Родригес К. З.,

Фишер Г., Смолски С.

Linux: азбука ядра

Последовательное рассмотрение на архитектурах x86 и PowerPC

КУДИЦ-ПРЕСС
2007

ББК 32.973.26-018.2

Родригес К. З., Фишер Г., Смолски С.
Linux: азбука ядра / Пер. с англ. - М.: КУДИЦ-ПРЕСС, 2007. - 584 с.

Книга представляет собой справочное руководство по администрированию сетей і среде Linux. Она будет интересна как начинающим, так и опытным пользователям все сторонним анализом популярных служб в системах Linux, описанием важнейших сете вых программ и утилит. Подробная информация по конфигурации и администрирова нию компонентов сети позволит администратору организовать работу в сети Linux н; качественно ином уровне.

Диапазон рассматриваемых тем широк. Обстоятельный подход авторов и продуман ная структура книги облегчит задачи, стоящие перед администратором.

Клаудия Зальзберг Родригес, Гордон Фишер, Стивен Смолски
Linux: азбука ядра

Перевод с англ. Легостаев И. В.
Научный редактор Мурашко И. В.

ООО «КУДИЦ-ПРЕСС»
190068, С.-Петербург, Вознесенский пр-т, д. 55, литер А, пом. 44.
Тел.: 333-82-11, ok@kudits.ru; *http://books.kudits.m*

Подписано в печать 15.01.2007.
Формат 70x90/16. Бум. офс. Печать офс.
Усл. печ. л. 42,7. Тираж 2000. Заказ 41

Отпечатано в ОАО «Щербинская типография»
117623, Москва, ул. Типографская, д. 10
Тел.: 659-23-27

ISBN 978-5-91136-017-7 (рус.)
ISBN 0-596-00794-9

Содержание

Введение

Здесь должны быть драконы. Так писали средневековые картографы о неизвестных или опасных территориях, и подобное же чувство вы испытываете, когда впервые вводите строку

```
cd /usr/src/linux ; Is
```

«С чего начать?» - думаете вы. «Что именно я здесь ищу? Как все это связано между собой и как работает на самом деле?»

Современные полнофункциональные операционные системы большие и сложные. Подсистем много, и зачастую они взаимодействуют друг с другом слишком тонко. И несмотря на то что *иметь* исходные коды ядра Linux (о котором немного позже) - это здорово, совершенно непонятно с чего начать, а также что и в какой последовательности искать.

Для этого и написана данная книга. Шаг за шагом вы узнаете про различные компоненты ядра, о том, как они работают и как связаны друг с другом. Авторы отлично знакомы с ядром и хотят поделиться своими знаниями; к концу прочтения этой книги вы станете хорошими друзьями ядра Linux с углубленным пониманием связей.

Ядро Linux - это «свободное» (от слова *свобода)* программное обеспечение. В Декларации свободного программного обеспечения *{The Free Software Definition*^ Ричард Столман определил свободы, которые делают программное обеспечение Свободным (с большой буквы). Свобода 0 - это свобода на запуск программного обеспечения. Это самая главная свобода. Сразу за ней следует Свобода 1, свобода изучения того, как программное обеспечение работает. Зачастую эта свобода упускается из виду. Тем не менее это очень важно, потому что лучший способ обучения - это наблюдение за тем, как это делают другие. В мире программного обеспечения это означает чтение программ других и выяснение того, что они сделали хорошо, а что плохо. Свободы GPL, по крайней мере на мой взгляд, одна из главных причин того, что GNU/Linux-системы стали такой важной силой среди современных информационных технологий. Эти свободы помогают вам каждый раз, когда вы используете вашу GNU/Linux-систему, и поэтому неплохой идеей будет остановиться и осмыслить каждую из них.

В этой книге мы воспользуемся преимуществами Свободы 1, позволяющей нам изучать глубины исходных кодов ядра Linux. Вы увидите, что одни вещи сделаны хорошо, а другие, скажем так, *менее хорошо.* Тем не менее благодаря Свободе 1 *вы имеете возможность все это увидеть* и почерпнуть для себя много нового.

1. http://www.gnu.org/philosophy/free-sw.html.

Это возвращает меня к серии книг *Prentice Hall Open Source Software Development,* к которой принадлежит и данная книга. Идея создания этой серии основана на том, что чтение - один из лучших способов обучения. Сегодня, когда мир осчастливлен обилием свободного и открытого программного обеспечения, его исходный код только и ждет (даже жаждет!), чтобы его прочитали, поняли и оценили. Целью этой серии является облегчение кривой обучения разработке программного обеспечения, так сказать, помочь вам в обучении с помощью демонстрации как можно более реального кода.

Я искренне надеюсь, что эта книга вам понравится и поможет в обучении. Также я надеюсь, что она вдохновит вас занять свое место в мире свободного программного обеспечения и открытых исходных кодов, что является наиболее интересным способом принять участие в этом процессе.

Удачи!

Арнольд Роббинс,
редактор серии

Благодарности

Мы хотим поблагодарить множество людей, без которых эта книга не могла бы быть написана.

Клаудия Зальзберг Родригес: Я хочу заметить, что зачастую сложно при недостатке места выделить главных помощников в достижении широкоизвестного успеха среди массы людей, которые неисчислимым и несчетным количеством способов помогали тебе достичь этого успеха. Поэтому я хочу поблагодарить всех помощников ядра Linux за их тяжелый труд, который они посвятили разработке той операционной системы, которой она стала сейчас, - за любовь к игре. Я испытываю глубочайшую признательность по отношению к ключевым учителям и наставникам на пути от пробуждения и поощрения ненасытной любознательности о том, как работают различные вещи, до обучения меня тому, как в них разбираться. Также я хочу поблагодарить свою семью за ее неизменную любовь, поддержку и оптимистический настрой в тех ситуациях, когда я уже готова была сдаться. Наконец, я хочу поблагодарить Йоза Рауля за бережное отношение к моему времени и за умение каждый раз находить способ подбодрить меня в сложной ситуации. **Гордон Фишер:** Я хочу поблагодарить всех программистов, которые терпеливо разъясняли мне сложности ядра Linux, когда я еще был новичком. Также я хочу поблагодарить Grady и Underworld за отличную кодерскую музыку.

Также мы хотим поблагодарить нашего главного редактора Марка Л. Тауба за знание того, что нужно для того, чтобы сделать книгу лучше на каждом шаге, и за то, что он вел нас в правильном направлении. Спасибо за последовательность и одновременную разумность, понимание, требовательность и безмерную открытость на протяжении написания этой книги. Кроме этого, мы хотим поблагодарить Джима Маркхема и Эрику Джемисон. Джиму Маркхему мы благодарны за его ранние редакторские комментарии, которые сослужили нам хорошую службу в течение написания оставшейся части рукописи. Эрике Джемисон мы благодарны за редакторские отзывы во время написания последней версии рукописи. Нашу благодарность заслуживают и рецензенты, которые потратили множество часов на чтение и высказывание замечаний, которые помогли сделать книгу лучше. Спасибо вам за зоркий глаз и проницательные комментарии; ваши замечания и комментарии неоценимы. Рецензентами были Алессио Гаспар, Мел Горман, Бенджамин Херенштмидт, Рон Мак-Карти, Чет Рейми, Эрик Рейнольде, Арнольд Робинсоне и Питер Салус. Мы хотим поблагодарить Кайлу Даджер за то, что она помогла нам с литературной обработкой и корректурой, с неизменно хорошим настроением, и Джинни Бесс за ее орлиный корректорский глаз. Отдельное спасибо армии людей, отвечавших за корректуру, литературную обработку, верстку, маркетинг и печать и с которыми мы не встречались лично, за то, что создание этой книги стало возможным.

Об авторах

Клаудия Зальзберг Родригес работает в Центре Linux технологий ГВМ, где занимается разработкой ядра и связанных с ним утилит программирования. Является системным программистом Linux более 5 лет. Работала с Linux для Intel и PPC на различных платформах начиная со встраиваемых систем и заканчивая высокопроизводительными системами.

Гордон Фишер писал драйверы под Linux и UNIX для многих низкоуровневых устройств и использовал множество специализированных ядер Linux для Intel- и PPC-платформ.

Стрив Смолски 26 лет в полупроводниковом бизнесе. Он работал на производстве, тестировании, участвовал в разработке памяти, процессоров и ASICS; писал приложения и драйверы для Linux, ATX, Windows; работал со встроенными операционными системами.

Предисловие

Технология в целом и компьютеры в частности обладают магической привлекательностью, которая, кажется, поглощает тех, кто к ним приближается. Технологические разработки раздвигают установленные рамки и заставляют пересмотреть ненадежные идеи, бывшие ранее нежизнеспособными. Операционная система Linux стала главным помощником для потока значительных изменений в индустрии и способов ведения бизнеса. Принятие Публичной лицензии GNU (GNU Public License) и взаимодействие с GNU-программным обеспечением вызвало множество дебатов вокруг открытых исходных кодов, свободного программного обеспечения и концепции сообщества разработчиков. Linux - это чрезвычайно удачный пример того, насколько мощной может быть операционная система с открытым кодом и как эта основа позволяет своей привлекательностью удерживать программистов всего мира.

В настоящее время расширяется доступность использования Linux для большинства пользователей компьютеров. Со множеством дистрибутивов, поддержкой сообщества и с помощью индустрии Linux нашел тихую гавань для своего применения в университетах, индустрии и домах миллионов пользователей.

Рост использования приводит к увеличению необходимости в поддержке и новой функциональности. В свою очередь, все больше и больше программистов начинают интересоваться «внутренностями» ядра Linux по мере того, как новые архитектуры и устройства официально добавляются к уже обширному (и стремительно растущему) арсеналу.

Портирование ядра Linux на Power-архитектуру способствовало расцвету операционной системы среди высокопроизводительных серверов и встроенных систем. Необходимость понимания того, как Linux работает на Power-архитектуре, растет, потому что растет число компаний, которые закупают основаные на PowerPC системы для работы под Linux.

Предполагаемая аудитория

Эта книга предназначена как начинающим, так и бывалым системным программистам, энтузиастам Linux и разработчикам прикладных программ, желающим лучше понимать, что заставляет их программы работать и благодаря чему это возможно. Каждый, кто знает C, знаком с основными принципами Linux и хочет разобраться в том, как устроен Linux, найдет в этой книге описание базовых концепций, необходимых для его сборки и понимания. Таким образом, книга представляет собой букварь для понимания работы ядра Linux.

Неважно, ограничиваются ли ваши знания Linux умением зайти в систему и написать маленькую программу под Linux, или вы состоявшийся системный программист, пытающийся понять принципы работы одной из подсистем, - эта книга предоставит вам необходимую информацию.

Организация материала

Эту книгу можно условно разделить на три части, каждая из которых предоставляет читателю необходимые для изучения «внутренностей» Linux знания.

Часть 1 включает необходимые инструменты и знания для обзора строения ядра.

Гл. 1, «Обзор», содержит историю Linux и UNIX, список дистрибутивов и короткий обзор различных подсистем ядра с точки зрения пользователя.

Гл. 2, «Исследовательский инструментарий», содержит описание структур данных и язык, используемый применительно к ядру Linux. Кроме того, в ней вы найдете введение в ассемблер для архитектур x68 и PowerPC, а также обзор инструментов и утилит, используемых для получения информации, необходимой для понимания работы ядра.

Часть 2 предлагает читателю базовые концепции каждой подсистемы ядра и комментирует код, реализующий функциональность подсистем.

Гл. 3, «Процессы: принципиальная модель выполнения», описывает реализацию модели процессов. Мы увидим, как процессы запускаются, и обсудим процесс контроля пользовательских процессов из ядра и наоборот. Кроме этого, мы обсудим, как процессы реализованы в ядре, и обсудим структуры данных, связанные с выполнением процесса. Также эта глава описывает прерывания и исключения, а также их реализацию на каждой из архитектур и то, как они взаимодействуют с ядром Linux.

Гл. 4, «Управление памятью», описывает, как ядро Linux отслеживает и распределяет доступную память между пользовательскими процессами и ядром. Эта глава содержит описание способов классификации памяти и того, как принимаются решения о выделении и освобождении памяти. Также подробно описан механизм ошибок памяти и его аппаратная реализация.

Гл. 5, «Ввод-вывод», описывает взаимодействие процессора с другими устройствами, соответствующие интерфейсы ядра и управление этим взаимодействием. Также эта глава описывает различные виды устройств и их реализацию в ядре.

Гл. 6, «Файловые системы», содержит обзор реализации в ядре файлов и директорий. В этой главе представлена виртуальная файловая система и абстрактный слой для поддержки различных файловых систем. Также эта глава описывает реализацию связанных с файлами операций, таких, как операции открытия и закрытия файлов.

Гл. 7, «Планировщик и синхронизация ядра», описывает операции планировщика, позволяющие нескольким процессам выполняться таким образом, как будто этот процесс в системе единственный. Глава подробно описывает, как ядро выбирает, какой процесс выполнять, и как оно взаимодействует с аппаратной частью для переключения с одного

процесса на другой. Кроме этого, в главе описаны приоритеты и их реализация. И наконец, она описывает работу системного таймера и того, как его использует ядро для слежения за временем.

Гл. 8, «Загрузка ядра», описывает то, что происходит с момента включения и до момента выключения системы Linux. Она содержит описание различных процессорных обработчиков для загрузки ядра, включая описание BIOS, Open Firmware и загрузчиков. Глава отслеживает линейный процесс запуска и инициализации ядра, включая все подсистемы, описанные в предыдущих главах.

Часть 3 касается более ручного подхода к сборке ядра Linux и работе с ним.

Гл. 9, «Построение ядра Linux», описывает перечень утилит, необходимых для сборки ядра, и формат исполнимых объектных файлов. Также подробно описана работа системы построения ядра из исходников (Kernel Source Build) и варианты ее настройки.

Гл. 10, «Добавление вашего кода в ядро», описывает работу устройства /dev/random, которое можно увидеть в любой Linux-системе. Работа с устройствами уже была описана ранее с более практической точки зрения. Здесь же описано то, как можно добавить в ядро поддержку собственного устройства.

Наш подход

Эта книга предлагает вниманию читателя концепции, необходимые для понимания ядра. Мы используем подход сверху вниз следующими двумя способами.

Во-первых, мы ассоциируем работу ядра с исполнением операций в пользовательском пространстве, с которыми читатель, скорее всего, более знаком, и постараемся объяснить работу ядра на этих примерах. Если это возможно, мы будем начинать с примеров в пользовательском пространстве, а затем будем прослеживать их выполнение в коде до уровня ядра. Углубиться напрямую возможно не всегда, так как перед этим для понимания работы нужно рассказать о типах данных подсистем и подструктурах. В этом случае мы постараемся объяснить подсистему ядра на специальном примере и укажем связь с пространством пользовательских программ. Это дает двойной эффект: демонстрирует слои работы ядра и их взаимодействие с пользовательским пространством, с одной стороны, и с аппаратной частью, с другой, а также позволяет объяснить работу подсистем, проследив прохождение событий по мере их поступления. Мы считаем, что это позволит читателю получить представление о том, как ядро работает, как это сочетается с тем, что мы уже знаем, и как оно соотносится с работой остальной операционной системы.

Во-вторых, мы используем подход сверху вниз для обзора структур данных, общих для операций подсистем, и увидим, как они связаны с управлением работой системы. Мы постараемся наметить основные структуры для подсистемных операций и будем акцентировать на них свое внимание во время рассмотрения операций подсистем.

Соглашения

На протяжении этой книги вы увидите листинги исходного кода. Верхний правый угол будет содержать расположение исходного файла относительно корня дерева исходных кодов. Листинг будет напечатан таким шрифтом. Номера строк приведены для удобства комментирования кода. При описании подсистем ядра и того, как они работают, мы будем часто обращаться к исходному коду и пояснять его.

Опции командной строки, имена функций, результат выполнения функций и имена переменных будут напечатаны таким шрифтом.

Жирный шрифт используется при введении нового понятия.

Глава 1

Обзор

В этой главе:

Linux - это операционная система, которая появилась на свет благодаря хобби студента по имени Линус Товальдс в 1991 г. Тогда Linux был скромным и непритязательным по сравнению с тем, во что он превратился теперь. Linux был разработан для работы на процессорах x86-архитектуры с жестким диском AT. Первая версия включала командный интерпретатор bash и компилятор gcc. В то время еще не ставилась задача переносимости, равно как и широкое распространение в академических и индустриальных кругах. Не было никаких бизнес-планов или стратегий. Тем не менее с самого начала система была бесплатной.

Linux превратился в командный проект под руководством и при поддержке Линуса еще со времен ранней бета-версии. Он занял нишу операционной системы для хакеров, которым хотелось бы иметь бесплатную операционную систему для архитектуры x86. Эти хакеры разрабатывали код, добавлявший в систему поддержку необходимой им функциональности.

Часто говорят, что Linux - это разновидность UNIX. Технически Linux является клоном UNIX, потому как реализует спецификацию POSIX UNIX PI003.0. UNIX доминировал на неинтеловских платформах с момента своего появления в 1969 г. и заслуженно считается мощной и элегантной операционной системой. Относительно высокопроизводительных рабочих станций UNIX являлся единственной системой для исследовательских, академических и индустриальных нужд. Linux привнес возможности UNIX на Intel-персональные компьютеры домашних пользователей. Сегодня Linux широко используется в индустрии и образовании, а также поддерживает множество архитектур, включая PowerPC.

В этой главе представлен обзор с высоты птичьего полета основных концепций Linux. Она познакомит вас с обзором компонентов и возможностей ядра и представит некоторые особенности, которые делают Linux таким привлекательным. Для понимания концепции ядра Linux вам нужно получить базовые представления о его предназначении.

1.1 История UNIX

Мы упомянули, что Linux - это разновидность UNIX. Тем не менее Linux не был разработан на основе существующего UNIX, однако сам факт того, что он реализует стандарты UNIX, заставляет нас обратить внимание на историю UNIX.

MULTiplex Information and Computing Service (MULTICS)[1], которая считается предшественником операционной системы UNIX, возникла благодаря совместному начинанию Массачусетского технологического института (MIT), Bell Laboratories и General Electric Company (GEC), которая в тот период была вовлечена в бизнес по производству компьютеров. Разработка MULTICS была запущена благодаря желанию создать машину, которая

[1] Сложная информационная и компьютерная служба. *Примеч. пер.*

могла бы поддерживать одновременную работу сразу нескольких пользователей. Во времена этого партнерства, в 1965-м, операционные системы, несмотря на возможность **мультипрограммирования** (разделения времени между программами) и существование систем пакетной обработки, поддерживали работу только с одним пользователем. Время отклика между выдачей системе задания и получением результатов измерялось часами. Целью MULTICS было создание операционной системы, которая позволяла бы **многопользовательское разделение времени,** предоставляя в распоряжение каждого пользователя отдельный терминал. Благодаря тому что Bell Labs и General Electric со временем прекратили проект, наработки MULTICS были использованы во множестве других проектов.

Разработки UNIX началась с портирования упрощенной версии MULTICS для создания операционной системы для миникомпьютера PDP-7, который должен был поддерживать новую файловую систему. Эта новая файловая система была первой файловой системой UNIX. Эта операционная система, разработанная Кеном Томпсоном, поддерживала двух пользователей и обладала командным интерпретатором и программами для манипуляции с файлами в новой файловой системе. В 1970 г. UNIX был портирован на PDP-11 и получил поддержку большего количества пользователей. Технически это была первая версия UNIX.

В 1973 г. при создании четвертой версии UNIX Кен Томпсон и Денис Ритчи переписали UNIX на C (язык, недавно разработанный Ритчи). Это позволило операционной системе отойти от ассемблера и открыло двери портированию операционной системы. Задумайтесь над революционностью этого решения. До этого операционные системы были тесно связаны с архитектурными спецификациями систем, потому что язык ассемблера был слишком индивидуален для того, чтобы портировать его на другие платформы. Перенос UNIX на C стал первым шагом к лучшей портируемости (и читабельности) операционных систем, шагом, благодаря которому UNIX приобрела такую популярность.

1974 г. отмечен началом роста популярности UNIX среди университетов. Академики начали сотрудничать с группой систем UNIX из Bell Laboratories для создания новой, во многом инновационной версии. Эта версия распространялась между университетами в образовательных целях бесплатно. В 1979-м, после множества нововведений, подчистки кода и усилий по улучшению переносимости, появилась седьмая версия (V7) операционной системы UNIX. Эта версия включала в себя компилятор C и командный интерпретатор, известный как Bourn shell.

1980 г. ознаменовался расцветом персональных компьютеров. Рабочие станции были установлены на многих предприятиях и в университетах. На основе седьмой версии UNIX было разработано несколько новых вариантов. В том числе Berkley UNIX, разработанный в Калифорнийском университете в Беркли, и разработанные AT&T UNIX System Ш и V. Каждая версия превратилась в отдельную систему, такую, как NetBSD и OpenBSD (варианты BSD) и AIX (вариант System V от IBM). При этом все коммерческие версии UNIX берут свое начало от System V и BSD.

Linux появился в 1991 г., когда UNIX был чрезвычайно популярен, но недоступен на PC. Стоимость UNIX была настолько высокой, что была недоступна большинству пользователей за исключением тех, кто был связан с университетами. Изначально Linux представлял собой усовершенствованную версию Minix (простую операционную систему, написанную Эндрю Таненбаумом в образовательных целях).

В последующие годы ядро Linux[1] вместе с системным программным обеспечением, предоставленным GNU-проектом, Фонд свободного программного обеспечения (Free Software Foundation, FSF) превратил разработку Linux в достаточно цельную систему, привлекавшую внимание не только увлеченных хакеров. В 1994 г. была выпущена версия Linux 1.0. С этого момента начался стремительный рост Linux, породивший спрос на множество дистрибутивов и увеличив количество университетов, корпораций и индивидуальных пользователей, требующих поддержки различных архитектур.

1.2 Стандартные и общие интерфейсы

Общие интерфейсы позволяют преодолеть пропасть между различными видами UNIX. Пользовательское решение о том, какую версию UNIX применять, основывается на портируемости и, следовательно, потенциальном рынке. Если вы программист, для вас не составляет тайны тот факт, что рынок для вашей программы ограничен кругом людей, которые используют ту же операционную систему, что и вы, или такую, на которую вашу программу можно легко портировать. Стандарты появились благодаря необходимости стандартизировать общие программные интерфейсы, которые позволяют запускать программу, написанную для одной системы, на другой с минимальными изменениями или вообще без оных. Различные организационные стандарты легли в основу спецификаций UNIX. POSIC, разработанный Institute of Electronic Engineers (IEEE)[2], - это стандарт портируемых операционных систем для компьютерного обеспечения, которому стремится следовать Linux.

1.3 Свободное программное обеспечение и открытые исходники

Linux - это один из наиболее успешных примеров программного обеспечения с открытыми исходниками. Программное обеспечение с открытыми исходниками - это программное обеспечение, исходный код которого свободно доступен, так что каждый может модифицировать, изучать и распространять его. Этим оно отличается от программного обеспечения с закрытыми исходниками, распространяемого только в бинарном виде.

[1] **Linux часто называют GNU/Linux для обозначения принадлежности его компонентов GNU-проекту FSF.**

[2] **В оригинальном тексте опечатка: на самом деле аббревиатура IIEE расшифровывается как Institute of Electri cal and Electronics Engineers - в вольном переводе (обычно не переводится) - Институт инженеров электро ники и электротехники. *Примеч. науч. ред.***

Открытые исходники позволяют пользователю дорабатывать программное обеспечение для удовлетворения своих потребностей. В зависимости от лицензии на код налагается несколько ограничений. Преимущество такого подхода состоит в том, что пользователь не ограничен только тем, что разработали другие, а может свободно доработать код для удовлетворения своих нужд. Linux представляет собой операционную систему, которая позволяет каждому дорабатывать и распространять себя. Это привело как к быстрой эволюции Linux, так и к страшной путанице в разработке, тестировании и документировании.

Существует несколько лицензий с открытыми исходниками, в частности Linux лицензируется под лицензией GNU General Public License (GPL) версии 2[1]. Копию лицензии можно найти в корне исходного кода в файле с именем COPRYRIGHT. Если вы планируете доработать ядро Linux, вам стоит ознакомиться с условиями лицензии, чтобы вы смогли узнать, на каких условиях вы сможете распространять свою модификацию.

Существует два лагеря последователей бесплатного программного обеспечения и программного обеспечения с открытыми исходниками. Free Software Foundation и группа открытых исходников (open-source group) различаются между собой по идеологии. Free Software Foundation, как более старая из этих двух групп, придерживается идеологии, что свобода слова распространяется на программное обеспечение в той же степени, что и на обычное слово. Группа открытых исходников рассматривает бесплатное программное обеспечение и программное обеспечение с открытыми исходниками как методологию, отличную от проприетарного программного обеспечения. Более подробную информацию можно найти по адресу http: //www. fsf.org и http://www.opensource.org.

1.4 Краткий обзор дистрибутивов Linux

Мы уже упоминали ранее, что ядро Linux является только одной из частей того, что обычно называется «Linux». Дистрибутив Linux - это совокупность ядра Linux, утилит, оконного менеджера и множества других приложений. Многие из системных программ, используемых в Linux, разработаны и поддерживаются в рамках проекта FSF GNU. С ростом распространенности и популярности Linux компоновка ядра вместе с этими и другими утилитами стала распространенным и прибыльным делом. Группы энтузиастов и корпорации взвалили на себя задачу по созданию и распространению различных дистрибутивов Linux, предназначенных для различных целей. Не вдаваясь в подробности, мы рассмотрим далее основные дистрибутивы Linux. Кроме того, постоянно появляются новые дистрибутивы Linux.

Большинство дистрибутивов Linux объединяют инструменты и приложения в группы заголовочных и исполнимых файлов. Эти группы называются пакетами и дают преимущество в использовании дистрибутивов Linux перед самостоятельной закачкой и заголовочных файлов и компиляцией всего из исходников. В соответствии с GPL лицензия по-

[1] **Общая открытая лицензия GNU.** ***Примеч. пер.***

зволяет взымать плату за дополнительную стоимость программного обеспечения с открытыми исходниками, например за послепродажную поддержку программного обеспечения.

1.4.1 Debian

Debian[1]- это GNU/Linux-операционная система. Как и другие дистрибутивы, он состоит из множества приложений и утилит, относящихся к GNU-программному обеспечению, и ядра Linux. Debian обладает одним из лучших менеджеров пакетов, *apt* (advanced packaging tool - усовершенствованный инструмент управление пакетами). Главным недостатком Debian является начальная процедура инсталляции, которая приводит в недоумение многих начинающих пользователей Linux. Debian не связан с корпорациями и разрабатывается группой энтузиастов.

1.4.2 Red Hat/Fedora

Red Hat[2] (компания) - главный игрок на рынке разработок с открытыми исходными кодами. Red Hat Linux был Linux-дистрибутивом компании до недавнего прошлого (2002-2003 гг.), когда он был заменен двумя отдельными дистрибутивами: Red Hat Enterprise Linux и Fedora Core. Red Hat Enterprise Linux предназначен для бизнеса, правительства и других отраслей, где требуется стабильное и поддерживаемое Linux окружение. Fedora Core адресована индивидуальным пользователям и энтузиастам. Основное различие между этими двумя дистрибутивами - это стабильность против широкой функциональности. Fedora включает более новый, менее стабильный код, чем включенный в состав Red Hat Enterprise. Red Hat является корпоративным выбором Linux в Америке.

1.4.3 Mandriva

Mandriva Linux[3] (ранее - Mandrake Linux) возник как простая для инсталляции версия Red Hat Linux, но со временем превратился в отдельный дистрибутив, ориентированный на индивидуальных пользователей Linux. Главная особенность Mandriva Linux - простота конфигурации и настройки.

1.4.4 SUSE

SUSE Linux[4] - это еще один главный игрок на Linux-арене. SUSE ориентирован на бизнес, правительства, индустрию и индивидуальных пользователей. Главное достоинство SUSE - это утилита *Yast2* для инсталляции и администрирования. SUSE является корпоративным выбором Linux в Европе.

[1] http://www.debian.org.
[2] http://www.redhat.com.
[3] http://www.mandriva.com/.
[4] http://www.novell.com/linux/suse/.

1.4.5 Gentoo

Gentoo[1] - это новый дистрибутив Linux, завоевавший множество положительных отзывов. Главная особенность Gentoo Linux в том, что пакеты компилируются из исходников в соответствии с конфигурацией вашей машины. Это осуществляется с помощью системы портирования Gentoo.

1.4.6 Yellow Dog

Yellow Dog Linux[2] - это один из главных игроков среди PPC-дистрибутивов Linux. Несмотря на то что некоторые из вышеописанных дистрибутивов работают и на PPC, этот основан на версии i386 Linux. Yellow Dog Linux больше всего похож на Red Hat Linux, он разработан с поддержкой платформы PPC в общем и Apple-аппаратного обеспечения в частности.

1.4.7 Другие дистрибутивы

Пользователи Linux могут горячо отстаивать любимые дистрибутивы, которых существует целое множество: классический Slackware, Monta Vista для встроенных систем и другие знакомые вам дистрибутивы. Для дальнейшего ознакомления с разнообразием дистрибутивов Linux я рекомендую вам раздел в Wikipedia http: / /en. wikipedia. org/ wiki/Category :Linux_distributions. По этой ссылке можно найти самую свежую информацию или ссылку на другие источники в сети.

1.5 Информация о версии ядра

Как и в случае с любым программным проектом, понимание схемы нумерации версий окажется вашим незаменимым помощником в деле исключения путаницы. До версии ядра 2.6 сообщество разработчиков придерживалось довольно простой схемы нумерации веток разработки для пользователей и разработчиков. Релизы с четными числами (2.2,2.4 и 2.6) являются стабильными. В стабильную ветку отправляется код с исправленными ошибками. При этом разработка продолжается в отдельной ветке, которая нумеруется нечетными цифрами (2.1, 2.3 и 2.5). Со временем разработка ветки дерева прекращается и превращается в новый стабильный релиз.

В середине 2004 г. стандартная система выпуска новых версий изменилась: код, который должен был отправиться в ветку для разработчиков, был включен в стабильную версию 2.6. Точнее говоря, «... основное ядро будет быстрее и будет обладать большей функциональностью, но не будет являться наиболее стабильным. Конечная доводка будет осуществляться дистрибьюторами (как и происходит сейчас), которым придется опера-

[1] **http://www.gentoo.org/.**

[2] **http://www.yellowdoglinux.com/.**

тивно выпускать новые патчи» [Джонатан Корбет на http://kerneltrap.org/node/view/3513].

Так как это сравнительно новая разработка, только время покажет, во что выльется изменение системы выпуска новых версий в долговременной перспективе.

1.6 Linux на PowerPC

Linux on PowerPC (система Linux, работающая на процессорах Power или PowerPC) в последнее время приобретает достаточную популярность. В последнее время в бизнес-и корпоративной среде наблюдается рост спроса на основанные на PowerPC системы с намерением использовать совместно с Linux. Причиной роста закупок PowerPC микропроцессоров стал факт, который заключается в отличной масштабируемости архитектуры и ее приспособленности для самых различных нужд.

Архитектура PowerPC появилась и на рынке встраиваемых систем в виде 32-битовых одночиповых систем system-on-chip (SOC) AMCC PowerPC и Motorola PowerPC. Эти SOC представляют собой совокупность процессора, таймера, памяти, шин, контроллеров и периферии.

Среди компаний, лицензирующих PowerPC, стоит отметить AMCC, IBM и Motorola. Несмотря на то что эти компании разрабатывают свои чипы независимо, чипы имеют набор общих инструкций и, следовательно, являются совместимыми.

Linux работает на PowerPC-игровых консолях, мейнфреймах и настольных системах по всему миру. Быстрое распространение Linux на других набирающих популярность архитектурах стало возможным благодаря объединенным усилиям энтузиастов, таких, как http: //www.penguinppc.org, и собственным инициативам корпораций, таких, как Linux Technology Center в IBM.

Благодаря росту популярности Linux на этой платформе нам придется рассмотреть, как Linux взаимодействует и использует функциональность PowerPC.

Информацию, связанную с Linux на Power, можно найти на множестве сайтов, и мы будем упоминать некоторые из них в процессе наших исследований;

http: / /www. penguinppc . org следит за судьбой порта Linux PPC и объединяет сообщество разработчиков, интересующихся новостями Linux on PowerPC.

1.7 Что такое операционная система

Теперь мы рассмотрим основные концепции операционных систем, основы использования и особенности Linux и то, как они между собой связаны. В этой главе описаны концепции, которые мы подробно рассмотрим в следующих главах. Если вы знакомы с этими концепциями, вы можете пропустить эту главу и сразу перейти к гл. 2, «Исследовательский инструментарий».

Операционная система - это то, что превращает ваше аппаратное обеспечение в пригодный для использования компьютер. Он отвечает за распределение ресурсов, предос-

тавляемых аппаратными компонентами вашей системы, и предоставляет возможность выполнять и разрабатывать прикладные программы. Если бы операционной системы не существовало, каждой программе пришлось бы включать в себя драйверы для всего оборудования, на котором ее можно использовать, что было бы лишней головной болью для программистов.

Состав операционной системы зависит от ее типа. Linux - это UNIX-образный вариант **монолитной системы.** Когда мы говорим, что система монолитна, мы не обязательно имеем в виду, что она большая (тем не менее во многих случаях это утверждение справедливо). Скорее мы имеем в виду, что она состоит из одного модуля - единственного объектного файла. Структура операционной системы определяется большим количеством процедур, которые компилируются и линкуются в единое целое. То, как эти процедуры связаны, определяет внутреннюю структуру монолитной системы.

В Linux мы имеем **пространство ядра** и **пользовательское пространство** как две отдельные части операционной системы. Пользователь общается с операционной системой через пользовательское пространство, где он может разрабатывать и/или выполнять программы. Пользовательское пространство не имеет доступа к ядру (и следовательно, к аппаратным ресурсам) напрямую, а только через системные вызовы - внешний слой процедур, реализованных в ядре. В пространстве ядра содержится функциональность по управлению аппаратными средствами. Внутри ядра системные вызовы вызывают другие процедуры, которые недоступны из пользовательского пространства, и некоторые другие дополнительные функции.

Подпространство процедур, которые невидимы из пользовательского пространства, образуется функциями отдельных драйверов устройств и функциями подсистем ядра. Драйверы устройств также представляют собой строго определенные интерфейсы функций для системных вызовов или для доступа к подсистемам ядра. На рис. 1.1 показана структура Linux

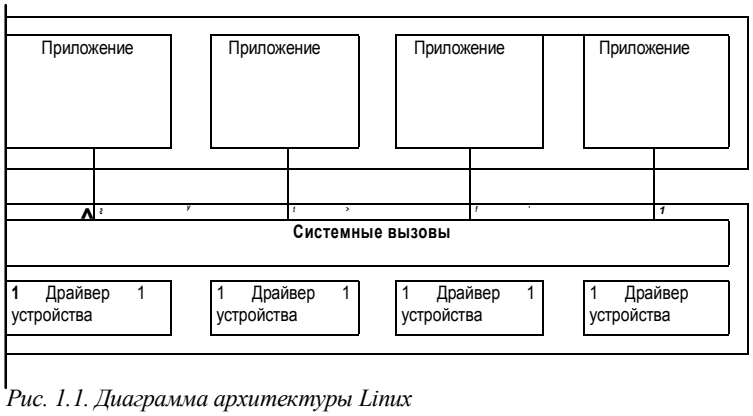


Рис. 1.1. Диаграмма архитектуры Linux

Кроме этого, Linux динамически загружает кучу драйверов устройств, нейтрализующих главный недостаток, присущий монолитным операционным системам. Динамически загружаемые драйверы устройств позволяют системному программисту внедрять системный код в ядро без необходимости перекомпиляции ядра в образ ядра. Это занимает значительное время (зависящее от производительности компьютера) и вызывает перезагрузки, что значительно замедляет для системного программиста разработку. При динамической загрузке драйверов устройств системный программист может загружать и выгружать свои драйверы устройств в реальном времени без необходимости перекомпиляции всего ядра и остановки системы.

На протяжении этой книги мы рассмотрим эти различные «части» Linux. Когда это возможно, мы будем использовать обзор сверху вниз начиная с простых пользовательских программ и далее проследим путь ее выполнения до системных вызовов и функций подсистем. Таким образом, вы сможете связать простую для понимания пользовательскую функциональность с компонентами ядра, ее реализующими.

1.8 Организация ядра

Linux поддерживает множество архитектур - это значит то, что его можно запускать на нескольких типах процессоров, включая alpha, arm, 1386, ia64, ppc, ppc64 и s390x. Пакет исходных кодов Linux включает поддержку всех этих архитектур. Основная часть кода написана на C и является аппаратно-независимой. Наиболее зависимая от аппаратуры часть кода написана на смеси C и ассемблера конкретной архитектуры. Сильно машинно-зависимые участки кода помещены в оболочку из нескольких системных вызовов, служащих интерфейсом. По мере чтения этой книги вы увидите архитектурно-зависимые части кода, связанные с инициализацией и загрузкой системы, обработкой векторов исключений, преобразованием адресов и вводом-выводом на устройства.

1.9 Обзор ядра Linux

Существуют различные компоненты ядра Linux. На протяжении этой книги мы будем использовать слова *компоненты* и *подсистемы* как взаимозаменяемые для обозначения категориальных и функциональных различий функций ядра.

В следующем разделе мы обсудим некоторые из этих компонентов и как они реализованы в ядре Linux. Также мы рассмотрим некоторые ключевые особенности операционной системы, позволяющие понять, как эти вещи реализованы в ядре. Мы разделим компоненты на файловую системы, процессы, планировщик и драйверы устройств. Тем не менее этот список далеко не полон, а только коротко излагает содержание книги.

1.9.1 Пользовательский интерфейс

Пользователи общаются с системой с помощью программ. Вначале пользователь регистрируется в системе через **терминал** или **виртуальный терминал.** В Linux программа, называемая **mingetty** для виртуальных терминалов или **agetty** для параллельных терминалов, следит за неактивными терминалами, ожидающими пользователей, чтобы сообщить, что они хотят зарегистрироваться в системе. Чтобы это сделать, они вводят имя своей учетной записи, и программа getty выполняет запрос к программе **login,** которая требует пароль, получает доступ к списку имен пользователей и паролей для аутентификации и позволяет им войти в систему в случае совпадения или выйти и завершить процесс, если совпадение не обнаружено. Программы getty каждый раз перезапускаются после завершения, что означает, что процесс перезапускается сразу после выхода.

После аутентификации в системе пользователи получают возможность сообщить программе, что они хотят выполнить. Если пользователь успешно идентифицирован, программа login запускает оболочку (shell). Таким образом, технически не являющаяся частью операционной системы оболочка становится главным интерфейсом операционной системы. Оболочка - это командный интерпретатор, представляющий собой ожидающий процесс. Затем ожидающий процесс (который блокируется до тех пор, пока ему не будет возвращен пользовательский ввод) интерпретирует и выполняет то, что набрал пользователь. Оболочка- это одна из программ, которую можно найти на верхнем слое на рис. 1.1.

Оболочка показывает командное приглашение (которое обычно конфигурируется, в зависимости от оболочки) и ожидает пользовательского ввода. Далее пользователь может обращаться к системным устройствам и программам, вводя их с помощью принятого в оболочке синтаксиса.

Программы, которые может вызывать пользователь, - это исполняемые файлы, которые хранятся файловой системой. Выполнение этих требований инициализируется оболочкой, порождающей дочерний процесс. Затем дочерний процесс может получить доступ к системным вызовам. После возврата из системных вызовов и завершения дочернего процесса оболочка возвращается к ожиданию пользовательского ввода.

1.9.2 Идентификация пользователя

Пользователь регистрируется по уникальному имени своей учетной записи. Кроме этого, он ассоциируется с уникальным идентификатором пользователя **user ID (UID).** Ядро употребляет этот DID для проверки прав пользователя на доступ к файлам. После регистрации он получает доступ к своей **домашней директории (home directory),** внутри которой может создавать, модифицировать и удалять файлы. В многопользовательских системах, таких, как Linux, важно идентифицировать пользователя с правами доступа и/или ограничениями для предотвращения для пользователя возможности вмешиваться в деятельность других пользователей или получать доступ к их данным. Суперпользова-

тель - superuser, или root, - это особенный пользователь, не имеющий ограничений; его пользовательский UID - 0.

Помимо этого, пользователь является членом одной или нескольких групп, каждая из которых имеет свой собственный групповой идентификатор (group ID, GED). При создании пользователя он автоматически становится членом группы с именем, идентичным его имени пользователя. Также пользователь может быть вручную «добавлен» в другие группы, определенные системным администратором.

Файл или программа (исполнимый файл) ассоциируются с правами, распространяемыми на пользователей и группы. Каждый отдельный пользователь может определить, кто имеет доступ к файлам, а кто нет. При этом файл ассоциируется с определенным UID и определенным GID.

1.9.3 Файлы и файловые системы

Файловая система предоставляет методы для хранения и организации данных. Linux поддерживает концепцию файла как устройствонезависимой последовательности байтов. Благодаря абстрагированию пользователь может получить доступ к файлу в независимости от устройства (например, жесткий диск, дискета или компакт-диск), на котором он хранится. Файлы группируются в некоторые хранилища, называемые директориями. Так как директории могут быть вложенными (каждая из директорий может содержать другие директории), структура файловой системы представляет собой иерархическое дерево. Корень (root) дерева является самым верхним узлом, к которому принадлежат все остальные хранимые директории и файлы. Он обозначается обратной косой чертой (/). Файловая система хранится на разделе жесткого диска или другом устройстве хранения информации.

1.9.3.1 Директории, файлы и имена путей

Каждый файл в дереве имеет свое имя пути, которое обозначает его имя и путь к нему. Также файл имеет директорию, к которой он принадлежит. Имя пути, которое начинается с текущей рабочей директории, или директории, в которой находится пользователь, называется относительным именем пути, потому что его имя является *относительным* к текущей рабочей директории. Абсолютное имя пути - это имя пути, которое начинается с корня файловой системы (например, имя, которое начинается с /). На рис. 1.2 абсолютным путем для файла file.c пользователя paul является путь /home/paul/ s re/file. c. Если мы находимся в домашней директории paul, относительным путем будет src/f ile. c.

Концепция абсолютных и относительных имен путей введена потому, что ядро ассоциирует процессы с текущей рабочей директорией и с корневой директорией. Текущая рабочая директория - это директория, из которой был вызван процесс; обозначается . (произносится как «дот»). Аналогично родительская директория - это директория которая

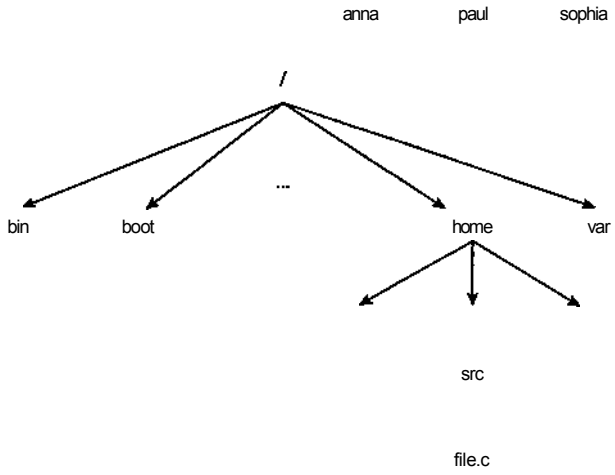


Рис. 1.2. Иерархическая файловая структура

содержит рабочую директорию; обозначается .. («дот дот»). Не забывайте, что после регистрации в системе пользователь «находится» в своей домашней директории. Если Anna говорит оболочке выполнить определенную программу, подобную Is, сразу после регистрации в системе, процесс, выполняющий Is, имеет /home/anna в качестве текущей рабочей директории (чья родительская директория /home) и / в качестве корневой директории. Корень всегда является собственным родителем.

1.9.3.2 Монтирование файловой системы

В Linux, как и во всех UNIX-подобных системах, подступ к файловой системе можно получить только в том случае, если она смонтирована. Файловая система монтируется с помощью системного вызова **mount** и размонтируется с помощью системного вызова **unmount.** Файловая система монтируется в точке монтирования, т. е. в директории, которая становится корнем для доступа к монтируемой файловой системе. Директория точки монтирования должна быть пустой[1]. Любые файлы, которые изначально располагаются в директориях, используемых в качестве точек монтирования, становятся недоступными после того, как файловая система смонтирована, и остаются недоступными до тех пор, пока файловая система не будет размонтирована. Файл /etc/mtab содержит таблицу смонтированных файловых систем[2], тогда как **/etc/f stab** содержит таблицу файловых систем, содержащую перечень всех файловых систем системы и их атрибуты.

[1] **Точка монтирования может и не быть пустой, т. е. может содержать файлы, однако «правила хорошего тона» требуют монтировать в пустые каталоги.** ***Примеч. науч. ред.***

[2] **Также информацию о смонтированных файловых системах можно получить из /proc/mounts.** ***Примеч. науч. ред.***

В /etc/mtab перечислены устройства смонтированных файловых систем, связанные с ними точки монтирования и любые опции, с которыми они были смонтированы[1].

1.9.3.3 Защита файла и права доступа

Файлы имеют права доступа в целях конфиденциальности и безопасности. Права доступа (access rights), или разрешения, хранятся в виде, в котором они применяются для трех различных групп пользователей: для самого пользователя, его группы и всех остальных. Этим трем группам пользователей могут быть назначены различные права доступа для трех типов доступа к файлу: для чтения, записи и выполнения. Когда мы вызываем список файлов с помощью Is -al, мы можем видеть разрешения файлов.

```
lkp  :-# Is  -al  /home/Sophia
drwxr-xr-x 22  sophia  sophia4096  Mar  14  15:13
drwxr-xr-x 24  root  root4096  Mar  7  18:47  ..
drwxrwx----  3  sophia department4096 Mar  4  08:37  sources
```

Первым элементом списка идет разрешение для домашней директории sophia. В соответствии с ним она разрешает всем видеть ее директорию, но не изменять ее. Сама она может ее читать, редактировать и исполнять[2]. Второй элемент означает права доступа к родительской директории /home; /home принадлежит к root, но тем не менее он позволяет всем ее читать и выполнять. В домашней директории sophia у нее есть директория с именем sources, которую позволено читать, изменять и запускать ей самой, членам ее группы с именем department и не позволено всем остальным.

1.9.3.4 Файловые режимы

В дополнение к правам доступа файл имеет три дополнительных режима: sticky, suid и sgid. Рассмотрим каждый из них.

sticky

Файл с включенным битом sticky содержит «t» в последнем символе поля mode (на пример, -rwx --------- 1). Возвращаясь к тем временам, когда доступ к дискам был значи тельно медленнее, чем сейчас, когда памяти было меньше и соответствующие методоло гии еще не были доступны[3], исполнимый файл имел активный бит sticky, позво ляющий ядру держать исполнимый файл в памяти во время его выполнения. Для активно используемых программ это позволяло увеличить производительность, снижая потреб ность в доступе к информации из файла на диске.

[1] Опции передаются в качестве параметров для системного вызова mount.

[2] Разрешение на выполнение применительно к директории означает, что пользователь может в нее входить. Разрешение на выполнение применительно к файлу означает, что он может быть запущен на выполнение и применяется только в отношении исполнимых файлов.

[3] Это связано с технологией использования принципа локальности с предпочтением загрузки частей программ. Более подробно это описано в гл. 4.

Когда бит sticky активен для директории, он бережет файлы от удаления и переименования со стороны пользователей, имеющих права на изменение данной директории (write permission) (за исключением root и хозяина файла).

suid
Исполнимый файл с установленным битом suid содержит «s», где символ «x» относится к биту пользовательского разрешения исполнения (например, -rws- ----------). Когда пользователь выполняет исполнимый файл, процесс ассоциируется с вызвавшим его пользователем. Если у исполнимого файла установлен бит suid, процесс наследует ЦШ владельца файла и доступ к соответствующим правам доступа. Таким образом, мы приходим к концепции **реального пользовательского ID (real user ID)** как противоположности **эффективного пользовательского ID (effective user ID).** Как мы вскоре увидим при рассмотрении процессов в главе «Процессы», реальный ID процесса соответствует пользователю, который запустил процесс. Эффективный UID зачастую совпадает с реальным ЦШ за исключением случаев, когда в файле стоит бит suid. В этом случае эффективный UID содержит UID владельца файла.

Файл suid часто используется хакерами, которые-вызывают исполнимый файл, принадлежащий к root с установленным битом suid, и перенаправляют программные операции для выполнения инструкций, которые им в противном случае не разрешено выполнять с разрешениями root.

sgid
Исполнимый файл с установленным битом sgid содержит «s» в том месте, где символ «x» относится к биту разрешения исполнения для группы (например, -rwxrws------). Бит действует почти так же, как и бит suid, но применяется по отношению к группе. Процесс также получает **реальный групповой ID (real group ID)** и **эффективный групповой ID (effective group ID),** которые содержат GID пользователя и GID группы файла соответственно.

1.9.3.5 Файловые метаданные

Файловые метаданные - это вся информация о файле, не включающая его содержание. Для примера, метаданные содержат тип файла, его размер, ЦШ его пользователя, права доступа и т. д. Как мы вскоре увидим, некоторые типы файлов (устройства, каналы и сокеты) не содержат данных, а только метадданые. Все метаданные файла, за исключением имени файла, хранятся в **inode** или в **индексном узле (index node);** inode - это блок информации, который имеет каждый из файлов. **Дескриптор файла (file descriptor)** - это внутриядерная структура данных для управления данными файла. Файловые дескрипторы назначаются, когда процесс обращается к файлу.

1.9.3.6 Типы файлов

UNIX-подобные системы имеют несколько типов файлов.

Обычные файлы

Обычные файлы обозначаются черточкой в первом символе поля mode (например, -rw-rw-rw-). Обычные файлы могут содержать ASCII или бинарные данные либо могут быть исполнимыми файлами. Ядро не интересуется тем, какой тип данных хранится в файле, и поэтому не делает различия между ними. Тем не менее пользовательские программы могут учитывать эту разницу. Обычные файлы могут хранить свои данные в нуле и большем количестве блоков данных[1].

Директории

Директории обозначаются буквой «d» в первом символе поля mode (например, drwx ----------). Директория - это файл, который хранит связь между файловыми именами и индексными узлами файла. Директория состоит из таблицы вхождений, каждая из которых относится к содержащемуся в директории файлу; Is -al перечисляет все содержимое директории и Ю связанного с ней inode.

