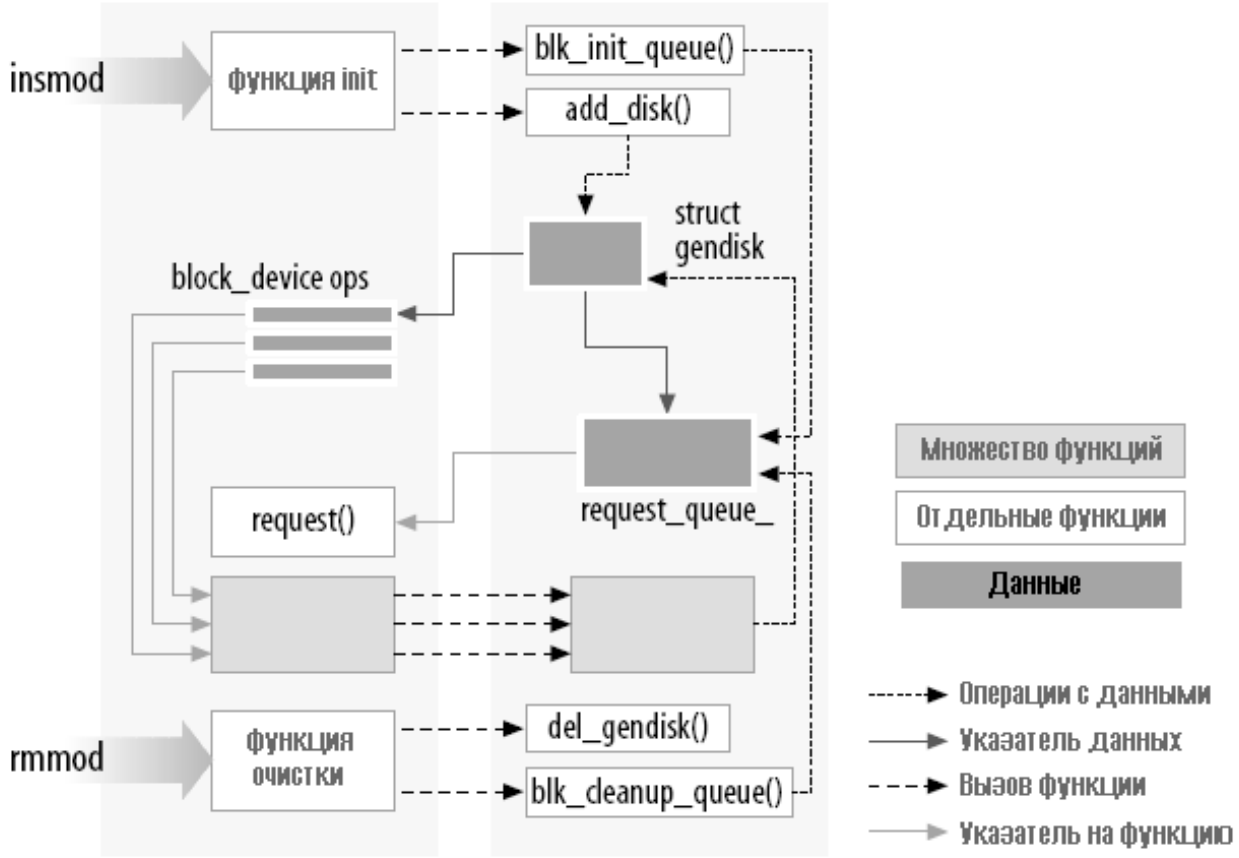
Большинство прикладных программ последовательно выполняют от начала до конца одну задачу. В отличае от программ модули ядра просто регистрируют себя в ядре для того, чтобы обслуживать в будущем запросы, и выполнение их функций инициализации немедленно прекращается. Иными словами, задача функции инициализации модуля заключается в подготовке функций модуля для последующего вызова; модуль как бы сообщает о себе: "Вот я и вот что я могу делать". Функция выгрузки модуля вызывается только непосредственно перед выгрузкой модуля. Она сообщает ядру: "Меня больше нет; не просите меня сделать что-нибудь ещё". Такой подход к программированию подобен программированию на основе обработки событий. Однако, в отличае от модулей ядра, пока не все приложения управляются событиями.

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, которые 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 программисты, 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 функции printf с небольшими отличиями, главным из которых является отсутствие поддержки плавающей точкой. На следующем рисунке показано, как используются в модуле вызовы функций и указатели на функции, чтобы добавить ядру новую функциональность.



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.

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.