Блочные устройства

Блочные устройства обозначаются буквой «Ь» в первом символе поля mode (напри мер, brw----------). Эти файлы представляют аппаратные устройства, ввод-вывод на ко торые выполняется блоками дискретного размера, кратными степени 2. К блочным устройствам относятся диски и ленточные накопители, доступные через директорию **/dev** файловой системы. Обращение к диску может требовать времени; поэтому передача данных для блочных устройств выполняется с помощью буфера кеширования (buffer cache) ядра, который хранит временные данные для уменьшения количества дорогих обращений к диску. В некоторые моменты ядро просматривает данные в буфере кеширо вания и синхронизирует с диском данные, которые нужно обновить. Таким образом значительно повышается производительность, однако в случае компьютерного сбоя это может привести к потере буферизированных данных, если они еще не были записаны на диск. Синхронизацию с диском можно вызвать принудительно с помощью системных вы зовов **sync, f sync** и **fdatasync,** которые сразу записывают буферизированные данные на диск. Блочное устройство не использует никаких блоков данных, поскольку не со держит никакой информации. Применяется только inode для хранения информации.

Символьные устройства

Символьные устройства обозначаются буквой «с» в первом символе поля mode (на пример, crw ----------). Эти файлы представляют аппаратные устройства, не имеющие блочной структуры и ввод-вывод которых представляет собой поток байтов, передавае-

[1] **Пустой файл занимает 0 блоков.**

мых напрямую между драйвером устройства и процессом-получателем. К этим устройствам относятся терминалы и параллельные устройства, доступные через директорию /dev файловой системы. Псевдоустройства (pseudo devices), или устройства, которые не представляют аппаратные устройства, а являются набором функций на стороне ядра, также относятся к символьным устройствам. Также эти устройства известны как raw-устройства, из-за того, что не содержат промежуточного кеша для хранения данных. Используется только inode для хранения информации.

Ссылки

Ссылки обозначаются буквой «l» в первом символе поля mode (например, lrw----------). Ссылка - это указатель на файл. Этот тип файлов позволяет создавать несколько ссылок на файл, данные которого хранятся в файловой системе в единственном экземпляре. Существует два типа ссылок: **жесткие ссылки (hard link)** и **символические (symbolic),** или **временные, ссылки (soft link).** Оба типа создаются с помощью вызова ln. Жесткие ссылки имеют ограничения, отсутствующие у символических. Это ограничения на связь файлов только внутри одной файловой системы, невозможность ссылаться на директории и несуществующие файлы. Ссылки сохраняют разрешения файла, на который они ссылаются.

Именованные каналы

Именованные каналы обозначаются буквой «p» в первом символе поля mode (например, prw----------). Канал - это файл, который осуществляет связь между программами, работая как канал данных; данные, записываются в него одной программой и считываются другой. Канал буферизует данные, вводимые первым процессом. Именованные каналы также известны как FIFO, потому что обычно записанные первыми данные извлекаются тоже первыми. Подобно файлам устройств, каналы не используют блоки данных, а только inode.

Сокеты

Сокеты обозначаются буквой «s» в первом символе поля mode (например, srw----------). Сокеты - это специальные файлы, которые также отвечают за связь между двумя процессами. Единственная разница между каналами и сокетами в том, что сокеты позволяют устанавливать связь между процессами на разных компьютерах, соединенных сетью. Файлы сокетов также не ассоциируются блоками данных. Так как в этой книге не рассматриваются вопросы, связанные с сетью, мы не будем касаться внутренней реализации сокетов.

1.9.3.7 Типы файловых систем

Файловая система Linux содержит интерфейс, позволяющий сосуществовать нескольким типам файловых систем. Тип файловой системы определяется способом разбиения блочных данных и манипуляциями с физическим устройством, а также типом этого фи-

зического устройства. В качестве примеров типов файловых систем можно привести системы на дисках, смонтированных по сети, такие, как NFS, и размещенные на локальных дисках, такие, как ext3, являющейся базовой файловой системой Linux. Некоторые специальные файловые системы, такие, как /proc, предоставляют доступ к данным и адресному пространству ядра.

1.9.3.8 Управление файлами

При обращении к файлу в Linux управление проходит через несколько уровней. Во-первых, программа, которая хочет получить доступ к файлу, делает системный вызов, такой, как open(), fseekЦ) или write (). Затем управление передается ядру, которое исполняет этот системный вызов. Существует высокоуровневая абстракция файловой системы, называемая VFS, которая определяет тип файловой системы (например, ext2, minix и ins do s), содержащей файл, и далее передает управление соответствующему драйверу файловой системы.

Драйвер файловой системы осуществляет работу с файлом на заданном логическом устройстве. Жесткий диск может иметь разделы msdos или ext2. Драйвер файловой системы знает, как интерпретировать данные, хранимые на диске, и использовать все связанные с ними метаданные. Поэтому драйвер файловой системы хранит содержащиеся в файле данные и дополнительную информацию, такую, как отметка времени, групповые и пользовательские режимы и разрешения файла (на запись/чтение/исполнение).

После этого драйвер файловой системы вызывает низкоуровневый драйвер, который выполняет все операции чтения данных с устройства. Этот низкоуровневый драйвер знает о блоках, секторах и всей аппаратной информации, необходимой для того, чтобы взять кусок данных и сохранить его на устройстве. Низкоуровневый драйвер передает информацию драйверу файловой системы, который интерпретирует и форматирует необработанную информацию и посылает ее VFS, которая, наконец, посылает блок данных обратно самой программе.

1.9.4 Процессы

Если мы представим операционную систему как каркас того, что сможет создать разработчик, мы можем представить процессы в виде базовых элементов, ответственных за выполнение этим каркасом необходимых действий. Точнее говоря, процесс - это выполняемая программа. Отдельная программа может быть выполнена много раз, поэтому с каждой программой может быть ассоциировано больше одного процесса.

Концепция процессов приобрела популярность с появлением многопользовательских систем в 1960-х. Представьте себе однопользовательскую операционную систему, где процессор выполняет только один процесс. В этом случае никакие другие программы не могут выполняться до тех пор, пока текущий процесс не будет завершен. Когда появилось несколько пользователей (или нам потребовалась многозадачная функциональность), нам нужно изобрести механизм переключения между задачами.

Модель процессов позволяет выполнять несколько задач благодаря реализации **контекста выполнения (execution context).** В Linux каждый процесс работает так, как будто он единственный. Операционная система управляет контекстами, назначая им процессорное время в соответствии с определенным набором правил. Эти правила назначает и выполняет **планировщик (scheduler).** Планировщик отслеживает продолжительность выполнения процесса и выключает его для того, чтобы ни одни из процессов не занимал все процессорное время.

Контекст выполнения состоит изо всего, что связано с программой, т. е. ее данных (и доступного для нее пространства адресов в памяти), ее регистров, стека, указателя стека и значения счетчика программы. За исключением данных и адресации памяти остальные компоненты процесса являются прозрачными для программиста. Тем не менее операционной системе требуется управлять стеком, указателем стека, счетчиком программы и машинными регистрами. В многопроцессорной системе операционная система должна дополнительно отвечать за **переключение контекстов (context switch)** между процессами и распределять между этими процессами ресурсы системы.

1.9.4.1 Создание процессов и управление ими

Процесс создается из другого процесса с помощью системного вызова **fork** (). Когда процесс вызывает f ork(), мы можем сказать, что процесс **порождает (spawned)** новый процесс. Новый процесс считается **дочерним процессом (child),** а первый считается **родительским процессом (parent).** Каждый процесс имеет родителя, за исключением процесса **init.** Все процессы, порожденные процессом init, запускаются во время загрузки системы. Это описано в следующих разделах.

В результате такой модели дочерних/родительских процессов система образует дерево процессов, описывающее характер отношений между запущенными процессами. Рис. 1.3 иллюстрирует такое дерево процессов.

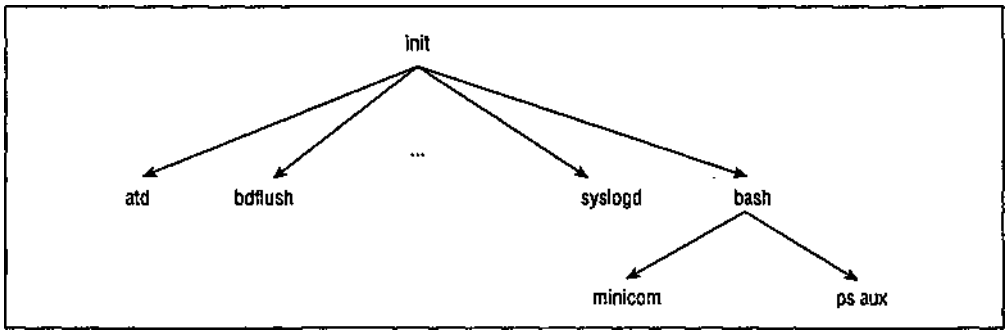


Рис. 1.3. Дерево процессов

После создания дочернего процесса родительскому процессу может понадобиться узнать, когда он будет завершен. Системный вызов **wait** () используется для приостановки родительского процесса до тех пор, пока дочерний процесс не завершится.

Процесс может заменить себя другим процессом. Это можно сделать, например, с помощью функции mingetty (), описанной ранее. Когда пользователю требуется доступ в систему, функция mingetty () запрашивает его пользовательское имя и заменяет себя процессом, выполняющим login (), в который в качестве параметра передается имя пользователя. Эта замена осуществляется с помощью вызова одного из системных вызовов **exec ().**

1.9.4.2 ID процесса

Каждый процесс обладает уникальным идентификатором, называемым **process ID (PID).** PID - это неотрицательное целое число. Идентификаторы процессов выделяются в инкрементной последовательности по мере создания процессов. По достижении максимального значения PID оно обнуляется и РГО начинают выделяться с наименьшего доступного числа, большего 1. Существует два специальных процесса: процесс 0 и процесс I[1]. Процесс 0 - это процесс, отвечающий за инициализацию и запуск процесса 1, который также известен как процесс **init.** Все процессы в запущенной системе Linux являются потомками процесса 1. После выполнения процесса 0 процесс init попадает в холостой цикл. Гл. 8, «Загрузка ядра», описывает этот процесс в разделе «Начало: start_kernel()».

Для идентификации процесса используются два системных вызова. Системный вызов getpid () возвращает РГО текущего процесса, а системный вызов getppid() возвращает РГО родителя этого процесса.

1.9.4.3 Группы процессов

Процесс может быть членом группы процессов, использующих один групповой ID. Группа процессов помогает образовывать наборы процессов. Это может потребоваться, например, если вы хотите быть уверенным, что несвязанные другим образом процессы получают сигнал kill в одно и то же время. Процесс, РГО которого идентичен ГО группы, считается лидером группы. ГО группы процессов можно манипулировать с помощью системных вызовов getpgidO и setpgidO, которые возвращают и устанавливают ГО группы процессов для указанных процессов соответственно.

1.9.4.4 Состояния процесса

Процессы могут находиться в различных состояниях в зависимости от планировщика и доступности требуемых процессу системных ресурсов. Процесс может быть в запущенном (runnable) состоянии, если он в данный момент находится в очереди выполнения **(run** queue), структуре, которая содержит ссылки на процессы, которые в данный

[1] **Процесс с PID 0 и процесс с PID 1.** ***Примеч. науч. ред.***

момент выполняются. Процесс может находиться в состоянии сна (sleep), если он ожидает освобождения ресурсов, занятых другим процессом, мертвым (dead), если он был убит, и покойным (defunct), или зомби (zombie), если процесс был завершен, прежде чем его родитель смог вызвать для него wait ().

1.9.4.5 Описатель процесса

У каждого процесса есть описатель, содержащий информацию об этом процессе. Описатель процесса содержит такую информацию, как состояние процесса, PID, пользовательскую команду на запуск и т. д. Эту информацию можно просмотреть с помощью вызова ps (состояние процесса). Вызов ps выводит нечто наподобие следующего:

```
lkp:~#ps  aux   |   more
USER   PID   TTY   STAT COMMAND
root     1          ?            S      init   [3]
root     2          ?            SN     [ksoftirqd/O]

root        10          ?       S<       [aio/0]

root        2026  ?      Ss  /sbin/syslogd  -a  /var/lib/ntp/dev/log
root        2029  ?      Ss  /sbin/klogd  -c  1  -2  -x

root       3324  tty2              Ss+  /sbin/mingetty tty2
root       332 5  tty3              Ss+  /sbin/mingetty tty3
root       332 6  tty4              Ss+  /sbin/mingetty tty4
root       3327  tty5              Ss+  /sbin/mingetty tty5
root       3328  tty6              Ss+  /sbin/mingetty tty6
root       3329  ttySOSs+  /sbin/agetty -L  9600  ttySO  vtl02
root       14914  ?             Ss  sshd:  root@pts/0

root        14917  pts/0     Ss  -bash
root        17682  pts/0     R+ ps  aux
root        17683  pts/0     R+ more
```

Список информации о процессах показывает, что процесс с PID 1 - это процесс init. Также в этом списке можно увидеть программы mingetty () и agetty (), ожидающие ввода от виртуального и параллельного терминалов соответственно. Обратите внимание, что они являются детьми предыдущих. И наконец, в списке можно увидеть сессию bash, в которой была использована команда ps aux | more. Заметьте, что |, которое используется для обозначения канала, - само не является процессом. Вспомните, что мы говорили о том, что каналы обеспечивают общение между процессами. В данном случае два процесса - это ps aux и more.

Как вы можете видеть, колонка STAT означает состояние процесса, где S означает спящий процесс, а R запущенный или запускаемый процесс.

1.9.4.6 Приоритет процесса

В однопроцессорных компьютерах мы можем выполнять в каждый момент времени только один процесс. Процессам назначаются приоритеты, и они борются друг с другом за время выполнения. Приоритет динамически изменяется ядром на основе того, сколько процессов в текущий момент запущено и каким приоритетом они до этого обладали. Процессу выделяется квант времени (timeslice) для выполнения, после которого планировщик заменяет его другим процессом, как будет описано ниже.

Сначала выполняются процессы с наивысшим приоритетом, а затем все остальные. Пользователь может устанавливать приоритеты процессов с помощью вызова nice (). Этот вызов создает процессу преимущество перед другими, определяя, сколько процессов должны его подождать. Наивысший приоритет обозначается отрицательным числом, а наименьший - положительным. Чем больший приоритет передается nice, тем большему числу процессов придется подождать.

1.9.5 Системные вызовы

Системные вызовы - это основной механизм, с помощью которого пользовательские программы общаются с ядром. Обычно системные вызовы применяют внутренние вызовы библиотек, устанавливающие регистры и данные для каждого системного вызова, необходимые для его выполнения. Пользовательские программы связываются с библиотекой с помощью определенных механизмов и делают запрос ядру.

Обычно системные вызовы обращены к одной из существующих подсистем. Это значит, что пользовательское пространство с помощью этого вызова может взаимодействовать с подсистемой из пространства ядра. Например, файлы требуют специального файлоидентифицирующего системного вызова и процессы выполняют соответствующий системный вызов. На протяжении этой книги мы рассмотрим системные вызовы, связанные с различными подсистемами ядра. Например, когда мы будем говорить о файловой системе, мы рассмотрим системные вызовы read(), write (), open() и close (). Так вы сможете увидеть, как реализована файловая система и как она управляется с помощью ядра.

1.9.6 Планировщик Linux

Планировщик Linux выполняет задачу передачи управления от одного процесса другому. За исключением процесса, имеющего преимущество ядра, в Linux 2.6 каждый процесс, включая ядро, может быть прерван практически в любой момент, а управление передано новому процессу.

Например, когда происходит прерывание, Linux должен прервать выполнение текущего процесса и обработать прерывание. Дополнительно в многозадачной операционной системе, такой, как Linux, нужно убедиться, что ни один из процессов не занимает процессор слишком долго. Планировщик отвечает за обе эти задачи: с одной стороны, он заме-

няет текущий процесс новым процессом; а с другой - следит за использование процессора процессами и заставляет их переключаться, если они занимают процессор слишком долго.

То, как планировщик Linux определяет, какому процессу передавать управление, подробно описано в гл. 7, «Планировщик и синхронизация ядра»; тем не менее, если говорить кратко, планировщик определяет приоритеты на основе прошлого быстродействия (сколько процессорного времени процесс занимал ранее) и критического характера быстродействия для процесса (прерывания имеют более критический характер, чем ведение лога системы).

Помимо этого, планировщик Linux управляет выполнением процессов на многопроцессорной машине (SMP). Существует несколько интересных особенностей для сбалансированной загрузки нескольких процессоров, таких, как привязка процесса к определенному процессору. Как было сказано ранее, базовая функциональность планировщика остается идентичной планировщику системы с одним процессором.

1.9.7 Драйверы устройств Linux

Драйверы устройств - это интерфейсы для работы ядра с жесткими дисками, памятью, звуковыми картами, сетевыми картами и другими устройствами ввода и вывода.

Ядро Linux обычно включает несколько драйверов по умолчанию; Linux не будет слишком полезен, если не сможет принять ввод с клавиатуры. Драйверы устройств выделены в отдельный модуль. Несмотря на то что Linux имеет монолитное ядро, он сохраняет высокую степень модульности, позволяя динамическую загрузку каждого драйвера. Тем не менее стандартное ядро может оставаться относительно небольшим и постепенно расширяться в зависимости от конфигурации системы, на которой запущен Linux.

В ядре Linux 2.6 драйверы устройств применяют два основных способа отображения их статуса пользователю системы: файловые системы /pros и /sys. При этом /proc обычно применяется с целью отладки и слежения за устройствами, а /sys используется для изменения настроек. Например, если у вас есть радиотюнер на встроенном Linux-устройстве, вы можете видеть частоту по умолчанию и возможность ее изменения в разделе устройств в sysf s.

В гл. 5, «Ввод-вывод», и 10, «Добавление вашего кода в ядро», мы подробно рассмотрим драйверы устройств для символьных и блочных устройств. Точнее говоря, мы коснемся драйвера устройства /dev/random и посмотрим, как он собирает информацию с других устройств Linux-системы.

1.10 Переносимость и архитектурные зависимости

По мере рассмотрения «внутренностей» ядра Linux время от времени мы будем обсуждать некоторые аспекты основного оборудования или *архитектуры.* Кроме того, ядро Linux - это большая совокупность кода, запускаемая на определенном типе процессоров и поэтому обладающая точными знаниями об этом процессоре (или процессорах),

включая набор инструкций и возможности. Тем не менее от каждого программиста ядра или системного программиста не требуется быть экспертом в микропроцессорах благодаря удачной идее *многослойного (layered)* строения ядра, что позволяет отлаживать многие возникающие проблемы прямо по мере их возникновения.

Ядро Linux создано таким образом, чтобы уменьшить количество аппаратно-зависимого кода. Когда требуется взаимодействие с аппаратной частью, вызываются соответствующие библиотеки, отвечающие за выполнение отдельных функций на данной архитектуре. Например когда ядро хочет выполнить переключение контекста, оно вызывает функцию **switch__to** (). Так как ядро компилируется под конкретную архитектуру (например, PowerPC или x86), оно линкуется (во время компиляции) с соответствующими include-файлами include/asm-ppc/system.h*uiH include/asm-i386/system.h соответственно. Во время загрузки архитектурно-зависимый код инициализации выполняет вызов к Firmware BIOS (BIOS - это программное обеспечение для загрузки, описанное в гл. 9, «Построение ядра Linux»).

В зависимости от целевой архитектуры с аппаратным обеспечением взаимодействуют различные слои программного обеспечения. Код ядра, ответственный за работу с этим аппаратным обеспечением, находится на более высоком слое.

Благодаря этому ядро Linux можно назвать *портабелъным (portable)* на различные архитектуры. Ограничения проявляются в тех случаях, когда невозможно портировать драйверы, по причине того, что такое аппаратное обеспечение несовместимо с данной архитектурой или она является недостаточно популярной для портирования на нее драйверов. Для создания драйвера устройства программист должен иметь спецификацию данного аппаратного обеспечения на уровне регистров. Не все производители предоставляют подобную документацию из-за проприетарного характера этого аппаратного обеспечения. Это в некоторой степени ограничивает распространение Linux на различные архитектуры.

Резюме

В этой главе представлен краткий обзор и описание тем, которые мы далее рассмотрим более подробно. Также мы упомянули некоторые особенности Linux, которым он обязан своей популярностью, и некоторые его недостатки. В следующей главе описываются базовые инструменты для эффективного изучения ядра Linux.

Упражнения

1. В чем разница между системой UNIX и UNIX-клоном?
2. Что означает термин «Linux on Powen>?
3. Что такое пользовательское пространство? Что такое пространство ядра?

4. Что является интерфейсом к функциональности ядра из пространства пользовательских программ?
5. Как связаны пользовательский UID и имя пользователя?
6. Перечислите способы связи файлов с пользователями.
7. Перечислите типы файлов, поддерживаемых Linux.
8. Является ли оболочка частью операционной системы?
9. Для чего нужны защита файла и его режимы?
10. Перечислите виды информации, которую можно найти в структуре, хранящей метаданные.
11. В чем заключается основное различие между символьными и блочными устройствами?
12. Какие подсистемы ядра Linux позволяют ему работать как многопоточная система?
13. Каким образом процесс становится родителем другого процесса?
14. В этой главе мы рассмотрели два иерархических дерева: дерево файлов и дерево процессов. В чем они похожи? Чем они отличаются?
15. Связаны ли Ш процесса и ID пользователя?
16. Для чего процессам назначаются приоритеты? Все ли пользователи могут изменять приоритеты процессов? Если могут или не могут, то почему.
17. Используются ли драйверы устройств только для добавления поддержки нового аппаратного обеспечения?
18. Что позволяет Linux быть портируемой на разные архитектуры системой?

Глава 2

Исследовательский инструментарий

В этой главе:

В этой главе приведен обзор основных конструкций программирования под Linux и описаны некоторые методы взаимодействия с ядром. Мы начнем с обзора основных типов данных Linux, используемых для эффективного хранения и получения информации, методов программирования и основ языка ассемблера. Это даст нам фундамент для более подробного анализа ядра в следующих главах. Затем мы опишем, как Linux компилирует и собирает исходный код в исполнимый код. Это будет полезно для понимания кросс-платформенного кода и заодно познакомит вас с GNU-набором инструментов. После этого будет описано несколько методов получения информации от ядра Linux. Мы проведем как анализ исходного и исполнимого кода, так и вставку отладочных сообщений в ядро Linux. Эта глава представляет собой сборную солянку обзора и комментариев по поводу принятых в Linux соглашений[1].

2.1 Типы данных ядра

Ядро Linux содержит множество объектов и структур, за которыми нужно следить. Для примера можно привести страницы памяти, процессы и прерывания. Способность быстро находить каждый из объектов среди всех остальных является залогом эффективности системы. Linux использует связанные списки и деревья бинарного поиска (вместе с набором вспомогательных структур), для того чтобы, во-первых, сгруппировать объекты внутри отдельных контейнеров и, во-вторых, для эффективного поиска отдельного элемента.

2.1.1 Связанные списки

Связанные списки (linked list) - это распространенные в компьютерной науке типы данных, повсеместно используемые в ядре Linux. Обычно в ядре Linux связанные списки реализуются в виде циклических двусвязных списков (рис. 2.1). Поэтому из каждого элемента такого списка мы можем попасть в следующий или предыдущий элемент. Весь код связанных списков можно посмотреть в include/linux/list .h. Этот подраздел описывает основные особенности связанных списков.

Связанный список инициализируется с помощью макросов LIST_HEAD и INIT_ LIST_HEAD:

```
include/linux/list .h
27
28   struct list_head   {
```

[1] Мы еще не углубляемся в глубины ядра. Здесь представлен обзор инструментов и концепций, необходимых для навигации в коде ядра. Если вы являетесь более опытным хакером, вы можете пропустить эту главу и сразу перейти к «внутренностям» ядра, описание которых начинается в гл. 3, «Процессы: принципиальная модель выполнения».

```
29      struct list_head    *next,                *prev;
30    };
31
32  #define    LIST HEAD INIT(name)          {     &(name),      &(name)        }
33
34  #define  LIST HEAD(name)    \
35       struct   list head name  =  LIST HEAD INIT(name)
36
37   #define  INIT LIST HEAD(ptr)   do   {   \
38      (ptr)->next   =   (ptr);    (ptr)->prev  =   (ptr);   \
39   }  while   (0)
```

Строка 34

Макрос LIST_HEAD создает голову связанного списка, обозначенную как name.

Строка 37

Макрос INIT_LIST_HEAD инициализирует предыдущий и следующий указатели структуры ссылками на саму себя. После обеих этих вызовов паше содержит пустой двусвязный список[1].

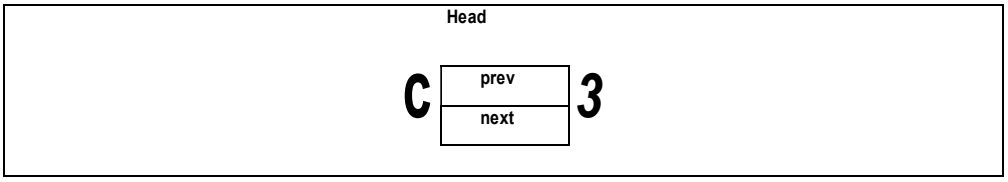


Рис. 2.1. Связанный список после вызова макроса INITJLISTJiEAD

Простые стек и очередь могут быть реализованы с помощью функций list_add () и list_add_tail () соответственно. Хорошим примером может послужить следующий отрывок из рабочего кода очереди:

kernel/workqueue.c
330 list_add(&wq->list, &workqueues);

Ядро добавляет wq->lis t к общесистемному списку рабочей очереди, workqueues. Таким образом, workqueues - это стек очередей.

Аналогично следующий код добавляет work->entry в конец списка cwq-> worklist. При этом cwq->worklist рассматривается в качестве очереди:

[1] Пустой связанный список определяется как список, для которого head->next указывает на голову списка.

kernel/workqueue.c
84 list_add_tail(&work->entry, &cwq->worklist);

Для удаления элемента из очереди используется **list_del** (), которая получает удаляемый элемент в качестве параметра и удаляет элемент с помощью простой модификации следующего и предыдущего узлов таким образом, чтобы они указывали друг на друга. Например, при уничтожении рабочей очереди следующий код удаляет рабочую очередь из общесистемного списка рабочих очередей:

```
kernel/workqueue.c
382 list_del(&wq->list);
```

В include/linux/list. h находится очень полезный макрос **list_f or__each_ entry:**

```
include/linux/list.h
349 /*
350 * list for  each entry -  проход по списку указанного типа
351 * @pos:  type * to используется как счетчик цикла.
3 52 * @head:  голова списка.
353 * ©member:  имя list_struct внутри структуры.
354 */
355 tdefine list_for__each__entry(pos, head, member)
356     for (pos = list_entry((head)->next, typeof(*pos), member),
357          prefetch(pos->member.next);
358      &pos->member != (head);
359      pos = list__entry(pos->member.next, typeof(*pos), member), 3 60
         prefetch(pos->member.next))
```

Эта функция перебирает весь список и выполняется со всеми его элементами. Например, при включении процессора он будит все процессы для каждой рабочей очереди:

```
kernel/workqueue.c
59  struct workqueue_jstruct   {
60      struct cpu workqueue  struct cpu  wq[NR  CPUS] ;
61      const char *name;
62      struct list head list;    /*  Пустая в однопоточном режиме  */
63  };
```

```
466     case CPU_ONLINE:
467       /* Удаление рабочих потоков. */
468       list for each  entry(wq,Sworkqueues,list)
469         wake up process(wq->cpu wq[hotcpu].thread) ;
470       break;
```

Макрос раскрывает и использует список list_head с помощью структуры workqueue_structwq для обхождения всех списков, головы которых находятся в рабочих очередях. Если это кажется вам немного странным, помните, что нам не нужно знать, в каком списке мы находимся, для того чтобы его посетить. Мы узнаем, что достигли конца списка тогда, когда значение указателя на следующий элемент текущего вхождения будет указывать на голову списка. Рис. 2.2 иллюстрирует работу списка рабочих очередей[1].

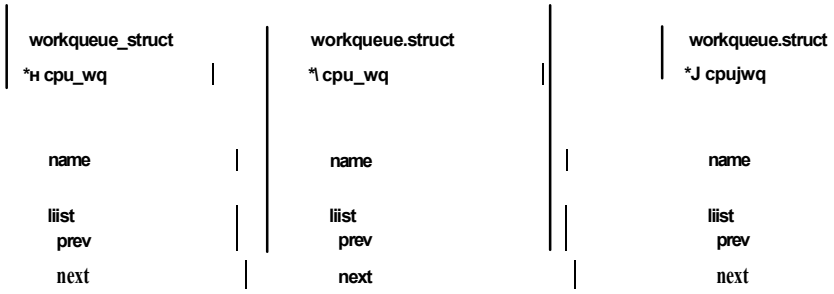


Рис. 2.2. Список рабочих очередей

Дальнейшее усовершенствование связанного списка заключается в такой реализации, где голова списка содержит только один указатель на первый элемент. В этом состоит главное отличие от двусвязного списка, описанного в предыдущем разделе. Используемый в хеш-таблицах (описанных в гл. 4, «Менеджмент памяти») единственный указатель головы не имеет указателя назад, на хвостовой элемент списка. Таким образом достигается меньший расход памяти, так как указатель хвоста в хеш-таблицах не используется.

```
include/linux/list.h 484
struct hlist_head {
```

[1] Кроме этого, можно использовать list_for_each_entry_reverse О для посещения элементов списка в обратном порядке.

```
485         struct hlist_node *first;
486   };

488   struct hlist node {
489         struct hlist node *next, **pprev;
490   };

492   #define HLIST_HEAD_INIT { .first = NULL }
493   #define HLIST_HEAD(name) struct hlist_head name = { .first = NULL }
```

Строка 492

Макрос HLIST_HEAD_INIT устанавливает указатель first в указатель на NULL.

Строка 493

Макрос HLIST_HEAD создает связанный список по имени и устанавливает указатель first в указатель на NULL.

Этот список создается и используется ядром Linux в рабочей очереди, как мы увидим далее в планировщике, таймере и для межмодульных операций.

2.1.2 Поиск

Подразд. 2.1.1 описывает объединение элементов в список. Упорядоченный список элементов сортируется по значению ключа каждого элемента (например, когда каждый элемент имеет ключ, значение которого больше предыдущего элемента). Если мы хотим обнаружить определенный элемент (по его ключу), мы начнем с головы и будем перемещаться по списку, сравнивая значение его ключа с искомым значением. Если значения не равны, мы переходим к следующему элементу, пока не найдем подходящий. В этом примере время, необходимое для нахождения нужного элемента, прямо пропорционально значению ключа. Другими словами, такой линейный поиск выполняется тем дольше, чем больше элементов в списке.

Большое 0

Для теоретической оценки времени работы алгоритма, необходимого для поиска заданного ключа поиска, используется нотация большое О (Big-O). Она показывает наихудшее время поиска для заданного количества элементов (п). Для линейного поиска Big-O нотация показывает 0(п/2), что означает среднее время поиска, т. е. перебор половины ключей списка. *Источник:* Национальный институт стандартов и технологий (www.nist.org)

При большом количестве элементов в списке для сортировки и поиска требуемых данных операционной системе требуются более быстрые методы поиска, чтобы подобные операции ее не тормозили. Среди множества существующих методов (и их реализаций) для хранения данных Linux использует *деревья*.

2.1.3 Деревья

Используемые в Linux для управления памятью деревья позволяют эффективно получать доступ и манипулировать данными. В этом случае эффективность измеряется тем, насколько быстро мы сможем сохранять и получать отдельные группы данных среди других. В этом подразделе представлены простые деревья, и в том числе красно-черные деревья, а более подробная реализация и вспомогательные элементы показаны в гл. 6, «Файловые системы». Деревья состоят из *узлов (nodes)* и *ребер (edges)* (см. рис. 2.3). Узлы представляют собой элементы данных, а ребра - связь между узлами. Первый, или верхний, узел является *корнем* дерева, или *корневым (root)* узлом. Связь между узлами описывается как *родителифагеЫ)\ дети (child),* или *сестры (sibling),* где каждый ребенок имеет только одного родителя (за исключением корня), каждый родитель имеет одного ребенка или больше детей, а сестры имеют общего родителя. Узел, не имеющий детей, называется *листом (leaf). Высота (height)* дерева - это количество ребер от корня до наиболее удаленного листа. Каждая строка наследования в дереве называется *уровнем (level).* На рис. 2.3 b и с находятся на один уровень ниже a, а d, e и f на два уровня ниже a. При просмотре элементов данного набора сестринских узлов упорядоченные деревья содержат элементы сестры с наименьшим значением ключа слева и наибольшим справа. Деревья обычно реализуются как связанные списки или массивы, а процесс перемещения по дереву называется *обходом (traversing)* дерева.

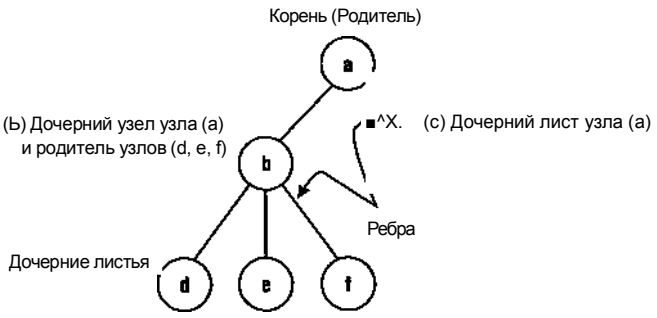


Рис. 2.3. Дерево с корнем

2.1.3.1 Бинарные деревья

До этого мы рассмотрели поиск ключа с помощью линейного поиска, сравнивая наш ключ на каждой итерации. А что если с каждым сравнением мы сможем отбрасывать половину оставшихся ключей?

Бинарное дерево (binary tree) в отличие от связанного списка является иерархической, а не линейной структурой данных. В бинарном дереве каждый элемент или узел ука-

зывает на левый и правый дочерние узлы и, в свою очередь, каждый дочерний узел указывает на левого и правого ребенка и т. д. Главное правило сортировки узлов заключается в том, чтобы у каждого левого дочернего узла значение ключа было меньше, чем у родителя, а у правого больше или равно родительскому. В результате применения этого правила мы знаем, что для значения ключа в данном узле левый дочерний узел и его потоки содержат меньшие значения ключей, чем у данного, а правый и его потомки - большее или равное значение ключа.

Сохраняя данные в бинарном дереве, мы уменьшаем данные для поиска на половину в каждой итерации. В нотации Big-O его производительность (с учетом количества искомых элементов) оценивается как **O log(n).** Сравните этот показатель с линейным поиском со значением Big-0 0(n/2).

Алгоритм, используемый для прохода по бинарному дереву, прост и отлично подходит для рекурсивной реализации, так как в каждом узле мы сравниваем значение нашего ключа и переходим в левое или правое поддерево. Далее мы обсудим реализации, вспомогательные функции и типы бинарных деревьев.

Как только что говорилось, узел бинарного дерева может иметь только одного левого, только одного правого потомка, обоих (левого и правого) потомков или не иметь потомков. Для упорядоченного бинарного дерева действует правило, что для значения узла (x) левый дочерний узел (и все его потомки) имеют значения меньше x, а правый дочерний узел (и все его потомки) имеют значение больше x. Следуя этому правилу, если в бинарное дерево вставляется упорядоченный набор значений, оно превращается в линейный список, что приводит к относительно медленному поиску значений. Например, если мы создаем бинарное дерево со значениями [0,1,2,3,4,5,6], 0 будет находиться в корне, 1 больше 0 и будет его правым потомком; 2 больше 1 и будет его правым потомком; 3 будет правым потомком 2 и т. д.

Сбалансированным по высоте (height-balanced) бинарным деревом является такое дерево, которое не имеет листьев, более удаленных от корня, чем остальные. По мере добавления узлов в дерево его нужно перебалансировать для более эффективного поиска; что выполняется с помощью *поворотов (rotation).* Если после вставки данный узел (e) имеет левого ребенка с потомками на два уровня больше, чем другие листья, мы должны выполнить правый поворот узла e. Как показано на рис. 2.4, e становится родителем h и правый ребенок e становится левым ребенком h. Если выполнять перебалансировку после каждой вставки, мы можем гарантировать, что нам нужен только один поворот. Это правило баланса (когда ни один из листьев детей не должен находиться на расстоянии больше одного) известно как AVL-дерево [в честь Дж. М. Адельсона-Велски (G. M. Adelson-Velskii) и Е. М. Лендис (E. M. Landis)].

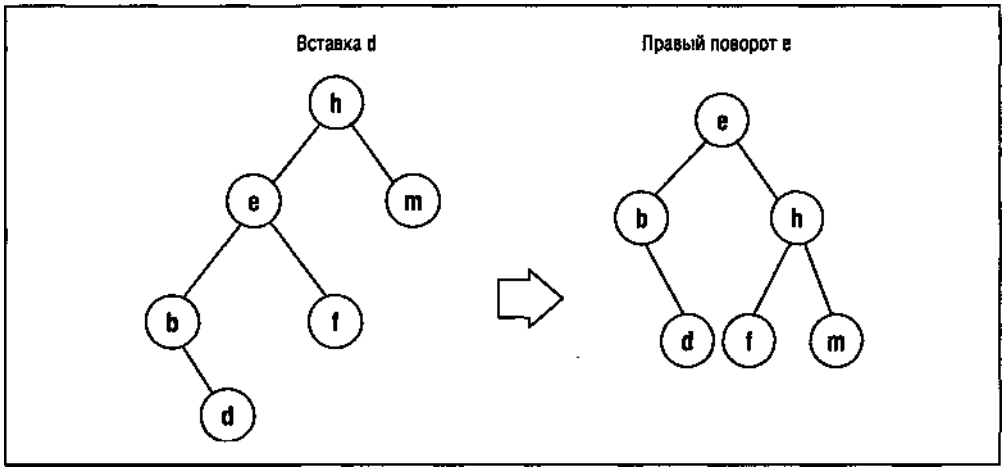


Рис. 2.4. Правый поворот

2.1.3.2 Красно-черные деревья

Красно-черное дерево, похожее на AVL-дерево, используется в Linux для управления памятью. Красно-черное дерево - это сбалансированное бинарное дерево, в котором каждый узел окрашен в красный или черный цвет. Вот правила для красно-черного дерева:

- **Все узлы являются либо красными, либо черными.**
- **Если узел красный, оба его потомка - черные.**
- **Все узлы-листья - черные.**
- **При перемещении от корня до листа каждый путь содержит одинаковое количество черных узлов.**

Как AVL-, так и красно-черные деревья имеют производительность O log(n) (в нотации Big-O), зависящую от количества вставленных данных (сортированных/несортированных) и поиска; каждый тип обладает своими преимуществами. [В Интернете можно найти несколько интересных книг, посвященных производительности деревьев бинарного поиска (BST).]

Как говорилось ранее, в компьютерной науке используются многие структуры данных и связанные с ними алгоритмы поиска. Целью этого раздела является помочь вам в ваших исследованиях концепций и структур данных, используемых для организации данных в Linux. Понимание основ списков и деревьев поможет вам понять более сложные операции, такие, как управление памятью и очереди, которые обсуждаются в следующей главе.

2.2 Ассемблер

Linux - это операционная система. Поэтому его часть тесно связана с процессором, на котором он работает. Авторы Linux проделали огромную работу по минимизации процессорно- (и архитектурно-) зависимого кода, стараясь писать как можно менее архитектурно-зависимый код. В этом разделе мы рассмотрим следующее:

- Каким образом некоторые функции реализуются на x86- и PowerPC-архитектурах.
- Как использовать макросы и встроенный ассемблерный код.

Целью этого раздела является раскрытие основ, необходимых вам для того, чтобы разобраться в архитектурно-зависимом коде ядра и не заблудиться в нем. Мы оставим серьезное программирование на языке ассемблера для других книг. Также мы рассмотрим некоторые тонкости применения языка ассемблера: встроенный ассемблер.

Чтобы конкретнее говорить о языке ассемблера для x86 и PPC, давайте поговорим об архитектуре каждого из этих процессоров.

2.2.1 PowerPC

PowerPC - это архитектура с ограниченным набором вычислительных инструкций **(Reduced Instruction Set Computing, RISC).** Архитектура RICS предназначена для увеличения производительности за счет упрощения выполнения набора инструкций за несколько циклов процессора. Для того чтобы воспользоваться преимуществами параллельных (суперскалярных) инструкций аппаратного обеспечения, некоторые из этих инструкций, как мы вскоре увидим, далеко не так просты. Архитектура PowerPC совместно разработана ЮМ, Motorola и Apple. В табл. 2.1 перечислен пользовательский набор регистров PowerPC.

Таблица 2.1. Набор регистров PowerPC

Имя регистра	Ширина регистра		Функция	Количество регистров
	32 бита	*64* бита		
CR	32	32	Регистр состояния	1
LR	32	64	Регистр связи	1
CTR	32	64	Регистр счетчика	1
GPR[0...31]	32	64	Регистр общего назначения	32
XER	32	64	Регистр исключений с фиксированной точкой	1
FPR[0...31]	64	64	Регистр с плавающей точкой	32

Таблица 2.1. Набор регистров PowerPC (Окончание)

FPSCR	32	64	Регистр контроля управления с плавающей точкой	1

Табл. 2.2 иллюстрирует применение бинарного интерфейса приложений для общих регистров и регистров с плавающей точкой. Переменные регистры могут использоваться в любое время, а постоянные-только для выполнения вызовов предусмотренных функций.

Бинарный интерфейс приложений [Application Binary Interface (ABI)]

ABI - это набор соглашений, позволяющий компоновщику объединять отдельные скомпилированные модули в один юнит без перекомпиляции, соглашений на вызовы, машинный интерфейс и интерфейс операционной системы. Помимо всего прочего, ABI определяет бинарный интерфейс между юнитами. Существует несколько разновидностей PowerPC ABI. Обычно они связаны с целевой операционной системой и/или оборудованием. Эти вариации и расширения основаны на разработанной в AT&T документации *UNIX System V Application Binary Interface* и ее более поздних вариациях из Santa Cruz*Operation (SCO). Соответствие ABI позволяет компоновать объектные файлы, откомпилированные различными компиляторами.

Таблица 2.2. Использование регистров ABI

Регистр	Тип	**Использование**
гО	Переменный	Пролог/эпилог, языково-зависимые
г1	Специальный	Указатель на стек
г2	Специальный	TOC
г3-г4	Переменный	Передаваемые параметры для ввода-вывода
г5-г10	Переменный	Передаваемые параметры
rll	Переменный	Указатель на окружение
г12	Переменный	Обработка исключений
г13	Постоянный	Зарегистрирован для вызовов
г14-г31	Постоянный	Зарегистрирован для вызовов
ГО	Переменный	Рабочий
fl	Переменный	Первый параметр с плавающей точкой, возвращает первое скалярное значение с плавающей точкой

Таблица 2.2. Использование регистров ABI (Окончание)

f2-f4	Переменный	Параметры со 2-го по 4-й, возвращают скалярное значение с плавающей точкой
F5-A3	Переменный	Параметры с 5-го по 13-й
fl4-f31	Постоянный	Зарегистрирован для вызовов

Архитектура PowerPC с 32 битами использует инструкции длиной 4 бита, выровненные по слову. Она оперирует байтами, полусловом, словом и двойным словом. Инструкции делятся на переходы, инструкции с фиксированной точкой и с плавающей точкой.

2.2.1.1 Условные инструкции

Регистр состояния (condition register, CR) применяется для всех условных операций. Он разбит на 8 4-битовых полей, которые можно явно изменить инструкцией move, неявно в результате выполнения инструкции или чаще всего в результате инструкции сравнения.

Регистр связывания (link register, LR) используется в некоторых видах условных операций для получения адреса перехода и адреса возврата из условной инструкции.

Регистр счетчика (count register, CTR) хранит счетчик циклов, увеличиваемый с помощью некоторых условных инструкций. Кроме этого, CTR хранит адрес перехода для некоторых из условных инструкций.

В дополнение к CTR и LR условные инструкции PowerPC могут выполнять переходы по относительному или абсолютному адресу. При использовании расширенных мнемоник становятся доступными еще множество различных условных и безусловных инструкций перехода.

2.2.1.2 Инструкции с фиксированной точкой

PPC не имеет вычислительных инструкций для изменения хранимых данных. Вся работа должна выполняться в одном из 32 регистров общего назначения (general-purpose registers, GPRs). Инструкции доступа к хранимым данным могут получать байты, полуслова, слова и двойные слова в обратном порядке байтов (Big Endian). При использовании расширенных мнемоник дополнительно доступны множество инструкций загрузки, сохранения, арифметические и с фиксированной точкой, а также специальные инструкции для перемещения из системных регистров или в них.

2.2.1.3 Инструкции с плавающей точкой

Инструкции с плавающей точкой можно разделить на две категории: вычислительные, включающие арифметические, трассировочные, преобразующие сравнительные; и невычислительные, включающие перемещение из хранящих или других регистров или в них. Доступно 32 регистра с плавающей точкой общего назначения; каждый может содержать данные в формате с плавающей точкой удвоенной точности.

Обратный порядок байтов/прямой порядок байтов (Big Endian/Litle Endian)

В процессорной архитектуре порядок байтов (Endianness) означает порядок следования байтов и операций. PowerPC работает с обратным порядком байтов, это значит, что самый старший байт имеет меньший адрес, а оставшиеся 3 байта следуют за ним (для 32-битового слова). Прямой порядок байтов принят в x86-архитектуре и означает противоположный порядок. Младший значащий байт имеет наименьший адрес, а оставшиеся 3 байта следуют за ним. Давайте рассмотрим это на примере представления числа 0x12345678 (рис. 2.5).

Рис. 2.5. Прямой и обратный порядок следования байтов

Споры о том, какая система лучше, выходят за рамки данной книги, тем не менее вам важно помнить, на какой системе вы пишете и отлаживаете свой код. Для примера ошибки, связанной с последовательностью байтов, можно привести драйвер PO-устройства, используемый на архитектуре с другой последовательностью байтов.

Термины *Big Endian* и *Little Endian* (тупоконечный и остроконечный) *були* придуманы Джонатаном Свифтом в *Путешествии Гулливера.* В этой истории Гулливер знакомился с двумя нациями, воевавшими из-за разногласия по поводу того, с какой стороны есть яйцо - с острой или с тупой.

2.2.2 x86

Архитектура x86 является архитектурой с полным набором вычислительных инструкций Complex Instruction Set Computing (CISC). Инструкции имеют различную длину в зависимости от назначения. В классе процессоров x86 Pentium существует три типа регистров: общего назначения, сегмента и статуса/управления. Далее описывается их базовый набор. Вот 8 регистров общего назначения и соглашение по их использованию:

- **EAX.** Аккумулятор общего назначения.
- **EBX** • Указатель на данные.
- **ECX**. Счетчик цикловых операций.
- **EDX**. Указатель ввода-вывода.
- **ESI**. Указатель на данные в сегменте DS.

- **EDI**. Указатель на данные в сегменте ES.
- **ESP**. Указатель на стек.
- **EBP**. Указатель на данные в стеке.

Шесть сегментных регистров используются в *реальном (real)* режиме адресации, когда память адресуется блоками. При этом каждый байт в памяти доступен по отступу из данного сегмента [например, ES: EDI указывает на память в ES (дополнительном сегменте) с отступом к значению в EDI]:

- CS. Сегмент кода.
- **SS.** Сегмент стека.
- **$ES_{\#}$ $DS_{\#}$ FS, GS.** Сегмент данных.

Регистр FLAGS показывает состояние процессора после каждой инструкции. Он может хранить такие результаты, как **нуль (zero), переполнение (overflow)** или **перенос (carry).** EIP - это регистр зарезервированного указателя, обозначающий отступ к текущей инструкции процессора. Обычно он используется с регистром сегмента кода для формирования полного адреса (например, CS:EIP):

- **EFLAGS**. Статус, управление и системные флаги.
- **EIP**. Указатель на инструкции, содержащие отступ от CS.

В архитектуре x86 используется прямой порядок следования данных. Доступ к памяти осуществляется порциями по байту (8 бит), слову (16 бит), двойному слову (32 бита) или учетверенному слову (64 бита). Преобразование адресов (и связанных регистров) описывается в гл. 4, но для этого раздела достаточно знать, что в архитектуре x86 код и данные делятся на три категории: **управляющие, арифметические** и **данные.**

2.2.2.1 Управляющие инструкции

Управляющие инструкции похожи на управляющие инструкции в PPC, позволяющие изменять поток выполнения программы. Архитектура x86 использует различные инструкции перехода («jump») и метки на выполняемый код, основанные на значениях регистра EFLAGS. В табл. 2.3 перечислены наиболее часто используемые из них. **Коды состояния condition codes** устанавливаются в соответствии с исходом определенных инструкций. Например, когда инструкция **cmp** (сравнение) обрабатывает два целых операнда, он модифицирует следующие флаги в регистре EFLAGS: OF (переполнение), **SF** (знаковый флаг), ZF (флаг нуля), **PF** (флаг четности) и CF (флаг переноса). Таким при вычислении значения инструкции **cmp** устанавливается нулевой флаг.

Таблица 2.3. Наиболее распространенные инструкции перехода

Инструкция	Функция	Коды состояния EFLAGS
je	**Переход, если равно**	**ZF=1**
jg	**Переход, если больше**	**ZF=0 и SF=OF**
jge	**Переход, если больше или равно**	**SF=OF**
jl	**Переход, если меньше**	**SF!=OF**
jle	**Переход, если меньше или равно**	**ZF=1**
jmp	**Безусловный переход**	**Безусловный**

В ассемблерном коде x86 метки состоят из уникальных имен, после которых ставится двоеточие. Метка может быть использована везде в ассемблерной программе и имеет тот же адрес, что и следующая за ней строка кода. Следующий код использует условный переход и метку:

```
100          pop eax
101    loop2
102    pop ebx
103    cmp eax, ebx
104          jge loop2
```

Строка 100

Получение значения из верха стека и помещение его в eax.

Строка 101

Это метка с именем loop2.

Строка 102

Получение значения из верха стека и помещение его в ebx.

Строка 103

Сравнение значений eax и ebx.

Строка 104

Переход, если eax больше или равно ebx.

Еще одним способом передачи программного управления являются инструкции **call** и **ret.** Обратимся к следующей строке ассемблерного кода:

```
call my_routine
```

Инструкция call передает программное управление на метку my_routine, передавая адрес инструкции, следующей в стеке сразу за инструкцией call. Инструкция ret (исполняемая из my_routine) извлекает возвращаемый адрес и осуществляет переход по нему.

2.2.2.2 Арифметические инструкции

Наиболее известны арифметические инструкции **add, sub, imul** (целочисленное умножение), **idiv** (целочисленное деление) и логические операторы **and, or, not** и **xor.** X86-инструкции с плавающей точкой и связанные с ними регистры выходят за рамки этой книги. Различные расширения архитектур Intel и AMD, такие, как **MMX, SSE, 3dNow, SIMD** и **SSE2/3,** значительно ускоряют приложения с интенсивными вычислениями, такие, как графические и аудио-вычисления.

2.2.2.3 Инструкции для работы с данными

Данные могут перемещаться между регистрами, между регистрами и памятью и из константы в регистр или память, но не из одного места в памяти в другое. Ниже приведено несколько примеров.

```
100    mov eax,ebx
101    mov eax,WORD PTR[data3]
102    mov BYTE PTR[char1],al
103    mov eax,Oxbeef
104    mov WORD PTR [my_data],Oxbeef
```

Строка 100

Передача 32 бит данных из ebx в eax.

Строка 101

Передача 32 бит данных из переменной data3 в памяти в eax.

Строка 102

Передача 8 бит данных из переменной charl в памяти в al.

Строка 103

Передача значения константы Oxbeef в eax.

Строка 104

Передача значения константы Oxbeef в переменную в памяти my_data.

Как видно из предыдущего примера, **push, pop** и их длинные версии **lpush, lpop** перемещают данные в стек и из него (по указателю SS: ESP). Подобно инструкции mov, операции push и pop могут применяться к регистрам, данным и константам.

2.3 Пример языка ассемблера

Мы можем написать простую программу, чтобы увидеть архитектурные различия преобразования одного и того же С-кода в ассемблер. Для этого эксперимента мы используем компилятор dcc, поставляемый с Red Hat 9, и gcc-кросс-компилятор для PowerPC. Мы приведем С-программу и затем для сравнения код для x86 и PowerPC.

Вас может озадачить количество ассемблерного кода, генерируемого из нескольких строк С. Так как мы просто компилируем из С в ассемблер, мы не будем связывать его ни с каким кодом окружения, таким, как С-библиотеки времени исполнения или создание-уничтожение локального стека, поэтому результат будет значительно меньше, чем настоящий ELF-исполнимый файл.

Помните, что в ассемблере вы приближаетесь к тому, что именно делает процессор цикл за циклом. Еще можно сказать, что вы получаете полный контроль над кодом и системой. Важно помнить, что даже при нахождении инструкций в памяти в определенном порядке, они не обязательно будут прочитаны именно в такой порядке. Некоторые архитектуры выстраивают операции загрузки и хранения отдельно.

Вот пример С-кода:

count.c

```
1  int  main()
2  {
3  int  i,j=0; 4
5  for(i=0;i<8;i++)
6  j=j+i; 7
8  return  0;
9  }
```

Строка 1

Здесь объявляется функция main.

Строка 3

Здесь инициализируются нулем переменные i и j.

Строка 5

Цикл for: пока значение i находится в интервале от 0 до 7, установить j равным j плюс i.

Строка 8

Метка перехода назад в вызывающую программу.

2.3.1 Пример x86-ассемблера

Вот код, сгенерированный для x86 с помощью команды gcc -S count. c из командной строки. После ввода кода база стека указывает на ss : ebp. Код выполнен в формате «AT&T», в котором регистры имеют префикс %, а константы - префикс $. Пример ассемблерных инструкций приведен в этом разделе для того, чтобы подготовить вас к будущим примерам программ, но перед этим нам нужно обсудить непрямую адресацию.

Для обозначения позиции в памяти (например, стека) ассемблер использует специальный синтаксис для индексированной адресации. Базовый регистр помещается в круглые скобки, а индекс ставится перед скобками. Результирующий адрес находится добавлением индекса к значению регистра. Например, если %ebp присвоено значение 20, эффективный адрес -8 (%ebp) будет (-8)+(20)= 12:

```
count.s
1             .file  "count.c"
2             .version  "01.01"
3             gcc2 compiled.:
4             .text
5             .align 4
6             .globl main
7             .type main,@function
8 main:
#Создание локальной области в памяти из 8 байт для for i и j.
9      pushl  %ebp
10     movl   %esp, %ebp
11     subl   $8, %esp

инициализация i (ebp-4) и j (ebp-8) нулем.
12     movl   $0, -8(%ebp)
13     movl   $0, -4(%ebp)
14     .p2align 2
15     .L3:

#Проверка для цикла for
16     cmpl   $7, -4(%ebp)
17      jle   .L6
18  jmp   .L4
19  .p2align 2
20  .L6:

#Это тело цикла for-loop
21  movl   -4(%ebp), %eax
22  leal   -8(%ebp), %edx
23  addl   %eax, (%edx)
24  leal   -4(%ebp), %eax
```

```
25  incl   (%eax)
26  jmp    .L3
27  .p2align 2
28  .L4:

#Конструкция для вызова из функции
29  movl  $0, %eax
3 0  leave
31  ret
```

Строка 9
Установка базового стекового указателя на стек.

Строка 10
Перемещение указателя на стек в базовый указатель.

Строка 11
Получение 8 байт стека mem начиная с ebp.

Строка 12
Помещение 0 в адрес ebp-8 (j).

Строка 13
Помещение 0 в адрес ebp-4 (i).

Строка 14
Ассемблерная директива, обозначающая выровненную по полуслову инструкцию.

Строка 15
Созданная на ассемблере метка с именем .L3.

Строка 16
Эта инструкция сравнивает значения i с 7.

Строка 17
Переход на метку .L6, если -4 (%ebp) меньше или равно 7.

Строка 18
В противном случае выполняется переход на метку .L4.

Строка 19
Выравнивание.

Строка 20
Метка .L6.

Строка 21
Перемещение i в eax.

Строка 22
Загрузка адреса j в edx.

Строка 23
Добавление i к адресу, на который указывает edx (j).

Строка 24
Перемещение нового значения i в eax.

Строка 25
Инкрементирование i.

Строка 26
Обратный переход в проверку цикла for.

Строка 27
Выравнивание, как описано в комментарии к строке 14.

Строка 28
Метка .L4.

Строка 29
Установка кода возврата в eax.

Строка 30
Освобождение локальной области в памяти.

Строка 31
Извлечение переменной из стека, извлечение адреса возврата, и обратный переход в вызывающий код.

2.3.2 Пример ассемблера PowerPC

Следующий результирующий PPC-ассемблерный код с С-программы. Если вы знакомы с языком ассемблера (и его акронимами), вам станут понятны большинство функций PPC. Тем не менее существует несколько различных форм базовых инструкций, обсуждаемых далее.

stwu, RS, D (RA) (Store Word with Update, сохранение слова с обновлением). Эта инструкция берет значение в регистре (GPR) RS и сохраняет его по эффективному адресу, формируемому как RA+D. После этого регистр RA (GPR) *обновляется* новым эффективным адресом.

li RT, RS, SI (Load Immediate, непосредственная загрузка). Эта расширенная мнемоника для загрузки инструкций с фиксированной точкой. Эквивалентна добавлению

RT, RS, S1, где сумма RS (GPR) и S1, являющаяся двоичным 16-битовым дополнением, сохраняется в RT. Если RS - это RO (GPR), значение SI сохраняется в RT. Обратите внимание, что используется только 16-битовое значение при том, что opcode, регистры и значения должны кодироваться в 32-битовые инструкции.

lwz RT, D(RA) (Load Word and Zero, загрузка слова и нуля). Эта инструкция формирует эффективный адрес наподобие **s** twu и загружает слово данных из памяти в RT (GPR); «and Zero» означает, что верхние 32 бита вычисляемого эффективного адреса устанавливаются в 0, если 64-битовая реализация запускается в 32-битовом режиме. (Более подробно ее реализация описана в *PowerPC Architecture Book L)*

blr (Branch to Link Register, переход к регистру ссылки). Эта инструкция безусловного перехода по 32-битовому адресу в регистре ссылки. При вызове этой функции вызывающий код помещает адрес возврата в регистр ссылки. Подобно x86-инструкции ret, blr является простым методом возврата из функции.

Следующий код сгенерирован в результате ввода в командную строку дсс -S count. c:

```
countppc. s
1   .file  "count, c"
2   .section     H. text"
3   .align 2
4   .globl main
5   .type main, ©function
6 main:
#Создание 32-битовой области в памяти из пространства стека
#и инициализация i и j.
7    stwu 1,-32(1)  #Сохранение 32 байт stack ptr (rl) в стек
8    stw 31,28(1)   #Сохранение слова г31 в нижнюю часть области памяти
9    mr 31,1       Перемещение содержимого rl в г31
10   li 0,0       # Загрузка 0 в г0
11   stw 0,12(31) # Сохранение слова г0 в эффективный адрес 12(r31), var j
12   li 0,0       # Загрузка 0 в r0
13   stw 0,8(31)  # Сохранение слова г0 в эффективный адрес 8(r31) , var i
14   .L2: #Проверка
цикла for
15   lwz 0,8(31)  # Загрузка 0 в г0
16   cmpwi 0,0,7 #Сравнение слова, следующего за г0 с целым значением 7
17   Ые 0, .L5  #Переход на метку L5, если меньше или равно
18   b .L3       #Безусловный переход на метку .L3
19   .L5:
#Тело цикла for
20   lwz 9,12(31)  #Загрузка j в г9
21   lwz 0,8(31)   # Загрузка i в г0
22   add 0,9,0    #Добавление г0 к г9 и помещение результата в г0
```

```
23  stw 0,12(31)  #Сохранение гО в j
24  lwz 9,8(31)   # Загрузка i в г9
25  addi 0,9,1   # Добавление 1 к г9 и помещение результата в гО
26  stw 0,8(31)   # Сохранение гО в i
27  Ь .L2
28  .1,3:
29  li 0,0   # Загрузка 0 в гО
30  mr 3,0   #перемещение гО в гЗ
31  lwz 11,0(1)    # Загрузка rl в rll
32  lwz 31,-4(11)  #Восстановление г31
33  mr 1,11       # Восстановление rl
34  Ыг             #Безусловный переход по содержимому регистра ссылки
```

Строка 7

Сохранение 32 байтов stackptr (rl) в стек.

Строка 8

Сохранение слова г31 в нижнюю часть области памяти.

Строка 9

Перемещение содержимого rl в г31.

Строка 10

Загрузка 0 в гО.

Строка 11

Сохранение слова гО в эффективный адрес 12 (r31), var j.

Строка 12

Загрузка 0 вгО.

Строка 13

Сохранение слова гО в эффективный адрес 8 (гЗ 1), var i.

Строка 14

Метка .L2 :.

Строка 15

Загрузка i в гО.

Строка 16

Сравнение слова, следующего за гО с целым значением 7.

Строка 17

Переход на метку .L5, если меньше или равно.

Строка 18

Безусловный переход на метку .L3.

Строка 19
Метка .L5:.

Строка 20
Загрузка j вг9.

Строка 21
Загрузка i в гО.

Строка 22
Добавление г0 к г9 и помещение результата в гО.

Строка 23
Сохранение г 0 в j.

Строка 24
Загрузка i вг9.

Строка 25
Добавление 1 к г9 и помещение результата в гО.

Строка 26
Сохранение г 0 в i.

Строка 27
Безусловный переход на метку .L2.

Строка 28
Метка .L3:.

Строка 29
Загрузка 0 в гО.

Строка 30
Перемещение г0 в г3.

Строка 31
Загрузка rl в г 11.

Строка 32
Восстановление г31.

Строка 33
Восстановление rl.

Строка 34
Безусловный переход в место, указанное содержимым регистра ссылки.

Сравнивая эти два ассемблерных файла, можно увидеть, что они состоят практически из одинакового количества строк. При более подробном изучении вы увидите, что PRISC (PPC)-процессоры характеризуются использованием множества инструкций загрузки и сохранения, тогда как COSC (x86) чаще всего используют инструкции mov.

2.4 Ассемблерные вставки

Еще одной формой кодирования, разрешенной gcc-компилятором, являются ассемблерные вставки. Как следует из их имени, ассемблерные вставки не требуют вызова отдельно скомпилированных ассемблерных программ. Используя определенные инструкции, мы можем указать компилятору, что данный блок кода нужно не компилировать, а ассемблировать. Несмотря на то что в результате получается архитектурно-зависимый файл, читаемость и эффективность С-функции сразу увеличивается. Вот конструкция ассемблерной вставки:

```
1 asm ассемблерная инструкция(ции)
2   : операнды вывода (опционально)
3   : операнды ввода (опционально)
4   : затираемые регистры (опционально)
5   );
```

Например, такая простая форма:

asm (*mov %eax, %ebx"); **может быть переписана в виде** asm ("movl %eax, %ebx" : : :) ;

Здесь мы «обманываем» компилятор, потому что мы затираем регистр ebx. (Читайте об этом дальше.)

Ассемблерные вставки являются настолько гибкими благодаря способности брать С-выражения, модифицировать их и возвращать их в программу с полной уверенностью, что компилятор их воспримет. Далее мы рассмотрим передачу параметров.

2.4.1 Операнды вывода

В строке 2 за двоеточием операнд вывода перечисляет С-выражения в круглых скобках начиная с условий. Для операндов вывода условия обычно имеют модификатор =, означающий, что они доступны только для чтения. Модификатор & показывает, что это операнд «ранней очистки», это значит, что операнд освобождается до того, как инструкция заканчивает его использовать. Каждый операнд отделяется запятой.

2.4.2 Операнд ввода

Операнд ввода в строке 3 использует тот же синтаксис, что и операнд вывода за исключением модификатора только для чтения.

2.4.3 Очищаемые регистры (или список очистки)

В нашей ассемблерной вставке мы можем модифицировать различные регистры и память. Для того чтобы дсс знал, что это за элементы, нам нужно перечислить их здесь.

2.4.4 Нумерация параметров

Каждый параметр получает порядковый номер начиная с 0. Например, если есть параметр вывода и два входных параметра, %0 указывает на параметр вывода, а % 1 и % 2 - на параметры ввода.

2.4.5 Ограничения

Ограничения обозначают то, как могут использоваться операнды. Документация GNU перечисляет полный список простых ограничений и машинных ограничений. В табл. 2.4 перечислены наиболее используемые ограничения для x86.

Таблица 2.4. Простые и машинные ограничения длях8б

Ограничение	Функция
a	Регистр eax
b	Регистр ebx
c	Регистр ecx
d	Регистр edx
S	Регистр esi
D	Регистр edi
I	Значение ограничения (0... 31)
q	Динамически выделяемый среди eax, ebx, ecx, edx регистр
r	То же, что и q + esi, edi
m	Позиция в памяти
a	То же, что и a + Ь; eax и ebx выделяются вместе в виде 64-битового регистра

2.4.6 asm

На практике (особенно в ядре Linux) ключевое слово **asm** может привести к ошибке при компиляции из-за наличия других инструкций с тем же именем. Зачастую можно увидеть выражения наподобие _ asm__ , означающие то же самое.

2.4.7 _volatile_

Еще одним часто используемым модификатором является___ **volatile** __. Этот моди фикатор важен для ассемблерного кода. Он указывает компилятору не оптимизировать содержимое ассемблерной вставки. Зачастую в программах аппаратного уровня компи лятор может подумать, что мы пишем слишком много ненужного кода, и попытается оп тимизировать наш код, насколько это возможно. При разработке прикладных программ это полезно, но на аппаратном уровне результат может оказаться полностью противопо ложным.

Например, представим, что мы пишем в спроецированный в память регистр, пред ставленный переменной **reg.** Затем мы выполняем действие, требующее опросить reg. Компилятор видит, что мы выполняем бессмысленное чтение той же самой области памя ти, и удаляет бесполезную команду. При использовании __ volatile _____компилятор бу дет знать, что оптимизировать доступ к этой переменной не нужно. Аналогично, когда вы видите asm volatile (...) в блоке ассемблерного кода, компилятор не будет оптими зировать код этого блока.

Теперь, когда мы знаем основы ассемблера и встроенного ассемблера дсс, мы мо-жем обратить свое внимание на сам встраиваемый ассемблерный код. Используя то, что мы только что изучили, мы сначала рассмотрим простой пример, а затем немного более сложный блок кода.

Вот первый пример кода, в котором мы передаем переменную внутрь встроенного участка кода:

```
6   int  foo(void)
7   {
8   int ee = 0x4000, ce = 0x8000, reg;
9     asm     volatile  ("movl %1, %%eax" ;
10   "movl %2, %%ebx";
11   "call setbits-
12   -movl %%eax, %0"
13   : "=r" (reg) // reg [параметр %0] - вывод
14   : "r" (ce) , Hr- (ee) // ce [параметр %1] , ee [параметр %2] - ввод
15   : -%eax- , -%ebx"   // %eax и % ebx очищаются
16   )
17  printf (Hreg=%x-, reg) ;
18     }
```

Строка 6
В этой строке находится обычное С-начало функции.

Строка 8
ее, се и req - локальные переменные, передаваемые в ассемблерную вставку в качестве параметров.

Строка 9
В этой строке находится обычное ассемблерное выражение. Помещение сев eax.

Строка 10
Помещение ее в ebx.

Строка 11
Вызов функции из ассемблера.

Строка 12
Возвращение значения в eax и его копирование в reg.

Строка 13
Эта строка содержит перечень выходных параметров. Параметр reg доступен только для чтения.

Строка 14
Эта строка содержит перечень входных параметров. Параметры сей ее - переменные регистров.

Строка 15
В этой строке находится перечень затираемых регистров. Далее изменяются регистры eax и ebx. Компилятор знает, что после этого выражения использовать данные значения нельзя.

Строка 16
Эта строка обозначает конец ассемблерной вставки.

Второй пример использует функцию switch_to() из include/asm-i386/ system, h. Эта функция - сердце переключения контекстов Linux. В этой главе мы рассмотрим только механизм ассемблерных вставок. Гл. 9, «Построение ядра Linux», описывает использование switch_to ().

```
include/asm-i386/system.h
012  extern struct task_struct * FASTCALL(__switch_to(struct
task struct *prev, struct task struct *next));

015  #define switch__to (prev,next, last) do {
```

```
16    unsigned long esi,edi;
17    asm volatile ("pushf l\n\t"
18    "pushl %%ebp\n\t"
19    "movl %%esp,%0\n\t"  /* сохранение ESP */
20    "movl %5,%%esp\n\t"  /* восстановление ESP */
21    "movl $lf,%l\n\t"  /* сохранение EIP */
22    "pushl %6\n\t"  /* восстановление EIP */
23    "jmp  _ switch_to\n"
23    "l:\t"
24    "popl %%ebp\n\t"
25    "popfl"
26    :"=m" (prev->thread.esp),"=m" (prev->thread.eip),
27    "=a" (last),"=S" (esi),"=D" (edi)
28    :"m" (next->thread.esp),"m" (next->thread.eip),
29    "2" (prev), "d" (next));
030   } while (0)
```

Строка 12

FASTCALL указывает компилятору передавать параметры в registers, asmlinkage указывает компилятору передавать параметры в stack.

Строка 15

Метод do {statements...} while (0) позволяет макросу представляться компилятору в более похожем на функцию виде. В этом случае он позволяет использовать локальные переменные.

Строка 16

Не смущайтесь: это просто имя локальной переменной.

Строка 17

Это ассемблерная вставка, не требующая оптимизации.

Строка 23

Параметр 1 используется как адрес возврата.

Строки 17-24

\n\t выполняется через интерфейс компилятора/ассемблера. Каждая ассемблерная инструкция записывается на своей строчке.

Строка 26

prev-> thread. esp и prev->thread. eip - выходные параметры: [%0] = (prev->thread, esp), только для чтения [%1] = (prev->thread, eip), только для чтения

Строка 27

%2] = (las t) - только для чтения в регистр eax:
[%3] = (esi), только для чтения в регистр esi
[%1] = (prev->thread, eip), только для чтения в регистр edi

Строка 28

Вот параметры ввода:
[%5] = (next->thread.esp), в памяти
[%б]= (next->thread.eip), в памяти

Строка 29

[%7] = (prev) - повторное использование параметра «2» (регистр eax) как входного:
[%8] = (next), назначен как ввод для регистра esx.
Обратите внимание, что здесь нет списка очистки.

Встроенный ассемблер для PowerPC практически идентичен по конструкции встроенному ассемблеру для x86. Простые ограничения, такие, как «т» и «п>, используются вместе с набором машинных ограничений PowerPC. Вот простой пример обмена 32-битового указателя. Обратите внимание, насколько этот встроенный ассемблер похож на ассемблер для x86:

```
include/asm-ppc/system.h
103  static __inline__ unsigned long
104  xchg_u32(volatile void *p, unsigned long val)
105  {
106   unsigned long prev;
107
108     asm     volatile   (M\n\
109  1:  lwarx %0,0,%2 \n"
110
111   "  stwcx. %3,0,%2 \n\
112    bne-  lb"
113    : "=&r" (prev), "=m" (*(volatile unsigned long *)p)
114    : Hr^H (p) , "r" (val), "m" (*(volatile unsigned long *)p)
115    : "cc", "memory");
116
117    return prev;
118   )
```

Строка 103

Эта подпрограмма выполняется на месте; она не вызывается.

Строка 104

Подпрограмма получает параметры p и val.

Строка 106

Локальная переменная prev.

Строка 108

Встроенный ассемблер. Не требует оптимизации.

Строки 109-111

lwarx вместе с stwcx формирует «атомарный обмен»; lwarx загружает слово из памяти и «резервирует» адрес для последующего сохранения из s twcx.

Строка 112

Переход, если не равно, на метку 1 (b = backward - в обратную сторону).

Строка 113

Вот операнды вывода:
[%0] = (prev), только для чтения, ранняя очистка
[%1]= (* (volatile unsigned long *)p), операнд в памяти только для чтения.

Строка 114

Вот операнды ввода:
[%2] = (p), операнд регистра
[%3] = (val), операнд регистра
[%4]= (* (volatile unsigned long *)p), операнд памяти

Строка 115

Вот операнды очистки:
[%5] = изменение кода регистра состояния
[% б] = очистка памяти

На этом мы завершаем обсуждение языка ассемблера и его использования в ядре Linux 2.6. Мы увидели, чем архитектуры PPC и x86 различаются и что у них общее. Ассемблерные технологии программирования используются в зависимости от платформы. Теперь обратим наше внимание на язык программирования C, на котором написана большая часть ядра Linux, и рассмотрим основные проблемы программирования на языке C.

2.5 Необычное использование языка C

Внутри ядра Linux действует множество соглашений, требующих много чего прочитать и изучить для понимания их использования и назначения. Этот раздел освещает некоторые неясности или неточности использования C, фокусируясь на принятых C-соглашениях, используемых в ядре Linux 2.6.

2.5.1 asmlinkage

asmlinkage указывает компилятору передавать параметры в локальный стек. Это связано с макросом FASTCALL, который указывает компилятору (аппаратно-зависимому) передавать параметры в регистры общего назначения. Вот макрос из include /asm/ linkage, c:

```
include/asm/linkage.h
4  #define asmlinkage CPP ASMLINKAGE   attribute   ((regparm(O) ) )
5  #define FASTCALL(x)  x   attribute   ((regparm(3)))
6  #define fastcall  _attribute_ ((regparm(3)))
```

Далее представлен пример asmlinkage.

asmlinkage long sys_gettimeofday (struct timeval _____ user *tv, struct timezone ___ user *tz)

2.5.2 UL

UL часто вставляется в конце численных констант для обозначения «unsigned long». UL (или L для long) необходимо вставлять для того, чтобы указать компилятору считать значение имеющим тип long[1]. На некоторых архитектурах таким образом можно избежать переполнения и выхода за границы типа. Например, 16-битовое целое может представлять числа от -32768 до +32767; беззнаковое целое может представлять числа от 0 до 65535. При использовании UL вы пишете архитектурно-независимый код для длинных чисел или длинных битовых масок.

Вот некоторые демонстрирующие это примеры из ядра:

```
include/linux/hash.h
18  idefine GOLDEN_RATIO_PRIME 0x9e370001UL

include/linux/kernel.h
23  #define ULONG_MAX  (-OUL)

include/linux/slab, h
39  #define SLAB_POISON   0x00000800UL  /* Ядовитые объекты */
```

[1] Очевидно, имеется в виду unsigned long. *Примеч. науч. ред.*

2.5.3 inline

Ключевое слово **inline** необходимо для оптимизации выполнения функций, интегрируя код этих функций в вызывающий код. Ядро Linux использует множество inline-функций, объявленных статическими; «static тНпе»-функция заставляет компилятор стараться внедрять код функции во все вызывающие ее участки кода и, если это возможно, избегать ассемблирования кода этой функции. Иногда компилятор не может обойтись без ассемблирования кода (в случае рекурсий), но в большинстве случаев функции, объявленные как static inline, полностью внедряются в вызывающий код.

Целью такого внедрения является устранение всех лишних операций, выполняемых при вызове функции. Выражение # define также позволяет убрать связанные с вызовом функции операции и обычно используется для обеспечения портируемости на другие компиляторы и встраиваемые системы.

Так почему бы не сделать встроенными все функции? Недостатком использования встраивания является увеличение бинарного кода и иногда замедление доступа к кешу процессора.

2.5.4 const и volatile

Эти два ключевых слова игнорируются многими начинающими программистами. Ключевое слово const не следует понимать как *константу,* а скорее как *только для чтения*. Например, *const int be* - это указатель на const-целое. При этом указатель может быть изменен, а целое число - нет. С другой стороны, *int const* *JC обозначает const-указатель на целое, когда число может быть изменено, а указатель - нет. Вот пример использования const:

```
include/asm-i3 86/processor.h
62 8  static inline void prefetch(const void *x)
629   {
630     asm     volatile    ("debt 0,%0" : : "r" (x) ) ;
631   }
```

Ключевое слово volatile (временный) означает переменную, которая не может быть изменена без замечания; volatile сообщает компилятору, что ему нужно перезагрузить помеченную переменную каждый раз, когда она встречается, а не сохранять и получать доступ к ее копии. Хорошим примером переменной, которую нужно отметить как временную, являются переменные, связанные с прерываниями, аппаратными регистрами, или переменные, разделяемые конкурирующими процессами. Вот пример использования volatile:

```
include/linux/spinlock.h
51  typedef struct {
```

```
volatile unsigned int lock;

58 } spinlock_t;
```

Учитывая то, что const следует трактовать как только для чтения, мы видим, что некоторые переменные могут быть одновременно const и volatile (например, переменная, хранящая содержимое регулярно обновляемого аппаратного регистра с доступом только для чтения).

Этот краткий обзор позволит начинающим хакерам ядра Linux чувствовать себя увереннее при изучении исходных кодов ядра.

2.6 Короткий обзор инструментария для исследования ядра

После успешной компиляции и сборки вашего ядра Linux вы можете захотеть посмотреть его «внутренности» до, после или даже во время процесса этой операции. Этот раздел коротко рассказывает об инструментах, используемых обычно для просмотра различных файлов ядра Linux.

2.6.1 objdump/readelf

Утилиты **objdump** и **readelf** отображают информацию об объектных файлах (objdump) или ELF-файлах (readelf). С помощью аргументов командной строки вы можете использовать команды для просмотра заголовков, размера или архитектуры данного объектного файла. Например, вот дамп для ELF-заголовка простой C-программы (a.out), полученный с помощью флага -h readelf:

```
Lwp> readelf -h a.out
ELF Header:
Magic:         7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:         ELF32
Data:         2's complement, little endian
Version:       1 (current)
OS/ABI:        UNIX - System V
ABI Version:   0
Type:         EXEC (Executable file)
Machine:       Intel 80386
Version:       0x1
Entry point address:        0x8048310
Start of program headers:   52 (bytes into file)
Start of section headers:   10596 (bytes into file)
Flags:         0x0
Size of this header:       52 (bytes)
```

```
Size of program headers:    32 (bytes)
Number of program headers:  6
Size of section headers:    40 (bytes)
Number of  section headers: 29
Section header  string  table  index: 2 6
```

А вот дамп заголовка программы, полученный с помощью readelf с флагом -1:

```
Lwp> readelf -l a.out
Elf file type is EXEC (Executable file)
Entry point  0x8048310
There are 6 program headers, starting at offset 52
Program Headers: Type  Offset  VirtAddr  PhysAddr FileSiz
 MemSiz Fig Align PHDR  0x000034 0x08048034 0x08048034 0x000c0
 0x000c0 R E 0x4
INTERP 0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R  0x1
  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD   0x000000 0x08048000 0x08048000 0x00498 0x00498 R E 0x1000
LOAD   0x000498 0x08049498 0x08049498 0x00108 0x00120 RW 0x1000
DYNAMIC 0x0004ac 0x080494ac 0x080494ac 0x000c8 0x000c8 RW 0x4
NOTE  0x000108 0x08048108 0x08048108 0x00020 0x00020 R  0x4 Section to
Segment mapping:
Segment Sections...
00
1    .interp
2    .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
.gnu.version r rel.dyn .rel.plt .init .pit .text .fini .rodata
3    .data .eh frame .dynamic .ctors .dtors .got .bss
4    .dynamic
5    .note.ABI-tag
```

2.6.2 hexdump

Команда **hexdump** отображает содержимое указанного файла в шестнадцатеричном, ASCII или восьмеричном формате. [Обратите внимание: на старых версиях Linux также использовался od (восьмеричный дамп). Теперь большинство систем используют вместо него hexdump.]

Например, чтобы посмотреть первые 64 бита ELF-файла a.out в шестнадцатеричном режиме, нужно набрать следующее:

```
Lwp>   hexdump -x -n 64 a.out
0000000  457f 151c 0101 0001 0000 0000 0000        0000
0000010  0002 0003 0001 0000 8310 0804 0034        0000
0000020  2964 0000 0000 0000 0034 0020 0006        0028
0000030 OOld 001a 0006 0000 0034 0000 8034        0804
0000040
```

Обратите внимание на магическое число заголовка ELF (с поменянными местами байтами) по адресу 0x0000000.

Это очень полезно при отладке действий; когда аппаратное устройство сбрасывает свое состояние в файл, обычный текстовый редактор интерпретирует его как набор управляющих символов. Hexdump позволяет вам увидеть, что же на самом деле содержится в файле без промежуточного преобразования редактором; hexdump - это редактор, который позволяет вам напрямую модифицировать файл без преобразования его содержимого в ASCII (или Unicode).

2.6.3 nm

Утилита run перечисляет все символы, находящиеся в объектном файле. Она отображает значения символов, их тип и имя. Эта утилита не так полезна, как остальные, но тем не менее может быть полезна при отладке файлов библиотек.

2.6.4 objcopy

Команда **obj copy** используется, когда вам нужно скопировать объектный файл и при этом пропустить или изменить некоторые его компоненты. Обычно objcopy используется для получения отладочных символов от тестируемого и работающего объектного файла. В результате размер объектного файла уменьшается и становится возможным его использование на встраиваемых системах.

2.6.5 ar

Команда **ar,** или **archive** (архивация), помогает поддерживать индексированные библиотеки, используемые компоновщиком. Команда ar собирает один или несколько объектных файлов в одну библиотеку. Кроме этого, она может выделять отдельный объектный файл из библиотеки. Чаще всего команду ar можно увидеть в Make-файле. Она часто используется для объединения используемых функций в одну библиотеку. Например, предположим, что у вас есть функция, выполняющая парсинг командного файла и извлечение некоторых данных или вызов для извлечения информации для указанных аппаратных регистров. Эта функция необходима нескольким исполнимым программам. Архивирование этой функции в отдельную библиотеку облегчит вам контроль за версиями, поместив функцию только в одно место.

2.7 Говорит ядро: прослушивание сообщений ядра

После того как Linux-система запущена и начинает выполняться, ядро самостоятельно ведет журнал сообщений и предоставляет информацию о своем состоянии на протяжении всех своих операций. В этом разделе описаны несколько наиболее распространенных способов того, как ядро Linux может общаться с пользователем.

2.7.1 printk()

Одной из базовых систем сообщений ядра является функция **printk** (). Ядро использует printk() как замену print f (), потому что стандартные библиотеки С ядром не компонуются; printk () использует тот же интерфейс, что и print f (), и позволяет выводить в консоль до 1024 символов. Функция printk () оперирует, пытаясь перехватить семафор консоли, поместить вывод в буфер журнала сообщений консоли и затем выполнить вызов драйвера консоли для сброса буфера. Если printk () не может перехватить семафор консоли, она помещает вывод в буфер журнала и полагается на процесс, получивший семафор для сброса буфера. Буфер журнала блокируется до того, как printk () помещает данные в буфер журнала, так что конкурирующие вызовы к printk () не помешают друг другу. Если семафор консоли занят, перед сбросом буфера журнала может накопиться несколько вызовов printk (). Поэтому не пытайтесь использовать вызовы printk () для вывода времени работы программ.

2.7.2 dmesg

Ядро Linux сохраняет свой журнал или сообщения несколькими разными способами; sysklogdO является комбинацией syslogdO и klogdO. (Более подробную информацию можно найти на man-страницах этих команд, а здесь мы просто рассмотрим их в целом.) Ядро Linux посылает сообщения через klogd (), которая связывает их с соответствующим уровнем предупреждений, и помещает сообщения всех уровней в /proc/kmsg; **dmesg-**это утилита командной строки для отображения буфера, хранимого в /proc/kmsg, и опционально фильтрующая сообщения на основе их уровня.

2.7.3 /val/log/messages

По этому адресу постоянно хранится большинство журналов сообщений системы. Программа syslogd () читает информацию из /etc/syslog. conf для получения позиции, где хранятся сохраненные сообщения. В зависимости от содержимого syslog. conf, которое может различаться для разных дистрибутивов Linux, сообщения журналов могут быть сохранены в нескольких файлах. Тем не менее чаще всего стандартным хранилищем является **/var/log/messages.**

2.8 Другие особенности

Этот раздел касается разнообразных особенностей, портящих жизнь начинающим исследователям кода ядра. Мы привели их здесь, чтобы помочь вам вникнуть во внутреннее строение Linux.

2.8.1 _init

Макрос __ **init** указывает компилятору, что связанная с ним функция или переменная используется только во время инициализации. Компилятор помещает весь код, помечен ный _ init, в специальную область памяти, которая освобождается по завершении фазы инициализации:

```
drivers/char/random.c
 679 static int ____ init batch_entropy_init (int size, struct entropy_store *r)
```

В качестве примера можно привести драйвер устройства случайных чисел, который инициализируется пулом энтропии во время загрузки. Во время загрузки драйвера раз личные функции используются для увеличения или уменьшения пула энтропии. Практика пометки инициализации драйверов устройств с помощью __ init является скорее об щепринятой, а не стандартной.

Аналогично, если данные применяются только во время инициализации, их нужно пометить как __ **initdata.** Ниже вы можете видеть, как _____ initdata используется в драйвере устройства ESP:

```
drivers/char/esp. c
 107 static char serial_name[] _____ initdata ="ESP serial driver";
 108 static char serial_version[] _____initdata = "2.2";
```

Кроме этого, существуют макросы **__exit и __exitdata,** которые используются при завершении работы. Их часто применяют во время выгрузки драйверов устройств.

2.8.2 likely() и unlikely()

likely () и **unlikely** () - это макросы, используемые разработчиками ядра Linux для того, чтобы давать подсказки компилятору и чипсету. Современные процессоры обладают мощным блоком предсказания, который пытается предсказывать поступающие команды и их порядок для оптимизации скорости выполнения. Макросы likely () и unlikely () позволяют разработчику указать процессору через компилятор наиболее вероятный участок кода, который следует предсказать, или наименее вероятный, который предсказывать не нужно.

Важность предсказания можно увидеть, только поняв *конвейер инструкций.* Современные процессоры выполняют предварительную выборку, т. е. они предварительно выбирают несколько следующих инструкций, которые будут исполнены, и загружают их в процессор. Внутри процессора эти инструкции исследуются и распределяются на несколько юнитов процессора (целые, с плавающей точкой и т. д.) в зависимости от того, как

их лучше выполнить. Некоторые инструкции могут застрять в процессоре, ожидая промежуточных результатов от предыдущих инструкций. Теперь представьте, что в поток инструкций загружены инструкции перехода. Процессор имеет два потока инструкций, из которых можно продолжить выполнение. Если процессор слишком часто предсказывает неправильную ветвь, он тратит слишком много времени на перезагрузку конвейера инструкций на выполнение. А что если процессор получит подсказку, какой путь выбирать? Одним из простейших методов предсказания условного перехода на некоторых архитектурах является анализ целевого адреса перехода. Если значение находится впереди текущего, есть большая вероятность, что этот переход является концом цикла предыдущего адреса, который много раз возвращается на первую позицию.

Программное обеспечение имеет возможность переопределять архитектурные предсказания переходов с помощью специальных мнемоник. Это свойство поддерживается компилятором с помощью функции **__builtin_expect** (), являющейся базой для макросов likely () и unlikely ().

Как было сказано ранее, предсказание переходов и процессорный конвейер инструкций достаточно сложен, чтобы входить в пределы рассмотрения данной книги, но способность «подстраивать» код в случае необходимости является большим плюсом в плане обеспечения производительности. Посмотрите на следующий блок кода:

```
kernel/time.c
 90 asmlinkage long sys_gettimeofday(struct timeval   user *tv,
                                     struct timezone   user *tz)
  91 {
  92   if (likely(tv != NULL)) {
  93     struct timeval ktv;
  94     do gettimeofday(&ktv);
  95     if (copy to  user (tv, &ktv, sizeof (ktv) ) )
  96       return -EFAULT;
  97   }
  98   if (unlikely(tz !=NULL)) {
  99     if (copy to  user (tz, &sys tz, sizeof (sys tz) ) )
 100       return -EFAULT;
 101   )
 102   return 0;
 103  }
```

В этом коде мы видим, что syscall, вероятнее всего, получает время для получения структуры tymeval, не равной нулю (строки 92-96). Если она равна нулю, мы не можем заполнить ее требуемым временем дня! Менее вероятно то, что временная зона не равна нулю (строки 98-100). Короче говоря, вызывающий код реже всего запрашивает временную зону и обычно запрашивает время.

Особые реализации likely () и unlikely () определяются следующим образом[1]:

```
include/linux/compiler.h
  45 #define likely(x)   __builtin_expect(!1(x), 1)
  46 #define unlikely(x)   __builtin_expect(!!(x), 0)
```

2.8.3 IS_ERR и PTR_ERR

Макрос **IS_ERR** декодирует отрицательное число ошибки в указатель, а макрос **PTR.JERR** получает код ошибки из указателя.

Оба макроса определены в include/linux/err. h.

2.8.4 Последовательности уведомлений

Механизм последовательностей уведомлений позволяет ядру регистрировать свой интерес к оповещению о появлении переменной асинхронного события. Этот обобщенный интерфейс распространяется на использование во всех подсистемах или компонентах ядра.

Последовательность уведомлений - это просто связанный список объектов **notifier_block:**

```
include/linux/notifier.h
14   struct notifier block
15   {
16   int(*notifier call)(struct notifier block *self, unsigned long,
void'*);
17   struct notifier block *next;
18   int priority;
19   };
```

notif ier_block содержит указатель на функцию **(notif ier_call)** для вызова при наступлении события. Параметры этой функции включают указатель на notif ier_ block, содержащий информацию, значение, соответствующее событию, или флаг, и указатель на тип данных, определяемый подсистемой.

Кроме этого, структура notifier_block содержит указатель на следующий notif ier_block в последовательности и описание приоритета.

[1] **Как видно из отрывка кода,___ builtin_expect О обнуляется до версии GCC 2.96, потому что до этой версии GCC не обладал возможностью влиять на предсказание переходов.**
Из этого отрывка это не следует; чтобы это увидеть, надо посмотреть на /usr/include/linux/compiler-gcc2.h.
Примеч. науч. ред.

Функции **not if ier_chain_register ()** и **not if ier_chain_unregister** () регистрируют и удаляют объект not if ier_block в указанной последовательности уведомлений.

Резюме

В этой главе описано достаточно информации, для того чтобы вы могли начать исследование ядра Linux. Было описано два метода динамических хранилищ: связанные списки и деревья бинарного поиска. Полученные базовые знания об этих структурах помогут вам в дальнейшем при обсуждении других тем, например таких, как процессы и процесс подкачки. Затем мы рассмотрели основы языка ассемблера, что поможет вам в разборе кода или отладке на машинном уровне, и, акцентировав внимание на ассемблерных вставках, мы показали возможность совмещения С и ассемблера внутри одной функции. Мы закончили эту главу обсуждением различных команд и функций, необходимых для изучения различных аспектов ядра.

Проект Hellomod

Этот раздел представляет базовые концепции, необходимые для понимания других Linux-концепций и структур, описанных далее в книге. Наши проекты концентрируются на создании загружаемых модулей, использующих новую, 2.6-архитектуру драйверов, и построении на базе этих модулей следующих проектов. Так как драйверы устройств могут быстро стать слишком сложными, нашей целью является только познакомить вас с базовыми конструкциями модулей Linux. Мы доработаем этот драйвер в следующих проектах. Данный модуль запускается как на PPC, так и на x86.

Шаг 1: написание каркаса модуля Linux

Первым модулем, который мы напишем, является символьный драйвер устройства «hello world». Сначала мы рассмотрим базовый код для модуля, а затем покажем, как его откомпилировать с помощью новой системы Makefile 2.6 (это обсуждается в гл. 9), и наконец, присоединим и удалим наш модуль от ядра с использованием команд insmod и rramod соответственно[1]:

```
hellomod.c
001
// hello world driver for Linux 2.6

4    #include <linux/module.h>
5    #include <linux/kernel.h>
```

[1] **Убедитесь, что в вашей конфигурации включена поддержка выгрузки модулей.**

```
6    #include <linux/init.h>
7    #MODULE_LICENCE("GPL"J[1];  //избавимся от ненужного сообщения

009  static int  _init lkp_init( void )
{
  printk(''<l>Hello/World! from the kernel space...\n");
  return 0;
013  }

015  static void  _exit lkp__cleanup ( void )
{
  printk("<l>Goodbye, World! leaving kernel space ...\n");
018  }

20    module_init(lkp_init);
21    module_exit(lkp_cleanup);
```

Строка 4

Все модули используют заголовочный файл module. h, который должен быть подключен.

Строка 5

Файл kernel. h содержит основные функции ядра.

Строка 6

Заголовочный файл init.h содержит макросы_____init и ____ exit. Эти макросы позволяют освободить память ядра. Рекомендуем вам бегло ознакомиться с кодом и комментариями в этом файле.

Строка 7

Для предупреждения об отсутствии GNU открытой лицензии в ядре начиная с версии 2.4 присутствует несколько специальных макросов. (За более подробной информацией обращайтесь к файлу modules .h.)

Строки 9-12

Это функция инициализации нашего модуля. Эта функция должна, например, содержать код создания и инициализации структур. В строке 11 мы можем послать сообщение из ядра с помощью printk (). Мы сможем увидеть это сообщение при загрузке нашего модуля.

[1] **В оригинальном тексте опечатка - должно быть '007 MODULE_LICENSE ("GPL") ;'. *Примеч. науч. ред.***

Строки 15-18

Это функция выхода из нашего модуля и очистки памяти. Здесь мы прибираем все, связанное с нашим драйвером при его уничтожении.

Строка 20

Это точка инициализации драйвера. Ядро вызывает ее во время загрузки для встроенных модулей или во время подгрузки для загружаемых модулей.

Строка 21

Для загружаемых модулей ядро вызывает функцию cleanup_module (). Для встроенных модулей она не дает никакого эффекта.

Мы можем иметь в нашем драйвере только одну точку инициализации (module__init) и одну точку очистки (module_exit). Эти функции ядро ищет при загрузке и выгрузке нашего модуля.

Шаг 2: компиляция модуля

Если вы используете старые методы построения модуля ядра (например, те которые начинаются с #define MODULE), новый метод немного изменился. Для тех, кто впервые использует модуль 2.6, это будет довольно просто. Далее приведен базовый Makefile для нашего одного модуля.

Makefile

```
002    # Makefile  for Linux Kernel  Primer module skeleton   (2.6.7)
006         obj-m += hellomod.o
```

Обратите внимание, что мы указываем системе сборки, что мы компилируем загружаемый модуль. Вызов для командной строки этого Makefile завернут в bash-скрипт, называемый doit следующего вида:

```
001 make  -C   /usr/src/linux-2.6.7  SUBDIRS=$PWD modules[1]
```

Строка 1

Опция -C указывает make изменить директорию исходников Linux (в нашем случае /usr/src/linux-2 .6.7) перед чтением Makefile или другими действиями.

Перед выполнением . /doit вы увидите примерно следующий вывод:

```
Lkp#   ./doit
make: Entering directory •/usr/src/linux-2.6.7•
```

[1] Удобно использовать следующую команду, которая не требует явным образом прописывать версию ядра: make -C /usr/src/linux-`uname -r` SUBDIRS=$PWD modules. *Примеч. науч.ред.*

```
  CC [M]  /mysource/hellomod.o
  Building modules, stage 2
  MODPOST
  CC    /mysource/hellomod.o
  LD  [M]    /mysource/hellomod.ko make:  Leaving
directory  •/usr/src/linux-2.6.7• lkp# _
```

Для тех, кто компилировал или создавал модули Linux с помощью более ранних версий Linux, мы можем сказать, что теперь у нас есть шаг связывания LD и что нашим выходным модулем является hellomod. ко.

Шаг 3: запуск кода

Теперь мы готовы вставить новый модуль в ядро. Мы сделаем это с помощью команды insmod следующим образом:

```
lkp# insmod hellomod.ко
```

Для проверки того, что модуль был вставлен правильно, вы можете использовать команду lsniod следующим образом:

```
lkp#  lsmod
Module     Size Used by
hellomod    2696  0
lkp#
```

Вывод нашего модуля генерируется с помощью printk (). Эта функция по умолчанию выполняет печать в файл /var/log/messages. Для его быстрого просмотра напечатайте следующее:

```
lkp#  tail  /var/log/mesasages
```

Это вывод 10 последних строк файла журнала. Вы увидите наше сообщение инициализации:

```
Mar  6  10:35:55  lkpl  kernel:  Hello,  World!  From the kernel  space...
```

Для удаления нашего модуля (и просмотра нашего сообщения выхода) используйте команду rnmod с именем модуля, которое можно увидеть с помощью команды insmod. Для нашей программы эта команда будет выглядеть следующим образом:

```
Lkp#  rnmod hellomod
```

И опять наш вывод в файле журнала будет выглядеть следующим образом:

```
Mar  6 12:00:05 lkpl kernel: Hello, World! From the kernel space...
```

В зависимости от настроек вашей X-системы или используемой вами командной строки вывод printk пойдет на вашу консоль наравне с файлом журнала. В нашем следующем проекте мы коснемся его снова при рассмотрении переменных системных задач.

Упражнения

1. Опишите, как в ядре Linux реализованы хеш-таблицы.
2. Структура, являющаяся членом дву связного списка, будет иметь структуру list_head. Перед вставкой структуры list_head в ядре структура будет иметь поля prev и next, указывающие на другие похожие структуры. Зачем создавать структуру только для хранения указателей prev и next?
3. Что такое ассемблерные вставки и почему вам может понадобиться их использовать?
4. Представьте, что вы пишете драйвер устройства, получающий доступ к регистрам последовательного порта. Объявите ли вы эти адреса как volatile? И если да, то почему.
5. Зная, что делает __ init, как вы думаете, какого рода функции могут использоваться с этим макросом?

Глава 3

Процессы: принципиальная модель выполнения

В этой главе:

Термин **процесс** понимается здесь как базовый элемент выполнения программы и является наиболее важной концепцией, которую необходимо понять для изучения работы операционной системы. Необходимо понять разницу между программой и процессом. Следовательно, мы подразумеваем под *программой* исполнимый файл, содержащий набор функций, а под *процессом* конкретный экземпляр данной программы. Процесс - это элемент операции, которая использует ресурсы, предоставляемые аппаратным обеспечением, и выполняется в соответствии с указаниями программы, которая его запустила.

Компьютеры выполняют множество вещей. Процессы могут выполнять задачи начиная с выполнения пользовательских команд и заканчивая управлением системными ресурсами для доступа к аппаратным ресурсам. Вкратце процесс можно назвать набором инструкций, которые он выполняет, содержимым регистров и программным счетчиком выполнения программы, а также **состоянием.**

Процесс как динамическая структура принимает множество состояний. Кроме этого, у процесса есть свой жизненный цикл: после создания процесса он живет в течение некоторого времени, во время которого проходит через множество состояний, а затем умирает. Рис. 3.1 демонстрирует общую картину жизненного цикла процесса.

Во время работы Linux-системы количество запущенных процессов не является постоянным. Процессы создаются и уничтожаются по мере необходимости.

Процесс создается уже существующим процессом с помощью вызова f ork(). Ответвившиеся процессы считаются **дочерними,** а процессы, от которых они ответвились, - родительскими. Дочерний и родительский процессы продолжают выполняться параллельно. Если родитель продолжает порождать новые дочерние процессы, эти процессы становятся **сестринскими** по отношению к первому ребенку. Дети, в свою очередь, могут сами порождать дочерние процессы. Так образуется иерархическая связь между процессами, определяющая их родство.

После создания процесса он готовится стать **выполняемым процессом.** Это значит, что ядро настраивает все структуры и получает необходимую информацию от процессора для выполнения процесса. Когда процесс готов стать выполнимым, но еще не выбран для выполнения, он находится в состоянии **готовности.** После того как он становится выполняемым, он может:

- Быть «исключенным» («deselected») и переведенным в это состояние планировщиком.
- Быть прерванным и помещенным в состояние ожидания или **блокировки.**
- Стать зомби на пути к своей смерти. Смерть процесса наступает при вызове exit ().

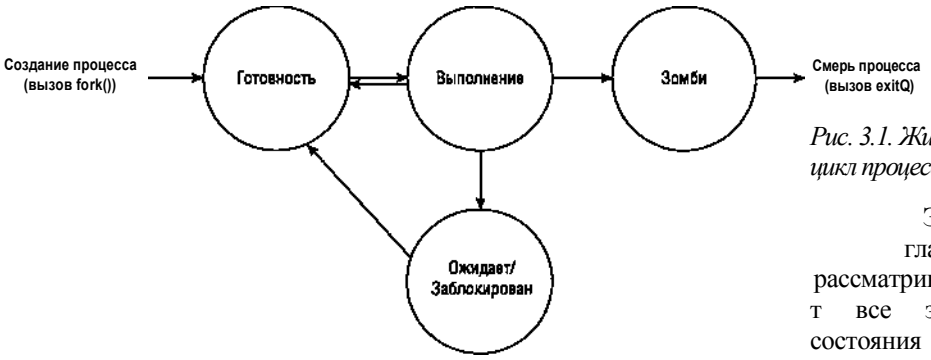


Рис. 3.1. Жизненный цикл процесса

Эта глава рассматривает все эти состояния и переход между ними. **Планировщик** управляет выбором и исключением процессов, выполняемых процессором. Гл. 7, «Планировщик и синхронизация ядра», описывает планировщик подробнее.