Функции, которые доступны из модулей

Стоит перечислить особенности, общие для всех функций API ядра, которые обеспечивают модулю интерфейс для использования всех доступных возможностей ядра.

Эти функции реализованы в ядре, и даже при совпадении по форме с вызовами стандартной библиотеки языка C (например, вызов sprintf()) - это совершенно другие функции. Не стоит обманываться внешней похожестью или «почти похожестью», даже при сходной функциональности они могут отличаться «трудноуловимыми» деталями реализации. Заголовочные файлы для функций пространства пользователя располагаются в /usr/include, а для API ядра — в совершенно другом месте, в каталоге/lib/modules/`uname -r`/build/include.

Разработчики ядра Linux не связаны требованиями совместимости снизу вверх, в отличие от очень жёстких ограничений для пользовательских API, налагаемых стандартом POSIX. Поэтому API ядра может изменяться даже между подверсиями ядра. Функции ядра довольно плохо документированы (по крайней мере, в сравнении с документацией POSIX-вызовов для пользовательского пространства). Источниками для изучения особенностей реализации функций API ядра могут служить (в порядке приоритета и полезности):

- Прототипы и определения из заголовочных файлов в дереве каталогов /lib/modules/`uname -r`/build/include.
- Комментарии в тех же заголовочных файлах.
- Файлы формата .txt из каталога «Documentation» в дереве исходных кодов ядра, если это дерево установлено в системе (при инсталляции системы оно не устанавливается и также не устанавливается из репозитариев дистрибутива). Но на самом деле — это не полноценная документация, а только заметки к реализации: иногда там можно найти весьма полезные мелочи, иногда — ничего.
- Программные файлы ядра (.c) из того же дерева исходных кодов ядра. Это самый достоверный источник информации, но добыть из него информацию не так и просто.

Других источников информации по API ядра, которым можно было доверять, фактически не существует, так как все остальные источники или устаревшие или ошибочные и т.д.

Существует общее правило (которое, правда, не всегда соблюдается), что функции API ядра в случае ошибки выполнения возвращают отрицательный результат завершения (код ошибки). Положительные возвращаемые результаты в API ядра используются для возврата численных результатов, а возвращаемое нулевое

значение — как логический признак в некоторых API. Если вспомнить соглашения POSIX API, то они радикально отличаются: зачастую POSIX-функции помещают положительный код ошибки в errno. В качестве отрицательного результата при возникновении ошибки API ядра возвращают те же числовые кодыошибок, что и POSIX, но со знаком минус. Этого же правила рекомендуют придерживаться и при написании собственных функций в теле модуля, в частности, функции инициализации модуля. Такие соглашения существуют в пространстве ядра.

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.

Обычно говорят 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.

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. Рассмотрим некоторые причины, почему вы можете написать драйвер в пользовательском пространстве.

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. 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, 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.

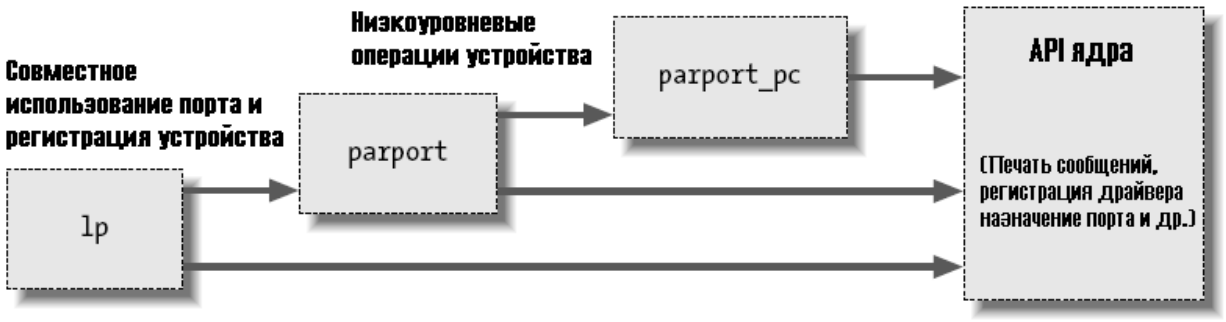
Случаем, в котором работа в пространстве пользователя может иметь смысл, является тот, когда вы начинаете иметь дело с новым и необычным оборудованием. Таким образом вы можете научиться управлять вашим оборудованием без риска подвешивания системы в целом. После того, как вы сделали это, выделение этого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

Пространство имен

После сборки модуля следующим шагом является загрузка его в ядро. Эту работу для нас делает команда 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 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.