Программа содержит множество компонентов, которые располагаются в памяти и запрашиваются процессом, исполняющим программу. Сюда входят **текстовый** сегмент, который содержит инструкции, выполняемые процессором; **сегменты данных,** которые содержат переменные, которыми манипулирует процесс; **стек,** который хранит автоматические переменные и данные функций; и **кучу (heap),** которая содержит динамически выделенную память. При создании процесса дочерний процесс получает копию родительного пространства данных, кучу, стек и **дескриптор (описатель) процесса (process descriptor).** Следующий раздел посвящен более детальному описанию дескриптора процесса Linux.

Процесс можно объяснить несколькими способами. Наш подход заключается в том, чтобы начать с высокоуровневого рассмотрения выполнения процесса и проследить его до уровня ядра, попутно объясняя назначение вспомогательных структур, которые это обеспечивают.

Как программисты, мы знакомы с написанием, компиляцией и выполнением программ. Но как они связаны с процессами? На протяжении этой главы мы обсудим пример программы, которую мы проследим от ее создания до выполнения своих основных задач. В этом случае процесс оболочки Bash создает процесс, становящийся экземпляром нашей программы; в свою очередь, наша программа порождает экземпляр дочернего процесса.

До того как мы перейдем к обсуждению процессов, нам нужно ввести несколько соглашений. Обычно мы используем слово *процесс* и слово *задача* для описания одного и того же. Когда мы говорим о выполняемом процессе, мы говорим о процессе, который выполняется процессором в данный момент.

Пользовательский режим против режима ядра

Что мы имеем в виду, когда говорим, что программа выполняется в пользовательском режиме или в режиме ядра? Во время жизненного цикла процесса он выполняет как собственный код, так и код ядра. Код считается кодом ядра, когда делается системный вызов, возникает исключение или происходит прерывание (и мы выполняем обработчик прерывания). Любой код, исполняемый процессом и не являющийся системным вызовом, считается кодом пользовательского режима. Такой процесс запускается в пользовательском режиме, и на него налагаются некоторые процессорные ограничения. Если процесс находится внутри системного вызова, мы можем сказать, что он выполняется в режиме ядра. С аппаратной точки зрения код ядра для процессоров Intel выполняется в кольце 0, а на PowerPC он выполняется в *супервизорском режиме (supervisor mode).*

3.1 Представление нашей программы

В этом разделе представлена программа-пример под названием createjprocess. Этот пример С-программы иллюстрирует различные состояния, в которых может находиться процесс, системные вызовы (которые генерируют перемещение между этими состояниями) и манипуляцию объектами ядра, которые поддерживают выполнение процессов. Идея заключается в получении глубокого понимания того, как программа превращается в процесс и как операционная система обрабатывает эти процессы.

create_process.c

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <sys/stat.h>
4  #include <fcntl.h>
5
6  int main(int argc, char *argv[])
7  {
8    int fd;
9    int pid;
11
12   pid = forkO ;
13   if (pid == 0)
14   {
15      execle("/bin/ls",NULL);
16      exit(2);
17

}
18
19   if (waitpid(pid) < 0)
20       printf("wait error\n");
21
```

```
22   pid =  fork();
23   if (pid == 0){
24        fd=open("Chapter 03.txt", 0 RDONLY);
25        close(fd);
26    }
27
28         if(waitpid(pid)<0)
29       printf("wait error\n");
30
31
32    exit(0);
33   }
```

Эта программа определяет **контекст выполнения,** включающий информацию о ресурсах, необходимых для удовлетворения требований, определяемых программой. Например, в каждый момент процессор выполняет только одну инструкцию, извлеченную из памяти[1]. Тем не менее эта инструкция не будет иметь смысла, если она не окружена **контекстом,** из которого становится ясно, как инструкция соотносится с логикой программы. Процесс обладает контекстом, составленным из значений, хранимых в счетчиках программы, регистрах, памяти и файлах (или используемом аппаратном обеспечении).

Эта программа компилируется и собирается в исполнимый файл, содержащий всю информацию, необходимую для выполнения программы. Гл. 9, «Сборка ядра Linux», уточняет разделение адресного пространства программы и то, как эта информация связана с программой, когда мы обсуждаем образы процессов и бинарные форматы.

Процесс содержит несколько характеристик, которые описывают этот процесс и делают его уникальным среди остальных процессов. Характеристики, необходимые для управления процессом, хранятся в одном типе данных, называемом **процессорным описателем (process descriptor).** Перед тем как углубиться в управление процессами, нам нужно познакомиться с этим описателем процесса.

3.2 Описатель процесса

В ядре описатель процесса представлен структурой под названием task_struct, которая хранит атрибуты и информацию о процессе. Здесь можно найти всю информацию ядра, связанную с процессом. На протяжении жизненного цикла процесса на процесс влияют многие аспекты ядра, такие, как управление памятью и планировщик. Описатель процесса хранит информацию, связанную с этим взаимодействием, вместе со стандартными UNIX-атрибутами процесса. Ядро хранит все описатели процессов в циклическом двусвязном списке, называемом **task_list.** Также ядро хранит указатель на task_

[1] Повторный вызов сегмента теста, указанного ранее.

struct текущего выполняемого процесса в глобальной переменной **current.** (Мы будем вспоминать current на протяжении книги, когда будем говорить о текущем выполняемом процессе.)

Процесс может состоять из одного или нескольких потоков. Каждый поток имеет ассоциированную с ним task_struct, включающую уникальный ГО. Потоки одного процесса разделяют адресное пространство этого процесса.

Следующие категории описывают некоторые типы элементов описателя процесса, за которыми необходимо следить на протяжении его жизненного цикла:

- атрибуты процесса,
- связи процесса,
- пространство памяти процесса,
- управление файлами,
- управление сигналами,
- удостоверение процесса,
- ресурсные ограничения,
- поля, связанные с планировкой.

Теперь мы подробнее рассмотрим поля структуры task_struct. Этот раздел описывает, для чего они нужны и в каких реальных действиях эти поля участвуют. Многие из них используются для вышеупомянутых задач, остальные выходят за пределы рассмотрения этой книги. Структура task_struct определена в include/linux/ sched.h:

```
include/linux/sched.h
3 84        struct task  struct {
385         volatile long  state;
3 86  struct thread info *thread info;
387           atomic  t usage;
3 88         unsigned long flags;
3 89         unsigned long ptrace;
390
3 91            int lock depth;
392
393         int prio, static prio;
3 94      struct list head run list;
3 95        prio array  t  *array;
396
3 97       unsigned long sleep avg;
3 98       long interactive_credit;
```

```
399    unsigned long long timestamp;
400    int activated;
401
402     unsigned long policy;
403     cpumask t cpus allowed;
404     unsigned int time_slice, first_time_slice; 405
406     struct list Jiead tasks;
407     struct list head ptrace children;
408     struct list head ptrace list; 409
410    struct mm_struct *mm, *active_mm;

413     struct linux  binfmt *binfmt;
414     int exit_code,  exit_jsignal;
415     int pdeath_signal;

419     pid_t pid;
420     pid_t  tgid;

426     struct task__struct *real_parent;
427     struct  task struct *parent;
428     struct list head children;
429     struct list head sibling;
430     struct task_struct *group_leader;

433    struct pid   link pids[PIDTYPE MAX];
434
43 5    wait queue head t wait chldexit;
43 6    struct completion *vfork done;
437     int __user *set_child_tid;
438     int   user *clear child tid;
439
440     unsigned long rt_priority;
441     unsigned long it real value, it prorevalue, it  virt value;
442     unsigned long it real incr, it prof. incr, it virt incr;
443     struct timer list real  timer;
444     unsigned long utime, stime, cutime, cstime;
445     unsigned long nvcsw, nivcsw, cnvcsw, cnivcsw;
446     u64 start_time;

450     uid  t uid, euid, suid, f suid;
451     gid t gid,egid, sgid, fsgid;
452     struct group info *group info;
453     kernel cap t cap effective, cap inheritable, cap  permitted;
454    int keep_capabilities:1;
```

```
455     struct user_struct  *user;

457     struct rlimit rlim[RLIM_NLIMITS];
458     unsigned short used math;
459     char coram[16];

461     int link_count, total_link_count;

467     struct fs__struct *fs;

469     struct files_struct *files;

509     unsigned long ptrace_message;
510     siginfo_t *last__siginfo;

516     };
```

3.2.1 Поля, связанные с атрибутами процесса

К категории атрибутов процесса мы относим характеристики процесса, связанные с состоянием и идентификацией процесса. Рассматривая значения этих полей в произвольное время, хакер ядра может узнать текущее состояние процесса. Рис. 3.2 иллюстрирует поля task_struct, которые связаны с атрибутами процесса.

3.2.1.1 state

Поле state отслеживает состояние процесса, в котором процесс находится во время своего жизненного цикла. Может хранить значения TASK__RUNNING, TASK_INTERRUPTIBLE, TASK..UNINTERRUPTIBLE, TASK_ZOMBIE, TASK_STOPPED и TASK_DEAD (см. более детальное описание в разделе «Жизненный цикл процесса»).

3.2.1.2 pid

В Linux каждый процесс имеет уникальный идентификатор процесса (process identifier, **pid.** Его тип можно привести к целочисленному, а максимальное знаение по умолчанию равно 32768 (наибольшее значение для short int).

3.2.1.3 flags

Флаги определяют специальные атрибуты, принадлежащие процессу. Флаги процессов определены в файле include/linux/sched.h и включают флаги, перечисленные в таблице 3.1. Значения флагов позволят хакеру получить больше информации о работе процесса.

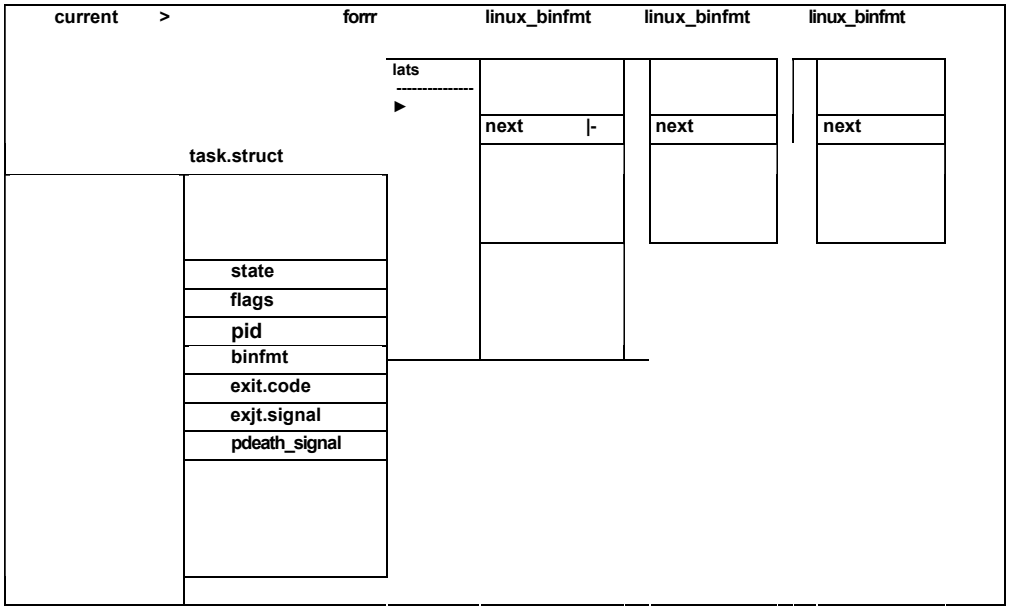


Рис. 3.2. Поля, связанные с атрибутами процесса

Таблица 3.1. Избранные значения флагов task_struct

Имя флага	Когда устанавливается
PF_STARTING	При создании процесса
PF_EXITING	Во время вызова do_exit ()
PF_DEAD	Во время вызова exit__notif y () при выходе из процесса. В этой точке процесс находится в состоянии TASK__ZOMBIE или TASK_DEAD
PF FORKNOEXEC	Родитель устанавливает этот флаг перед ветвлением

3.2.1.4 bitfmt

Linux поддерживает несколько исполнимых форматов. Исполнимый формат определяет, как структура кода вашей программы будет загружена в память. Рис. 3.2 иллюстрирует связь между task_struct и структурой linux_binfmt, которая содержит всю информацию, относящуюся к определенному бинарному формату (см. гл. 9).

3.2.1.5 exit_code и exit_signal

Поля exit_code и exit_signal хранят код выхода процесса и сигнал завершения (если он был использован). Таким образом выходное значение дочернего процесса передается его родителю.

3.2.1.6 pdeath.signal

pdeath__signal - это сигнал, посылаемый при смерти родителя.

3.2.1.7 comm

Чаще всего процесс создается с помощью вызова исполнимого файла из командной строки. Поле comm хранит имя исполнимого файла, вызванного из командной строки.

3.2.1.8 ptrace

ptrace устанавливается системным вызовом ptrace (), вызываемым при измерении производительности процесса. Обычно флаги ptrace () определяются в файле include /linux/ptrace.h.

3.2.2 Поля, связанные с планировщиком

Операции с процессами выполняются так, как будто они выполняются на собственном виртуальном процессоре. Тем не менее на самом деле он делит процессор с другими процессами. Чтобы поддерживать переключение между выполняемыми процессами, каждый процесс тесно взаимодействует с планировщиком (более подробно это описано в гл. 7).

Тем не менее для того чтобы разобраться с назначением некоторых полей, вам необходимо понять несколько базовых концепций планировщика. Когда для запуска готово более одного процесса, планировщик решает, какой из них запустить первым и на какое время. Планировщик достигает равномерной производительности и эффективности, выделяя каждому процессу **временной срез (timeslice)** и **приоритет (priority).** Временной срез определяет длительность времени, отводимую процессу до того, как будет выполнено переключение на другой процесс. Приоритет процесса - это значение, определяющее относительный порядок, в котором процессу будет позволено выполняться с учетом ожидающих процессов, - чем выше приоритет процесса, тем быстрее планировщик его запустит. Поля, показанные на рис. 3.3, отслеживают значения, необходимые планировщику.

3.2.2.1 prio

В гл. 7 мы увидим, что динамический приоритет процесса - это значение, зависящее от истории планировщика процессов и определяющее значение nice. (Более подробно значение nice описано в следующей вставке.) Оно обновляется во время sleep, когда процесс не выполняется и когда будет использован следующий временной срез. Это

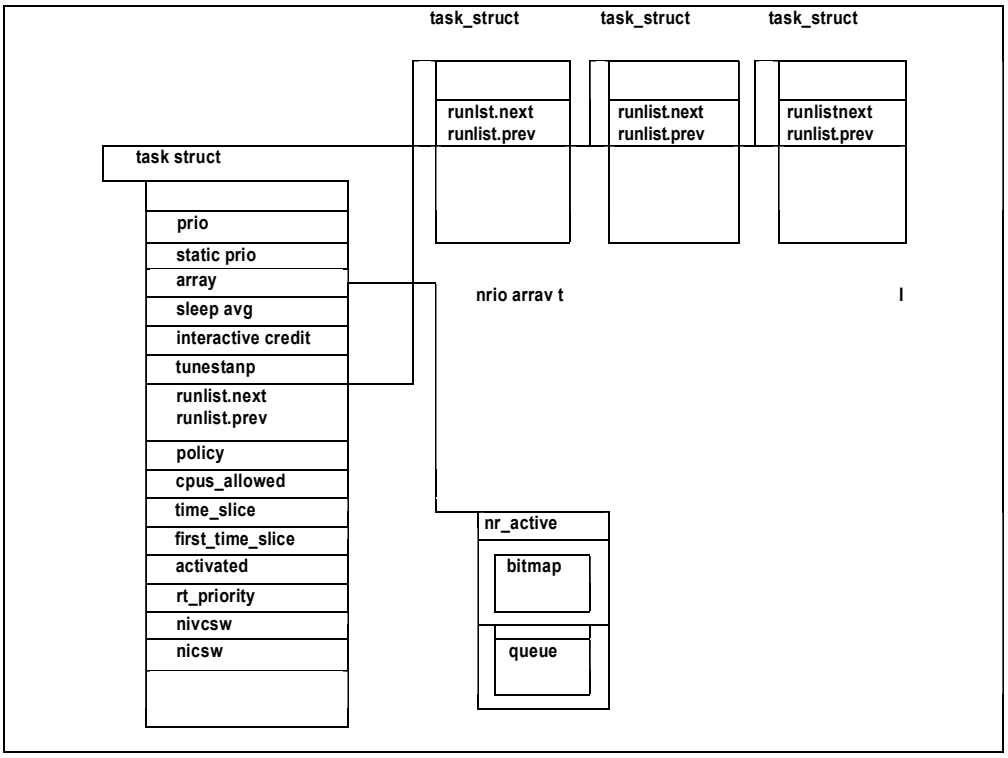


Рис. 3.3. Поля, связанные с планировщиком

значение prio связано со значением поля static_prio, описанного далее. Поле prio хранит +/- 5 к значению static_prio, зависящие от истории процесса; он получает бонус +5, если слишком долго спал, и штраф -5, если удерживал процессор в течение слишком многих временных срезов.

3.2.2.2 static_prio

s tatic_prio - это эквивалент значения nice. Значение static_prio по умолчанию равно MAX_PRIO-20. В нашем ядре значение по умолчанию для MAX_PRIO составляет 140.

3.2.2.3 runjist

Поле run_list указывает на runqueue; runqueue содержит список всех выполняемых процессов. (См. раздел «Базовые структуры» для получения более подробной информации о структуре runqueue.)

Nice

Системный вызов nice () позволяет пользователю изменять статический приоритет планировщика для процесса. Значение nice может варьироваться от -20 до 19. Далее функция nice () вызывает set_user_nice () для установки поля static_prio структуры task_struct. Значение static_prio рассчитывается из значения nice с помощью макроса PRIO_TO_NICE. Аналогично значение nice рассчитывается из значения static_prio с помощью вызова NICE_TO_PRIO.

```
---------------------------------------------- kernel/sched.c
tdefine NICE TO PRIO(nice)  (MAX RT PRIO + nice + 20)
tdefine PRIO_TO_NICE (prio)  (prio - MAX_RT_PRIO - 20)
```

3.2.2.4 array

Поле array указывает на массив приоритетов runqueue. (Раздел «Слежение за процессом: базовые конструкции планировщика» в этой главе описывает этот массив более подробно.)

3.2.2.5 sleep_avg

Поле sleep__avg используется *для* расчета эффективного приоритета задачи, равного среднему количеству тиков счетчика, потраченному задачей на сон.

3.2.2.6 timestamp

Поле timestamp используется для расчета sleep__avg, когда задача приостановлена или спит.

3.2.2.7 interactive_credit

Поле interactive__credit используется вместе со sleep_avg и активизирует поля для расчета sleep_avg.

3.2.2.8 policy

policy определяет тип процесса (например, разделяющий время или работающий в реальном времени). Тип процесса сильно зависит от приоритета планировщика. (Более подробно это поле описано в гл. 7.)

3.2.2.9 cpus_allowed

Поле cpus__allowed указывает, какой процессор обрабатывает задачу. Это один из способов указания конкретного процессора для данной задачи при работе на многопроцессорной системе.

3.2.2.10 time.slice

Поле time_slice определяет максимальный отрезок времени, разрешенный процессу для выполнения.

3.2.2.11 firstjime_slice

Поле first_time_slice периодически устанавливается в 0 и отслеживается планировщиком.

3.2.2.12 activated

Поле activated отслеживает инкрементирование и декрементирование среднего времени сна. Если выполняется непрерываемая задача, это поле устанавливается в -1.

3.2.2.13 rtpriority

rt_priority - это статическое значение, которое может быть обновлено только вызовом schedule (). Это значение необходимо для поддержки задач реального времени.

3.2.2.14 nivcsw и nvcsw

Существуют различные типы контекстов. Ядро отслеживает их с целью профилирования. Глобальный счетчик переключений устанавливается одним из четырех контекстных счетчиков переключений в зависимости от типа перехода выполняемого при переключении контекста (переключение контекстов описано в гл. 7). Ниже представлены счетчики для базовых переключений контекста.

- **Поле nivcsw (количество непринудительных переключений контекста) хранит счетчик приоритета ядра, примененного к задаче. Он увеличивается только при возвращении задачи на основе приоритета ядра, когда счетчик переключений устанавливается с помощью nivcsw.**
- **Поле nvcsw (количество принудительных переключений контекста) хранит счетчик переключений контекста, основанных на приоритете ядра. Счетчик переключений устанавливается в nvcsw, если предыдущее состояние не было активным приоритетом.**

3.2.3 Поля, связанные с отношениями между процессами

Следующие поля структуры task_struct связаны с отношениями между процессами. Каждая задача или процесс р имеет родителя, который его создал. Процесс р тоже может создавать процессы и поэтому тоже может иметь детей. Так как родитель р может создать больше одного процесса, вполне возможно, что процесс р будет иметь сестринские процессы. Рис. 3.4 иллюстрирует, как task_struct хранит эти связи процессов.

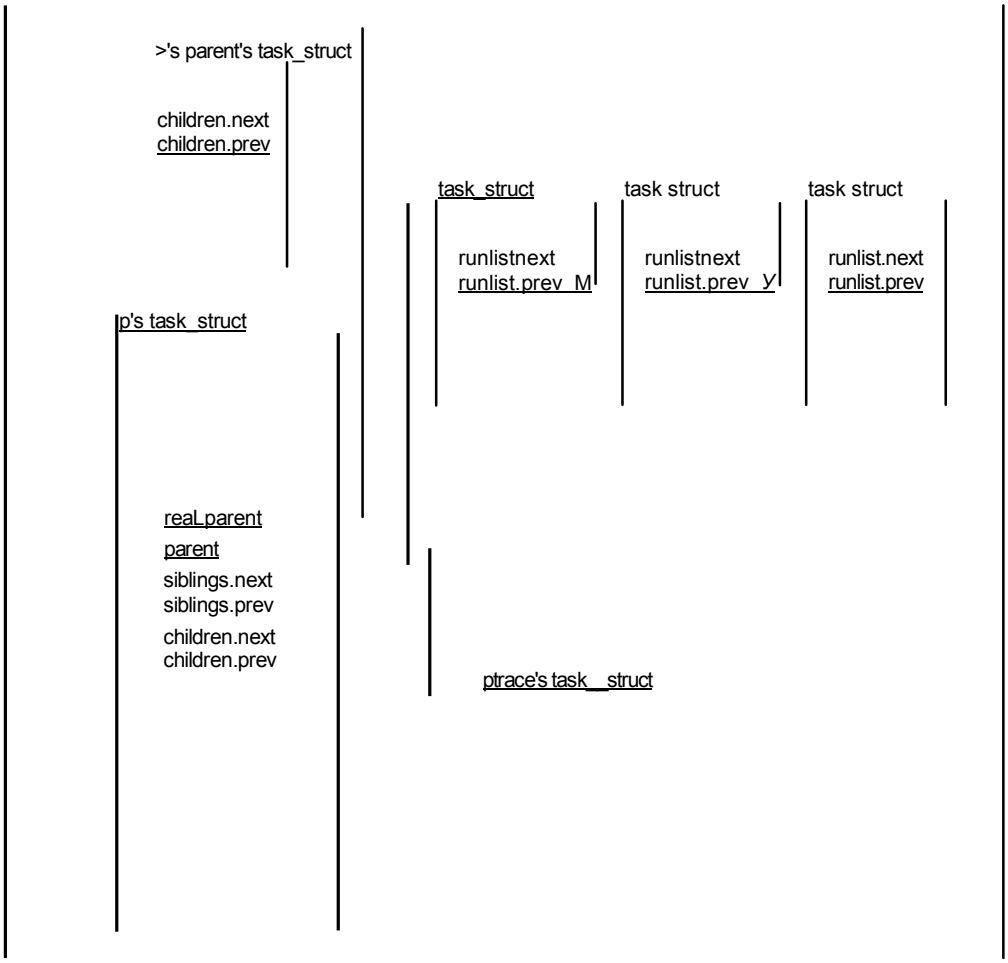


Рис. 3.4. Поля, связанные с отношениями между процессами

3.2.3.1 reaLparent

real__parent указывает на описатель родителя текущего процесса. Он указывает на дескриптор процесса init (), если оригинальный родитель этого процесса был уничтожен. В предыдущих ядрах это называлось p_opptr.

3.2.3.2 parent

parent - это указатель на описатель родительского процесса. На рис. 3.4 видно, что он указывает на ptrace_task_struct. Когда для процесса запускается ptrace, родительское поле task__struct указывает на процесс ptrace.

3.2.3.3 children

children - это структура, указывающая на список детей текущего процесса.

3.2.3.4 sibling

sibling - это структура, указывающая на список сестринских процессов текущего процесса.

3.2.3.5 groupjeader

Процесс может быть членом группы процессов, и каждая группа процессов имеет процесс, являющийся ее лидером. Если наш процесс является членом группы, group_ leader указывает на описатель лидера этой группы. Лидер группы обычно получает tty, с которого был создан данный процесс и который называется **управляющим терминалом (controlling terminal).**

3.2.4 Поля, **связанные с удостоверением процесса**

В многопользовательских системах необходимо разделять процессы, создаваемые различными пользователями. Это необходимо делать в целях безопасности и для защиты пользовательских данных. Поэтому у каждого процесса имеется удостоверение, позволяющее системе определить, к чему у него есть доступ, а к чему нет. Рис. 3.5 иллюстрирует поля task__struct, связанные с удостоверением процесса.

3.2.4.1 uid и gid

Поле uid содержит число ГО пользователя, который создал процесс. Это поле используется для защиты и обеспечения безопасности. Аналогично поле gid содержит групповой идентификатор группы, к которой принадлежит процесс; uid и gid со значением 0 относятся к пользователю root и его группе.

3.2.4.2 euid и egid

Эффективный пользовательский идентификатор обычно хранит то же самое значение, что и поле пользовательского идентификатора. Он изменяется, если у выполняемой программы установлен бит UID (SUID). В этом случае эффективный идентификатор пользователя представляет собой идентификатор владельца файла программы. Обычно это применяется для того, чтобы позволить пользователям запускать отдельные программы

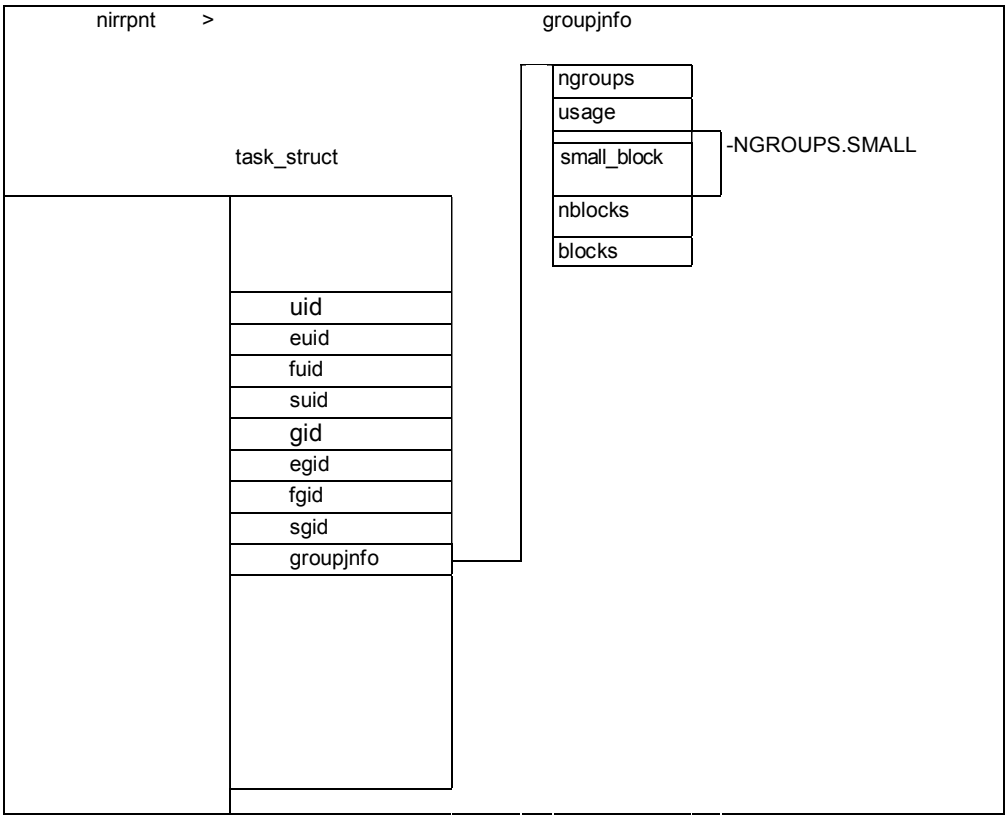


Рис. 3.5. Поля, связанные с удостоверением процесса

с теми же правами, что и у других пользователей (например, root). Эффективный идентификатор группы работает точно так же и хранит значение, отличное от поля gid, только в том случае, если установлен бит группового идентификатора (SGID).

3.2.4.3 suid и sqid

suid (сохраненный пользовательский идентификатор) и sqid (сохраненный групповой идентификатор) употребляются в системных вызовах setuid ().

3.2.4.3 fsuidn fsqid

Значения fsuidn fsqid используются для процессов файловой системы. Обычно они содержат те же значения, что uid и gid, за исключением случаев, когда выполняется системный вызов setuid ().

3.2.4.5 groupjnfo

Пользователь в Linux может быть членом одной или нескольких групп. Эти группы могут обладать различными правами на доступ к системе и данным. Поэтому процесс должен наследовать эти удостоверения. Поле group_inf o указывает на структуру типа group__inf o, содержащую информацию, связанную с группами, членом которых является процесс.

Структура group_inf o позволяет процессу ассоциироваться с несколькими группами, количество которых ограничено только размером памяти. На рис. 3.5 вы можете видеть, что поле group_info с именем small_block является массивом NGROUP_ SMALL (в нашем случае из 32 элементов) элементов gid_t. Если задача принадлежит более чем 32 группам, ядро может выделить дополнительные блоки или страницы, которые будут хранить необходимое количество gid__t за пределами NGROUP_SMALL. Поле nblocks хранит количество выделенных блоков, а ngroups хранит значения элементов в массиве small_block, хранящих значение gid_t.

3.2.5 Поля, связанные с доступными возможностями

Традиционно UNIX-системы предлагают процессно-ориентированную защиту на действия и доступ к некоторым объектам, определяя каждый процесс как привилегированный (суперпользовательский или UID = 0) или непривилегированный (для любого другого процесса). Возможности были введены в Linux для отделения действий, которые ранее были доступны только в суперпользовательском режиме. Таким образом, возможности - это индивидуальные «привилегии», которые могут быть выданы процессу независимо от других процессов и от его UID. Таким образом, отдельные процессы могут получать возможность выполнять отдельные администраторские задачи без необходимости получать полные привилегии или являться собственностью суперпользователя. Такие возможности представляют собой отдельные администраторские операции. Рис. 3.6 демонстрирует поля, связанные с возможностями процесса.

3.2.5.1 cap_effective, capjnheritable, cap_permitted и keep_capabilities Структура используется для поддержки модели возможностей, определенной в include /linux/security.h как unsigned 32 битовое значение. Каждые 32 бита маски соответствуют набору возможностей; на каждую возможность отводится 1 бит:

- **cap__ef f active.** Возможность, уже используемая процессом.
- **cap_inheritable.** Возможность, передаваемая через вызов execve.
- **cap permitted.** Возможность, которая может быть как эффективной, так и наследуемой.

  Понять разницу между этими тремя типами можно, если представить их подобными упрощенному генному пулу, доступному одному из родителей. Генетические спо-

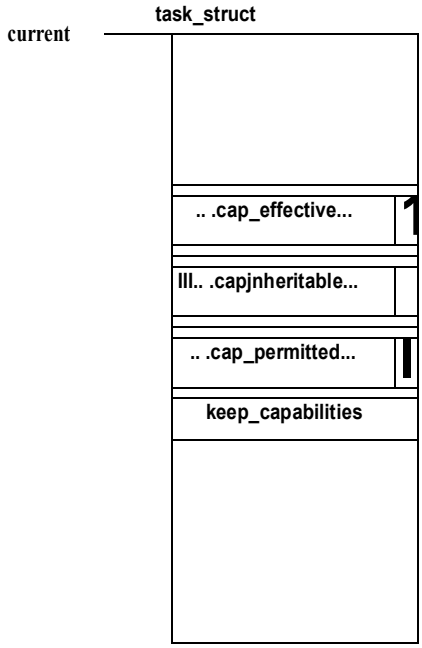


Рис. 3.б. Поля, связанные с доступными возможностями

собности, доступные одному из родителей, мы можем просто перечислить (эффективные возможности) и/или передать (наследование). Разрешенные возможности более похожи на потенциальные возможности, а эффективные возможности - на реальные.

Поэтому cap__ef f ective и cap_inheritable всегда являются подмножеством cap_permitted.

- **keep_capabilities. Следит за тем, как процесс теряет или получает свои возможности при вызове setuid ().**

В табл. 3.2 перечислены некоторые поддерживаемые возможности, определенные в `lunux/include/capability.h`.

Таблица 3.2. Избранные возможности

Возможность	Описание
CAP_CHOWN	Игнорирует ограничения, налагаемые chown ()
CAP_FOWNER	Игнорирует ограничения на доступ к файлам
CAP_PSETID	Игнорирует setuid и setgid ограничения для файлов
CAPJCILL	Игнорирует ruid и euid при посылке сигналов
CAP_SETGm	Игнорирует связанные с группой проверки
CAP_SETUID	Игнорирует связанные с uid проверки
CAP_SETCAP	Наделяет процесс полным набором возможностей

Ядро проверяет, какие из возможностей установлены при вызове capable () с передачей значений возможностей в качестве параметров. Обычно функция проверяет, какие из битов возможностей установлены в cap_ef f ective; если они установлены, функция устанавливает current-> flags в PF_SUPERPRIV, что означает получение возможностей. Функция возвращает 1, если возможность получена, и 0, если возможность не может быть получена.

С манипуляцией возможностями связано три системных вызова: capgetO, capset () и prctl (). Первые два позволяют процессу получать и устанавливать возможности, а системный вызов prctl () позволяет манипулировать current->keep_capabilities.

3.2.6 Поля, связанные с ограничениями процесса

Задача использует множество ресурсов, предоставляемых аппаратным обеспечением и планировщиком. Перечисленные ниже поля служат для отслеживания и использования ограничений, налагаемых на процесс.

3.2.6.1 rlim

Поле rlim содержит массив, позволяющий контролировать ресурсы и поддерживать значения ограничений на ресурсы. Рис. 3.7 иллюстрирует поля rlim структуры task_ struct.

Linux распознает необходимость ограничивать количество определенных ресурсов, которыми разрешено пользоваться процессу. В силу того что тип и количество используемых ресурсов может отличаться от процесса к процессу, необходимо хранить информацию о каждом процессе. Где же эту информацию разместить, как не в описателе процесса.

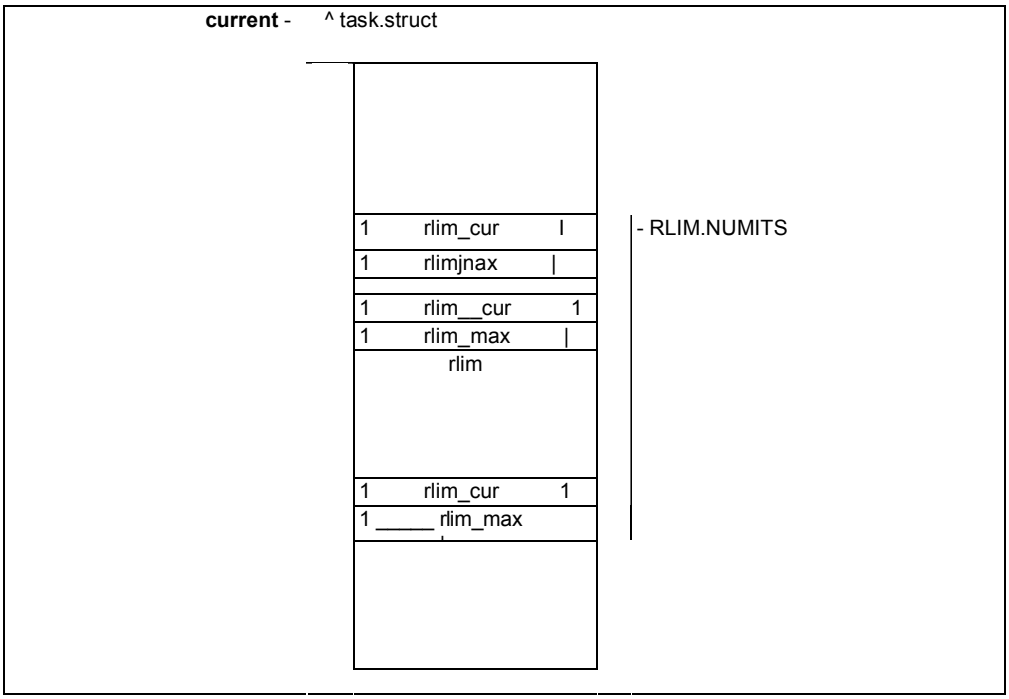


Рис. 3.7. Ресурсные ограничения task_struct

Описатель rlimit (include/linux/resource.h) имеет поля rlim_cur и rlim_max, представляющие собой текущее и максимальное ограничения, налагаемые на ресурс. Тип ограничения варьируется от ресурса к ресурсу в зависимости от его типа.

```
include/linux/resource.h
struct rlimit {
   unsigned long  rlim cur;
   unsigned long  rlim_max;
};
```

В табл. 3.3 перечислены ресурсы, для которых в include/asm/resource.h определены ограничения. При этом x86 и PowerPC имеют одни и те же ограничения на ресурсы и их значения по умолчанию.

Когда значение установлено в RLIMIT_INGINITY, ресурс для данного процесса не ограничен.

Таблица 3.3. Значения ограничений ресурсов

Имя ограничения ресурсов	Описание	Значение по умолчанию rlim_cur	Значение по умолчанию rlimjnax
RLIMIT_CPU	Количество процессорного времени в секундах, выдаваемое процессу на выполнение	**RLIMIT_INGINITY**	**RLIMIT_INGINITY**
RLIMIT_FSIZE	Размер файла в блоках по 1 кб	**RLIMIT_INGINITY**	RLIMIT_INGINITY
RLIMIT_DATA	Размер кучи в байтах	**RLIMIT_INGINITY**	RLIMIT_INGINITY
RLIMIT_STACK	Размер стека в байтах	**_STK_LIM**	RLIMIT_INGINITY
RLIMIT_CORE	Размер файла сброса ядра	**0**	RLIMIT_INGINITY
RLIMIT_RSS	Максимальный резидентный размер (реальной памяти)	RLIMIT_INGINITY INR OPEN	RLIMIT_INGINITY INR_OPEN
RLIMIT_NPROC	Количество процессов, которые принадлежат данному процессу		
RLIMIT_NOFILE	Количество открытых файлов, которое этот процесс может иметь в каждый момент	RLIMIT_INGINITY	RLIMIT_INGINITY
RLIMIT_MEMLOCK	Физическая память, которая может быть заблокирована (не свопирована)	RLIMIT_INGINITY	RLIMIT_INGINITY
RLIMIT_AS	Размер адресного пространства процесса в байтах		
RLIMIT_LOCKS	Количество блокировок файлов		

Текущее ограничение (rlim_cur) - это мягкое ограничение, которое может быть изменено с помощью вызова setrlilimO. Максимальное ограничение определяется rlim_max и не может быть обойдено непривилегированным процессом. Системный вызов getrlimit () возвращает значение ограничения на ресурс. И setrlimit () и де-trlimit () получают в качестве параметра имя ресурса и указатель на структуру типа rlimit.

3.2.7 Поля, связанные с файловой системой и адресным пространством

Процессы могут быть тесно связаны с файлами на протяжении своего жизненного цикла, выполняя задачи наподобие открытия, закрытия, чтения и записи; task_struct имеет два поля, связанные с данными файлов и файловой системы: f s и files (см. гл. 6, «Файловые системы»). С адресным пространством связаны переменные active_mm и mm (см. описание mm_struct в гл. 4, «Управление памятью»). На рис. 3.8 показаны поля task_struct, связанные с файловой системой и адресным пространством.

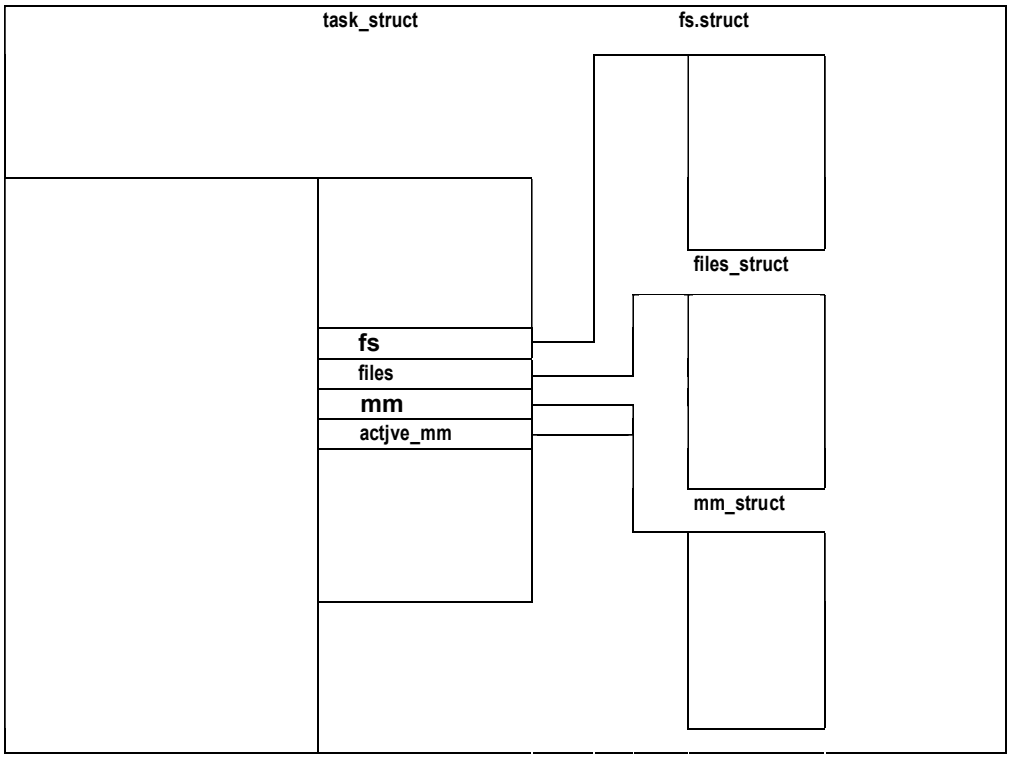


Рис. 3.8. Поля, связанные с файловой системой и адресным пространством

3.2.7.1 fs

Поле f s содержит указатель на информацию о файловой системе.

3.2.7.2 files

Поле files хранит указатель на таблицу описателей файлов задачи. Этот файловый описатель хранит указатель на файлы (точнее говоря, на их описатели), открытые задачей.

3.2.7.3 mm

iran указывает на адресное пространство и связанную с управлением памятью информацию.

3.2.7.4 active_mm

active_mm указывает на адресное пространство, которое чаще всего используется. Оба поля, mm и active_mm, вначале указывают на одну и ту же mm_struct.

Оценка описателя процесса подводит нас к идее типа данных, с которым процесс связан на протяжении всего своего жизненного цикла. Теперь мы можем рассмотреть, что происходит на протяжении жизненного цикла процесса. Следующий раздел объясняет различные этапы и состояния процесса и построчно комментирует программу-пример для того, чтобы объяснить, что происходит в ядре.

3.3 Создание процессов: системные вызовы fork(), vfork() и cloneQ

После того как код примера откомпилирован в файл (в нашем случае исполнимый ELF[1]), мы можем вызвать его из командной строки. Посмотрим, что происходит после того, как мы нажимаем клавишу Return. Мы уже говорили, что любой процесс запускается другим процессом. Операционная система предоставляет функциональность, необходимую для этого в лице системных вызовов fork (), vf ork () и clone ().

Библиотека С предоставляет три функции, запускающие эти системные вызовы. Прототипы этих функций объявлены в <unistd. h>. Рис. 3.9 показывает, как процесс, вызывающий fork (), выполняет системный вызов sys__fork (). Этот рисунок показывает, как ядро производит создание процесса. Аналогично vf ork () вызывает sys_f ork () и clone () вызывает sys_clone ().

В конце концов все эти три системных вызова вызывают do_f ork () - функцию ядра, выполняющую большое количество действий, необходимых для создания процесса. Вас может удивить, почему для создания процесса существует три функции. Каждая функция создает процесс немного иначе, и существуют объективные причины, почему в разных случаях следует использовать разные функции.

Когда мы нажимаем кнопку Return в строке shell, оболочка создает новый процесс, исполняющий нашу программу с помощью вызова fork (). Поэтому, если мы вводим команду Is в оболчке и нажимаем Return, псевдокод оболочки в этот момент выглядит примерно следующим образом:

```
if ( (pid = fork()) == 0 )
  execve(*foo");
else
  waitpid(pid);
```

[1] **ELF - формат исполнимого файла, поддерживаемый Linux. В гл. 9 описана структура ELF-формата.**

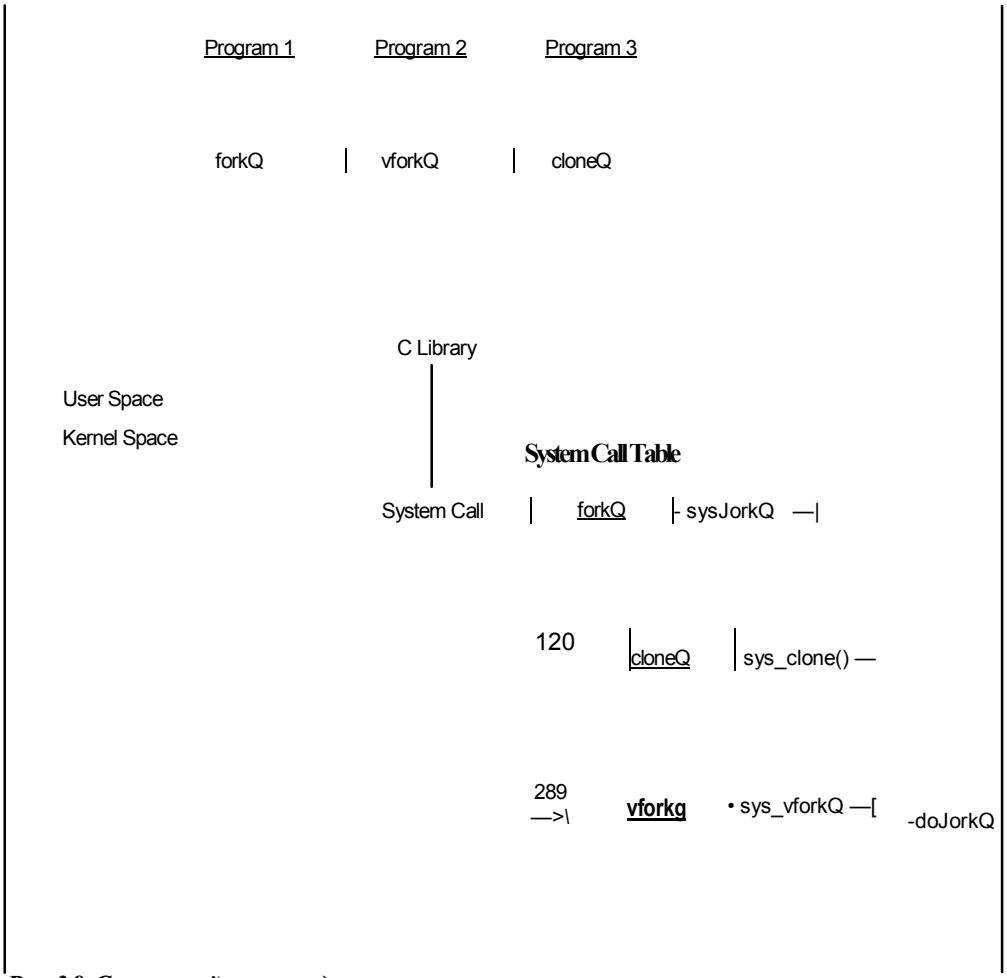


Рис. 3.9. Системный вызов создания процесса

Теперь мы можем рассмотреть эти функции и проследить их выполнение до уровня системного вызова. Несмотря на то что наша программа вызывает fork (), она может также легко вызывать vf ork () или clone (), которые мы тоже рассмотрим в этом разделе. Первой функцией, которую мы рассмотрим, будет fork (). Мы проследим ее через вызовы fork (), sys__f ork () и do__f ork (). Затем мы рассмотрим vf ork () и, наконец, clone (), тоже до вызова do_f ork ().

3.3.1 Функция fork()

Функция fork () возвращает два значения: одно родительскому и второе дочернему процессу. Если она возвращает значение дочернему процессу, fork () возвращает 0. Если она возвращает значение родительскому процессу, fork () возвращает PIII дочернего процесса. При вызове функции fork () функция помещает необходимую информацию в соответствующие регистры, включая индекс в таблице системных вызовов, где находится указатель на системный вызов. Процессор, на котором мы работаем, сам определяет регистры, в которых хранится информация.

На этом этапе, если вы хотите продолжить последовательную передачу событий, посмотрите раздел «Прерывания» в этой главе, чтобы увидеть, как вызывается sys_f ork (). Тем не менее понимать, как создается процесс, вовсе не обязательно.

Давайте теперь посмотрим на функцию sys_f ork (). Эта функция работает немного иначе, чем вызов функции do_f ork (). Обратите внимание, что функция sys_f ork () архитектурно зависима, потому что передаваемые в функцию параметры передаются через системные регистры.

```
arch/i386/kernel/process.c
asmlinkage int sys_fork(struct pt_regs regs)
{
   return do__fork(SIGCHLD, regs.esp, &regs, 0, NULL, NULL) ;
}
```

```
arch/ppc/kernel/process.c
int  sys_fork(int pi,  int  p2,  int p3,  int p4,  int p5,  int p6,
                    struct pt_regs  *regs) {
            CHECK_FULL_REGS(regs);
            return do_fork(SIGCHLD,  regs->grp[1],  regs,  0,  NULL,  NULL); }
```

Две архитектуры получают разные параметры для системных вызовов. Структура pt_regs хранит информацию, подобную указателю стека. По соглашению на PPC указатель на стек содержится в gpr [1], а на x86 он содержится в %esp[1].

3.3.2 Функция vfork()

Функция vf ork () похожа на функцию fork () за исключением того, что родительский процесс блокируется до тех пор, пока дочерний не вызовет exit () или exec ().

[1] Помните, что этот код выполнен в формате «AT&T», в котором регистры имеют префикс %.

```
sys_vforк()
arch/i386/kernel/process.c
asmlinkage int sys_vfork(struct pt_regs regs)
{
return do_fork(CLONE_VFORK | CLONE_VM | SIGCHLD, regs.ep, &regs, 0,
NOLL, NULL); }
```

```
arch/ppc/kernel/process.c
int sys_vfork(int pi, int p2, int p3, int p4, int p5, int p6,
                    struct pt_regs *regs) {
          CHECK_FULL_REGS(regs);
return do_fork (CLONE__VFORK | CLONE_VM | SIGCHLD, regs->gpr[1], regs, 0, NULL,
NULL); }
```

Единственная разница между вызовами do_f ork () в sys_vf ork () и sys_ fork () заключается во флагах, передаваемых do_fork (). Наличие этих флагов проверяется позднее для определения того, будет ли выполнено описанное выше поведение (блокирование родителя).

3.3.3 Функция clone()

Библиотечная функция clone () в отличие от f ork() и vf ork() получает указатель на функцию в качестве аргумента[1]. Дочерний процесс, который создается с помощью do_f ork (), вызывает эту функцию сразу же после своего создания.

```
sys_clone()
arch/i386/kernel/process. c
asmlinkage int sys__clone (struct pt_regs regs) {
unsigned long clone flags;
        unsigned long newsp;
        int   user *parent tidptr, *child tidptr;
        clone_flags = regs.ebx;
        newsp = regs.ecx;
        parent_tidptr = (int _ user *)regs.edx;
        child_tidptr = (int _ user *)regs.edi;
```

[1] **Библиотечная функция clone О имеет следующий прототип: int clone (int (*fn) (void *),void *child_stack, int flags, void *arg), где первый параметр - та функция, которая будет запущена сразу же после создания процесса.** ***Примеч. науч. ред.***

```
        if (Inewsp)
        newsp = regs.esp; return do_fork(clone_flags &
        ~CLONE IDLETASK, newsp, &regs, 0, parent  tidptr,
        child_tidptr) ;
}

arch/ppc/kernel/process, c
int sys_clone(unsigned long clone_flags, unsigned long usp,
              int   user *parent tidp, void   user *child  thread ptr,
              int   user *child tidp,   int pб,
              struct pt_regs  *regs) {
        CHECK_FULL_REGS(regs); if
        (usp  ==  0)
usp = regs->gpr[1];  /* указатель  на стек дочернего процесса  */ return do_fork(clone_flags
& ~CL0NE_IDLETASK,  usp,   regs,  0, parent_tidp,  child_tidp); }
```

Как показано в табл. 3.4, единственная разница между fork (), vf ork () и clone () заключается во флагах, установленных в соответствующем вызове do_f ork ().

Таблица 3.4. Флаги, передаваемые в doJorkQ, vforkQ и cloneQ

	forkQ	vfork()	clone()
SIGCHLD	X	X	
CLONE_VFORK		X	
CLONE_VM		X	

И наконец, мы переходим к do_f ork (), которая выполняет настоящее создание процесса. Вспомним, что до этого мы только выполнили из родителя вызов fork (), породивший системный вызов sys_f ork O, и у нас еще нет нового процесса. Наша программа f oo до сих пор является исполнимым файлом на диске. В памяти она еще не запущена.

3.3.4 Функция do_fork()

Мы проследим выполнение ядром функции do_f ork () построчно и прокомментируем детали создания нового процесса.

```
kernel/fork.c
1167    long do_fork(unsigned long clone_flags,
1168            unsigned long stack_start,
```

```
1169        struct pt__regs *regs,
1170        unsigned long stack size,
1171        int    user *parent tidptr,
1172        int  _ user *child_tidptr)
1173    {
1174      struct task struct *p;
1175      int trace = 0;
1176      long pid;
1177
1178      if (unlikely(current->ptrace)) {
1179         trace = fork traceflag (clone flags);
1180         if (trace)
1181           clone_flags |= CLONE_PTRACE;
1182      }
1183
1184      p = copy_process (clone__f lags,  stack_start,  regs,
                           stack_size,  parent_tidptr,  child_tidptr) ;
```

Строки 1178-1183

Код начинается с проверки того, хочет ли родительский процесс сделать новый процесс трассируемым (ptraced). Трассировка имеет приоритет среди функций, имеющих дело с процессом. Эта книга объясняет назначение ptrace только на самом высоком уровне. Для определения, какой дочерний процесс должен быть отслежен, fork_traceflag() должна подтвердить значение clone__flags. Если в clone_f lags установлен флаг CLONE_VFORK и SIGCHLD не перехвачен родителем или если текущий процесс обладает также установленным флагом PT__TRACE_FORK, дочерний процесс отслеживается до тех пор, пока не будут выставлены флаги CLONE_UNTRACED или CLONE_IDLETASK.

Строка 1184

В этой строке создается новый процесс и из регистров извлекаются необходимые значения. Функция copy_process () выполняет все необходимые действия для создания пространства процесса и заполнения полей его описателя. Тем не менее запуск нового процесса происходит позже. Подробности copy__process () будут более уместны при рассмотрении работы планировщика. (См. раздел «Слежение за процессами: базовые конструкции планировщика», где более подробно описано происходящее здесь.)

`kernel/fork.c`

```
1189    pid = IS ERR(p) ? PTR  ERR(p) : p->pid;
1190
```

```
1191   if (!IS_ERR(p)) {
1192      struct completion vfork;
1193
1194      if (clone flags & CLONE VFORK) {
1195        p->vfork done = &vfork;
1196        init_completion(&vfork);
1197

}
1198
1199     if ( (p->ptrace & PT PTRACED) || (clone flags & CLONE STOPPED)) {

1203          sigaddset(&p->pending.signal, SIGSTOP);
1204          set tsk thread flag(p, TIF SIGPENDING);
1205        }
```

Строка 1189

Здесь выполняется проверка ошибок указателей. Если мы обнаруживаем ошибку в указателях, мы возвращаем ошибку указателя без выполнения дальнейших действий.

Строки 1194-1197

Здесь выполняется проверка того, что do_f ork () была вызвана из vfork (). Если это так, выполняются специфичные для vfork () действия.

Строки 1199-1205

Если родитель отслеживается или клонирование установлено в режим CLONE_ STOPPED, дочерний процесс получает сигнал SIGSTOP до начала выполнения и поэтому запускается сразу в остановленном состоянии.

```
kernel/fork.c
1207   if (!(clone_flags & CLONE_STOPPED)) {

1222         wake__up_forked_process (p) ;
1223   } else {
1224      int cpu = get_cpu();
1225
1226      p->state = TASK_STOPPED;
1227      if (!(clone flags & CLONE STOPPED))
1228        wake up forked process(p); /* делается в последнюю очередь */
1229      ++total forks;
1230
1231      if (unlikely (trace)) {
1232        current~>ptrace_message = pid;
1233        ptrace notify ((trace « 8) | SIGTRAP);
1234      }
```

```
1235
1236      if (clone flags & CLONE VFORK) {
1237        wait for completion(&vfork);
1238        if (unlikely (current->ptrace & PT TRACE  VFORK DONE) )
1239          ptrace notify ( (PTRACE EVENT VFORK DONE « 8) |  SIGTRAP) ;
1240      } else

1248        set__need_resched () ;
1249    }
1250    return pid;
1251}
```

Строки 1226-1299

В этом блоке мы устанавливаем состояние задачи в TASK_STOPPED. Если флаг CLONE_STOPPED в clone_f lags не установлен, мы будим дочерний процесс. В противном случае мы оставляем его ожидать сигнала пробуждения.

Строки 1231-1234

Если для родителя установлен ptracing, мы посылаем уведомление.

Строки 1236-1239

Если производится вызов vf ork (), здесь блокируется родитель и посылается уведомление о начале слежения. Это реализуется с помощью помещения родителя в очередь ожидания, где он остается в состоянии TASK_UNINTERRUPTIBLE до того момента, как дочерний процесс вызовет exit () или execve ().

Строка 1248

Для текущей задачи (родителя) устанавливается need_resched. Это позволяет дочернему процессу запуститься первым.

3.4 Жизненный цикл процесса

Теперь, когда мы рассмотрели, каким образом процесс создается, нам нужно увидеть, что происходит в течение его жизни. За это время процесс может побывать в различных состояниях. Переход между этими состояниями зависит от выполняемых процессом действий и от характера устанавливаемых сигналов. Наша программа-пример может находиться в состояниях TASK_INTERRUPTIBLE и TASK__RUNNING (текущее состояние).

Первое состояние процесса - TASK__INTERRUPTIBLE. Это происходит при создании процесса в функции copy_process (), которая вызывается do_f ork (). Второе состояние процесса - TASK_RUNNING устанавливается перед выходом из do_f ork (). Эти два состояния гарантированно присутствуют в жизни любого процесса. Следом за этими состояниями процесс может оказаться еще в двух состояниях. Последнее состоя-

ние, в которое устанавливается процесс, - это TASK_ZOMBIE, во время вызова do_exit (). Давайте рассмотрим различные состояния процессов и способы перехода между этими состояниями. Мы рассмотрим, как наш процесс переходит из одного состояния в другое.

3.4.1 Состояния процесса

Когда процесс выполняется, это значит, что его контекст загружен в регистры процессора и в память, а определяемая контекстом программа выполняется. В каждый момент времени процесс может не выполнятся по самым различных причинам. Процесс может не иметь возможности продолжать работу по причине ожидания ввода, который не происходит, или планировщик может решить, что он выполнялся в течение максимально разрешенного времени, и это может стопорить другой процесс. Процесс считается готовым, когда он не выполняется, но может быть выполнен (после перепланировки) или блокирован, когда ожидает ввода.

Рис. 3.10 показывает абстрактные состояния процесса и перечисляет возможные состояния задачи в Linux, соответствующие каждому состоянию. Табл. 3.5 раскрывает четыре перехода и показывает, как они осуществляются. Табл. 3.6 связывает абстрактные состояния со значениями, используемыми в ядре Linux для обозначения этих состояний.

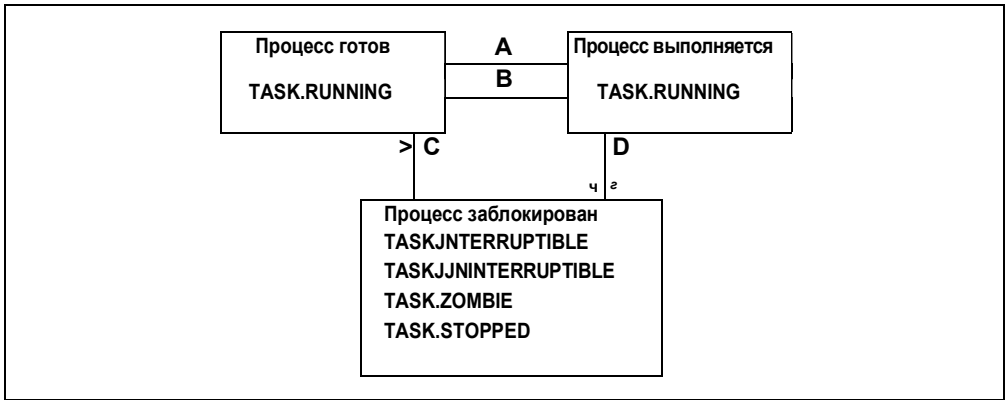


Рис. 3.10. Переходы между состояниями процесса

Таблица 3.5. Краткий перечень переходов

Переход	Агент перехода
Готов - выполняется (А)	Выбран планировщиком
Выполняется - готов (В)	Временной отрезок кончился (неактивен). Процесс приостановлен (активен)

Таблица 3.5. Краткий перечень переходов (Окончание)

Заблокирован - готов (C)	Поступают сигналы. Ресурс становится доступным
Выполняется - заблокирован (D)	Процесс спит или чего-то ожидает

Таблица 3.6. Связь флагов Linux с абстрактными состояниями процесса

Абстрактное состояние	**Состояние задачи Linux**
Готов	TASK_RUNNING
Выполняется	TASK_RUNNING

Заблокирован	TASK_INTERRUPTIBLE
	`TASK UNINTERRUPTIBLE`
	`TASK ZOMBIE`
	`TASK_STOPPED`

ПРИМЕЧАНИЕ. Состояние set_current_process () может быть установлено, если имеется прямой доступ к настройке структуры задачи:
current->state = TASK_INTERRUPTIBLE.
Вызов set__current_process (TASK_INTERRUPTIBLE) даст тот же эффект.

3.4.2 Переход между состояниями процесса

Теперь мы рассмотрим типы событий, которые заставляют процесс переходить из одного состояния в другое. Абстрактные процессы перехода (см. табл. 3.5) включают переход из состояния ожидания в состояние выполнения, переход из состояния выполнения в состояние готовности, переход из заблокированного состояния в состояние готовности и переход из состояния выполнения в заблокированное состояние. Каждый переход может переводить в более чем один переход между различными состояниями задач Linux. Например, переход из блокированного состояния в выполняемое может происходить из одного из четырех состояний задач Linux: TASK_INTERRUPTIBLE, TASK_ZOMBIE, TASK_UNINTERRUPTIBLE или TASK_STOPPED в состояние TASK_RUNNING. Рис. 3.11 и табл. 3.7 описывают эти переходы.

Теперь мы опишем различные переходы между состояниями применительно к переходам между состояниями задачи Linux, подпадающими под основные категории переходов процесса.

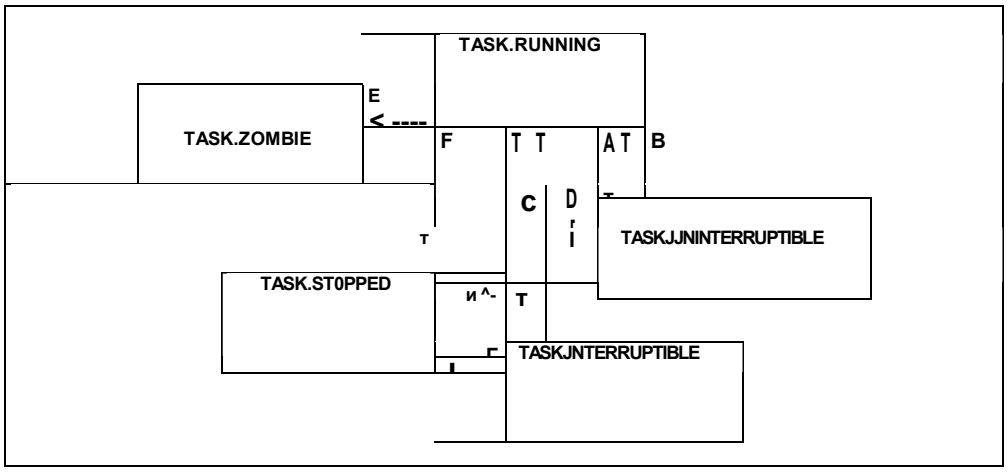


Рис. 3.11. Переход между состояниями задачи

Таблица 3.7. Краткий перечень переходов задачи

Начальное состояние задачи Linux	Конечное состояние задачи Linux	Агент перехода
TASK__RUNNING	TASKJJNINTERRUPTIBLE	Процесс входит в очередь ожидания
TASK_RUNNING	TASK_INTERRUPTIBLE	Процесс входит в очередь ожидания
TASK_RUNNING	TASK_STOPPED	Процесс получает сигнал SIGSTOP или процесс отслеживается
TASK RUNNING	TASK_ZOMBIE	Процесс убит, но родитель не вызвал sys_wait4 ()
TASK_INTERRUPTIBLE		В течение получения сигнала
	TASK_STOPPED	В течение пробуждения
TASK_UNINTERRUPTIBLE	TASK_STOPPED	Процесс получает ожидаемый ресурс.
TASK_UNINTERRUPTIBLE	TASK_RUNNING	Процесс получает ожидаемый ресурс или установлен на выполнение в результате полученного сигнала
TASK INTERRUPTIBLE TASK RUNNING	TASK RUNNING TASK_RUNNING	Выключен и включен планировщиком

3.4.2.1 Готовность к выполнению

Абстрактный переход между состояниями процесса «готовность к выполнению» не соотносится с существующими переходами между состояниями Linux, потому что это состояние не изменяется (остается в TASK_RUNNING). Тем не менее процесс переходит из очереди готовности в состояние выполнения **(очередь выполнения),** когда он действительно выполняется на процессоре.

TASK_RUNNING в TASK_RUNNING

Linux не имеет специального состояния для различения задачи, выполняемой на процессоре в данный момент, и задачи, остающейся в состоянии TASK_RUNNING, даже когда задача перемещается из очереди и его контекст выполняется. Планировщик выбирает задачу из очереди выполнения. Гл. 7 описывает, как планировщик выбирает следующую задачу для установки на выполнение.

3.4.2.2 Состояние выполнения в состояние готовности

В этой ситуации состояние задачи не изменяется, даже если сама задача претерпевает изменение. Абстрактное состояние процесса поможет нам понять, что происходит. Как и в предыдущем случае, процесс переходит из состояния выполнения в состояние готовности, когда процесс переходит из состояния выполнения на процессоре и помещается в очередь выполнения.

TASK_RUNNING в TASKJIUNNING

Так как в Linux нет специального состояния для задачи, контекст которой выполняется на процессоре, задача в Linux в этом случае не претерпевает перехода между состояниями и остается в состоянии TASK_RUNNING. Планировщик выбирает, когда переключить эту задачу из состояния выполнения и поместить ее в очередь выполнения в соответствии со временем, потраченным задачей на выполнение и ее приоритетом. (Подробности описываются в гл. 7.)

3.4.2.3 Состояние выполнения в состояние блокировки

Когда процесс блокируется, он может быть в одном из следующих состояний: TASK_ INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_ZOMBIE ИЛИ TASK_STOPPED. Теперь опишем, как задача переходит из состояния TASK_RUNNING в каждое из этих состояний, как описано в табл. 3.7.

TASKJIUNNING в TASKJNTERRUPTIBLE

Это состояние обычно вызывается с помощью блокирования функций ввода-вывода, которые ожидают поступления сообщения или ресурса. Что это значит для задачи, находящейся в состоянии TASK_INTERRUPTIBLE? Она просто остается в очереди выполнения, так как она не готова для выполнения. Задача в состоянии TASK_INTERRUPTIBLE просыпается, если ее ресурс становится доступным (время или аппаратура) или поступает

сигнал. Завершение оригинального системного вызова зависит от реализации обработчика прерывания. В примере кода дочерний процесс получает доступ к файлу на диске. Драйвер диска определяет, когда устройство станет доступным и к данным можно будет получить доступ. Поэтому код драйвера будет выглядеть примерно следующим образом:

```
while(1) {
if(resource_available) break();
set_current_state(TASK_INTERRUPTIBLE); schedule(); }
set_current_state(TASK_RUNNING);
```

В этом примере процесс входит в состояние TASK_INTERRUPTIBLE за время, в течение которого выполняется вызов open (). В этой точке он выходит из состояния выполнения с помощью вызова schedule (), а другой процесс из очереди выполнения становится выполняемым процессом. После того как ресурс становится доступным, процесс удаляется из цикла и его состояние изменяется на TASK_RUNNING, которое помещает его обратно в очередь обработки. После этого он ждет, пока планировщик не решит запустить процесс на выполнение.

Следующий листинг демонстрирует функцию interruptible_sleep_on (), которая устанавливает задачу в состояние TASK_INTERRUPTIBLE.

```
kernel/sched.c
2504    void interruptible_sleep_on(wait_queue_head_t  *q)
2505    {
2506      SLEEP_ON_VAR
2507
2508    current->state = TASK INTERRUPTIBLE;
2509
2510      SLEEP_ON_HEAD
2511      scheduleO;
2 512     SLEEP_ON_TAIL
2513   }
```

Макросы SLEEP_ON_HEAD и SLEEP_ON_TAIL заботятся о добавлении и удалении задачи из очереди ожидания (см. раздел «Очередь ожидания» в этой главе). Макрос SLEEP_ON_VAR инициализирует запись о задаче в очереди ожидания, которая добавляется в очередь ожидания.

TASK_RUNNING в TASKJJNINTERRUPTIBLE

Состояние TASK__UNINTERRUPTIBLE похоже на TASK_INTERRUPTIBLE за исключением того, что процесс не формирует сигналы, получаемые, когда он находится в режиме ядра. Это состояние также является состоянием по умолчанию, в которое задача устанавливается при инициализации в процессе ее создания с помощью do_fork(). Функция sleep_on () устанавливает задачу в состояние TASK_UNINTERRUPTIBLE.

kernel/sched. c

```
2 545   long fastcall _____ sched sleep_on(wait_queue_head_t *q)
2546    {
2 547   SLEEP_ON_VAR
2548
2549    current->state = TASK UNINTERRUPTIBLE;
2550
2551    SLEEP ON HEAD
2552    schedule*);
2553    SLEEP ON TAIL
2554
2555    return timeout;
2556   )
```

Эта функция устанавливает задачу в очередь ожидания, устанавливает ее состояние и вызывает планировщик.

TASKJOJNNING в TASKJZOMBIE

Процесс в состоянии TASKJZOMBIE называется зомби-процессом. Каждый процесс в течение своего жизненного цикла проходит через это состояние. Длительность времени, в течение которого процесс остается в этом состоянии, зависит от родителя. Чтобы это понять, представьте, что в UNIX-системах каждый процесс может получить статус выхода дочернего процесса с помощью вызовов wait () или waitpid () (см. раздел «Оповещение родителей и sys_wait4()»). Поэтому родительскому процессу должен быть доступен минимум информации, даже когда дочерний процесс уничтожен. Оставлять процесс живым только для того, чтобы родитель мог получить о нем информацию, - слишком накладно, поэтому используется состояние зомби, в котором ресурсы процесса освобождаются и он возвращается, но его описатель остается.

Это временное состояние устанавливается во время вызова sys_exit () (см. более подробную информацию в разделе «Завершение процесса»). Процесс в этом состоянии никогда снова не запустится. Из этого состояния он может перейти только в состояние TASK_STOPPED.

Если задача остается в этом состоянии слишком долго, родительский процесс не убивает своих детей. Задача зомби не может быть убита, так как она уже не является живой.

Это значит, что задач для убиения не существует, а существуют только описатели, ожидающие освобождения.

TASK_RUNNING в TASKJTOPPED

Этот переход выполняется в двух случаях. Первый случай - это когда отладчик или утилита трассировки манипулирует процессом. Второй случай - это когда процесс получает SIGSTOP или один из сигналов на остановку.

TASKJJNINTERRUPTIBLE или TASKJNTERRUPTIBLE в TASKJSTOPPED

TASK_STOPPED управляет процессами в SMP-системах или в течение обработки сигнала. Процесс устанавливается в состояние TASK_STOPPED, когда процесс получает сигнал на пробуждение или если ядру необходимо, чтобы именно этот процесс не отвечал ни на какие сигналы (как будто он установлен, например, в TASK_INTERRUPTIBLE).

Если задача не находится в состоянии TASK_ZOMBIE, процесс устанавливается в состояние TASK_STOPPED до того, как он получит сигнал SIGKILL.

3.4.2.4 Состояние блокировки в состояние готовности

Переход из блокированного состояния в состояние готовности происходит после получения данных или доступа к оборудованию, которого ожидает процесс. Два специфичных для Linux перехода, подпадающие под эту категорию, - это из TASKJNTERRUPTIBLE В TASK_RUNNING и ИЗ TASK_INTERRUPTIBLE в TASK_RUNNING.

3.5 Завершение процесса

Процесс может завершаться добровольно явным образом и добровольно неявным образом либо принудительно. Добровольное завершение может быть выполнено двумя способами:

1. В результате возврата из функции main () (неявное завершение).
2. С помощью вызова exit () (явное завершение).

Выполнение возвращения из функции main () преобразуется в вызов exit (). При этом компоновщик вставляет вызов exit ().

Принудительное завершение может быть получено несколькими способами:

1. Процесс получает сигнал, который не может обработать.
2. Во время выполнения в режиме ядра происходит исключение.
3. Программа получает SIGABRT или другой сигнал на завершение.

Завершение процесса обрабатывается различным образом в зависимости от того, ж^ его родитель или нет. Процесс может:

- завершиться до родителя,

- завершиться после родителя.

В первом случае дети превращаются в зомби-процессы до того момента, как родитель сделает вызов wait () /waitpid(). Во втором случае статус родителя дочернего процесса будет наследоваться от процесса init(). Таким образом, при завершении процесса ядро проверяет все активные процессы на предмет того, что завершаемый процесс является их родителем. Если такие процессы были найдены, то PID их родителя устанавливается в I[1].

Посмотрим на пример еще раз и проследим его до самого его конца. Процесс явно вызывает exit (0). (Обратите внимание, что, кроме этого, может быть вызван _exit (), return (0) или программа просто дойдет до конца main без всяких дополнительных вызовов.) Библиотечная С-функция exit () выполняет, в свою очередь, системный вызов sys_exit (). Мы можем просмотреть следующий код и увидим, что происходит с процессом далее.

Теперь мы посмотрим функцию, завершающую процесс. Как говорилось ранее, наш процесс f oo вызывает exit (), который вызывает первую рассматриваемую нами функцию - sys_exit(). Мы проследим вызов sys_exit() и углубимся в детали do_exit ().

3.5.1 Функция sys_exit()

```
kernel/exit.c
asmlinkage long sys_exit(int error_code)
{
  do_exit( (error_code&0xff )«8) ;
}
```

sys_exit () для различных архитектур не различается, а их работа довольно понятна - все они выполняют вызов do_exit () и преобразуют код выхода в формат, требуемый ядром.

3.5.2 Функция do_exit()

```
kernel/exit.c
7 07  NORET TYPE void do exit(long code)
708   (
709    struct ta?k struct *tsk = current;
710
711    if (unlikely (in_interrupt()))
```

[1] **То есть их родителем становится процесс init ().** ***Примеч. науч. ред.***

```
712    panic("Aiee, killing interrupt handler!");
713   if (unlikely(!tsk->pid))
714    panic("Attempted to kill the idle task!");
715   if (unlikely(tsk->pid == 1))
716    panic("Attempted to kill init!");
717   if (tsk->io context)
718    exit_io_context();
719   tsk->flags |= PF EXITING;
720   del  timer sync (&tsk->real  timer) ;
721
722   if (unlikely(in_atomic()))
723    printk(KERN INFO "note:%s[%d] exited with preempt count %d\n",
724       current->comm, current->pid,
725       preempt_count());
```

Строка 707

Код параметра представляет собой код выхода, который процесс возвращает родителю.

Строки 711-716

Проверка маловероятных, но возможных непредвиденных ситуаций. Включает следующее:

1. Проверку, что мы не внутри обработчика прерывания.
2. Проверку, что мы не в задаче idle (PID=0) или в задаче init (PID=l). Обратите внимание, что процесс init убивается только при завершении работы системы.

Строка 719

Здесь мы устанавливаем PF_EXITING в поле flags структуры задачи. Это означает, что процесс завершается. Например, такая конструкция используется при создании временного интервала для заданного процесса. Флаги процесса проверяются, и тем самым достигается экономия процессорного времени.

kernel/exit.c

```
727   profile exit  task(tsk) ;
728
729   if (unlikely(current->ptrace & PT_TRACE_EXIT)) {
73 0    current->ptrace message = code;
731    ptrace_notify((PTRACE_EVENT_EXIT « 8) | SIGTRAP);
732    }
733
734   acct_process(code);
```

```
735      exit mm(tsk);
736
737   exit sem(tsk) ;
73 8      exit f iles (tsk) ;
739       exit fs(tsk) ;
740    exit  namespace(tsk) ;
741    'exit_thread() ;
```

Строки 729-732

Если процесс отслеживается и установлен флаг PT_TRACE_EXIT, мы передаем код выхода и уведомляем об этом родительский процесс.

Строки 735-742

Эти строки выполняют очистку и перераспределение ресурсов, используемых задачей, и больше не нужны;___exit__mm() освобождает выделенную для процесса память и освобождает структуру mm_struct, ассоциированную с процессом; exit_sem() убирает связь задачи с любыми семафорами IPC;_____ exit_f iles () освобождает любые файлы, используемые процессом, и декрементирует счетчик файла;__ exit_f s () освобождает все системные данные.

kernel/exit.c

```
744   if (tsk->leader)
745    disassociate ctty(l);
746
747   module put(tsk->thread info->exec domain->module);
748   if (tsk->binfmt)
749    module_put(tsk->binfmt->module);
```

Строки 744-475

Если процесс является лидером сессии, можно ожидать, что он имеет контрольный терминал или tty. Эта функция убирает связь между задачей-лидером и контролирующим tty.

Строки 747-749

В этом блоке мы уменьшаем счетчик ссылок для модуля:

kernel/exit.c

```
751   tsk->exit_code = code;
```

```
752                     exit_notify(tsk);
753
754     if (tsk->exit_signal == -1 && tsk->ptrace == 0)
755       release_task(tsk);
756
757     schedule();
758     BUGO;
759     /* Избегание  "noreturn  function does  return".  */
760     for(;;);
761   }
```

Строка 751

Устанавливает код выхода в поле exit_code структуры task_struct.

Строка 752

Родителю посылается сигнал SIGCHLD, а состояние задачи устанавливается в TASK_Z0MBIE; exit_notify() уведомляет всех, кто связан с задачей, о ее приближающейся смерти. Родитель информируется о коде выхода, а в качестве родителя детей процесса назначается процесс init. Единственным исключением из этого правила является ситуация, когда другой существующий процесс выходит из той же группы процессов: в этом случае существующий процесс используется как суррогатный родитель.

Строка 754

Если exit_signal равен -1 (что означает ошибку) и процесс не является ptraced, ядро вызывает планировщик для освобождения описателя процесса этой задачи и для освобождения его временного среза.

Строка 757

Передача процессора новому процессу. Как мы увидим в гл. 7, вызов schedule () не возвращается. Весь код после этой строки обрабатывает неправильные ситуации или избегает замечаний компилятора.

3.5.3 Уведомление родителя и sys_wait4()

Когда процесс завершается, об этом уведомляется его родитель. Перед этим процесс находите^ состоянии зомби, когда все ресурсы возвращаются в ядро, и остается только описатель процесса. Родительская задача (например, оболочка Bash) получает сигнал SIGCHLD, посылаемый ядром, когда дочерний процесс завершается. В примере оболочка вызывает wait (), когда хочет получать уведомления. Родительский процесс может игнорировать сигнал, не реализуя обработчик прерывания, и может вместо этого выбрать вызов wait () [или waitpid ()] в любой точке.

Семейство функций wait служит для решения двух основных задач:

- **Гробовщик.** Получение информации о смерти задачи.
- **Гробокопатель.** Избавление ото всех отслеживаемых процессов.

Наша родительская программа может выбирать вызов одной из четырех функций в семействе wait:

- pid_t wait(int *status)
- pid_t waitpid(pid_t pid, int *status, int options)
- pid_t wait3(int *status, int options, struct rusage *rusage)
- pid__t wait4 (pid_t pid, int *status, int options, struct rusage *rusage)

Каждая функция, в свою очередь, вызывает sys_wait4 (), который порождает множество уведомлений.

Процесс, вызывающий функцию wait, блокируется до того, как один из его дочерних процессов завершается или возвращается сразу, если дочерний процесс уже завершен (или если у процесса нет дочерних процессов). Функция sys_wait4 () показывает нам, как ядро управляет этим уведомлением:

```
kernel/exit.c
1031  asmlinkage long sys  wait4 (pid t pid, unsigned int *  stat addr,
int options, struct rusage * ru)
1032   (
1033     DECLARE_WAITQUEUE(wait,  current);
1034     struct task_struct  *tsk;
1035     int  flag,  retval; 1036
1037     if (options  &  ~(WNOHANG|WUNTRACED| ____ WNOTHREAD| ___ WCLONE  | ___ WALL))
1038       return -EINVAL;
1039
1040     add_wait_queue(&current->wait_chldexit,&wait);
1041   repeat:
1042     flag  =  0;
1043     current->state  =  TASK__INTERRUPTIBLE;
1044     read__lock(&tasklist_lock) ;
```

Строка 1031

Параметры включают PIII целевого процесса, адрес, куда помещается статус выхода дочернего процесса, флаги для sys_wait4 () и адрес, по которому размещена информация об используемых ресурсах.

Строки 1033 и 1040

Определение очереди ожидания и добавление в нее процесса. (Более подробно это описано в разделе «Очередь ожидания».)

Строки 1037-1038

Этот код в основном проверяет ошибочные состояния. Функция возвращает код ошибки, если в системный вызов переданы неправильные параметры. В этом случае возвращается ошибка EINVAL.

Строка 1042

Переменная flag устанавливается в начальное значение 0. Эта переменная изменяется, как только аргумент pid оказывается принадлежащим к одной из дочерних задач вызова.

Строка 1043

Это код, в котором вызывающий код блокируется. Состояние задачи изменяется С TASK_RUNNING на TASK_INTERRUPTIBLE.

kernel/exit.c

```
1045   tsk = current;
1046   do {
1047    struct task struct *p;
1048    struct list_head *_p;
1049    int ret;
1050
1051    list_for_each(_p,&tsk->children) {
1052     p = list entry( p,struct task struct,sibling);
1053
1054     ret = eligible child(pid, options, p);
1055     if (lret)
1056       continue;
1057     flag = 1;
1058     switch (p->state) {
1059     case TASK STOPPED:
1060       if (!(options & WUNTRACED) &&
1061         !(p->ptrace & PT PTRACED))
1062       continue;
1063     retval = wait_task_stopped(p, ret == 2,
```

```
1064          stat addr, ru);
1065      if (retval != 0) /* Освобождает блокировку. */
1066       goto end wait4;
1067      break;
1068     case TASK ZOMBIE:

1072        if (ret == 2)
1073         continue;
1074        retval = wait task  zombie(p, stat addr, ru);
1075        if (retval != 0) /* Освобождает блокировку. */
1076         goto end wait4;
1077        break;
1078       }
1079     }

1091     tsk = next thread(tsk);
1092     if (tsk->signal != current->signal)
1093      BUG();
1094    } while (tsk != current);
```

Строки 1046 и 1094

Цикл do while выполняется один раз за цикл при поиске себя и затем продолжается при поиске других задач.

Строка 1051

Повтор действия для каждого процесса в списке детей задачи. Помните, что при этом родительский процесс ожидает завершения детей. Процесс, находящийся в состоянии TASK__INTERRUPTIBLE, перебирает весь список своих детей.

Строка 1054

Определение, имеет ли передаваемый параметр pid допустимое значение.

Строки 1058-1079

Проверка состояния каждой дочерней задачи. Действия выполняются, только если ребенок остановлен или в состоянии зомби. Если задача спит, готова или выполняется (предыдущее состояние), ничего не делается. Если дочерний процесс находится в состоянии TASK_STOPPED и используется опция UNTRACED (что означает, что задача не останавливается по причине отслеживания процесса), мы проверяем состояние дочернего процесса, о котором получена информация, и возвращаем информацию об этом дочернем процессе. Если дочерний процесс находится в состоянии TASK_ZOMBIE, он убирается.

kernel/exit.c

```
1106    retval = -ECHILD;
1107   end_wait4:
1108    current->state = TASK_RUNNING;
1109    remove_wait_queue(&current->wait_chldexit,&wait) ;
1110    return retval;
1111   }
```

Строка 1106

Если мы добрались до этой точки, переданный параметр PID не является дочерним процессом вызывающего процесса. ECHILD - это ошибка, используемая для уведомления об этой ошибке.

Строки 1107-1111

В этой точке весь список дочерних процессов обработан и все дочерние процессы, которые нужно было удалить, удалены. Блокировка родителя снимается, и его состояние опять устанавливается в TASK_RUNNING. Наконец, удаляется очередь ожидания.

В этой точке вам должны быть знакомы различные состояния, в которых процесс может побывать на протяжении своего жизненного цикла, реализующие их функции ядра и структуры, которые ядро использует для отслеживания всей этой информации. Теперь мы рассмотрим, как планировщик манипулирует и управляет процессами для создания эффекта многопоточной системы. Также мы увидим подробности перехода процесса из одного состояния в другое.

3.6 Слежение за процессом: базовые конструкции планировщика

До этого места мы говорили о концепции состояний и переходов между состояниями процессов с позиции процессов. Мы еще не говорили об управлении переходами и инфраструктуре ядра, выполняющих запуск и остановку процессов. Планировщик обрабатывает все эти подробности. Заканчивая исследование жизненного цикла процесса, мы теперь представим вашему вниманию основы планировщика и того, как он взаимодействует с функцией do_f ork () при создании процесса.

3.6.1 Базовая структура

Планировщик оперирует структурой, называемой **очередью выполнения.** В системе присутствует по одной очереди выполнения на каждый процессор. Основой структуры очереди выполнения являются два приоритетно-отсортированных массива. Один из них

содержит активные задачи, а другой - отработавшие. Обычно активная задача выполняется в течение определенного времени, длиной во временной срез или квант времени, а затем вставляется в массив отработавших задач, где ожидает следующей порции процессорного времени. Когда активный массив становится пустым, планировщик меняет местами эти два массива, меняя активный и отработанный указатели. Далее планировщик начинает выполнять задачи из активного массива.

Рис. 3.12 иллюстрирует массив приоритетов в очереди ожидания. Структура массива приоритетов определена следующим образом:

```
kernel/sched.c
192  struct prio_array {
193   int nr active;
194   unsigned long bitmap[BITMAP_SIZE];
195   struct list head queue [MAX  PRIO] ;
196  };
```

Структура prio_array имеет следующие поля:

- **nr_active**. Счетчик, хранящий количество задач, находящихся в массиве приоритетов.
- **bitmap**. Следит за приоритетами в массиве. Настоящая длина bitmap зависит от размера unsigned long в системе. Ее всегда достаточно для хранения MAX_PRIO бит, но может быть и больше.
- **queue**. Массив, который хранит список задач. Каждый список хранит задачи с определенными приоритетом. Поэтому queue [0] хранит список всех задач с приоритетом 0, queue [1] хранит список всех задач с приоритетом 1 и т. д.

С этим базовым пониманием организации очереди выполнения мы можем проследить работу планировщика с одной задачей на однопроцессорной системе.

3.6.2 Пробуждение после ожидания или активация

Вспомните, что, когда процесс вызывает fork (), создается новый процесс. Как говорилось ранее, процесс, вызывающий fork (), называется родительским, а новый процесс - дочерним. Новый процесс нужно передать планировщику для того, чтобы он получил доступ к процессору. Это происходит в функции do_f ork ().

В функции do_f ork () операции, связанные с пробуждением процесса, выполняют две строчки; copy__process (), вызываемая в строке 1184 linux/kernel/fork, c, вызывает функцию sched_f ork (), которая инициализирует процесс для следующей его вставки в очередь выполнения планировщика; wake_up_f orked_process (), вызываемая в строке 1222 linux/kernel/f ork. c, получает инициализированный процесс

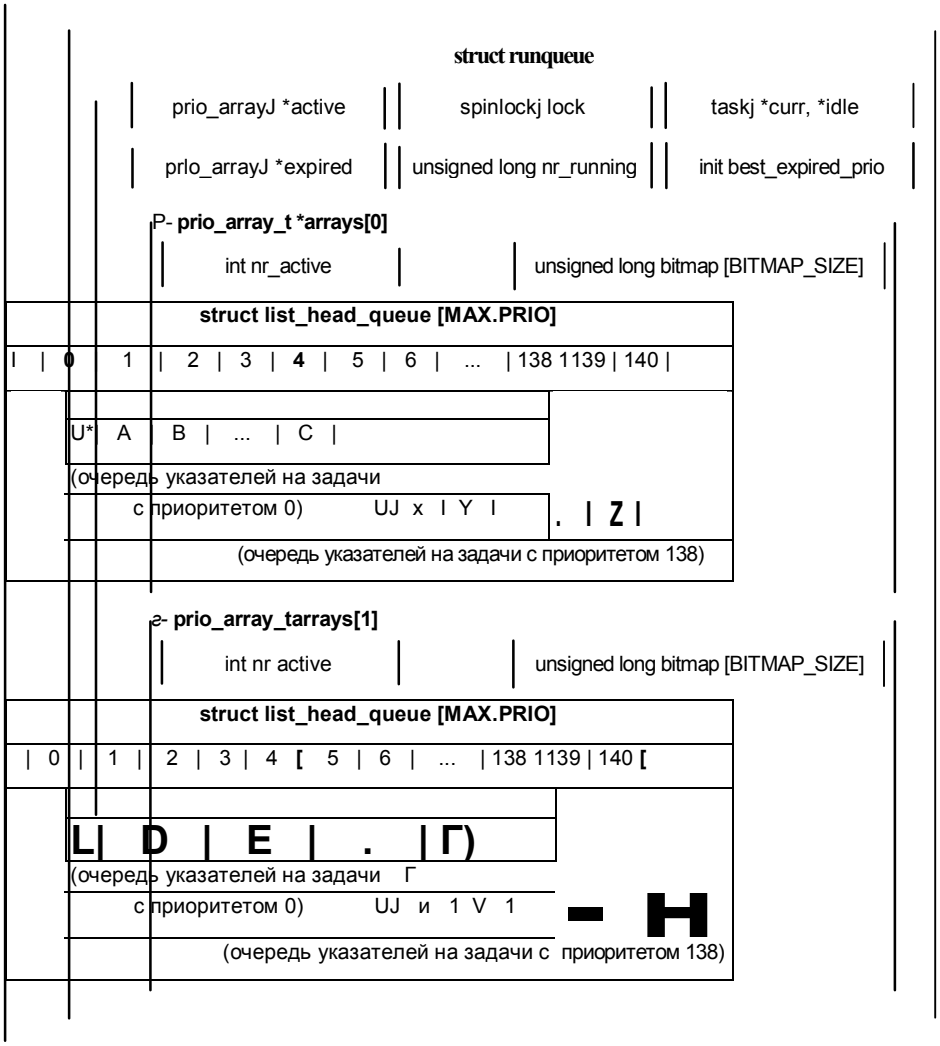


Рис. 3.12. Массив приоритетов в очереди выполнения

и устанавливает его в очередь выполнения. Инициализация и вставка разделяются для того, чтобы новый процесс можно было убить или завершить до того, как он попадет в планировщик. Процесс будет передан планировщику только в том случае, если он будет создан, инициализирован и не будет получать сигналов ожидания.

3.6.2.1 sched_fork(): инициализация планировщика для нового ответвленного процесса
Функция sched_fork() выполняет настройку инфраструктуры, требуемой планировщику для нового ответвленного процесса:

```
kernel/sched.c
719 void sched fork(task t *p)
720 {
721   /*
722   * Здесь мы помечаем процесс выполняемым, но еще не помещаем
723   * в очередь выполнения. Это дает гарантию, что никто не запустит
724   * его на выполнение и что сигнал или другое внешнее
72 5  * событие не сможет его разбудить и вставить в очередь выполнения.
726   */
727   p->state = TASKJRUNNING;
728   INIT LIST HEAD(&p->run list);
729   p->array = NULL;
730   spin_lock_init(&p->switch_lock);
```

Строка 727

Процесс обозначается выполняемым с помощью установки атрибута state в структуре задачи в TASK_RUNNING для того, чтобы удостовериться, что в очередь выполнения не добавлены события, а запуск процесса до do__f or k () и copy_process () проверяет, что процесс был создан правильно. Когда проверка завершена, do__f ork () добавляет его в очередь ожидания с помощью wake_up_f orked__process ().

Строки 728-730

Инициализация поля run_list-npouecca. Когда процесс активируется, поле run_list связывает вместе очередь структуры массива приоритетов и очередь выполнения. Поле процесса array устанавливается в NULL для того, чтобы обозначить, что он не является частью ни одного из массивов приоритетов очереди выполнения. Следующий блок - sched_fork() со строки 731 до 739 работает с приоритетом обслуживания ядра. (Более подробную информацию об этом можно увидеть в гл. 7.)

```
kernel/sched.c
740 /*
741 * Разделение временного среза между родителем и детьми, при котором
742 * общее количество временного среза, выделенного системой,
743 * не изменяется, обеспечивая более честное планирование.
744 */
```

```
745   local irq disable();
746   p->time_slice = (current->time_slice +1) » 1;
747 /*
748 * Остатки первого временного среза возвращаются
749 * родителю, если дочерний процесс слишком рано закончит работу.
750 */
751   p->first time slice = 1;
752   current->time slice »= 1;
753   p->timestamp = sched_clock();
754   if (!current->time slice) {
755   /*
756   * Это редкий случай, возникающий, когда у родителя остается только
757   * один миг от его временного среза. В этом случае блокировка
758   * очереди выполнения не представляет никаких сложностей.
759   */
760     current->time slice = 1;
761     preempt_disable();
762     scheduler tick(0, 0) ;
763     local irq enable();
764     preempt enable();
765    } else
766     local__irq_enable () ;
767 }
```

Строки 740-753

После отключения локальных прерываний мы делим родительский временной срез между родителем и его детьми, используя оператор сдвига. Первый временной срез нового процесса устанавливается в 1, потому что он еще не выполнялся, а его время создания инициализируется в текущее время в наносекундах.

***Строки** 754-767*

Если временной срез родителя равен 1, в результате деления родителю остается время 0 на выполнение. Так как родитель является текущим процессом планировщика, нам понадобится планировщик для выбора нового процесса. Это делается с помощью вызова scheduler_tick() (в строке 762). Приоритет отключается для того, чтобы удостовериться, что планировщик выбирает новый процесс без прерывания. Как только это сделано, мы включаем приоритет и восстанавливаем локальные прерывания.

В этой точке новый созданный процесс имеет специфическую для планировщика инициализированную переменную и имеет начальный временной срез, равный половине оставшегося временного среза родительского процесса. Заставляя процесс пожертвовать частью своего времени и передать его дочернему процессу, ядро позволяет процессам не

отнимать слишком большую часть процессорного времени. Если процессу выделить слишком много времени, нехороший процесс может наплодить множество дочерних и быстро съесть все процессорное время.

После успешной инициализации процесса и проверки инициализации do_fork () вызывает wake_up_forked_process ():

```
kernel/sched.c
922 /*
923 * wake up forked  process  - будит свежеответвленный процесс.
924 * Эта функция производит некоторые статистические приготовления
92 5 * планировщика, которые необходимо выполнить для каждого нового
92 6 * созданного процесса.
927 */
928 void fastcall wake up forked process(task t * p)
929 {
93 0  unsigned long flags;
931  runqueue t *rq = task rq lock(current, &flags);
932
933  BUG ON(p->state != TASK RUNNING);
934
935  /*
93 6  * Мы уменьшаем среднее время сна родительского процесса
937  * и дочернего процесса для предотвращения порождения
938  * сверхинтерактивных задач от сверхинтерактивных задач.
939  */
940  current->sleep_avg = JIFFIES_TO_NS(CURRENT_BONUS( current) *
941    PARENT PENALTY / 100 * MAX SLEEP AVG / MAX BONUS) ;
942
943  p->sleep avg = JIFFIES TO NS (CURRENT BONUS (p) *
944    CHILD PENALTY / 100 * MAX SLEEP AVG / MAX BONUS) ;
945
946  p->interactive credit = 0;
947
948  p->prio = effective prio(p);
949  set task cpu(p, smp processor id());
950
951  if (unlikely(!current->array))
952      activate task(p, rq) ;
953  else {
954   p->prio = current->prio;
955   list add tail(&p->run list, &current->run list);
956   p->array = current->array;
957   p->array->nr_active++;
958   rq->nr_running++;
```

```
959   }
960   task__rq__unlock (rq, &flags) ;
961 }
```

Строки 930-934

Первое, что делает планировщик, - это блокирование структуры очереди ожидания. Любое изменение очереди ожидания производится с заблокированной структурой. Также выдается сообщение об ошибке, если процесс не помечен как TASK_RUNNING. Процесс должен устанавливаться в это состояние в функции sched__f ork () (см. строку 727 в ранее приведенном kernel/sched. c).

Строки 940-947

Планировщик вычисляет среднее время сна родительского и дочернего процессов. Значение среднего времени сна показывает, сколько времени процесс проводит в состоянии сна в сравнении с тем, сколько времени он выполняется. Оно увеличивается во время сна процесса и уменьшается каждый тик времени во время выполнения. Интерактивный, или ограниченный вводом-выводом, процесс затрачивает большую часть времени на ожидание ввода и обычно имеют высокое среднее время сна. Неинтерактивные, или ограниченные процессором, процессы тратят большую часть отводимого им времени на выполнение на процессоре вместо ожидания ввода-вывода и имеют низкое время сна. Так как пользователи желают видеть результат из ввода в виде клавиатурных нажатий или перемещений мыши, интерактивные процессы получают от планировщика больше преимуществ, чем неинтерактивные процессы. При этом планировщик принудительно вставляет интерактивный процесс в массив активных приоритетов, после того как их временной срез завершен. Для предотвращения порождения неинтерактивных дочерних процессов от интерактивных и, таким образом, избегания диспропорционального разделения процессора эти формулы используются для понижения среднего времени сна родителя и его детей. Если новый ответвленный процесс является интерактивным, после продолжительного сна он получит достаточный приоритет чтобы наверстать упущенные преимущества.

Строка 948

Функция ef f ective_prio () изменяет статический приоритет процесса. Она возвращает приоритет в пределах от 100 до 139 (от MAX_RT_PRIO до MAX_PRI0-1). Статический приоритет процесса может быть изменен до 5 в каждом направлении на основе использования процессора в прошлом и времени, потраченного на сон, но он всегда остается в этих пределах. Из командной строки мы сообщаем о значении nice-процесса, который варьируется и от +19 до -20 (от наименьшего до максимального приоритета). Приоритет nice, равный 0, соответствует статическому приоритету 120.

Строка 949

Процесс устанавливает свой атрибут процессора в значение текущего процессора.

Строки 951-960

В этом блоке кода новый процесс или дочерний процесс копирует информацию для планировщика из родителя, который является текущим, и затем вставляет себя в соответствующее место очереди выполнения. Мы закончили наше изменение очереди сообщении и теперь должны ее разблокировать. Разд. 3.7 и рис. 3.13 освещает этот процесс более подробно.

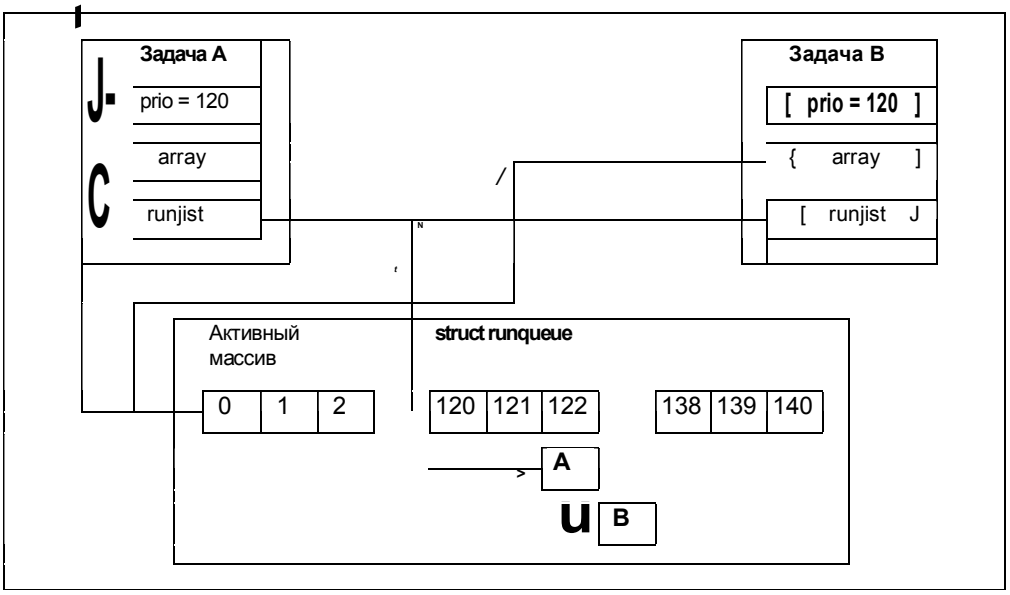


Рис. 3.13. Вставка в очередь выполнения

Указатель **array** указывает на массив приоритетов в очереди выполнения. Если текущий процесс не указывает на массив приоритетов, это означает, что текущий процесс завершен или спит. В этом случае поле текущего процесса runlist не находится в массиве приоритетов очереди выполнения, что значит, что операция list__add_tail () (в строке 955) провалится. Вместо этого мы вставляем новый созданный процесс, используя __activeate_task (), которая добавляет новый процесс в очередь без обращения к его родителю.

В нормальной ситуации, когда текущий процесс ожидает процессорного времени в очереди выполнения, процесс добавляется в очередь в слот p__prior массива приоритета. Массив, в который добавляется процесс, имеет собственный счетчик процесса

nr_ac tive, который мы увеличиваем, как увеличиваем и собственный счетчик процесса nr_running. Наконец, мы снимаем блокировку с очереди выполнения.

В случае, когда текущий процесс не указывает на массив приоритетов и очереди выполнения, полезно видеть, как планировщик управляет очередью выполнения и атрибутами массива приоритетов.