Как было сказано выше, insmod разрешает неопределённые символы на таблицу публичных символов ядра. В таблице содержатся адреса глобальных объектов ядра - функций и переменных, которые необходимы для выполнения драйверов-модулей ядра. 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 модулей поверх других модулей (стек модулей). Стек модулей встроен также и в исходнике основного ядра: файловая система ms-dos 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 следующем рисунке; Стрелки показывают связи между модулями и интерфейсом программирования ядра.



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.

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“), который используется ядром во время загрузки, чтобы найти переменные, экспортируемые модулем.

Адресное пространство

В операционной системе Linux память пользователя и память ядра являются независимыми друг от друга и расположены в различных адресных пространствах. Проще говоря, адресные пространства виртуализированы, т. е. абстрагированы от физической памяти. Поскольку адресные пространства являются виртуальными, в системе их может быть много. Фактически само ядро и каждый из процессов располагаются в своих собственных изолированных адресных пространствах. Эти адресные пространства состоят из адресов виртуальной памяти, что позволяет нескольким процессам с независимыми адресными пространствами обращаться к физическому адресному пространству (т. е. к памяти, физически установленной на устройстве), имеющему значительно меньший объем. Такой подход обеспечивает не только удобство, но и безопасность, поскольку все адресные пространства являются независимыми и изолированными друг от друга.

Однако такая безопасность не дается даром. Поскольку ядро и каждый из процессов могут иметь одинаковые адреса, ссылающиеся на различные области физической памяти, при использовании общей памяти могут возникать задержки. К счастью, существует несколько решений этой проблемы. Совместное использование памяти пользовательскими процессами может обеспечиваться с помощью POSIX-механизма совместно используемой памяти shmem (shared memory mechanism), причем процессы могут иметь различные виртуальные адреса, ссылающиеся на одну и ту же область физической памяти.

Отображение виртуальной памяти на физическую осуществляется через аппаратно реализованные страничные таблицы.

Аппаратная часть непосредственно выполняет отображение, а ядро управляет таблицами и их конфигурацией. Обратите внимание на то, что процесс может иметь большое, но разбросанное адресное пространство (как показано на рисунке); это означает, что небольшие области (страницы) адресного пространства отображаются на физическую память через страничные таблицы. Это позволяет процессу иметь большое адресное пространство, занятое только теми страницами, с которыми необходимо работать в текущий момент.

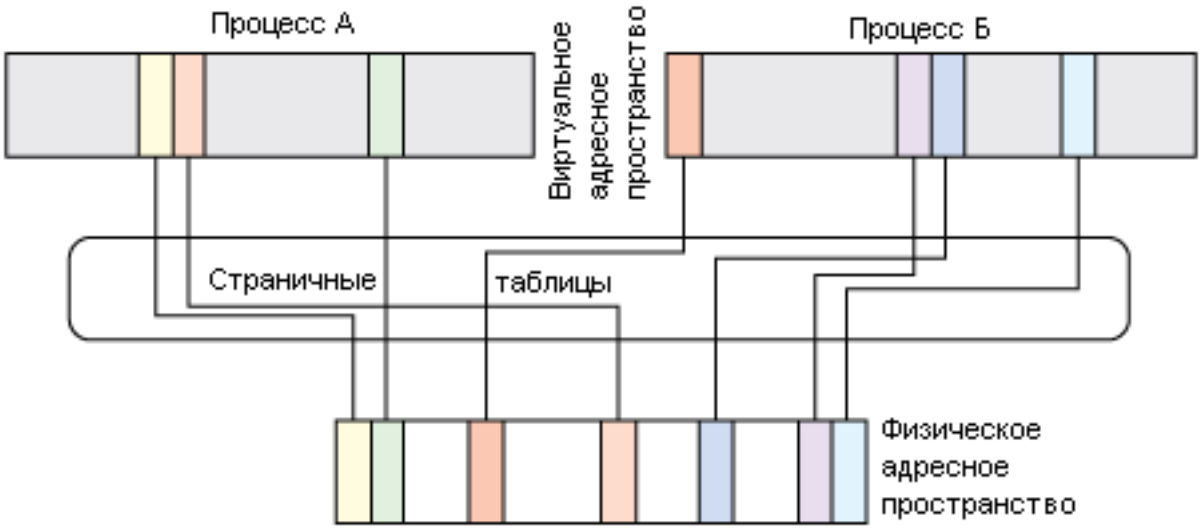
Старший и младший номер устройства

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.