```
kernel/sched.c
3 66 static inline void   activate task(task t *p, runqueue t *rq)
367 {
3 68   enqueue task(p, rq->active) ;
3 69   rq->nr running++;
370 }
```

_ activate_task () помещает данный процесс p в активный массив приоритетов в очереди выполнения rq и увеличивает поле nr_running, которое является счетчиком общего количества процессов, находящихся в очереди выполнения.

```
kernel/sched.c
311  static void enqueue_task  (struct  task_struct  *p,  prio_array_t  *array)
312  {
313    list add tail(&p->run list, array->queue + p->prio);
314      set bit(p->prio, array->bitmap);
315    array->nr active++;
316    p->array = array;
317  }
```

Строки 311-312

enqueue_task() берет процесс p и помещает его в массив приоритетов array при инициализации различных аспектов массива приоритетов.

Строка 313

run_list процесса добавляется в хвост очереди, находящейся в p->prio в массиве приоритетов.

Строка 314

Битовая карта массива приоритетов p->prio устанавливается при запуске планировщика, когда он видит, что существует процесс, выполняемый с приоритетом p->prio.

Строка 315

Счетчик массива приоритетов процесса увеличивается, отражая добавление нового процесса.

***Строка** 316*

Массив указателей процесса устанавливается в массив приоритетов, в который он только что был добавлен.

Итак, акт добавления нового, напрямую ответвленного процесса даже при подробном рассмотрении кода может сбивать с толку из-за похожести используемых в планировщике имен. Процесс помещается в конец списка массива приоритетов очереди выполнения, в слот, определяемый приоритетом процесса. Затем процесс записывает в свою структуру местоположение в массиве приоритетов и список, в котором он находится.

3.7 Очередь ожидания

Мы обсудили процесс перехода между состояниями TASK_INTERRUPTIBLE и TASK_RUNNING или TASK_UNINTERRUPTIBLE. Теперь мы рассмотрим другую структуру, вовлеченную в этот переход. Когда процесс ожидает наступления внешнего события, он удаляется из очереди выполнения и помещается в **очередь ожидания.** Очередь ожидания - это дву связный список структур wait_queue_t. Структура wait_queue_t получает всю необходимую для слежения за ожидающим процессом информацию. Все задачи, ожидающие определенного события, помещаются в очередь ожидания. Задачи данной очереди ожидания пробуждаются, как только наступает ожидаемое ими событие, убирают себя из очереди ожидания и переходят обратно в состояние TASK_RUNNING. Вы можете вспомнить, что системный вызов sys_wait4 () использует очередь ожидания, когда родитель посылает требование на получение статуса ответвленного дочернего процесса. Обратите внимание, что задача, ожидающая внешнего события (и поэтому больше не находящаяся в очереди выполнения[1]), может находиться либо в состоянии TASK_UNINTERRUPTIBLE, либо в состоянии TASK_INTERRUPTIBLE.

Очередь ожидания представляет собой двусвязный список структур **wait_queue_t,** хранящих указатели на структуры процесса для заблокированных процессов. Каждый список имеет в заголовке структуру **wait_queue_head_t,** которая обозначает голову списка и служит разделителем начала и конца списка, позволяющим избежать лишних пробегов по всему списку wait_queue_t. Рис. 3.14 иллюстрирует реализацию очереди ожидания. Теперь мы рассмотрим структуры wait_queue_t и wait_queue_head_t.

```
include/linux/wait.h
19  typedef struct  _wait_queue wait_queue_t;

23  struct  _ wait_queue {
```

[1] **Задача убирается из очереди выполнения, как только она засыпает, и, следовательно, передает управление другому процессу.**

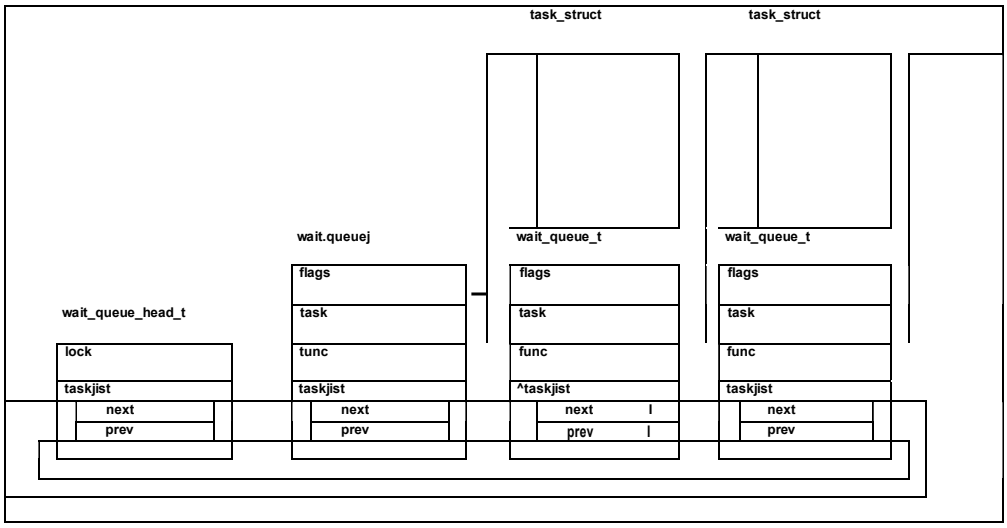


Рис. 3.14. Структура очереди ожидания

```
24   unsigned int flags;
25  ttdefine WQ FLAG EXCLUSIVE 0x01
26 struct task struct * task;
27   wait_queue_func_t func;
28   struct list_head task_list;
29  };
30
31  struct   wait queue head {
32   spinlock_t lock;
33   struct list_head task_list;
34  };
35 typedef struct __wait_queue_head wait_queue_head_t;
```

Структура wait_queue_t состоит из следующих полей:

- **flags.** Может хранить значение WO FLAG EXCLUSIVE, соответствующее 1, и ~WO FLAG EXCLUSIVE, соответствующее 0. Флагом помечаются эксклюзивные процессы. Эксклюзивные и неэксклюзивные процессы описываются в подразд. 3.7.1.
- **task.** Указатель на описатель процесса, помещаемого в очередь ожидания.
- **func.** Структура, хранящая функцию, используемую для пробуждения задачи в очереди ожидания. Поле по умолчанию использует def ault_wake_f unction (), которая описывается в подразд. 3.7.2.

- **wait_gueue_t** определена следующим образом:

```
include/1inux/wai t.h
typedef int (*wait_queue_func__t) (wait_queue__t *wait,
unsigned mode, int sync);
```

  где wait - указатель на очередь ожидания, mode имеет значение TASK_INTERRUPTIBLE или TASK_UNINTERRUPTIBLE, а sync определяет, нужно ли синхронизировать пробуждение.

- **task__list.** Структура, хранящая указатели на предыдущий и следующий элементы в очереди ожидания.

Структура__ wait__queue__head__t - является головой списка очереди ожидания и состоит из следующих полей:

- **lock.** Единственная блокировка очереди позволяет синхронизировать добавление и удаление элементов из очереди ожидания.
- **task_list.** Структура, указывающая на первый и последний элементы очереди ожидания.

Раздел «Очередь ожидания» в гл. 10 («Добавление вашего кода в ядро») описывает пример реализации очереди ожидания. В общих чертах способ, которым процесс погружает себя в сон, требует вызова одного из макросов wait_event* (кратко описывающихся там же) или выполнения такой последовательности шагов, показанных в примере в гл. 10:

1. Описывая очередь ожидания, процесс засыпает с помощью DECLARE_WAITQUEUE.
2. Добавляет себя в очередь ожидания с помощью add_wait_queue() или add_wait_queue_exclusive ().
3. Состояние процесса изменяется на TASK_INTERRUPTIBLE или на TASK_UNINTERRUPTIBLE.
4. Проверяется наступление внешнего события или вызывается schedule (), если оно еще не наступило.
5. После наступления внешнего события устанавливается в состояние `TASK_RUNNING`.
6. Удаляет себя из очереди ожидания с помощью вызова remove_wait_queue ().

Процесс пробуждения процесса обрабатывается с помощью вызова одного из макросов wake_up. Эти макросы пробуждают все процессы, которые принадлежат к ука-

занной очереди ожидания. Это устанавливает задачу в состояние TASK_RUNNING и помещает ее обратно в очередь выполнения.

Давайте теперь рассмотрим, что происходит, когда мы вызываем функцию add_wait_queue ().

3.7.1 Добавление в очередь ожидания

Для добавления спящего процесса в очередь ожидания используется две функции: add_wait_queue () и add_wait_queue_exclusive (). Для обеспечения двух видов сна процесса существует две функции. Неэксклюзивный ожидающий процесс - это процесс, который ожидает наступления состояния, не разделяемого с другими ожидающими процессами. Эксклюзивный ожидающий процесс ожидает наступления состояния, которое ожидает другой процесс, что потенциально может привести к соревнованию между процессами.

Функция add_wait__queue () вставляет неэксклюзивный процесс в очередь ожидания. Неэксклюзивный процесс в любом случае будет разбужен ядром, как только наступит событие, которого он ожидает. Функция устанавливает поле flags структуры очереди ожидания в соответствующее спящему процессу значение 0, устанавливает блокировку очереди ожидания для избежания прерывания при доступе к той же очереди и возникновения соревновательной ситуации, добавляет структуру в список очереди ожидания и затем снимает с очереди ожидания блокировку, делая ее доступной для других процессов.

kernel/fork.c

```
93  void add_wait_queue(wait_queue_head_t *q,  wait_queue_t * wait)
94   {
95    unsigned long flags;
96
97    wait->flags &= ~WQ_FLAG_EXCLUSIVE;
98    spin lock irqsave(&q->lock, flags); 9
9     add_wai t_queue(q, wait);
100   spin_unlock_irqrestore(&q->lock,  flags);
101   }
```

Функция add_wait_queue_exclusive() вставляет эксклюзивный процесс в очередь ожидания. Функция устанавливает поле flags структуры очереди ожидания в 1 и далее работает аналогично add_wait_queue O, за исключением того, что эксклюзивный процесс добавляется в конец очереди. Это означает, что в конкретной очереди ожидания неэксклюзивные процессы находятся в начале, а эксклюзивные - в конце. Это необходимо, как мы увидим далее при рассмотрении пробуждения процессов, для того, чтобы процессы в очереди ожидания будились именно в такой последовательности.

```
kernel/fork.c
105   void add__wait_queue_exclusive(wait_queue_head_t *q,
                                                wait_queue_t * wait)
106 {
107 unsigned long flags;
108
109 wait->flags |= WQ FLAG EXCLUSIVE;
110 spin lock irqsave(&q->lock, flags);
111       add wait queue tail(q, wait);
112 spin__unlock irqrestore(&q->lock, flags)
113 }
```

3.7.2 Ожидание события

Интерфейсы sleep_on (), sleep_on__timeout () и interruptible_sleep__on () , еще поддерживаемые в 2.6, в версии 2.7 будут удалены. Поэтому мы рассмотрим только интерфейс wait_event* (), пришедший на смену интерфейсу sleep_on* ().

Интерфейс wait_event* () включает в себя wait_event O, wait_event_ interruptible () и wait__event_interruptible_timeout (). Рис. 3.15 демонстрирует основные используемые для этого функции.

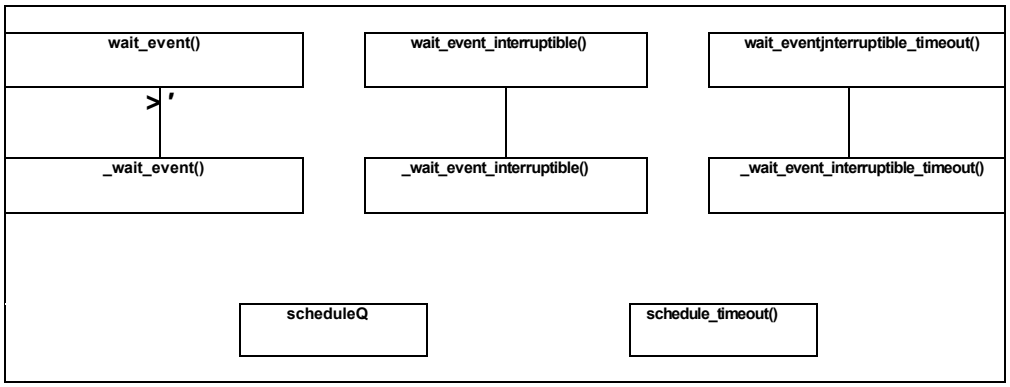


*Рис. 3.15. График вызова wait_event*0*

Мы отследим и опишем интерфейсы, связанные с wait_event (), и опишем ее от личия от других двух функций. Интерфейс wait__event () является оберткой вокруг вы зова __wai t_event () с бесконечным циклом, который прерывается, только если насту пает ожидаемое событие; wait_event_interruptible__timeout () передает тре тий параметр вызова ret типа int, используемый для передачи времени ожидания.

wait_event_interruptible () является единственным из трех интерфейсов, который возвращает значение, которое равно -ERESTARTSYS, если сигнал прерывает ожидаемое событие, или 0, если событие насупило:

```
include/linux/wait.h
137   #define wait_event(wq, condition)
138   do {
139    if (condition)
140     break;
141    __ wait_event(wq, condition);
142   } while (0)
```

Интерфейс __ wait_event () выполняет всю работу по изменению состояния процесса и манипуляции с описателем:

```
include/linux/wait.h
121   #define   wait event(wq, condition)
122   do {
123    wait  queue t   wait;
124    init waitqueue entry(&   wait, current);
125
126    add wai t queue (&wq, &  wait);
127    for (;;) {
128     set current state(TASK_UNINTERRUPTIBLE);
129     if (condition)
13 0    break;
131     scheduleO;
132    }
133    current->state = TASK RUNNING;
134    remove wai t queue (&wq, &  wai t) ;
135   } while (0)
```

Строки 124-126

Инициализация описателя очереди ожидания для текущего процесса и добавление описателя в передаваемую очередь ожидания. До этого момента wait_event_ interruptible и wait_event_interruptible_timeout выглядят иден тично _ wait_event.

Строки 127-132

Этот код устанавливает бесконечный цикл, который будет прерван только по наступлении ожидаемого события. Перед блокировкой по наступлению события мы установим состояние процесса в TASK_INTERRUPTIBLE с помощью макроса

set__current_s tate. Вспомним, что этот макрос связан с указателем на текущий процесс, поэтому нам не нужно передавать в него информацию о процессе. Как только он заблокирован, он передает процессор следующему процессу с помощью вызова scheduler (). В этом смысле wait_event_interruptible () имеет значительное отличие; он устанавливает поле состояния процесса в TASK_UNINTERRUPTIBLE и ожидает вызова для текущего процесса signal_pending; __wait_event_interruptible_timeout очень похож на wait_event_interruptible за исключением вызова schedule_timeout () вместо schedule () при вызове планировщика; schedule_timeout получает в качестве параметра длительность времени ожидания, который передается в оригинальный интерфейс wait__event_interruptible_timeout.

Строки 133-134

В этом участке кода наступает ожидаемое событие, или для двух других интерфейсов может быть получен сигнал, или будет исчерпано время ожидания. Поле state описателя процесса устанавливается в TASK_RUNNING (планировщик помещает процесс в очередь выполнения). Наконец, запись удаляется из очереди ожидания. Функция remove_wait__queue () блокирует очередь ожидания перед удалением записи и снимает блокировку перед выходом.

3.7.3 Пробуждение

Процесс будет разбужен, как только наступит ожидаемое им событие. Обратите внимание, что процесс может заснуть самостоятельно, но не может самостоятельно проснуться. Для пробуждения задач, находящихся в очереди ожидания, может использоваться множество макросов, но все они применяют три основные функции пробуждения. Макросы wake__up, wake_up_nr, wake_up_all/ wake_up_interraptible, wake_up_ interruptible_nr и wake_up_interraptible_all вызывают функцию __wake_up() с разными параметрами. Макросы wake_up_all_sync и wake_up_interruptible_sync вызывают функцию wake_up_sync () с разными параметрами. И наконец, макрос wake_up_locked по умолчанию вызывает функцию _wake_up_locked ():

```
include/linux/wait.h
116 extern void FASTCALL( __wake up(wait queue head t *q,
                          unsigned int mode, int nr) ) ;
117 extern void FASTCALM __wake up locked(wait queue head t *q,
                          unsigned int mode));
118 extern void FASTCALM   wake up svnc (wait queue head t *q,
                          unsigned int mode, int nr));
119
120 #define wake_up(x)    __wake_up( (x) ,TASK_UNINTERRUPTIBLE |
```

```
                                          TASK INTERRUPTIBLE, l)
121 #define wake_up_nr(x, nr)  _ wake_up( (x) ,TASK UNINTERRUPTIBLE |
                                          TASK_INTERRUPTIBLE, nr)
122 #define wake_up_all(x)     _ wake_up( (x) ,TASK UNINTERRUPTIBLE |
                                         TASK_INTERRUPTIBLE, 0)
123 #define wake up all sync (x)    wake up sync ( (x) ,TASK UNINTERRUPTIBLE
                                                | TASK__INTERRUPTIBLE, 0)
124 #define wake_up_interruptible(x)    wake up((x),
                          TASK_INTERRUPTIBLE, 1)
125 #define wake up interruptible nr (x, nr)
             _wake_up( (x) ,TASK_INTERRUPTIBLE, nr)
126 #define wake up interruptible all(x)
               wake up((x),TASK INTERRUPTIBLE, 0)
127 #define wake up locked(x)     wake up locked((x),
                               TASK_UNINTERRUPTIBLE | TASK_INTERRUPTIBLE)
128 #define wake_up_interruptible__sync (x)     wake up sync ( (x) ,
                                              TASK INTERRUPTIBLE, 1
129 )
```

Давайте посмотрим на__ wake___up ():

```
kernel/sched.c
2336    void fastcall _______ wake_up(wait_queue_Jiead_t *q,
                                      unsigned int mode,   int nr_exclusive)
2337    {
2338     unsigned long flags;
2339
2340     spin_lock_irqsave(&q->lock, flags);
2341        wake up common(q,  mode,  nr   exclusive,  0);
2342     spin_unlock_irqrestore(&q->lock,   flags);
2343    }
```

Строка 2336

Параметры, передаваемые **в** __wake_up, включают q, указатель на очередь ожида ния; mode, индикатор типа пробуждаемого потока (определяется состоянием пото ка); и nr_exclusive, который указывает эксклюзивное и неэксклюзивное пробу ждение. Эксклюзивное пробуждение (когда nr_exclusive=0) пробуждает все за дачи в очереди (как эксклюзивные, так и неэксклюзивные), тогда как неэксклюзивное пробуждение пробуждает все неэксклюзивные задачи и только одну эксклюзивную задачу.

Строки 2340, 2342

Эти строки устанавливают и снимают кольцевую блокировку очереди ожидания. Блокировка устанавливается перед вызовом ____ wake_up_common () для ис ключения возможности возникновения соревнования.

Строка 2341

Функция __ wake_up_common () выполняет основные операции функции wakeup:

```
kernel/sched.c
2313    static void  _ wake up .common(wait .queue head t *q,
                          unsigned int mode, int nr exclusive, int sync)
2314    {
2315     struct list_head *tmp, *next;
2316
2317     list for  each safe(tmp, next, &q->task list) {
2318        wait queue t *curr;
2319        unsigned flags;
2320       curr = list_entry(tmp, wait_queue_t, task_list);
2321        flags = curr->flags;
2322        if (curr->func(curr, mode, sync) &&
2323          (flags & WQ FLAG EXCLUSIVE) &&
2324          !—nr exclusive)
2325           break;
2326      }
2327     }
```

Строка 2313

Параметры, передаваемые в __ wake__up_common (), - это q, указатель на очередь ожидания; mode, тип пробуждаемого потока; nr_exclusive, тип предыдущего пробуждения, и sync, который указывает, должно ли пробуждение быть синхронизированным.

Строка 2315

Здесь мы устанавливаем временные указатели для работы с элементами списка.

Строка 2317

Макрос list_for_each_safe сканирует каждый элемент очереди ожидания. Это начало нашего цикла.

Строка 2320

Макрос list__enrty возвращает адрес структуры очереди ожидания, хранимый в переменной tmp.

Строка 2322

Вызов функции поля wait_queue_t. По умолчанию def ault_wake_function () вызывается, как показано ниже.

```
kernel/s ched.c
2296   int default_wake_function(wait_queue_t *curr,
                                 unsigned mode, int sync)
2297   {
2298    task_t *p = curr->task;
2299    return try_to_wake_up(p, mode, sync);
2300   }
```

Эта функция вызывает try_to_wake_up () (kernel / sched. c) для задачи, на которую указывает структура wai t_queue_t. Эта функция выполняет основную работу по пробуждению процесса, включая его помещение в очередь выполнения.

Строки 2322-2325

Цикл завершается, если разбуженный процесс является первым эксклюзивным процессом. Это имеет смысл, когда мы понимаем, что все выполняемые процессы оказались в хвосте очереди ожидания. После того как мы встречаем первую эксклюзивную задачу в очереди ожидания, все оставшиеся задачи тоже являются эксклюзивными, и поэтому мы не станем их пробуждать, а просто прервем цикл.

3.8 Асинхронный поток выполнения

Мы говорили, что процессы могут переходить из одного состояния в другое с помощью прерываний, например из состояния TASK_INTERRUPTIBLE в TASK_RUNNING. Один из способов, которым этого можно добиться, - это асинхронное выполнение с использованием исключений и прерываний. Мы говорили, что процессы могут переходить между режимом ядра и режимом пользователя. Теперь мы перейдем к описанию работы исключений и проследим, как работают прерывания.

3.8.1 Исключения

Исключения, или **синхронные прерывания (synchronous interrupts),** - это аппаратно-возникающие события внутри процессора. Эти события синхронизированы с работой процессора; при этом они возникают не до, а после выполнения инструкции кода. Примером процессорного исключения может служить обращение к виртуальной позиции в памяти, которая физически не существует [известное как ошибка страницы (page **fault)]** и вычисление, в процессе которого возникает деление на 0. Важно запомнить, что обычно исключения (иногда называемые программными irqs, т. е. прерываниями) про-

исходят именно после выполнения инструкции. Их отличие от внешних или асинхронных событий (asynchronous events) обсуждается далее в подразд. 3.8.2.

Большинство современных процессоров (включая x86 и PowerPC) позволяют программистам инициировать исключения с помощью определенных инструкций. Эти инструкции можно рассматривать как обеспеченные аппаратно-функциональные вызовы. В качестве примера могут служить системные вызовы (system calls).

3.8.1.1 Системные вызовы

Linux предоставляет программам в пользовательском режиме точку входа в ядро, через которую можно затребовать сервисы ядра или доступ к аппаратным средствам. Эти точки входа стандартизованы и определены в ядре. Программам в пользовательском режиме доступно множество библиотечных С-функций, наподобие функции fork() на рис. 3.9, связывающих в одной функции код и один или несколько системных вызовов. Когда пользовательский процесс вызывает одну из этих функций, некоторые значения помещаются прямо в соответствующие регистры процессора и генерируют программные прерывания. Далее это программное прерывание вызывает точку входа ядра. Несмотря на то что это не рекомендуется, системные вызовы (syscall) также могут быть вызваны из кода ядра. То, откуда следует вызывать системные вызовы, порождает массу дискуссий, потому что вызов системного вызова из ядра позволяет получить преимущество в скорости. Обратной стороной этого увеличения производительности является увеличение сложности кода и его удобочитаемости. В этом подразделе мы рассмотрим «традиционную» реализацию системных вызовов, когда системные вызовы вызываются из пользовательского пространства.

Системные вызовы обладают свойством перемещать данные между пользовательским пространством и пространством ядра. Для этой цели служат две функции: copy_to_user () и copy_f rom_user (). Как и везде в программировании ядра, здесь важна критическая проверка перемещаемых данных (указателей, длины, описателей и разрешений). Эти функции обладают встроенными проверками. Любопытно, что они возвращают число *непереданных* байтов.

По своей натуре реализация системных вызовов является аппаратно-зависимой. Традиционно на Intel-архитектуре все системные вызовы используют программное прерывание 0x80[1].

Параметры системных вызовов передаются через регистры общего назначения с уникальным номером системного вызова в %eax. Реализация системного вызова нах86-архитектуре ограничивает количество параметров пятью. Если требуется более пяти пара-

[1] **В целях увеличения производительности на новых (PIV+) Intel-процессорах работа выполняется с помощью реализации vsyscall. Виртуальные системные вызовы основаны на вызовах в пользовательском пространстве памяти (обычно на странице «vsyscall») и используют быстрые инструкции sysenter и sysexit (если они доступны) вместо традиционных вызовов 0x80. Аналогичные меры по увеличению производительности присутствуют во многих PPC-реализациях.**

метров, возможна передача указателя на блок параметров. Перед выполнением ассемблерной инструкции int 0x80 с помощью механизма обработки исключений процессора вызывается специальная функция ядра. Рассмотрим пример инициализации входа в системный вызов.

set_system_gate(SYSCALL_VECTOR/&system_call);

Этот макрос создает описатель пользовательской привилегии в записи 128 (SYSCALL_VECTOR), указывающий на адрес обработчика системного вызова в entry. S (sys tem_cal 1).

Как мы увидим в подразд. 3.8.2, посвященном прерываниям, функции прерываний PPC привязаны к определенным позициям в памяти; внешние обработчики прерываний привязаны к адресу 0x500, системный таймер - к адресу 0x900 и т. д. Инструкция системного вызова sc указывает на адрес OxcOO. Давайте посмотрим на участок кода в head. S, где устанавливается обработчик для системного вызова PPC.

```
arch/ppc/kernel/head.S
484   /*Системный вызов*/
485      . = OxcOO
486   SystemCall
487    EXCEPTION PROLOG
488    EXC_XFER_EE_LITE(OxcOO,  DoSyscall)
```

Строка 485

Привязка к адресу. Эта строка указывает загрузчику, что следующая инструкция находится по адресу OxcOO. Так как метки следуют похожим правилам, метка SystemCall вместе с первой строкой кода в макросе EXCEPTION_PROLOG по адресу OxcOO.

Строка 488

Этот макрос отсылает обработчик DoSyscall ().

На обеих архитектурах номер системного вызова и любые его параметры хранятся в регистрах процессора.

Когда обработчик исключения x86 обрабатывает int 0x8 0, он индексируется в таблице системного вызова. Файл arch/ i3 8 б /kernel/ entry. S содержит низкоуровневые функции, обработчики прерываний и таблицу системных вызовов sys_call_table. Это верно и для низкоуровневых PPC-функций в arch/ppc/kernel/entry.S и sys_call_table в arch/ppc/kernel/misc. S.

Таблица системных вызовов - это реализованный на ассемблере массив С с 4-байтовыми элементами. Каждый элемент инициализируется адресом функции. По соглашению мы должны ставить перед началом нашей функции «sys_». Так как позиция в таблице

определяет номер системного вызова, мы должны добавить имя нашей функции в конец списка. Даже у разных языков ассемблера таблицы практически идентичны невзирая на архитектуру. Тем не менее в момент написания этой книги на PPC таблица состоит только из 255 элементов, тогда как в x86-таблице их 275.

Файлы include/asm-i386/unist .h и include/asm-ppc/unistd.h ассоциируют системные вызовы с их номерами позиций в sys_call_table. В этом файле «sys_» заменяется на « ___NR_». Также в этом файле содержится макрос, помогающий пользовательским программам загружать параметры в регистры. (См. описание С- и ассемблерных переменных в разделе о программировании на ассемблере в гл. 2, «Исследовательский инструментарий».)

Посмотрим, как мы можем добавить системный вызов с именем sys_ourcall. Системный вызов необходимо добавить в sys_call_table. Ниже показано добавление нашего системного вызова в sys_call_table x86.

```
arch/i386/kernel/entry.S
607   .data
608   ENTRY(sys_call_table)
609   .long sys_restart_syscall  /*  0  - старый системный вызов  "setupO",
                                      *  используемый для перезапуска*/

878   .long sys_tgkill              /*  270  */
879   .long sys_utimes
880   .long sys_fadvise64_64
881   .long sys_ni_syscall    /*  sys_vserver  */
882   .long sys_ourcall             /*  нашим системным вызовом будет  274   */
883
884   nr_syscalls=(.-sys_call_table)/4
```

На x86 наш системный вызов будет иметь номер 274. Если мы добавим системный вызов с именем sys_ourcall на PPC, его номером будет 255. Далее показано, как будет выглядеть связь нашего системного вызова с номером позиции в include/asm-ppc/ unis td. h;____ NR_ourcall - номер элемента 255 в конце таблицы.

```
include/asm-ppc/unistd.h
/*
* Этот файл содержит номера системных вызовов.
*/
#define   NR restart syscall  0
#define  NR exit   1
#define  _ NR_fork   2
```

```
#define   NR utimes   271
#define _ NR_fadvise64_64   272
#define   NR vserver  273
ttdefine _ NR_ourcall   274
/* #define NR syscalls 274 это старое значение перед нашим системным
 * вызовом */ #define
NR_syscalls 275
```

Следующий подраздел описывает прерывания и вовлеченную в оповещение ядра аппаратную часть, необходимую для их обработки. Исключения группируются по действиям, выполняемым их обработчиками. Несмотря на то что исключения используют во время обработки те же пути, что и прерывания, исключения предпочитают посылать сигналы обратно в текущий процесс, вместо того чтобы работать с драйверами устройств.

3.8.2 Прерывания

Прерывания асинхронны выполнению процессора, это означает, что прерывания происходят между инструкциями. Процессор получает уведомление о прерывании через внешний сигнал через свой контакт (INTR или NMI). Этот сигнал, поступающий от драйвера устройства, называется **контроллером прерывания (interrupt controller).** Прерывания и контроллеры прерываний являются аппаратно- и системно-зависимыми. От архитектуры к архитектуре существует множество различий в дизайне и реализации контроллеров прерываний. Этот подраздел затрагивает основные аппаратные различия и функции и отслеживает код ядра от архитектурно-зависимой до архитектурно-независимой части.

Контроллер прерывания необходим для того, чтобы процессор мог общаться в каждый отдельный момент с несколькими периферийными устройствами. Старые x86-компьютеры использовали каскадную пару контроллера прерывания Intel 8259, настроенную таким образом[1], что процессор мог отличать 15 дискретных линий прерываний (IRQ) (см. рис. 3.16). Когда на контроллер прерываний поступает прерывание (например, при нажатии кнопки), он выделяет ему линию INT, связанную с процессором. Затем процессор распознает сигнал, добавляя распознанную INTA-линию в контроллер прерываний. В этот момент контроллер прерываний передает данные IRQ-процессору. Эта последовательность называется **циклом распознавания прерывания (interrupt-acknowledge cycle).**

Более новые x86-процессоры имеют локальный усовершенствованный программируемый контроллер прерываний Advanced Programmable Interrupt Controller (APIC). Локальный APIC (встроенный в блок процессора) получает сигналы прерываний от следующих источников:

[1] **IRQ первого 8259(обычно IRQ2) связано с выходом второго 8259.**

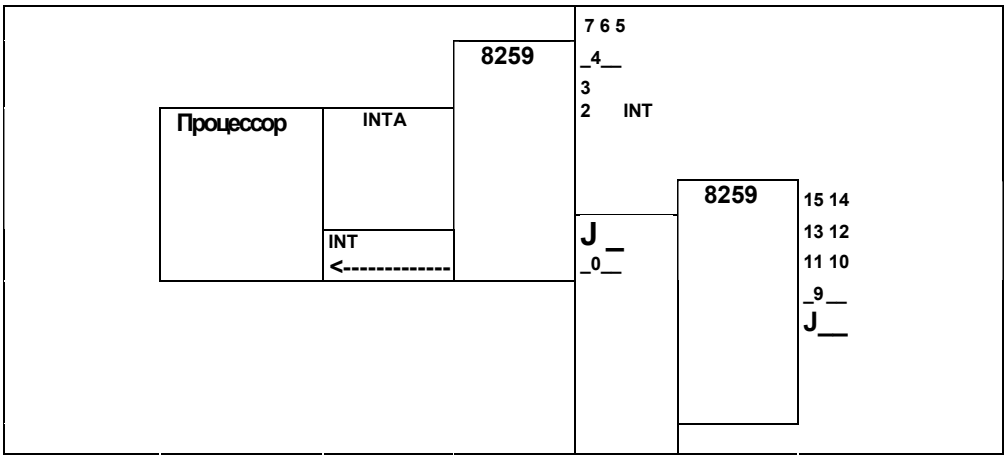


Рис. 3.16. Каскадный контроллер прерывания

- каналов процессорных прерываний (LINTO, LINT1);
- внутреннего таймера;
- внутреннего монитора производительности;
- внутреннего температурного датчика;
- внутренних ошибок APIC;
- другого процессора (межпроцессорные прерывания);
- внешнего APIC ввода-вывода (через APIC-шину на многопроцессорных системах).

После того как APIC получает сигнал прерывания, он передает сигнал ядру процессора (вовнутрь процессора). Ввод-вывод APIC изображен на рис. 3.17 как часть чипсета процессора и предназначен для получения 24 программируемых вводов прерывания.

x86-процессоры с локальным APIC могут быть сконфигурированы с контроллером типа 8259 вместо ввода-вывода через архитектуру APIC (или APIC может быть сконфигурирован в качестве интерфейса для контроллера 8259). Для того чтобы узнать, использует ли система архитектуру APIC, нужно ввести в командную строку следующее:

```
lkp:~# cat  /proc/interrupts
```

Если вы увидите строку I/O-APIC, значит, используется именно этот контроллер. В противном случае вы увидите XT_P1C, что означает использование архитектуры типа 8259.

Контроллер прерываний Power PC для Power Mac G4 и G5 интегрирован в контроллеры Key Largo и K2 I/O. Ввод в командную строку:

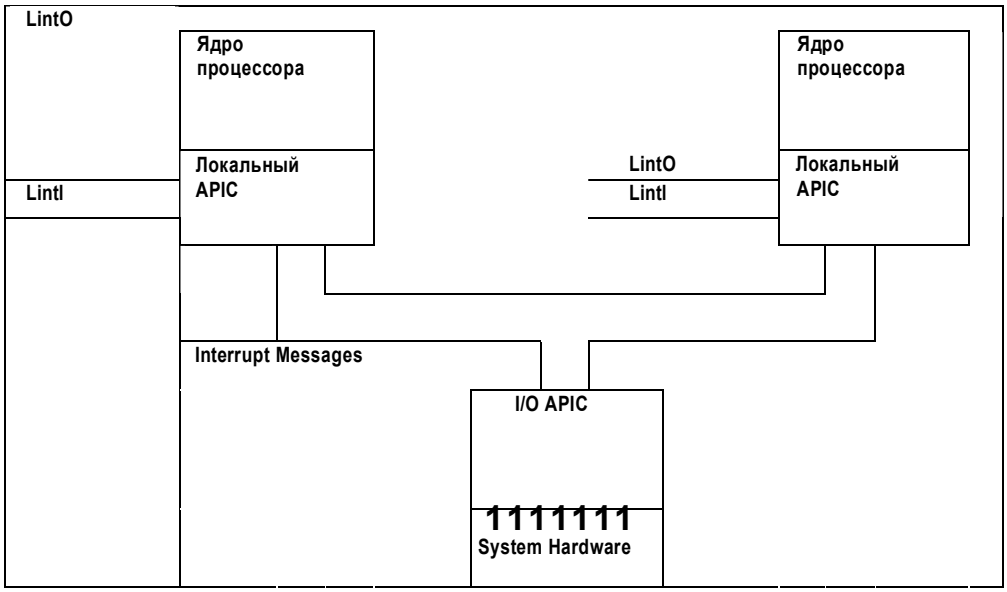


Рис. 3.17. I/O APIC

lkp:~# cat /proc/interrupts

на 04-машине выдаст OpenPIC, т. е. стандарт «Открытый программируемый контроллер прерываний» (Open Programmable Interrupts Controller), разработанный AMD и Cyrix в 1995 г. для многопроцессорных систем. MPIC - это реализация OpenPIC от IBM, которая используется в нескольких моделях их CHRP. Более старые Apple-компьютеры имели встроенный контроллер прерываний и для встроенных процессоров 4xx, ядро контроллера прерываний было интегрировано в AP1C-чип.

Теперь, когда у нас есть представление о том, как и когда прерывания передаются ядром аппаратуре, мы можем проанализировать реальный пример обработки ядром прерывания аппаратного системного таймера и проследить, как это прерывание передается. По мере продвижения через код системного таймера мы увидим, что во время прерывания аппаратно-программный интерфейс реализован как на x86-, так и на PPC-архитектуре с таблицами переходов, которые выбирают соответствующий код обработчика для данного прерывания.

Каждому прерыванию на x86-архитектуре назначается уникальный номер, или вектор. Во время прерывания этот вектор используется для индексирования в таблице **описателей прерываний (Interrupt Descriptor Table, (IDT).** (См. формат x86-описателя в *Intel Programmer's Reference.)* Таблица ШТ позволяет аппаратно помогать программе находить адреса и проверять код обработчика во время прерывания. Архитектура PPC немного

отличается тем, что таблица прерываний создается во время компиляции для выполнения соответствующего обработчика прерывания. (Далее в этом разделе описываются некоторые аспекты программной инициализации и использование таблиц перехода при сравнении обработчика прерывания системного таймера на x86 и PPC. Подразд. 3.8.2.1 описывает обработчики прерываний и их реализацию, подразд. 3.8.2.3 - системный таймер в качестве примера реализации на Linux прерываний и связанных с ними обработчиков.)

3.8.2.1 Обработчики прерываний

Прерывания и обработчики прерываний выглядят как обычные С-функции. Они могут и зачастую выполняют аппаратно-специфические задачи. Обработчики прерываний Linux можно разделить на высокопроизводительную верхнюю половину и низкопроизводительную нижнюю половину.

- **Верхняя половина.** Должны выполняться настолько быстро, насколько это воз можно. Обработчики верхней половины в зависимости от того, как они зарегистрированы, могут выполняться при выключении всех локальных (для дан ного процессора) прерываний (быстрый обработчик). Код в обработчиках верхней половины необходимо ограничивать для выполнения только аппаратно-критических или критических по времени задач. Слишком долгое нахождение в обработчике верхней половины может значительно сказаться на производительности системы. Для того чтобы производительность оставалась высокой и латентность (время, в течение которого задача занимает устройство) низкой, служит архитектура нижней половины.
- **Нижняя половина.** Позволяет записывающему обработчику отложить наименее критическую работу до момента, когда у процессора появится свободное время[1]. Помните, что прерывания поступают асинхронно выполнению системы; ядро в этот момент может заниматься чем-то более критическим. В архитектуре нижней поло вины записывающий обработчик может позволить ядру выполнить наименее критический код обработчика немного позднее.

Табл. 3.8 иллюстрирует четыре наиболее используемых метода обработки прерываний нижней половины.

В ранних версиях Linux использовались обработчики системного таймера верхней половины/нижней половины. Потом они были переписаны и стали употреблять только верхнюю половину.

Таблица 3.8. Методы обработки прерываний нижней половины

«Старые» нижние разделители	Эти возникшие до SMP обработчики постепенно были вытеснены, потому что независимо от количества процессоров в каждый момент может работать только одна нижняя половина. Эта система была удалена из ядра 2.6 и упоминается здесь только для справки
Рабочая очередь	Код верхней половины запускается в так называемом контексте прерываний (interrupt context), который не ассоциируется с процессом. Без ассоциации с процессом код не может заснуть или быть блокированным. Рабочая очередь запускает контекст процесса (process context) и получает все способности любого потока ядра. Рабочая очередь обладает широким набором функций для создания, планирования^тмены и т.д. Более подробную информацию о рабочих очередях можно найти в разделе «Рабочие очереди и прерывания» в гл. 10
Программные прерывания (softirqs)	Программные прерывания выполняются в контексте прерывания и похожи на нижнюю половину за исключением того, что программные прерывания одного типа могут выполняться на многопроцессорной системе одновременно. В системе доступно только 32 программных прерывания. Системный таймер использует программное прерывание
Тасклеты (tasklet)	Похожи на программные прерывания за исключением отсутствия ограничений. Все тасклеты проходят через одно программное прерывание, и один тасклет не может одновременно выполняться на нескольких процессорах. Интерфейс тасклетов проще для использования и реализации по сравнению с программными прерываниями

3.8.2.2 Структуры IRQ

Вся связанная с IRQ информация хранится в трех структурах: irq_desc_t, irqac-tion и hw_interrupt_type (рис. 3.18).

Структура irqjiescjt

Структура ircj_desc_t - это основной описатель IRQ. Структура ircr_desc_t хранит глобальный массив доступа размера NR__TRQS (с архитектурно-зависимым значением) с названием irq_desc.

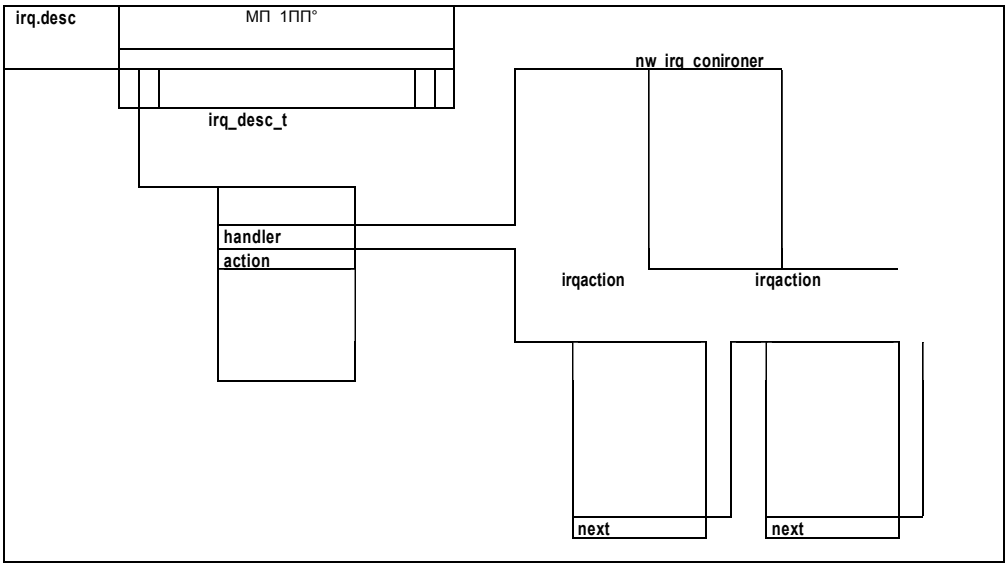


Рис. 3.18. Структуры IRQ

```
include/linux/irq.h
60  typedef struct irq desc {
61   unsigned int status;   /* статус IRQ */
62   hw irq controller *handler;
63   struct irqaction *action; /* список действий IRQ */
64   unsigned int depth;  /* вложенные отключения irq */
65   unsigned int irq count; /*Для обнаружения поврежденных прерываний*/
66   unsigned int irqs unhandled;
67   spinlock t lock;
68   ) __ cacheline_aligned irq_desc_t;
69
70   extern irq_desc_t irq_desc [NR_IRQS];
```

Строка 61

Значение поля status определяется установкой флага, описывающего статус линии IRQ. Флаги показаны в табл. 3.9.

Таблица 3.9. Флаги irq_desc_t->field

Флаг	Описание
IRQ_INPROGRESS	Указывает, что мы, в процессе выполнения обработчика для линии IRQ
IRQ_DISABLED	Указывает, что IRQ аппаратно отключается таким образом, что обработчик не выполняется, даже если он включен для аппаратной линии
ORQ_PENDING	Среднее состояние, обозначающее, что прерывание получено, но обработчик не выполняется
IRQ_REPLAY	Предыдущее IRQ не было получено
IRQ_AUTODETECT	Состояние линии IRQ устанавливается после проверки
IRQ_WAITING	Используется при проверке
IRQ_LEVEL	Срабатывает уровень IRQ вместо контура
IRQ_MASKED	В коде ядра этот флаг не используется
IRQ_PER_CPU	Используется для обозначения того, что линия IRQ является локальной для процессорного вызова

***Строка** 62*

Поле handler указывает на hw_irq_controller; hw_ircr_controller определен для структуры hw_interrupt_type, описатель контроллера прерываний которого используется для низкоуровневого аппаратного описания.

Строка 63

Поле action хранит указатель на структуру irqaction. Эта структура, более подробно описываемая ниже, отслеживает действия, необходимые для обработки прерывания при включении IRQ.

***Строка** 64*

Поле depth - это счетчик отключений вложенных IRQ. Флаг IRQ DISABLE очищается, только когда значение этого поля равно 0.

Строки 65-66

Поле irq__count вместе с полем irqs_unhandled обозначает IRQ, которые могут застрять. Они используются на x86 и PPC в функциях note_interrupt () (arch/<arch>/kernel/irq.c).

Строка 67

Поле lock хранит циклическую блокировку описателя.

Структура irqaction

Ядро использует структуру irqaction для отслеживания обработчиков прерываний и связывания их с IRQ. Давайте посмотрим на структуру и поля, которые мы будем рассматривать в следующих подразделах.

```
include/linux/interrupt.h
35 struct irqaction {
36  irqreturn t (*handler) (int, void *, struct pt regs *);
31  unsigned long flags;
38  unsigned long mask;
3 9   const char *name;
40   void *dev id;
41   struct irqaction *next;
42  };
```

Строка 36

Поле handler - это указатель на обработчик прерывания, вызываемый при возникновении прерывания.

Строка 37

Поле flags может хранить флаги наподобие SA_INTERRUPT, обозначающего, что обработчик прерывания будет работать при всех выключенных прерываниях, или SA_SHIRQ, обозначающего, что обработчик прерывания может разделять линию IRQ с другими обработчиками.

Строка 39

Поле name хранит имя зарегистрированного обработчика.

Структура hwjinterruptjtype

Структура hw__interrupt_type или hw_irq__controller содержит всю информацию, связанную с системным контроллером прерываний. Сначала мы посмотрим на структуру, а затем посмотрим, как она реализована для нескольких контроллеров прерываний.

```
include/linux/irq.h
40  struct hw interrupt type {
41   const char * typename;
42   unsigned int (*startup)(unsigned int irq);
43   void (*shutdown)(unsigned int irq);
44   void (*enable)(unsigned int irq);
45   void (*disable)(unsigned int irq);
46   void (*ack)(unsigned int irq);
```

```
47    void (*end)(unsigned int irq);
48    void (*set_affinity)(unsigned int irq, cpumask_t dest);
49   };
```

Строка 41

typename хранит имя программируемого контроллера прерывания (Programmable Interrupt Controller, PIC). (Далее PIC описаны подробнее.)

Строки 42-48

Эти поля хранят указатель на PIC-специфические функции программирования.

Теперь давайте посмотрим на наш PPC-контролер. В этом случае мы рассмотрим PIC для PowerMac.

```
arch/ppc/platforms/pmac pic.c
170   struct hw  interrupt  type pmac pic = {
171    * PMAC-PIC *,
172    NULL,
173    NULL,
174    pmac unmask irq/
175    pmac mask irq/
176    pma c_ma s k_and_ac k_i rq,
177    pmac end irq,
178    NULL
179   };
```

Как вы можете видеть, имя этого PIC - PMAC-PIC и для него определено 4 из 6 функций. Функции pmac_unamsk_irq и pmac_mask_irq включает и отключает линию IRQ соответственно. Функция pmac_mask__and_ack_irq проверяет получение IRQ, а pmac_ end_irq занимается очисткой, когда выполнение обработчика прерывания завершается.

```
arch/i386/kernel/i8259.c
59   static  struct hw_interrupt_type  i82 59A_irq_type              = {
60     "XT-PIC",
61    startup 8259A irq/
62    shutdown 82 59A irq/
63    enable 82 59A irq/
64    disable 8259A irq,
65    mask and ack 82 59A,
66    end 8259A irq,
67    NULL
68   };
```

PIC для x86 8259 называется XT-PIC и определяет первые 5 функций. Первые две, startup_8259A_irq и shutdown_8259A__irq, начинают и завершают реальную линию IRQ соответственно.

3.8.2.3 Пример прерывания: системный таймер

Системный таймер представляет собой пульс операционной системы. Системный таймер и прерывания выставляются во время инициализации системы во время загрузки. Инициализация прерывания в это время использует интерфейс, отличный от того, который используется для прерываний, зарегистрированных во время выполнения. Мы рассмотрим эти различия на нашем примере.

После того как стали выпускаться чипы с большим набором возможностей, дизайн ядра стал предусматривать несколько вариантов исходников системного таймера. Наиболее распространенная реализация таймера для архитектуры x86 - это **программируемый интервальный таймер (Programmable Interval Timer, (PIT),** а для PowerPC - это **декрементор (decrementer).**

Архитектура x86 исторически реализует PIT с помощью таймера Intel 8254; 8524 использует 16-битовый счетчик вниз, прерывающийся при отсчете терминала. Поэтому значения, записываемые 8254 в регистр, уменьшаются до тех пор, пока не достигнут 0. В этот момент активизируется прерывание для ввода IRQ 0 на контроллере прерываний 8259, ранее упоминаемом в этом разделе.

Системный таймер, реализованный на PowerPC-архитектуре, представляет собой декрементирующиеся часы в виде 32-битового счетчика вниз, работающего с частотой процессора. Аналогично 8259, он активизирует прерывания при каждом отсчете терминала. В отличие от Intel-архитектуры декрементор встроен в процессор.

Каждый раз при отсчете таймера он активизирует прерывание, известное как **тик (tick).** Частота этих тиков устанавливается с помощью переменной HZ.

Теперь мы начинаем подбираться к коду инициализации системного таймера и связанных с ним прерываний. Обработчик системного таймера вставляется ближе к концу инициализации ядра; мы возьмем участок кода в start_kernel () - основной функции инициализации, выполняемой при загрузке, в которой сначала выполняется вызов trap_init (), затем init_IRQ () и, наконец, time_init ():

```
init/main.c
3 86  asmlinkage void  _init start_kemel (void)
387  {

413  trap_init();

415  init_IRQ();
```

HZ

HZ - это аббревиатура от Herzh (Hz), имени Генриха Герца (1857-1894). Один из первооткрывателей радиоволн, Герц сумел подтвердить теорию Максвелла об электричестве и магнетизме, получив искру в катушке проволоки. Маркони продолжил эти эксперименты, и в результате было изобретено радио. В честь этого человека и его вклада в науку элементарная единица частоты была названа его именем; один цикл в секунду равен одному герцу.

HZ определена в include/asm-xxx/param.h. Давайте рассмотрим эти значения для x86 и PPC.

```
include/asm-i38б/param.h
5    #ifdef   KERNEL
6    #define HZ     1000   /* внутренняя частота таймера ядра */
```

```
include/asm-ppc/param.h
8    #ifdef   KERNEL
9    #define HZ          100      /*  внутренняя частота  таймера ядра  */
```

Значение HZ обычно равно 100 для большинства архитектур, однако по мере роста производительности машин количество тиков постепенно увеличивается. Рассматривая в этой книге две основные архитектуры, мы увидим (ниже), что тик по умолчанию для обеих архитектур равен 1000. Период одного тика равен 1/HZ. Таким образом, период (время между прерываниями) равен 1 мс. Мы можем заметить, что как только значение HZ увеличивается, мы получаем большее количество прерываний за то же время. Несмотря на то что для времясберегающих функций это лучше, следует помнить, что при этом больше процессорного времени будет тратиться на ответ ядра прерываниям системного таймера. При экстремальных значениях это может замедлить реакцию пользовательских программ. Как и для любых других прерываний, здесь важно подобрать правильный баланс.

```
419  time_init(); }
```

Строка 413

Макрос trap_init () инициализирует элементы исключения в таблице описателей прерываний (Interrupt Descriptor Table, ШТ) для архитектуры x86, выполняемых в защищенном режиме. IDT - это таблица, записываемая в память. Адрес ГОТ устанавливает регистры IDTR-процессора. Каждый элемент таблицы описателей прерываний — это один из трех вентилей. Вентиль - это адрес защищенного режима x86, состоящий из селектора, отступа и уровня привилегии. Вентиль предназначен для передачи программного управления. Три типа вентилей в IDT включают сие-

темные (system), когда управление передается другой задаче; **прерывания (interrupt),** когда управление передается в обработчик прерывания с отключенными прерываниями, и **ловушки (trap),** когда управление передается в обработчик прерывания без изменения прерываний.

Архитектура PPC выполняет переход по определенному адресу, в зависимости от исключения. Функция trap_init () не является **операцией (no-op)** для PPC. Позднее в этом подразделе, когда мы будем рассматривать код системного таймера, мы увидим различия между таблицей прерываний PPC и таблицей описателей прерываний x86.

```
arch/i386/kernel/traps.c
900  void __ init trap_init(void)
901  {
902  #ifdef CONFIG EISA
903   if (isa readl(0x0FFFD9) == *E' +(4'«8) + (S'«16) + (A'«24) ) {
904    EISA_bus = 1;
905   }
906  #endif
907
908  #ifdef CONFIG X86 LOCAL APIC
909  init apic_mappings();
910  #endif
911
912  set trap gate(0,&divide error);
913  set intr gate(l,&debug) ;
914  set intr gate(2,&nmi);
915  set system  gate(3, &int3) ;  /* int3-5 можно вызывать отовсюду */
916  set system gate(4,&overflow);
917  set system  gate (5, &bounds) ;
918  set trap gate(б,&invalid  op);
919  set_trap_gate(7,&device_not_available);
920  set task gate(8,GDT ENTRY DOUBLEFAULT TSS);
921  set trap gate(9,&coprocessor segment  overrun);
922  set_trap_gate(10/&invalid_TSS);
923  set trap gate(ll/&segment not present);
924  set trap gate(12/&stack segment);
925  set trap gate(13,&general protection);
926  set intr gate(14,&page fault);
927  set trap gate(15/&spurious interrupt  bug);
928  set_trap_gate(16/&coprocessor_error);
92 9    set  trap gate (17, &alignment check) ;
93 0           #ifdef CONFIG X86 MCE
931  set_trap_gate(18/&machine_check);
932 #endif
```

```
933                 set_trap_gate(19,&simd__coprocessor_error);
934
935             set_systeitL_gate (SYSCALL_VECTOR,&system_call) ;
936
937   /*
938    * LDT по умолчанию - это одинарный вентиль вызова 1ca117 из iBCS
939    * и вентиль вызова 1ca1127 для бинарных файлов  Solaris/x86
940    */
941   set_call_gate(&default_ldt[0],103117) ;
942   set_call_gate(&default_ldt[4],1031127);
943
944   /*
945    * Служит барьером для любых внешних состояний процессора
946   */
947   cpu_init(); 948
949   trap_init_hook();
950  }
```

Строка 902

Поиск сигнатуры EISA; eisa_read() - это вспомогательная функция, позволяющая читать шину EISA, отображая ввод-вывод в память с помощью ioremap ().

Строки 908-910

Если усовершенствованный программируемый контроллер прерываний (APIC) существует, его адрес добавляется в фиксированную карту системных адресов. См. «специальные» функции для системных адресов в include/asm-i3 86/ fixmap.h и set_f ixmap_nocache (); init_apic_niappings () использует эти функции для получения физических адресов APIC.

Строки 912-935

Инициализация ШТ с вентилями ловушек, системными вентилями и вентилями прерываний.

Строки 941-942

Эти специальные межсегментные вызовы вентилей обеспечивают поддержку Бинарного стандарта совместимости Intel (Intel Binary Compatibility Standard) для запуска других UNIX-бинарных файлов на Linux.

Строка 947

Для текущего выполняющегося процессора инициализируется его таблица и регистры.

Строка 949

Используется для инициализации системно-специфического оборудования, такого, как разные типы APIC. На платформе x86 это не операция.

Строка 415

Вызов init_IRQ инициализирует аппаратный контроллер прерывания. Обе архитектуры, x86 и PPC, имеют несколько реализаций устройств. Для архитектуры x86 мы рассмотрим устройство i8259. Для PPC мы рассмотрим код, связанный с Power Mac.

На PPC реализация init_IRQ () находится в arch/ppc/kernel/irq. c. В зависимости от конкретной аппаратной конфигурации init_IRQ() вызывает несколько вспомогательных функций для инициализации PIC. Для конфигурации Power MAC функция pmac_pic_init () из arch/ppc/platforms/pmac_pic .c вызывается для контроллеров ввода-вывода G3, G4 и G5. Эти аппаратно-зависимые функции пытаются опознать тип контроллера ввода-вывода и выполняют соответствующие настройки. В этом примере PIC является частью устройства контроллера вода-вывода. Процесс инициализации прерывания похож на x86 с тем небольшим различием, что запуск таймера осуществляется не с помощью PPC версии init_IRQ(), а с помощью функции time_init (), описываемой далее в этом подразделе.

На платформе x86 существует несколько вариаций PIC. Как было сказано ранее, более старые системы используют каскадный 8259, а более современные - архитектуру 10-APIC. Этот код показывает PIC с эмуляцией контроллера типа 8259.

```
arch/i386/kernel/i8259.c
342  void    init init ISA irqs (void)
343  {
344   int i;
345  #ifdef CONFIG X86 LOCAL APIC
346   ini t bsp  APIC () ;
347  #endif
348   init_8259A(0);

3 51      for (i = 0; i < NR IRQS; i++) {
352     irq desc[i].status = IRQ DISABLED;
353     irq desc[i].action = 0; 3
54      irq desc[i].depth = 1;
355
356     if (i < 16) {
357      /*
3 58 * 16 INTA-циклов прерываний в старом стиле:
359       */
360      irq desc[i].handler = &i8259A irq type;
361     } else {
```

```
362     /*
3 63      * xhigh' PCI IRQs заполняемые по запросу
364       */
365     irq_desc[i].handler = &no_irq_type;
366    }
367   }
368  }

409
410  void   init init IRQ(void)
411  {
412   int i;
413
414   /* настройки перед инициализацией вентиля вызова */
415   pre_intr_init_hook();

422   for (i = 0; i < NR IRQS; i++) {
423    int vector = FIRST_EXTERNAL_VECTOR + i;
424    if (vector != SYSCALL VECTOR)
425     set intr gate(vector, interrupt[i]);
426   }

431  intr  init hook () ;
437  setup_timer () ;

}
```

Строка 410

Это точка входа в функцию, вызываемая из start_Jkernel (), которая является основной функцией инициализации ядра при запуске системы.

Строки 342-348

Если доступен и предпочитается локальный APIC, он инициализируется и виртуально связывается с каналом 8259. После этого инициализируется устройство 8259, с помощью регистров ввода-вывода в init_8259A(0).

Строки 422-426

В строке 424 системные вызовы не включаются в этот цикл, потому что они уже проинсталлированы ранее в trap_init (). Linux использует вентиль прерываний Intel (код, инициированный ядром) как описатель прерываний. Это делается с помощью макроса set_intr_gate () (в строке 425). Исключения используют системный вентиль Intel и ловушку вентиля, устанавливаемую set_system_gate() и set_trap_gate() соответственно. Эти макросы можно найти в arch/i386/kernel/traps.c.

Строка 431

Устанавливает обработчики прерываний для локального APIC (если он используется) и вызывает setup_irq () из irq. с для каскадного 8259.

Строка 437

Запуск 8253 PIT с помощью регистров ввода-вывода.

Строка 419

Теперь мы проследим time_init () для установки обработчика прерываний системного таймера как для PPC, так и для x86. В PPC системный таймер (сокращенное для простоты название) инициализирует декрементор:

```
arch/ppc/kernel/time.c
void _ init time_init (void)
{

317  ppc_md.calibrate_decr();

351   set_dec(tb_ticks_per_jiffy);

}
```

Строка 317

Подбор значения для системной переменной HZ.

Строка 351

Установка декрементора в начальное состояние.

Архитектура PowerPC и реализация Linux не требуют инсталляции прерывания таймера. Вектор декрементора прерывания устанавливается в 0x900. Вызов обработчика жестко закодирован на эту позицию и не изменяется:

```
arch/ppc/kernel/head.S
/* декрементор */
479   EXCEPTION(0x900,  Decremented  timer_interrupt,  EXC_XFER_LITE)
```

Макрос EXCEPTION для работы с декрементором более подробно описан далее в этом подразделе. Обработчик декрементора теперь готов к выполнению по изменению счетчика.

Следующий фрагмент кода показывает инициализацию системного таймера для x86.

```
arch/i386/kernel/time.c
void __init time_init(void)
{

340   time init hook();
}
```

Функция time_init() обращается к time_init_hook(), находящейся в машинно-специфичном файле setup. c:

```
arch/i386/machine-default/setup.c
072    static struct irqaction irqO = { timer_interrupt, SA_INTERRUPT, 0,
                                                          "timer", NULL, NULL};
81     void ___init time_init_hook(void)
82     {
083      setup_irq(0, &irq0);
084    }
```

Строка 72

Мы инициализируем структуру irqaction таким образом, чтобы она соответствовала irqO.

Строки 81-84

Вызов функции setup_irq(0, &irq0) помещает структуру irqaction, содержащую обработчик timer_interrupt (), в очередь разделяемых прерываний, связанных с irqO.

Этот фрагмент кода действует подобно вызову request_irq() для обработчика общего случая (не загружаемого во время инициализации ядра). Код инициализации для прерывания времени помещает ярлык обработчика в irq_sedc [ ]. Код времени выполнения использует disable_irq() , enable_irq() , request_irq() и f ree_irq() в файле irq.c. Все эти функции позволяют работать с IRQ и использовать структуру irci_desc.

Время прерывания

Для PowerPC декрементор находится внутри процессора и имеет собственный вектор прерывания 0x900. Архитектура x86 отличается тем, что PIT является **внешним прерыванием (external interrupt),** идущим от контроллера прерывания. Внешний контроллер PowerPC использует вектор 0x500. Похожая ситуация происходит и в x86, если системный таймер работает через локальный APIC.

Табл. ЗЛО и 3.11 описывают таблицу векторов прерываний для x86 и PPC архитектур соответственно.

Таблица 3.10. Таблица векторов прерываний x8б

Номер вектора/IRQ	Описание
0	Ошибка деления
1	Отладочное исключение
2	Прерывание NMI
3	Точка останова
4	INTO-обнаруженное переполнение
5	Пределы BOUND превышены
6	Неверный код операции
7	Устройство недоступно
8	Двойная ошибка (два одновременных прерывания)
9	Переполнение сегмента сопроцессора (зарезервировано)
10	Неверное состояние сегмента задачи
11	Сегмент отсутствует
12	Ошибка стека
13	Общее нарушение защиты
14	Ошибка страницы
15	(Зарезервировано Intel. Не используется)
16	Ошибка плавающей точки
17	Проверка размещения
18	Проверка машины
19-31	(Зарезервировано Intel. Не используется)
32-255	Маскируемые прерывания

Таблица 3.11. Отступы PPC для векторов прерываний

Отступ (шестнадцатеричный)	Тип прерывания
00000	Зарезервировано
00100	Системный сброс
00200	Проверка машины
00300	Сохранение данных
00400	Сохранение инструкций
00500	Внешнее
00600	Размещение
00700	Программа
00800	Плавающая точка недоступна
00900	Декрементор
00A00	Зарезервировано
00B00	Зарезервировано
00C00	Системный вызов
00D00	Трассировка
00E00	Поддержка плавающей точки
00E10	Зарезервировано
00FFF	Зарезервировано
01000	Зарезервировано, зависит от реализации
02FFF	(Конец размещения вектора прерываний)

Обратите внимание на общие черты двух архитектур. Эти таблицы представляют оборудование. Программным интерфейсом для таблицы векторов исключений прерываний Intel является таблица описателей прерываний (IDT), ранее упоминавшаяся в этой главе.

Теперь мы можем видеть, как архитектура Intel обрабатывает аппаратные прерывания с помощью IRQ, таблица переходов в entry. S, для вызова вентилей (описателей) и, наконец, кода обработчиков. Это продемонстрировано на рис. 3.19.

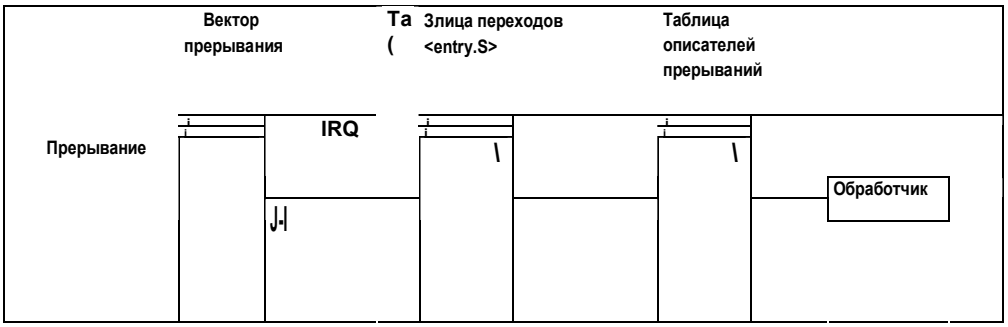


Рис. 3.19. Путь прерываний нах86

С другой стороны, PowerPC указывает на определенный отступ в памяти, где находится переход на соответствующий обработчик. Как мы увидим в дальнейшем, таблица переходов PPC в head.S индексируется с помощью фиксированного расположения в памяти. Это можно увидеть на рис. 3.20.

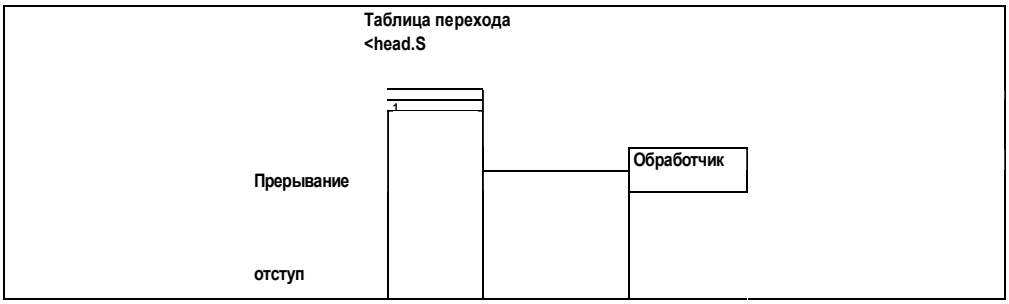


Рис. 3.20. Поток прерываний на PPC

После того как мы рассмотрим внешнее прерывание PPC(смещение 0x500) и обработчик прерывания таймера PPC (смещение 0x900), вам станут понятны многие неясные моменты.

Обработка внешнего вектора прерываний PowerPC

Как было описано ранее, процессор переходит по адресу 0x500 в случае внешнего прерывания. Перед тем как продолжить рассмотрение макроса EXCEPTION () в файле head. S, мы рассмотрим следующие строки кода, которые связываются, загружаются

и отображаются в память таким образом, чтобы они находились по отступу 0x500. Эта аппаратная таблица переходов работает аналогично ГОТ на x86.

```
arch/ppc/kernel/head.S
453    /* External interrupt */
454   EXCEPTION(0x500, Hardwarelnterrupt, do_IRQ, EXC_XFER_LITE)
```

Далее вызывается третий параметр, **do_IRQ().** Давайте рассмотрим соответствующую функцию.

```
arch/ppc/kernel/irq.c
510   void do_IRQ(struct pt_regs *regs)
511   {
512   int irq, first = 1;
513   irq_enter();

523   while ((irq = ppc md.get irq(regs)) >= 0) {
524    ppc_irq_dispatch_handler(regs, irq);
525    first = 0;
526   }
527   if (irq != -2 && first)
528   /* С точки зрения SMP это небезопасно ... но кого это волнует ? */
529  ppc spurious interrupts++;
53 0 irq exit () ;
531   }
```

Строки 513-530

Сообщают коду приоритетного обслуживания прерываний, что мы находимся внутри аппаратного прерывания.

Строка 523

Чтение из контроллера прерываний незаконченного прерывания и преобразование его в номер **IRQ** (до тех пор, пока не будут обработаны все прерывания).

Строка 524

ppc_ircx_dispatch_handler () обрабатывает прерывание. (Мы рассмотрим эту функцию позже.)

Функция ppc_irq__dispatch_handler () практически идентична функции do_IRQ () для x86:

```
arch/ppc/kernel/irq.c
```

```
428  void ppc__irq_dispatch_handler(struct pt_regs *regs, int  irg)
429  {
43 0                        int status;
431  struct irqaction *action;
432  irq_desc__t *desc = irq_desc + irq; 433
434  kstat this cpu.irqs[irq]++;
435  spin lock(&desc->lock); 43
6    ack_irq(irq);

441  status = desc->status & ~(IRQ_REPLAY | IRQ_WAITING);
442  if (!(status & IRQ PER CPU))
443   status |= IRQ_PENDING; /* мы хотим это обработать */

449   action =  NULL;
450   if (likely(! (status & (IRQ  DISABLED | IRQ INPROGRESS) ) ) ) {
451     action = desc->action;
452    if (!action || !action->handler) {
453     ppc spurious interrupts++;
454     printk(KERN JDEBUG "Unhandled interrupt %x, disabled\n",irq) ;
455      /* Мы не будем вызывать здесь disable irq, потому что иначе
            попадем в тупик */
456      ++desc->depth;
457      desc->status |= IRQ DISABLED;
458      mask irq(irq);
459      /* Это настоящее прерывание, находящееся в eoi,
460         к которому мы хотим перейти */
461      goto out;
462      }
463      status &= ~IRQ PENDING; /* поручение обработки*/
464      if (!(status & IRQ PER CPU))
465      status |= IRQ_INPROGRESS; /* мы выполняем обработку */
466    }
567                   desc->status = status;

489   for (;;) {
490    spin_unlock(&desc->lock);
491    handle irq event(irq, regs, action);
492    spin_lock(&desc->lock); 493
494    if (likely(!(desc->status & IRQ PENDING)))
495     break;
496    desc->status &= ~IRQ PENDING;
497   }
498  out:
499  desc->status &= -IRQ_INPROGRESS;
```

```
511 }
```

Строка 432

Получает IRQ из параметров и получает доступ к соответствующей irq_desc.

Строка 435

Получает циклическую блокировку описателя IRQ в случае конкурентного доступа к тому же прерыванию другим процессором.

Строка 436

Посылает подтверждение оборудованию. Оборудование реагирует соответствующим образом, предотвращая дальнейшую обработку прерываний этого типа до тех пор, пока обработка данного не будет завершена.

Строки 441-443

Очистка флагов IRQ REPLAY и IRQ WAITING. В этом случае IRQ_REPLAY демонстрирует, что IRQ сбрасывается ранее и перепосылается. IRQ WAITING обозначает, что IRQ протестировано. (Оба случая выходят за рамки данного обсуждения.) В однопроцессорных системах устанавливается флаг IRQ_PENDING, который показывает, что мы выполняем обработку прерывания.

Строка 450

Этот блок кода проверяет состояние, при котором мы не будем обрабатывать прерывание. Если установлены IRQ DISABLED или IRQ_INPROGRESS, мы можем пропустить этот блок кода. Флаг IRQ_DISABLED устанавливается, когда мы не хотим, чтобы система отвечала на определенную линию IRQ. Указывает, что прерывание будет обрабатываться процессором. Этот флаг используется в том случае, когда второй процессор многопроцессорной системы пытается обработать то же самое прерывание.

Строки 451-462

Здесь мы проверяем существование обработчика. Если нет, мы делает перерыв и переходим на метку «out» в строке 498.

Строки 463-465

В этой точке мы очищаем все три состояния для необслуживаемых прерываний, как было поручено. Устанавливается флаг IRQ_INPROGRESS, а флаг IRQ PENDING снимается, это означает, что прерывание обработано.

Строки 489-497

Здесь производится обслуживание прерываний. Перед обслуживанием прерываний снимается циклическая блокировка с описателя прерывания. После того как бло-

кировка снята, вызывается вспомогательная функция handle_irq_event (). Эта функция выполняет обработчик прерывания. По завершении снова устанавливается блокировка описателя. Если флаг IRQ PENDING не установлен (другим процессором) во время обработки IRQ, цикл прерывается. В противном случае служба опять прерывается.

Обработка прерывания системного таймера PowerPC

Как указано в timer_init (), декрементор жестко привязан к 0x900. Мы можем считать, что счетчик терминала достигнут и вызван обработчик timer_interrupt () в arch/kernel/time.c:

```
arch/ppc/kernel/head.S
  /* Декрементор */ 479  EXCEPTION(0x900, Decremented
timer_interrupt, EXC_XFER_LITE)
```

Вот функция timer_interrupt ():

```
arch/ppc/kernel/time.c
145  void timer interrupt(struct pt regs * regs)
146  {

152  if (atomic read(&ppc_n_lost_.interrupts) != 0)
153   do__IRQ(regs) ;
154
155 irq_enter();

159   if (!user_mode(regs))
160    ppc_do_profile(instruction_pointer(regs));

165 write_seqlock(&xtime_lock);
166
167  do_timer (regs) ;

189 if (ppc_md.set__rtc_time(xtime. tv_sec+l + time_offset) == 0)

195  write_sequnlock(&xtime_lock);

198   set_dec(next_dec);

208   irq_exit();
209  }
```

Строка 152

Если прерывание было потеряно, возвращаемся и вызываем внешний обработчик в 0x900.

Строка 159

Выполняется профилирование для отладки работы ядра.

Строки 165 и 195

Эти участки кода блокируются.

Строка 167

Этот код аналогичен используемому для прерывания таймера на x86.

Строка 189

Обновление RTC.

Строка 198

Перезапуск декрементора для следующего прерывания.

Строка 208

Возвращение из прерывания.

Код прерывания теперь запущен в нормальном режиме до следующего прерывания.

Обработка прерывания системного таймера на x86

Перед активацией прерывания (в нашем примере PIT отсчитывается вниз до нуля и активизирует IRQ0) контроллер прерывания активизирует линию прерывания, проходящую через процессор. Код ассемблера в entry. S имеет точку входа, соответствующую каждому описателю **в** ЮТ. IRQ0 является первым внешним прерыванием и вектором 32 **в** IDT. Теперь код готов для перехода в точку входа 32 в таблице перехода в entry. S:

```
arch/i386/kernel/entry.S
385 vector=0
3 86  ENTRY(irq entries start) 3
87  .rept NR IRQS
388    ALIGN
389   1: pushl $vector-256
3 90  jmp common interrupt
3 91  .data
3 92  .long lb
3 93  .text
3 94  vector=vector+l
3 95  .endr
396
3 97  ALIGN
3 98  common_interrupt:
```

```
3 99  SAVE_ALL
400   call  do_IRQ
401   jmp  ret_from_JLntr
```

Этот код представляет собой отличную демонстрацию магии ассемблера. Конструкция повтора . rept (в строке 387) и его завершающая инструкция (в строке 395) создают таблицу переходов прерывания во время компиляции. Обратите внимание, что в этом циклически созданном блоке кода номер вектора, находящийся в строке 389, декрементируется. Помещая в стек вектор, код ядра теперь знает, с каким IRQ он работает во время прерывания.

Когда мы покидаем отслеживаемый код для x86, код переходит в соответствующую точку входа в таблице переходов и сохраняет IRQ в стек. Затем код переходит в общий обработчик в строке 398 и вызывает do_IRQ () (arch/ i3 8 б /kernel /irq. c) в строке 400. Эта функция практически идентична ppc___irq_dispatch_handler (), описаной в подразделе «Обработка внешнего вектора прерываний PowerPC», поэтому мы не будем описывать ее еще раз.

На основе поступающего IRQ функция do_irq () получает доступ к соответствующему элементу irq_desc и переходит к каждому обработчику в цепочке структур действий. Здесь мы наконец выполняем сам вызов функции обработчика для PIT: timer__interrupt (). Посмотрите на следующий фрагмент кода из time.c. Придерживаясь того же порядка, что и в файле, начнем со строки 274.

arch/i386/kernel/time.c

```
274  irqreturn_t timer_interrupt (int irq,  void *dev__id,   struct pt__regs *regs)
275   {

287    do_timer_interrupt(irq, NULL, regs);

290    return  IRQ_HANDLED;
291   }
```

Строка 274

Это точка входа в системный обработчик прерывания таймера.

Строка 287

Это вызов do_timer__interrupt ().

```
arch/i386/kernel/time.c
208  static inline void do timer interrupt(int irq, void *dev id,
209       struct pt regs *regs)
210  {
```

```
227    do_timer_interrupt_hook(regs); 250
}
```

Строка 227

Вызов do_timer_interrupt_hook(). Эта функция представляет собой обертку для вызова do_timer (). Давайте на нее посмотрим.

```
include/asm-i38 б/mach-default/do_timer.h
16     static  inline void do_timer_interrupt_hook(struct pt_regs     *regs)
17     {
018     do_timer(regs);
025   x86_do_profile(regs);
030   }
```

Строка 18

Здесь выполняется вызов do__timer (). Эта функция выполняет основную часть работы по обновлению системного таймера.

Строка 25

Функция x86_do_profile() проверяет регистр iep на наличие кода, который возвращается перед прерыванием. В остальное время эти данные демонстрируют, как часто выполняется процесс.

В этом месте прерывание системного таймера возвращается из do_irq () в entry . S для выполнения уборки и продолжения потока прерывания.

Как было описано ранее, системный таймер является пульсом операционной системы Linux. В качестве примера мы рассматривали в этом подразделе прерывание таймера, остальные прерывания операционной системы обрабатываются аналогично.

Резюме

Процессы разделяют процессор с другими процессами и определяют индивидуальные контексты исполнения, хранящие информацию, необходимую для выполнения процесса. За время их выполнения они проходят через множество состояний, которые можно абстрагировать для состояния блокировки, выполнения и готовности к выполнению.

Ядро хранит информацию, соответствующую описателю task_struct. Поля task_struct могут разделяться между различными функциями, взаимодействующими с процессом, включая атрибуты процесса, взаимоотношения процесса, доступ процесса

к к памяти, связанные с процессом файловые манипуляции, уведомления, ресурсные ограничения и планирование. Все эти поля необходимы для слежения за контекстом процесса. Процесс может включать один или несколько потоков, разделяющих адресное пространство. Каждый процесс обладает собственной структурой.

Создание процесса начинается с системного вызова fork () , vfork () или clone (). Все три системных вызова заканчиваются вызовом функции ядра do_f ork (), выполняющего основные действия по созданию нового процесса. Во время выполнения процесс переходит из одного состояния в другое. Процесс переходит из состояния готовности в состояние выполнения, когда его выбирает планировщик, из выполняемого состояния в состояние готовности, когда его временной срез завершился или захвачен другим процессом, из заблокированного состояния в состояние готовности при поступлении ожидаемого сигнала и из выполняемого состояния в заблокированное при ожидании ресурса или во время сна. Смерть процесса наступает по системному вызову exit ().

Далее мы углубились в базовые конструкции планировщика и используемые им структуры, включая очередь выполнения и очередь ожидания, а также то, как планировщик управляет этими структурами для отслеживания состояния процессов.

Завершается процесс обсуждением асинхронного потока выполнения процессов, включая исключения и прерывания с рассмотрением аппаратной обработки прерывания на x86 и PPC. Мы увидели, как ядро Linux управляет прерыванием, после того как оно поступает через аппаратуру, на примере прерывания системного таймера.

Проект: текущая системная переменная

Этот раздел описывает task_struct и текущую системную переменную, которая указывает на структуру task_struct выполняемой задачи. Целью проекта является подтверждение идеи, что ядро предпочитает редко изменяемые серии связанных структур, которые создаются и уничтожаются во время выполнения программы. Как мы видели, структура task_struct является одной из наиболее важных структур ядра, в которой содержится вся необходимая ядру информация о задаче. Этот проект получает доступ к структуре так же, как и ядро, и может служить основой для дальнейших экспериментов читателей.

В этом проекте мы получаем доступ к текущей task_s truct и печатаем различные элементы из этой структуры. Начиная с файла include/linux/sched.h мы ищем trask_struct. Используя sched.h, мы возьмем current->pid и current-> comm, идентификатор текущего процесса и его имя и проследим pid и согшпэтих структур. Далее мы углубимся в функцию из printk () и пошлем сообщение обратно текущему tty-терминалу, который мы используем.

> **ПРИМЕЧАНИЕ. После запуска программы (hellomod), как вы думаете, каким будет имя текущего процесса, которое распечатает current->comm? Каким будет имя процесса parent? (См. нижеследующее обсуждение кода.)**

Рассмотрим код процесса.

Исходный код процесса[1]

```
currentptr.c
1       #include <linux/module.h>
2       #include <linux/kernel.h>
3       #include <linux/init.h>
4       #include <linux/sched.h>
5       #include <linux/tty.h>
006
007     void tty_write_messagel(struct tty_struct *, char *) ;
008
009      static int my_init( void )
010

{
011
012        char *msg="Hello tty!";
013
014         printk("Hello, from the kernel...\n");
015        printk("parent pid =%d(%s)\n", current->parent->pid,
            current->parent->comm);
016          printk("current pid =%d(%s) \n", current->pid, current->comm) ;
017
18             tty write messagel(current->signal->tty,msg) ;
19             return 0;
020     }
022 static void my_cleanup( void )
{
  printk("Goodbye, from the kernel...\n"); }
27  module_init(my_init);
28  module_exit(my_cleanup);

//Эта функция извлечена из <printk.c>
032 void tty write messagel(struct tty struct *tty, char *msg)
{
  if (tty & tty->driver->write) tty->driver-
   >write(tty, 0, msg, strlen(msg));
  return;
037  }
```

[1] **Вы можете использовать исходный код примера в качестве отправной точки в исследовании ядра. В ядре присутствует множество полезных функций, которые стоит рассмотреть, таких, как внутренние [например, strace ()]; построение вашего собственного инструмента наподобие этого проекта поможет вам пролить свет на некоторые аспекты ядра Linux.**

Строка 4

sched.h содержит struct task__struct { }, к которой мы обращались через Ш процесса (->pid), и имя текущего процесса (->comm), в то время как родительский PID (->parent) указывает на структуру родительской задачи. Также мы ищем указатель на структуру сигнала, содержащую ссылку на структуру tty (см. строки 18-22).

Строка 5

tty. h содержит struct tty_struct {}, используемую функцией, которую мы нашли в printk. c (см. строки 32-37).

Строка 12

Это просто строка сообщения, которую мы хотим послать обратно на терминал.

Строка 15

Здесь мы обращаемся к родительскому РШ и его имени из структуры текущей задачи. Ответ на предыдущий вопрос заключается в том, что родителем нашей задачи является текущая программа-оболочка; в нашем случае Bash.

Строка 16

Здесь мы обращаемся к текущему РГО и имени из структуры текущей задачи. Отвечая на вторую половину предыдущего вопроса, мы вводим insmod в командной строке Bash, и это будет респечатано в качестве текущего процесса.

Строка 18

Эта функция взята из kernel /printk. c. Она используется для перенаправления событий на определенный tty. Для демонстрации нашей текущей точки мы передаем этой функции структуру tty_struct из tty (окна или командной строки), от которого мы порождаем нашу программу. Мы получаем эту информацию из current->signal->tty. Строка msg param объявлена в строке 12.

Строки 32-38

Функция tty write проверяет существование tty и затем вызывают соответствующий драйвер устройства с помощью сообщения[1].

Запуск кода

Откомпилируйте код и запустите insmod (), как в нашем первом проекте.

[1] **Этот вызов (драйвера) действителен именно для ядра 2.6.7, исходные тексты которого используются в данной книге, и не всегда действителен для других ядер, например для ядра 2.6.15 следует использовать следующий вызов: tty->driver->write (tty, msg, strlen (msg)). *Примеч. науч.ред.***

Упражнения

1. Когда мы описывали состояния процесса, мы описали «ожидающее или заблокированное состояние» как состояние процесса, который не выполняется и не готов к выполнению. В чем разница между ожидающим и заблокированным процессами? В каком состоянии будет процесс, обнаруживший себя ожидающим, и в каком он будет состоянии, если он заблокирован?
2. Найдите код ядра, где состояние процесса меняется с выполняемого на заблокированное? Другими словами, найдите, где состояние current->s tate меняется С TASK_RUNNING на TASK_STOPPED.
3. Чтобы понять, сколько времени понадобится счетчику, чтобы отмотаться до конца, проделайте следующие вычисления. Если 64-битовый декрементор работает на частоте 500 МГц, сколько ему потребуется времени на завершение при следующих начальных значениях:
   a) 0x0000000000000fff,
   b) OxOOOOOOOOffffffff,
   c) Oxffffffffffffffff?
4. Старые версии Linux использовали sti () и cli () для отключения прерываний, при которых раздел кода не будет прерываться. Новые версии Linux используют вместо этого spin_lock (). Какое главное преимущество циклической блокировки?
5. Каким образом функция x86 do_IRQ() и функция PPC ppc_irq__dispatch_ handler () позволяют разделять прерывания?
6. Почему не рекомендуется получать доступ к системному вызову из кода ядра?
7. Сколько очередей выполнения на один процессор доступно в ядре Linux версии 2.6?
8. Когда процесс ответвляет новый процесс, требуется ли Linux выделить ему новый временной срез? И если да, то почему.
9. Как процесс может быть вставлен в текущий массив приоритетов очереди выполнения, после того как его временной срез закончился? Каков диапазон обычного приоритета процесса? Что вы можете сказать о процессах реального времени?

Глава 4

Управление памятью

В этой главе:

Управление памятью - это метод, с помощью которого запускаемая на компьютере программа получает доступ к памяти, используя комбинации аппаратных и программных манипуляций. Работа подсистемы управления памятью заключается в выделении доступной памяти требующему ее процессу и освобождении памяти, более этому процессу не требующейся, а также в слежении за всей доступной памятью.

Жизненный цикл операционной системы разделяется на две фазы: нормальное выполнение и загрузку. Фаза загрузки использует память временно. Фаза нормального выполнения разделяет память на порции, из которых одна постоянно назначена коду и данным ядра, а другая порция назначается для динамического управления памятью. Запросы к динамической памяти происходят по мере создания и роста процесса. Эта глава концентрируется на нормальном выполнении.

Мы должны понять несколько концепций высокого уровня, связанных с управлением памятью, перед тем как погрузиться в подробности реализации и связи отдельных компонентов. Глава начинается с описания того, что такое система управления памятью и что такое виртуальная память. Далее мы обсудим различные структуры ядра и алгоритмы, помогающие управлять памятью. Далее мы разберемся, как ядро управляет памятью, увидим, как разделяется и управляется память процесса и как это связано со структурами ядра более высокого уровня. После этого мы рассмотрим процесс накопления памяти, управление ею и ее освобождение, рассмотрим ошибки страниц и как они обрабатываются на двух архитектурах - Power PC и x86.

Простейшим типом системы управления памятью является такая, при которой выполняемый процесс получает полный доступ к памяти. Для работающего таким образом процесса необходимо включать в себя весь код, необходимый для управления любым присутствующим в системе аппаратным обеспечением, необходимо следить за всеми своими адресами памяти и иметь все данные загруженными в память. Этот подход налагает на разработчика полную ответственность и предполагает, что процесс полностью помещается в доступную память. Для более-менее сложных программ выполнить все эти требования практически невозможно, поэтому доступная память обычно делится между операционной системой и пользовательскими процессами, с передачей задачи управления памятью операционной системе.

На современные операционные системы налагается требование возможности разделения системных ресурсов между множеством программ и возможностью обхода ограничений памяти прозрачным для разработчика программы образом. **Виртуальная память** - это метод, позволяющий предоставить программе доступ к большему, чем физически присутствует в системе, объему памяти и обеспечить эффективное разделение памяти между множеством программ. Физическая память, или память ядра, - это память в планках памяти, установленный в системе. Виртуальная память позволяет программам считать, что им прозрачно доступен больший объем памяти, чем присутствует в памяти

ядра за счет места на диске. Пространство диска, как более дешевое и обладающее большим объемом, чем физическая память, может использоваться в качестве расширения внутренней памяти. Мы вызываем эту виртуальную память так, как будто дисковое пространство является обычной оперативной памятью. Рис. 4.1 иллюстрирует связь между различными уровнями хранения данных.

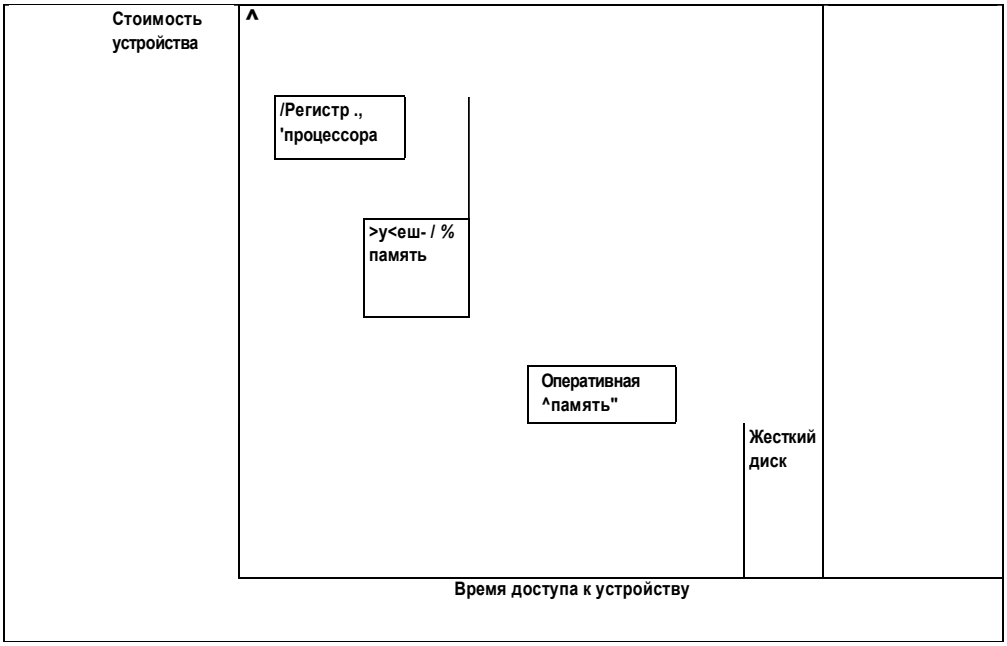


Рис. 4.1. Иерархия доступа к данным

Для использования виртуальной памяти, данные программы разделяются на базовые единицы, которые могут перемещаться из памяти на диск и обратно. Таким образом, используемые части программы могут находиться в памяти, получая преимущества быстрого времени доступа. Неиспользуемые части временно размещаются на диске, что минимизирует влияние недостаточной скорости доступа диска, хотя при этом все равно требуется некоторое время на чтение данных в память. Эти единицы данных, или блоки, виртуальной памяти называются **страницами (pages).** Таким же образом физическую память приходится разделять на сегменты, хранящие эти страницы. Эти сегменты называются **фреймами страниц (page frames).** Когда процесс запрашивает адрес, содержимое страницы загружается в память. Все запросы к данным на этой странице требуют доступа к этой странице. Если к адресу на этой странице не был получен доступ ранее, страница не загружена в памяти. Первый запрос к адресу на странице получается неудачным или

вызывает **ошибку страницы (page fault),** потому что она не находится в памяти и должна быть загружена с диска. Ошибка страницы - это ловушка. Когда это происходит, ядро должно выбрать фрейм страницы и записать его содержимое (страницу) обратно на диск, заменив его содержимым страницы, затребованной программой.

Когда программа выбирает данные из памяти, она использует адреса для обозначения порции памяти, к которой требуется доступ. Эти адреса называются **виртуальными адресами (virtual addresses)** и образуют **виртуальное адресное пространство (virtual address space)** процесса. Каждый процесс имеет собственный диапазон виртуальных адресов, позволяющий предотвратить чтение или запись данных другой программы. Виртуальная память позволяет процессу «использовать» больше памяти, чем доступно физически. К тому же операционная система может предоставить каждому процессу свое собственное виртуальное линейное адресное пространство[1].

Размер этого адресного пространства определяется размером слова данной архитектуры. Если процессор может хранить 32-битовое значение в своих регистрах, виртуальное пространство программы, запускаемой на этом процессоре, составляет 2il адресов[2]. Виртуальная память не только позволяет расширить доступное количество памяти, но и делает прозрачной для программиста пользовательского пространства свойства физической памяти. Например, программисту не нужно управлять пробелами в памяти. В нашем 32-битовом примере у нас есть виртуальное адресное пространство от 0 до 4 Гб. Если в системе присутствует 2 Гб оперативной памяти, ее физический адрес находится в пределах от 0 до 2 Гб. Наши программы могут быть программами 4 Гб, главное, чтобы они помещались в доступную память. Все программы хранятся на диске, и **страницы** перемещаются в память по мере необходимости.

Действие перемещения страницы из памяти на диск и обратно называется **страничной подкачкой (paging).** Подкачка включает преобразование виртуальных адресов программы в физический адрес в памяти.

Управление памятью - это часть операционной системы, которая следит за связью между виртуальным и физическим адресом и управляет подкачкой. **Модуль управления памятью (Memory Management Unit, MMU),** являющийся аппаратным агентом, выполняет настоящее преобразование[3]. Ядро использует **таблицу страниц (page tables),** индек-

[1] **Адресация процесса подразумевает несколько допущений относительно процесса использования памяти. Первое заключается в том, что процесс не использует всю затребованную память единовременно. Второе заключается в том, что два или более экземпляра процесса из одного исполнимого файла загружают исполняемый объект только один раз.**

[2] **Несмотря на то что ограничение на доступную память технически равно сумме памяти и файла подкачки, адресуемый предел зависит от размера слова на данной архитектуре. Это значит, что, даже если в системе есть больше 4 Гб памяти, процесс может выделить более 3 Гб (с учетом того, что до 1 Гб назначается ядру).**

[3] **Некоторые микропроцессоры, такие, как Motorola 68000 (68 Кб), нуждаются в собственном MMU; uCLinux - это Linux-дистрибутив, который специально портирован для работы на системах без MMU. Без MMU, виртуальные и физические адреса совпадают.**

сирующую список доступных страниц и их связь с адресами, использующимися MMU при преобразовании адресов. Таблица обновляется при загрузке страницы в память.

Посмотрев на общую картину управления памятью, рассмотрим теперь, как в ядре реализовано управление памятью и как реализованы страницы.

4.1 Страницы

В качестве базовой единицы памяти, которой управляет менеджер памяти, страница обладает большим количеством состояний, за которыми необходимо следить. Например, ядру необходимо знать, когда страница становится свободной для повторного выделения. Для этого ядро использует описатель страниц. Каждой физической странице в памяти назначается свой описатель страницы.

Этот раздел описывает различные поля в описателе страницы и то как их использует менеджер памяти. Структура страницы определена в include/linux/mm.h.

```
include/linux/mm.h
170  struct page {
171   unsigned long flags;
172
173   atomic t count;
174   struct list head list;
175   struct address space *mapping;
176   unsigned long index;
177   struct list head lru;
178
179   union {
180    struct pte chain *chain;
181
182    pte addr t direct;
183   } pte;
184   unsigned long private; 185

196  #if defined(WANT_PAGE_VIRTUAL)
197   void *virtual;
198
199  #endif
200  };
```

4.1.1 flags

Атомарные флаги описывают состояние страниц фреймов. Каждый флаг представлен одним битом 32-битового значения. Некоторые вспомогательные функции позволяют нам манипулировать отдельными флагами и тестировать их. Кроме этого, некоторые вспомогательные функции позволяют нам получать доступ к значениям битов, соответствующих отдельным флагам. Сами флаги, как и вспомогательные функции, определены в include/linux/page-flags.h. Табл. 4.1 описывает некоторые флаги, которые могут быть установлены в поле flags структуры страницы.

Таблица 4.1. Значение flag для page->flags

Флаг	Описание
PG_locked	Страница заблокирована, и ее нельзя трогать. Этот бит используется при вводе-выводе на диск, устанавливается перед операцией ввода-вывода и снимается после нее
PG_error	Обозначает, что для этой страницы произошла ошибка
PG_referenced	Обозначает, что эта страница была запрошена для выполнения операции ввода-вывода. Используется для определения того, находится ли страница в списке активных или неактивных страниц
PG__uptodate	Обозначает, что содержимое страницы верно и устанавливается после операции чтения в эту страницу. Этот флаг взаимоисключаем с PG_error
PG_dirty	Обозначает модифицированную страницу
PG_lru	Страница находится в списке наименее часто используемых [least recently used (lru)]. См. более подробное описание структуры lru далее в этом разделе
PG_active	Обозначает, что страница находится в списке активных страниц
PG_slab	Эта страница принадлежит к блоку памяти, созданному выделителем блоков, описывается в разделе «Выделение секций» в этой главе
PG_highmem	Означает что эта страница находится в верхней области памяти (ZONE_HIGHMEM) и поэтому не может быть сразу отображена в виртуальное адресное пространство ядра. Страницы из верхней области памяти определяются во время загрузки в mem_init () (см. подробности в гл. 8, «Загрузка ядра»)
PG_checked	Элемент файловой системы ext2. Убран в версии 2.5
PG_arch_l	Бит архитектурно-специфического состояния страницы

Таблица 4.1. Значение flag для page->flags (Окончание)

PG_reserved	Помечает страницу, которую нельзя выгрузить в своп память, которая не существует или выделена при загрузке системы
PG_private	Обозначает, что страница верна и устанавливается, если page->private содержит правильное значение
PG_writeback	Обозначает, что страница перезаписывается
PG_mappedtodisk	Эта страница содержит блоки, выделенные на системном диске
PG_reclaim	Обозначает, что страницу можно перераспределить
PG_compound	Обозначает, что страница является частью страницы более высокого уровня

4.1.1.1 count

Поле count служит в качестве счетчика ссылок на страницу. Значение 0 означает, что фрейм станицы доступен для повторного использования. Положительное значение означает количество процессов, могущих получить доступ к данным этой страницы[1].

4.1.1.2 list

Поле list - это структура, хранящая указатели на следующий и предыдущий элементы двусвязного списка. Двусвязный список, к которому принадлежит данная страница, определяется частью, связанной с состоянием страницы.

4.1.1.3 mapping

Каждая страница может быть ассоциирована со структурой address_space, хранящей информацию для отображения файла в память. Поле mapping является указателем на address_space, членом которого является данная страница; address_space - это набор страниц, принадлежащих объекту памяти (например, inode). Более подробно использование address_space описывается в гл. 7, «Планировщик и синхронизация ядра», в подразд. 7.14.

4.1.1.4 lru

Поле lru хранит указатели на следующий и предыдущий элементы в списке последних использованных (Least Recently Used, LRU). Этот список связан с перераспределением памяти и состоит из двух списков: active_list, содержащего используемые страницы, и inactive_list, хранящего страницы, годные для повторного использования.

[1] Страница освобождается, когда хранимые в ней данные больше не требуются.

4.1.1.5 virtual

virtual - это указатель на соответствующий странице виртуальный адрес. В системе с верхней памятью[1], отображение памяти может выполняться динамически, для чего необходимо выполнять перерасчет виртуального адреса. В этом случае это значение становится равным NULL.

Составные страницы

Составные страницы - это страницы более высокого уровня. Для включения поддержки составных страниц в ядре во время компиляции необходимо включить «Huge TLB Page Support». Составные страницы объединяют более одной страницы, первая из которых называется головной страницей, а последняя хвостовой страницей. У всех составных страниц устанавливается бит PG_compound в page->f lags, а page->lru. next указывает на голову страницы.

4.2 Зоны памяти

Не все создаваемые страницы равноценны. На некоторых компьютерных архитектурах определены константы, в рамках которых можно использовать некоторые физические адреса. Например, на x86 некоторые шины ISA могут адресовать только 16 Мб оперативной памяти. Несмотря на то что на PPC таких констант нет, концепция зон памяти портируется и на эту платформу для упрощения архитектурно-независимой части кода. В архитектурно-зависимой части PPC кода эти зоны перекрываются. Другие подобные константы могут использоваться, если в системе присутствует больше оперативной памяти, чем можно адресовать линейным способом.

Зоны памяти составляются из фреймов страниц или физических страниц, поэтому фреймы страниц выделяются из определенных зон памяти. В Linux существует три зоны памяти: ZONE__DMA (использующая фреймы страниц DMA), ZONE_NORMAL (не DMA-страницы с виртуальным отображением в память) и ZONE_HIGHMEM (страницы, чьи адреса не находятся в пространстве виртуальных адресов).

4.2.1 Описатель зоны памяти

Как и любой объект, управляемый ядром, зона памяти имеет структуру zone, хранящую всю информацию об этой зоне. Структура zone определена в include/linux/ mmzone.h. Далее мы рассмотрим поближе несколько наиболее часто используемых полей этой структуры:

[1] Верхняя память - это физическая память, которая превышает адресуемое виртуально пространство. См. разд. 4.2, «Зоны памяти».

include/linux/mmzone.h бб
struct zone {

```
70  spinlock t   lock;
71  unsigned long  free pages;
72  unsigned long  pages_min, pages_low, pages_high; 73
74  ZONE PADDING ( padl )
75
76  spinlock t   lru lock;
77  struct list head active JList;
78  struct list head  inactive list;
79  atomic_t  refill_counter;
80  unsigned long nr active;
81  unsigned long nr inactive;
82  int all_unreclaimable;     /* Все страницы закрепляются */
83  unsigned long pages scanned; /* с момента последнего восстановления */ 84
85        ZONE_PADDING (_pad2_)

103   int temp priority;

104   int prev__priority;

109   struct free_area free_area[MAX_ORDER];

135   wait_queue_head_t  * wait_table;
13 6  unsigned long wait table size;
137   unsigned long wait table bits;
138
13 9  ZONE_PADDING (_pad3_)

157   } ___ cacheline_maxaligned_in_smp;
```

4.2.1.1 lock

Описатель зоны нужно блокировать во время работы с этой зоной для предотвращения ошибок чтения-записи. Поле lock хранит кольцевую блокировку, защищающую описатель от этих ошибок.

Блокировка касается только описателя, а не самого диапазона памяти, с которым он ассоциирован.

4.2.1.2 free_pages

Поле f ree_pages хранит количество оставшихся в зоне свободных страниц. Это unsigned long-число увеличивается каждый раз, когда из зоны выделяется станица, и уменьшается после того, как страница освобождается. Общее количество свободной оперативной памяти, возвращаемое nr_f ree_pages (), рассчитывается с помощью сложения значений для всех трех зон.

4.2.1.3 pages_min, pagesjow и pagesjiigh

Поля pages_min, pages_low и pages_high хранят значения водяных знаков. Когда количество доступных станиц достигает каждого из этих водяных знаков, ядро отвечает на нехватку памяти соответствующим каждому из этих значений образом.

4.2.1.4 rujock

Поле ru_lock хранит циклическую блокировку для списка свободных страниц.

4.2.1.5 acrtivejlst и inactivejist

acrtive_list и inactive_list используются при перераспределении функциональности страниц. Первый - это список активных страниц, а второй - список страниц, годных для повторного использования.

4.2.1.6 all_unreaclaimable

Поле all_unreaclaimable устанавливается в 1, если все страницы в зоне закреплены. Использовать повторно их можно только с помощью демона страниц kswapd.

4.2.1.7 pages_scanned, temp_priority и prev_priority

Поля page s__s c armed, temp_priority и prev_priority связаны с функциональностью перераспределения памяти, выходящей за рамки рассмотрения данной книги.

4.2.1.8 free_area

Дружественная система, использующая битовую карту f ree_агеа.

4.2.1.9 waiMable, wait_table_size и wait_table_bits

Поля wait_table, wait_table_size и wait_table_bits связаны с запросами к зонам страниц из очереди ожидания процессов.

4.2.2 Вспомогательные функции для работы с зонами памяти

Когда к объекту применяется действие, обычно необходимо получить информацию об этом объекте. Проще всего такую информацию можно получить с помощью вспомогательных функций. Далее представлено несколько вспомогательных функций для манипуляции с зонами памяти.

Выравнивание кеша и заполнение зон памяти

Выравнивание кеша выполняется для увеличения производительности при доступе к полю описателя. Выравнивание кеша увеличивает производительность за счет минимизации количества инструкций, необходимых для копирования части данных. Возьмем случай 32-битового значения, не выровненного по слову. Процессору придется выполнить две инструкции «загрузка слова» для получения данных из регистров вместо одного. ZONE_PUDDING показывает, как выполняется выравнивание в зоне памяти:

```
include/linux/mmzone.h #if
defined(CONFIG_SMP) struct
zone padding { int x;
}      cacheline  maxaligned in smp;
#define ZONE_PADDING(name)  struct zone_padding name;
#else
#define ZONE PADDING(name)
#endif
```

Если вы хотите узнать больше о выравнивании в Linux, см. файл include/linux/ cache. h.

4.2.2.1 for__each_zone()

Макрос f or_each_zone () итерационно перебирает все зоны:

```
include/linux/mmzone.h
2 68  #define for each zone(zone) \
2 69   for (zone = pgdat_list->node_zones; zone; zone = next_zone(zone) )
```

4.2.2.2 is_highmem() и is_normal()

Функции is__highmem() и is_normal() проверяют структуры зон в верхней или в нормальной зоне соответственно:

```
include/linux/mmzone.h
315   static inline int is_highmem(struct zone *zone)
316   {
317    return (zone - zone->zone_pgdat->node_zones == ZONE_HIGHMEM) ;
318   }
319
32 0  static inline int is normal(struct zone *zone)
321   (
```

```
return (zone - zone->zone_pgdat->node_zones == ZONE_NORMAL); 323 }
```

4.3 Фреймы страниц

Фреймы страниц - это единицы памяти, хранящие страницы. Когда процессу требуется память, ядро вызывает фрейм страницы. Таким же образом, когда фрейм страницы больше не используется, ядро освобождает его и делает доступным для другого процесса. Для выполнения этих операций служат следующие функции.

4.3.1 Функции для затребования страниц фреймов

Для запроса страниц фреймов можно использовать несколько функций. Мы можем раз делить эти функции на две группы, в зависимости от типа возвращаемого ими значения. Одна группа возвращает указатель на структуру страницы (типа void*), что соответст вует требуемому для выделения фрейму страницы. Сюда входят alloc__pages () и alloc_page (). Вторая группа функций возвращает 32-битовое значение виртуально го адреса (типа long) первой выделенной страницы. Сюда входят ____ get_ f ree_page() и ____ get_dma_pages(). Многие из них являются обычными обертками низ коуровневых интерфейсов. Рис. 4.2 и 4.3 демонстрируют графики вызовов этих функций.

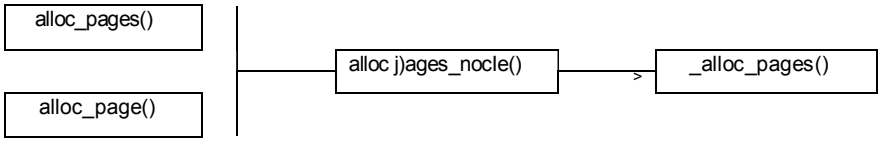


Рис. 4.2. Граф вызова alloc_ *0

Следующие макросы и функции ссылаются на число обработанных страниц (затребованных или освобожденных) в степени 2. Страницы выделяются и освобождаются последовательностями длины, равной степени двойки. Мы можем запросить 1,2,4,16 и т. д. групп страниц[1].

4.3.1.1 alloc_pages() и alloc_page()

a11oc_раде () запрашивает одну страницу и поэтому не имеет параметра порядка (order). Эта функция выставляет этот параметр в 0 при вызове alloc_pages_node(); alloc_pages (), наоборот, позволяет затребовать несколько страниц (2 в степени order):

[1] Группы запрашиваемых и освобождаемых страниц всегда являются последовательными.

```
include/linux/gfp.h
75  #define alloc pages(gfp mask, order) \
76    alloc pages  node(numa node id() , gfp mask# order)
77  #define alloc page(gfp mask) \
78    alloc_pages_node(numa_node_id(), gfp_mask, 0)
```

Как вы можете видеть на рис. 4.2, далее оба макроса вызывают alloc_ pages_node (), передавая соответствующие параметры; alloc_pages_node()- это функция-обертка, используемая для проверки порядка требуемых фреймов страниц:

```
include/linux/gfp.h
67  static inline struct page * alloc_pages_node (int nid,
                 unsigned int gfp mask, unsigned int order)
68  {
69   if (unlikely(order >= MAX__ORDER) )
70    return NULL;
71
72   return  _ alloc_pages(gfp_mask, order,
          NODE DATA(nid)->node zonelists + (gfp mask & GFP ZONEMASK) ) ;
73  }
```

Вы можете увидеть, что, если порядок требуемых страниц больше доступного максимума (MSX___ORDER), запрос на выделение страницы не проходит. В alloc _j?age() это значение всегда устанавливается в 0 и поэтому всегда проходит. MSX_ORDER определено в linux/mmzone. h и равно 11. Поэтому мы можем затребовать до 2048 страниц.

Функция ___alloc_pages () выполняет основную работу по запросу страницы. Эта функция определена в mm/page_alloc. с и требует знания зон памяти, о которых говорилось в предыдущем разделе.

4.3.1.2 _getjree_page() и _get_dma_pages()

Макрос __ get_f ree_раде () предполагает затребование только одной страницы. Как и в a11oc_раде (), в него передается 0 в качестве количества требуемых страниц для _ get_f ree_раде (), выполняющей основные действия. Рис. 4.3 иллюстрирует иерархию вызова этих функций.

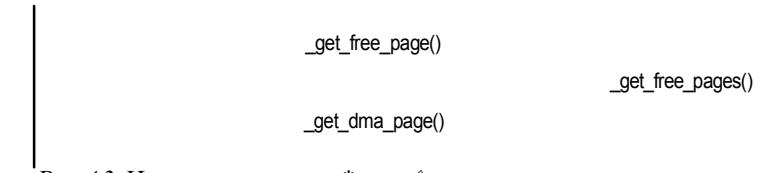


Рис. 4.3. Иерархия вызова get_* _page()

include/linux/gfp. h
```
83    #define _____get_free_page(gfp_mask)  \
84        __get_free_pages ( (gfp_itiask) , 0)
```

Макрос __ ge t_dma_pages () указывает, что требуемые страницы будут выделены из ZONE_DMA с помощью добавления флага к маске флагов страницы. ZONE_DMA указывают на порцию памяти, зарезервированную для доступа DMA:

```
include/linux/gfp. h
86    #define  _ get_dma__pages (gfp_mask, order) \
87     __ get_free_pages((gfp_mask) | GFP_DMA,(order))
```

4.3.2 Функции для освобождения фреймов страниц

Существует много функций для освобождения фреймов страниц: два макроса и две функции, для которых они служат обертками. Рис. 4.4 демонстрирует иерархию вызова функций, связанных с освобождением страниц. Эти функции также можно разделить на две группы. На этот раз разделение проводится по типу получаемых параметров. Первая группа, включающая __ f ree_page () и ____ f ree_pages (), получает указатель на описатель страницы, связанный с освобождаемыми страницами. Вторая группа, f ree_page () и f ree_pages (), получает адрес первой освобождаемой страницы.

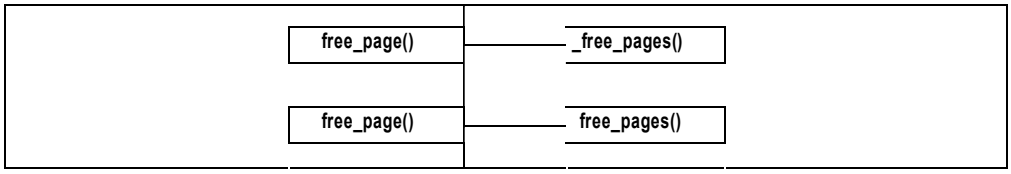


*Рис. 4.4. Иерархия вызова *free_page*0*

Макросы __ f ree_page () и f ree_page () освобождают одну страницу. Они передают 0 в качестве порядка освобождаемой страницы в функцию, выполняющую основную работу в _ f ree_page () и f ree_page () соответственно:

```
include/linux/gfp.h
94  #define   free page(page)    free pages((page), 0)
95  tdefine free_page(addr) free_pages((addr) , 0)
```

В конце концов f ree_page () вызывает _____free_pages_bulk(), являющуюся функцией реализации в Linux системы близнецов (buddy system). Мы рассмотрим совместную систему более подробно в следующем подразделе.

4.3.3 Система близнецов (buddy system)

При выделении и освобождении фреймов страниц, система сталкивается с проблемой фрагментации памяти, называемой **внешней фрагментацией (external fragmentation).** Это происходит, когда доступные фреймы страниц оказываются разбросаны по памяти таким образом, что невозможно выделить непрерывную последовательность страниц достаточной длины для удовлетворения запроса программы. При этом доступные фреймы страниц перемежаются одной или несколькими занятыми фреймами страниц, которые их разрывают. Уменьшить внешнюю фрагментацию можно несколькими способами. Linux использует реализацию алгоритма менеджера памяти, называемую **системой близнецов (buddy system).**

Система близнецов содержит список доступных блоков памяти. Каждый список указывает на блок памяти разного размера, каждый из которых равен степени двойки. Количество списков зависит от реализации. Фреймы страниц выделяются из списка свободных блоков наименьшего доступного размера. Система придерживает наибольшие доступные блоки для обслуживания больших запросов. Когда возвращаются выделенные блоки, система близнецов выполняет поиск свободных списков доступных блоков памяти, имеющих тот же размер, который имеет и возвращаемый блок. Если любой из доступных блоков прилегает к возвращаемому блоку, они объединяются блок в 2 раза большего размера. Эти блоки (возвращаемый и следующий за ним доступный) называются близнецами, откуда и происходит название «система близнецов». При этом ядро проверяет, чтобы блоки большего размера стали доступными сразу после освобождения фрейма страницы.

Теперь посмотрим на функции, реализующие систему близнецов в Linux. Функция выделения фрейма страницы__ alloc_pages () (mm/page__alloc. c). Фрейм страни цы освобождается с помощью функции __ f ree_pages_bulk ():

```
mm/page alloc.c
585  struct page * fastcall
586    alloc pages(unsigned int gfp mask, unsigned int order,
587      struct zonelist *zonelist)
588  {
589    const int wait = gfp mask &   GFP  WAIT;
590    unsigned long min;
591    struct zone **zones;
592    struct page *page;
593    struct reclaim state reclaim state;
594    struct task struct *p = current;
595    int i;
```

```
596     int alloc_type;
597     int do_retry;
598
599         might_sleep_if(wait) ;
600
601     zones = zonelist->zones;
602     if (zones[0] == NULL) /* В списке зон нет зон */
603       return NULL;
604
605             alloc_type = zone_idx(zones[0]);

608     for (i = 0; zones [i] != NULL; i++) {
609       struct zone *z = zones[i];
610
611       min = (1«order) + z->protection[alloc_type] ;

617     if (rt_task(p))
618       min -= z->pages_low » 1;
619
620     if (z->free_pages >= min ||
621          (!wait && z->free_pages >= z->pages_high) ) {
622       page = buffered_rmqueue(z, order, gfp_mask);
623       if (page) {
624        zone_statistics(zonelist, z);
625         goto got_pg;
626       }
627     }
628   }
629
63 0/* у нас мало памяти, и мы не смогли найти то, что нам нужно */
631     for (i = 0; zones[i] != NULL; i++) {
632       wakeup_kswapd(zones [i]);
633
634     /* Снова проходим по списку зон, с учетом        GFP_HIGH */
635    for (i = 0; zones[i] != NULL; i++) {
636       struct zone *z = zones[i];
637
638       min = (1«order) + z->protection[alloc_type] ;
639
640       if (gfp_mask &  __ GFP_HIGH)
641         min -= z->pages_low » 2;
642       if (rt_task(p))
643         min -= z->pages_low » 1;
644
645       if (z->free_pages >= min ||
646            (!wait && z->free_pages >= z->pages_high)) {
```

```
647         page = buffered rmqueue(z, order, gfp mask) ;
648         if (page) {
649           zone statistics(zonelist, z);
650           goto got_pg;
651         }
652       }
653     }

720   nopage:
721     if (!(gfp mask &   GFP NOWARN) && printk ratelimit () ) {
722       printk(KEKN WAKNING "%s: page allocation failure."
723           * ordered, mode: 0x%x\n" ,
724          p->comm, order, gfp mask);
725       dump s tack();
726     }
727     return NULL;
728   got pg:
729     kernel map pages(page, 1 « order, 1) ;
730    return page;
731   }
```

Система близнецов в Linux разделена на зоны, что означает, что список поддерживаемых доступных фреймов разделен на зоны. При этом каждый поиск свободного фрейма страниц может выполняться в одной из трех возможных зон, из которых можно получить фреймы страниц.

Строка 586

Целочисленное значение gf p_mask позволяет вызывающему___alloc_pages () коду определять способ поиска фреймов страниц (модификаторов действия). Возможные значения определены в include/linux/gfp .h и перечислены в табл. 4.2.

Таблица 4.2. Модификаторы действия для gfpjnask при выделении страниц памяти

Флаг	Описание
__GFP_WAIT	Позволяет ядру блокировать процесс, ожидающий фрейм страницы. Пример его использования можно увидеть в строке 537 page_alloc. c
__GFP_COLD	Требуется кеширование холодных страниц
__GFP_HIGH	Фрейм страницы можно найти в экстренном пуле памяти
__GFP_IO	Возможно выполнение передачи ввода-вывода
__GFP_FS	Позволяет вызвать низкоуровневые операции файловой системы

Таблица 4.2. Модификаторы действия для gfpjnask при выделении страниц памяти (Окончание)

GFP_NOWARN	При ошибке выделения фрейма страницы функция выделения посылает предупреждение об ошибке. Если выбран этот модификатор, сообщение не выводится. Пример применения этого флага можно увидеть в строках 665-666 page_alloc. c
_GFP_REPEAT	Повторная попытка выделения
_GFP_NORETRY	Запрос не требуется повторять из-за возможности ошибки
_GFP_DMA	Фрейм страницы находится в ZONE_DMA
_GFP_HIGHMEM	Фрейм страницы находится в ZONE_HIGHMEM

В табл. 4.3 представлены указатели на список зон, соответствующих модификаторам из gf p_mask.

Таблица 4.3. Список зон

Флаг	Описание
GFP__USER	Означает, что память выделяется не в пространстве ядра
GFP_KERNEL	Означает, что память выделяется в пространстве ядра
GFP_ATOMIC	Используется в обработчиках прерываний с помощью вызова kmalloc, так как предполагается, что выделитель памяти не засыпает
GFP_DMA	Означает, что память выделяется из ZONE_DMA

Строка 599

Функция might_sleep_if () получает значение переменной wait, хранящей ло гический бит операции AND для gfp__mask и значения _____ GFP_WAIT. Значение wait равно 0, если _____ GFP_WAIT не установлено, и 1, если установлено. Если при конфигурации ядра включена проверка сна при циклической блокировке (в меню Kernel Hacking), эта функция позволяет ядру блокировать текущий процесс на значение времени задержки.

Строки 608-628

В этом блоке мы проходим список описателей зон и ищем зону с достаточным для удовлетворения запроса количеством свободных страниц. Если количество свободных страниц удовлетворяет требуемому или если процессу разрешены ожидания, а количество свободных страниц больше либо равно верхнему пороговому значению зоны, вызывается функция buf f ered_rmqueue ().

Функция buf f ered__rmqueue () получает три аргумента: описатель зоны с доступными фреймами страниц, порядок количества требуемых фреймов страниц и температуру требуемых страниц.

Строки 631-632

Если мы попали в этот блок, мы не можем выделить страницу, потому что у нас слишком мало доступных фреймов страниц. Здесь предпринимается попытка вернуть фреймы страниц для удовлетворения запроса. Функция wakeup_ kswapd () выполняет эти действия и корректирует зоны с соответствующими фреймами страниц. При этом обновляются описатели зон.

Строки 635-653

После попытки возвращения фреймов страниц в предыдущем блоке кода мы снова проходим по зонам и ищем свободные фреймы страниц.

Строки 720-727

В этот блок кода мы попадаем, когда понимаем, что доступных фреймов страниц нет. Если выбран модификатор GFP_NOWARN, функция выводит сообщение об ошибке выделения страницы, включающее имя команды, вызванной для текущего процесса, порядок требуемых фреймов страниц и применяемую к запросу gf p_mask. Эта функция возвращает NULL.

Строки 728-730

Переход в этот блок кода выполняется после того, как требуемые страницы обнаружены. Функция возвращает адрес описателя страницы. Если требуется более одного фрейма страницы, она возвращает адрес описателя страницы для первого выделенного фрейма страницы.

При возвращении блока памяти система близнецов старается объединить их в блок большего размера, если доступен близнец такого же порядка. Это действие выполняет функция __f ree_pages_bulk (). Посмотрим, как она работает.

```
mm/page alloc.c
178  static inline void  _free_pages_bulk (struct page *page,
                                           struct page *base,
179      struct zone *zone, struct free area *area, unsigned long mask,
180      unsigned int  order)
181  {
182    unsigned long page idx, index;
183
184    if (order)
185      destroy compound page(page, order);
186    page_idx = page - base;
```

```
187     if (page idx & -mask)
188       BUG () ;
189     index = page idx » (1 + order);
190
191    zone->free j?ages -= mask;
192     while (mask + (1 « (MAX ORDER-l))) {
193       struct page *buddyl, *buddy2;
194
195       BUG_ON(area >= zone->free_area + MAX_ORDER);
196       if (! __test_and_change_bit(index, area->map))

206        buddyl = base + (page idx Л -mask);
2 07            buddy2 = base + page idx;
208       BUG  ON(bad range(z one, buddyl));
209       BUG ON(bad range(z one, buddy2));
210       list del(&buddyl->lru);
211       mask «= 1;
212      area++;
213       index »=  1;
214      page idx &= mask;
215     }
216     list add(&(base + page idx)->lru, &area->free list) ;
217   }
```

Строки 184-215

Функция __ free_pages_bulk() перебирает размеры блоков, соответствующих каждому из списков свободных блоков. (MAX_ORDER - это порядок блока наибольшего размера.) Для каждого порядка, и пока не будет достигнут максимальный порядок или найден наименьший близнец, она вызывает __test_and_ change_bit (). Эта функция проверяет, выделена ли страница близнеца для возвращаемого блока. Если это так, мы прерываем цикл. Если нет, она смотрит, можно ли найти близнеца с большим порядком, с которым можно объединить наш освобождаемый блок фреймов страниц.

Строка 216

Свободный блок вставляется в соответствующий список свободных фреймов страниц.

4.4 Выделение секций

Мы говорили, что страницы являются для менеджера памяти базовой единицей памяти. Тем не менее обычно процесс требует память указанного порядка байтов, а не порядка страниц. Для удовлетворения запросов по выделению меньших объемов памяти исполь-

зуется вызов функции kmalloc (), ядро реализует выделитель секций (slab allocator), являющийся слоем менеджера памяти, работающего с затребованными страницами.

Выделитель секций пытается уменьшить затраты на выделение, инициализацию, уничтожение и освобождение областей памяти, поддерживая кеш недавно использованных областей памяти. Этот кеш хранит выделенные и инициализированные области памяти, готовые к размещению. Когда пославший требование процесс больше не нуждается в области памяти, эта область возвращается в кеш.

На практике выделитель секций содержит несколько кешей, каждый из которых хранит области памяти разного размера. Кеши могут быть специализированными (specialized) или общего назначения (general purpose). Например, описатель процесса task_struct хранится в кеше, поддерживаемом выделителем секций. Область памяти, занимаемая этим кешем равна sizeof (task_struct). Аналогично хранятся в кеше структуры данных inode и dentry. Кеши общего назначения создаются из областей памяти предопределенного размера. Области памяти могут иметь размер 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65535 и 131072 байта[1].

Если мы запустим команду cat /proc/stabinf o, будет выведен список существующих кешей выделителя. В первой колонке вывода мы можем увидеть имена структур данных и группу элементов в формате size-*. Первый набор соответствует специализированным объектам кеша; набор букв соответствует кешам, хранящим объекты общего назначения указанного размера.

Кроме этого, вы можете заметить, что кеши общего назначения имеют два вхождения одинакового размера, у одного из которых в конце стоит DMA. Это происходит потому, что области памяти могут быть затребованы как из обычной зоны, так и из зоны DMA. Выделитель секций поддерживает кеши обеих типов памяти для удовлетворения всех запросов. Рис. 4.5 демонстрирует вывод /proc/slabinf o , где видны кеши обеих типов памяти.

В свою очередь, кеш делится на контейнеры, называемые секциями (slabs). Каждая секция составляется из одного или более прилегающих фреймов страниц, из которых выделяются области памяти меньшего размера. Поэтому мы будем говорить, что секции содержат объекты. Сами объекты представляют собой интервалы адресов предопределенного размера внутри фрейма страницы, который принадлежит к данной секции. Рис. 4.6 демонстрирует анатомию выделителя секций.

Выделитель секций использует три главные структуры для поддержания информации об объекте: описатель кеша, называемый kmem_cache; общий описатель кеша, называемый cache_size, и выделитель секции, называемый slab. Рис. 4.7 показывает общую картину связей между всеми описателями.

[1] **Все кеши общего назначения для повышения производительности выравниваются по L1.**

size-131072(DMA)	0	0	131072	1	32	: tunables	8	4	0	slab data	0	0	0
size-131072	0	0	131072	1	32	: tunables	8	4	0	slabdata	0	0	0
size-65536 (DMA)	0	0	65536	1	16	: tunables	8	4	0	slabdata	0	0	0
size-65536	1	1	65536	1	16	: tunables	8	4	0	slabdata	1	1	0
size-32768 (DMA)	0	0	32768	1	8	: tunables	8	4	0	slabdata	0	0	0
size-32768	0	0	32768	1	8	: tunables	8	4	0	slabdata	0	0	0
size-16384 (DMA)	0	0	16384	1	4	: tunables	8	4	0	slabdata	0	0	0
size-16384	0	0	16384	1	4	: tunables	8	4	0	slabdata	0	0	0
size-8192(DMA)	/%A	0	8192	1	2	: tunables	8	4	0	slabdata	0	0	0
size-8192		68	8192	1	2	: tunables	8	4	0	slabdata	64	68	0
size-4096 (DMA)	0	0	4096	1	1	: tunables	24	12	0	slabdata	0	0	0
size-4096	65	65	4096	1	1	: tunables	24	12	0	slabdata	65	65	0
size-204cV(DMA)	0	0	2048	2	1	: tunables	24	12	0	slabdata	0	0	0
size.-2048	102	102	2048	2	1	: tunables	24	12	0	slabdata	51	51	0
siz6-1024(DMA)	0	0	1024	4	1	: tunables	54	27	0	slabdata	0	0	0
size-1024	73	100	1024	4	1	: tunables	54	27	0	slabdata	25	25	0
size-512(DMA)	0	0	512	8	1	: tunables	54	27	0	slabdata	0	0	0
size-512	288	288	512	8	1	: tunables	54	27	0	slabdata	36	36	0
size-256 (DMA)	0	0	256	15	1	: tunables	120	60	0	slabdata	0	0	0
size-256	149	165	256	15	1	: tunables	120	60	0	slabdata	11	11	0
size-128(DMA)	0	0	128	30	1	: tunables	120	60	0	slabdata	0	0	0
size-128	4906	10290	128	30	1	: tunables	120	60	0	slabdata	343	343	0
size-64 (DMA)	0	0	64	58	1	: tunables	120	60	0	slabdata	0	0	0
size-64	1565	2323	64	58	1	: tunables	120	60	0	slabdata	40	40	0

Рис. 4.5. cat /proc/slabinfo

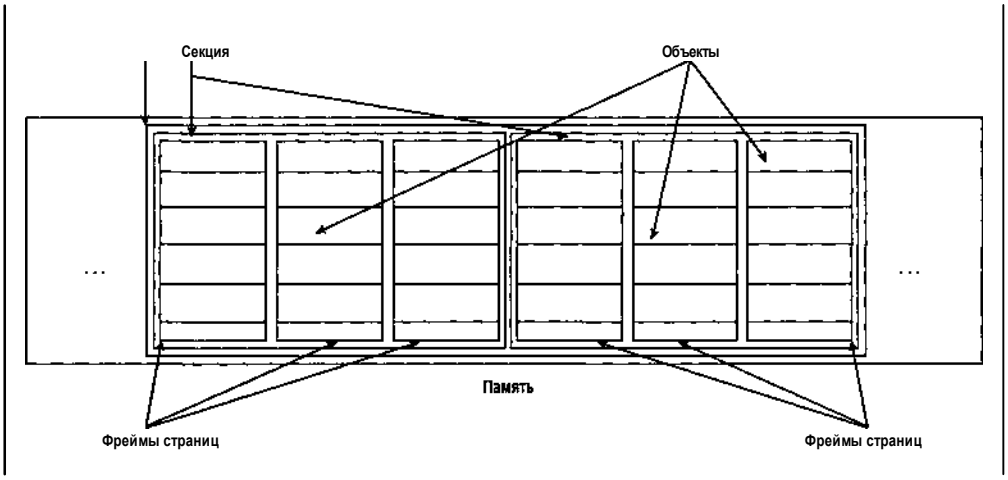


Рис. 4.6. Анатомия выделителя секций

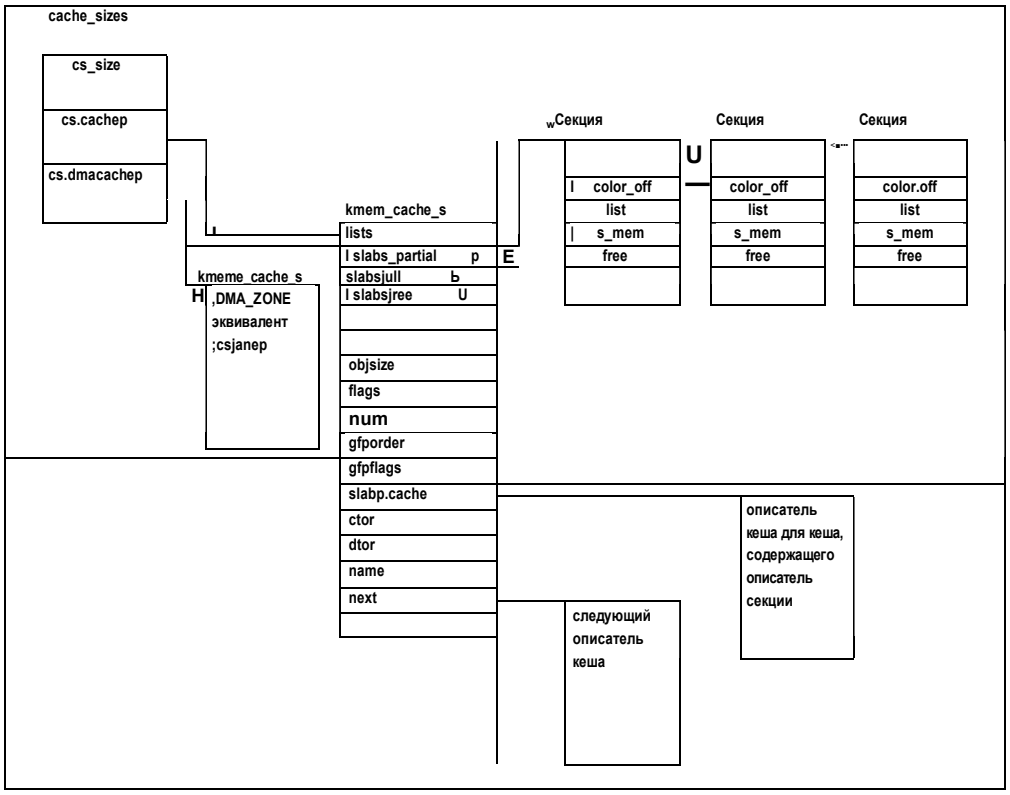


Рис. 4.7. Структура выделителя секций

4.4.1 Описатель кеша

Каждый кеш имеет собственный описатель кеша типа kmem_cache_s, хранящий о нем информацию. Большинство этих значений устанавливается или рассчитывается во время создания кеша в kmem_cache_create () (mm/slab. c).)Мы обсудим эту функцию в следующем разделе.) Для начала давайте посмотрим на поля описателя кеша и попытаемся понять, что он хранит.

```
mm/slab.c

246  struct kmem_cache_s {

252   struct kmem_list3  lists;

2 54  unsigned int  objsize;
```

```
255   unsigned int  flags;  /* флаги констант */
256   unsigned int num;  /* количество объектов в секции */

263  unsigned int  gfporder;
264
265   /* принудительные GFP-флаги, т. е. GFP_DMA */
266    unsigned int  gfpflags;
267
268    size_t   color; /* диапазон раскраски кеша */
269    unsigned int   color off;  /* цветовой отступ */
270    unsigned int  color next; /* раскраска кеша */
271    kmem c ache t   * s1abp cache;
272    unsigned int  dflags;    /* динамические флаги */
273   /* функция-конструктор */
274    void (*ctor)(void *, kmem cache t *, unsigned long);
275
276   /* функция-деструктор */
277    void (*dtor)(void *, kmem cache t *, unsigned long);
278
279   /* 4) создание,уничтожение кеша */
280    const char   *name;
281    struct list head next;
282

301  };
```

4.4.1.1 lists

Поле lists - это структура, хранящая головы трех списков, соответствующих трем состояниям, в которых может находиться секция: частично заполненному, полному или свободному. Кеш может иметь одну или несколько секций в любом из этих состояний. Именно таким образом эта структура данных связана с секциями. Сам список является двусвязным списком, поддерживаемым полем описателя секции list. Он описывается в подразделе «Описатель секции» далее в этой главе.

mm/slab.c

```
217   struct kmeiti_list3  {
218    struct list head slabs partial;
219    struct list head slabs full;
220    struct list_head slabs_free;
223    unsigned long next reap;
224    struct array__cache  * shared;
225   };
```

lists.slabsjpartial

lists . slabs_partial - это голова списка секций, с лишь частично выделенными объектами. Поэтому секция в частичном состоянии имеет несколько выделенных объектов и несколько свободных, готовых к использованию.

lists, slabsJull

lists . slabs_full -это голова списка секций, с полностью выделенными объектами. Эти секции не содержат доступных для выделения объектов.

lists, slabsJree

lists . slabs_f ree - это голова списка секций, с полностью свободными для выделения объектами. Ни один из объектов не выделен.

Поддержание этих списков снижает время, требуемое для нахождения свободных объектов. Когда из кеша затребывается объект, ядро имеет частично заполненную секцию. Если список частично заполненных секций пуст, поиск производится в списке свободных секций. Если список свободных секций пуст, создается новая секция.

lists.next_reap

Секции имеют фреймы выделенных для них страниц. Если эти страницы не используются, лучше вернуть их в общий пул памяти. Пригодные для этого кеши уничтожаются. Это поле хранит время уничтожения следующего кеша. Оно устанавливается в kmem_cache_create () (mm/slab.c) в момент создания кеша и обновляется в cache_reap () (mm/slab. c) при каждом вызове.

4.4.1.2 objsize

Поле obj size хранит размер (в байтах) объектов в кешах. Определяется во время создания кеша на основе требуемого размера и размещения кеша.

4.4.1.3 flags

Поле flags хранит маску флагов, описывающую отдельные характеристики кеша. Возможные флаги определены в include/linux/slab. c и приведены в табл. 4.4.

Таблица 4.4. Флаги секции

Флаг	Описание
SLAB_POISON	Запрос, тестирующий запись шаблона a5a5a5a5 в секцию во время создания. Может использоваться для проверки инициализированной памяти
SLAB_NO_REAP	Когда запрос памяти встречает недостаток памяти, менеджер памяти начинает уничтожать области памяти, которые больше не используются. Этот флаг устанавливается для того, чтобы быть уверенным, что этот кеш не будет автоматически уничтожен

Таблица 4.4. Флаги секции (Окончание)

SLAB_HWCACHE_ALIGN	Запрос на то, чтобы объект был выровнен в аппаратной линии кеша процессора для увеличения производительности с помощью сокращения циклов памяти
SLAB_CACHE_DMA	Означает, что следует использовать DMA-память. При запросе нового фрейма страницы флаг GFP_FLAG передается системе близнецов
CLAB_PANIC	Означает, что следует вызывать панику, если по какой-либо причине вызов kmem_cache_create () оказался неудачным

4.4.1.4 пит

Поле пит хранит количество объектов в каждой секции кеша. Это поле определяется во время создания кеша [также в kmem_cache__create ()] на основе значения gf porder (см. следующее поле), размера создаваемых объектов и требуемого размещения.

4.4.1.5 gfporder

gf porder - это порядок (на основе 2) количества последовательных фреймов страниц, находящихся в секторе кеша. По умолчанию имеет значение 0 и устанавливается при создании кеша с помощью вызова kinem_cache__create ().

4.4.1.6 gfpflags

Флаги gfpflags определяют тип фреймов страниц, требуемых для секции данного кеша. Они определяются на основе требуемых флагов области памяти. Например, если область памяти предназначена для использования с DMA, поле gfpflags устанавливается в GFP_DMA и передается далее при запросе фрейма страницы.

4.4.1.7 slabs_cache

Описатель секции хранится внутри самого кеша или снаружи его. Если описатель секции для секции в этом кеше хранится вовне по отношению к кешу, поле slabp_cache хранит указатель на описатель кеша, хранящего объект типа описателя секции. (См. более подробную информацию о хранении описателя секции в подразделе «Описатель секции».)

4.4.1.8 ctor

Поле ctor хранит указатель на конструктор[1], связанный с кешем, если он существует.

Если вы знакомы с объектно-ориентированным программированием, для вас не будет новостью концепция конструкторов и деструкторов. Поле описателя кеша ctor позволяет запрограммировать функцию, которая будет вызываться каждый раз, когда создается новый описатель кеша. Аналогично поле dtor хранит указатель на функцию, которая будет вызываться каждый раз, когда описатель кеша уничтожается.

4.4.1.9 dtor

Практически так же, как и поле с tor, поле dtor хранит указатель на деструктор, связанный с кешем, если такой существует.

Конструктор и деструктор определяются во время создания кеша и передаются в качестве параметров kmem_cache_create ().

4.4.1.10 name

Поле name хранит читабельную для человека строку с именем, отображаемым при открытии /proc/slabinf o. Например, для кеша, хранящего файловый указатель, значением этого поля будет f ilp. Проще всего это понять, если выполнить cat /proc/ slabinf o. Поле секции name должно хранить уникальные значения. Во время создания имя требуемой секции сравнивается с именами всех остальных секций в списке. Повторения не допускаются. Если существует еще одна секция с тем же именем, создание секции завершается ошибкой.

4.4.1.11 next

next - это указатель на следующий описатель кеша в односвязном списке описателей кешей.

4.4.2 Описатель кеша общего назначения

Как было сказано ранее, кеши всегда хранят объекты предопределенного размера общего назначения в виде пар. Один кеш - для выделения объектов из зоны DMA, другой - для стандартного выделения из обычной памяти. Если вы вспомните, что такое зоны памяти, вы поймете, что DMA-кеши размещаются в ZONE_DMA, а стандартные - в ZONE_ NORMAL. Структура cache_sizes является удобным средством для хранения всей связанной с размерами кешей информации.

```
include/linux/slab.h
69  struct cache_sizes {
70  size_t     cs_size;
71  kmem_cache_t    *cs_cachep;
72  kmem_cache_t    *cs_dmacachep;
73  };
```

4.4.2.1 cs_size

Поле cs_size хранит размер объекта памяти, находящегося в кеше.

4.4.2.2 cs_cachep

Поле cs__cachep хранит указатель на описатель кеша обычной памяти, выделяемых из ZONE_NORMAL.

4.4.2.3 cs_dmacachep

Поле cs_dmacachep хранит указатель на описатель кеша обычной памяти, выделяемых **ИЗ ZONE_NORMAL.**

На ум приходит вопрос: «Где хранятся описатели кешей?» Выделитель секций обладает специально зарезервированным для этого кешем. Кеш cache_cache хранит объекты типа описателя кеша. Кеш секции инициализируется статически при загрузке системы, для того чтобы можно было быть уверенным, что место для описателей кеша есть.

4.4.3 Описатель секции

Каждая секция в кеше имеет описатель, хранящий связанную с этой секцией информацию. Необходимо заметить, что описатель кеша хранится в специальном кеше, называемом cache__cache. Описатель секции, в свою очередь, хранится в двух местах: внутри самой секции (точнее говоря, фрейм первой страницы) или снаружи, в первом кеше общего назначения, в объекте достаточного размера для хранения описателя секции. Он заполняется при создании кеша на основе оставшегося после объекта размещения. Это пространство определяется при создании кеша.

Давайте рассмотрим несколько полей описателя секции.

mm/slab.c

```
173  struct slab {
174   struct list_head list;
175   unsigned long  coloroff;
176   void   *s  mem;  /* включает цветовой отступ */
177   unsigned int  inuse;  /* количество активных объектов в секции */
178   kmem_bufctl_t   free;
179  };
```

4.4.3.1 list

Как вы помните из обсуждения описателя кеша, секция может находиться в одном из трех состояний: free, partial или full. Описатель кеша хранит описатель секции в трех списках - по одному для каждого состояния. Все секции в определенном состоянии хранятся в двусвязном списке в зависимости от значения поля list.

4.4.3.2 sjnem

Поле s_mem хранит указатель на первый объект в секции.

4.4.3.3 inuse

Значение inuse отслеживает количество занятых в секции объектов. Для полностью или частично заполненных секций это число положительное, а для свободных секций равно 0.

4.4.3.4 free

Поле free хранит значение индекса для массива, элементы которого представляют объекты секции. В частности, поле free хранит значение индекса элемента массива, представляющего первый доступный объект в секции. Тип данных kmem_buf ctl_t связывает все объекты внутри секции. Этот тип является просто беззнаковым целым и определен в include /asm/ types .h. Этот тип данных определяет массив, всегда хранящийся сразу после описателя секции, в зависимости от того, хранится ли описатель секции внутри ее или снаружи. Все станет значительно понятней, если мы взглянем на функцию slab_buf ctl_t, возвращающую массив:

```
mm/slab.c
1614  static inline kmem bufctl t *slab bufctl(struct slab *slabp)
1615  {
1616   return (kmem_bufctl_t *)(slabp+1);
1617  }
```

Функция slab_buf c tl__t () получает указатель на описатель секции и возвращает указатель на область памяти, следующую сразу за описателем секции.

При инициализации кеша поле slab_f гее устанавливается в 0 (так как все объекты свободны, будет возвращен первый) и каждое вхождение в массив kniem__buf ctl_t устанавливается в значение индекса следующего члена массива. Это означает, что нулевой элемент хранит значение 1, первый элемент хранит значение 2 и т. д. Последний элемент в массиве хранит значение BUFCTL_END, что означает, что он является последним элементом в массиве.

Рис. 4.8 демонстрирует описатель секции, массив buf c tl и размещение объекта секции, когда описатель секции хранится внутри самой секции. В табл. 4.5 показаны возможные значения некоторых полей описателя секции в каждом из трех возможных состояний.

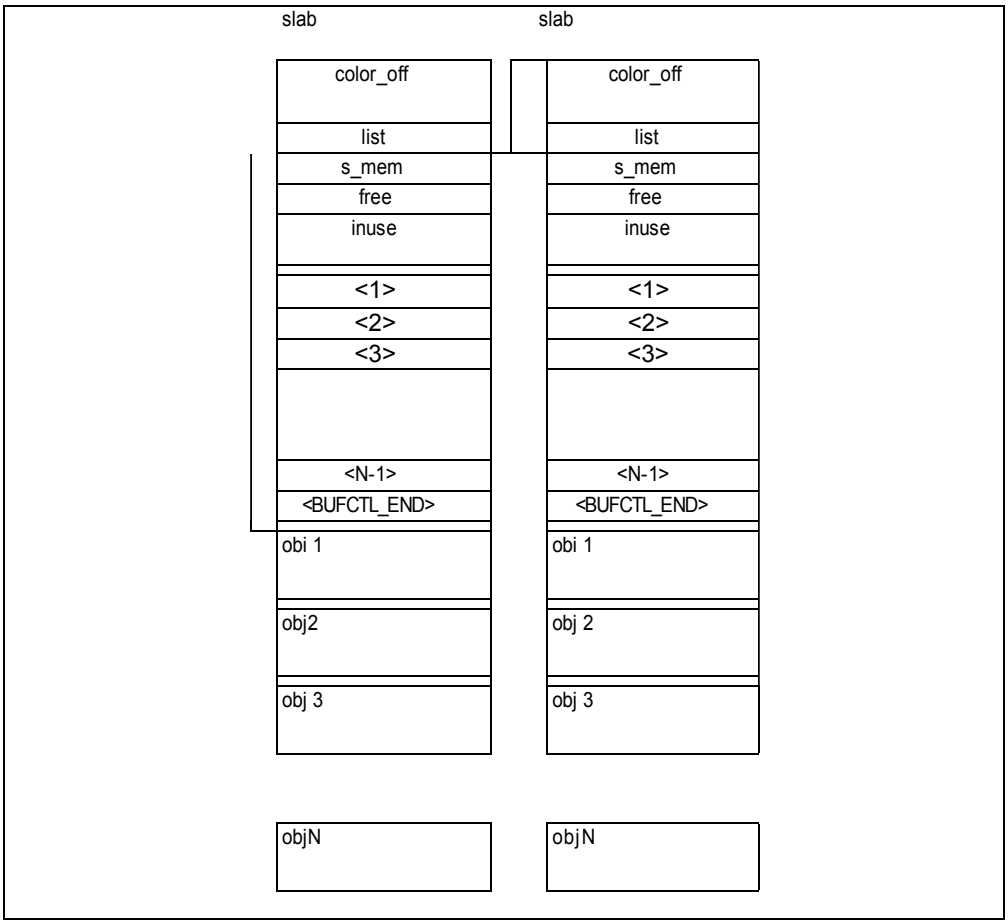


Рис. 4.8. Описатель секции и bufctl

Таблица 4.5. Состояние секции и значения полей описателя

	Free	**Partial**	Full
slab_inuse		X X	N N
slab->free			

ПРИМЕЧАНИЕ. N = количество объектов в секции; X = некоторое положительное число.

4.5 Жизненный цикл выделителя секции

Теперь мы рассмотрим взаимодействие кеша и выделителя секции на протяжении жизненного цикла ядра. Ядру нужно быть уверенным, что некоторые структуры находятся на своих местах для поддержки запрошенной процессом области памяти и при создании специализированного кеша в динамически загружаемом модуле.

Для выделителя секции важную роль играют несколько глобальных структур. Некоторые из них упоминались в этой главе. Давайте рассмотрим эти глобальные переменные.

4.5.1 Глобальные переменные выделителя секции

С выделителем секции связаны несколько глобальных переменных. Они включают:

- **cache_cache**. Описатель кеша, содержащий все остальные описатели кешей. Читабельное для человека имя этого кеша - kmem__cache. Это единственный статически выделяемый описатель кеша.
- **cache__chain.** Элемент списка, служащий в качестве указателя на список описателей кешей.
- **cache__chain_sem.** Семафор, контролирующий доступ к cache_chain[1]. Каждый раз, когда в последовательность добавляется новый элемент (новый описатель кеша), этот семафор получается с помощью down () и освобождается с помощью up ().
- **malloc_sizes [ ]**. Массив, хранящий описатели кеша для DMA- и неБМА-кешей, которые связаны с общим кешем.

Перед инициализацией выделителя секций эти структуры уже должны находиться на своих местах. Давайте посмотрим, как они создаются.

```
mm/slab.c
486  static kmem cache t cache cache = {
487    .lists = LIST3 INIT(cache cache.lists),
488    .batchcount = 1,
489    .limit   = BOOT_CPUCACHE_ENTRIES,
490    .objsize  = sizeof(kmem cache t),
491    .flags   = SLAB_NO_REAP,
492    .spinlock  = SPIN LOCK UNLOCKED,
493    .color_off  = Ll_CACHE_BYTES,
494    .name  = "ктет_саспе",
495   };
496
497   /* Защищенный доступ к последовательности кеша. */
498   static struct semaphore cache_chain_sem;
```

[1] **Семафоры подробно описываются в гл. 9, «Построение ядра Linux».**

```
499
500   struct  list_head cache_chain;
```

Описатель кеша cache_cache обладает флагом SLAB_NO_REAP. Даже если памяти мало, этот кеш сохраняется на протяжении всей жизни ядра. Обратите внимание, что семафор cache_chain только определяется, но не инициализируется. Инициализация происходит во время инициализации системы в вызове kmem_cache_init (). Здесь мы рассмотрим эту функцию подробнее.

```
mm/slab.c
462   struct cache sizes malloc sizes[] = {
463   #define CACHE(x) { .cs size = (x) },
464   #include <linux/kmalloc  sizes.h>
465     { 0, }
466   #undef CACHE
467   };
```

Этот фрагмент кода инициализирует массив malloc_sizes [ ] и устанавливает поле cs_size в соответствии со значением, определенным в include/linux/ kernel_sizes. h. Как было сказано ранее, размер кеша может варьироваться от 32 байт до 131072 байт в зависимости от специальных настроек ядра[1].

Когда глобальные переменные заняли свои места, ядро начинает инициализацию выделителя секций с помощью вызова kmem_cache_init () из init/main.c[2]. Эта функция берет на себя заботу об инициализации последовательности кешей, его семафора, общего кеша, кеша kmem_cache - короче говоря, всех глобальных переменных, используемых выделителем секций для управления секциями. В этом месте создаются специализированные кеши. Для создания кешей используется функция kmem_cache_create ().

4.5.2 Создание кеша

Создание кеша включает три шага:

1. Выделение и инициализация описателя.
2. Расчет раскраски секции и размера объекта.
3. Добавление кеша в список cache_chain.

[1] Существует несколько настроек, при которых размер общего кеша становится большим, чем 131072. Более подробную информацию можно увидеть в include/linux/kmalloc_sizes .h.

[2] Гл. 9 описывает процесс инициализации начиная с загрузки. Мы увидим, как kmem_cache_init () встраивается в процесс загрузки.

Общие кеши устанавливаются во время инициализации с помощью kmem_cache_ init () (mm/slab.c). Специализированные кеши создаются с помощью вызова kmem_cache_create ().

Рассмотрим каждую из этих функций.

4.5.2.1 kmem_cache_init()

Здесь создаются cache_chain и общие кеши. Эта функция вызывается во время про цесса инициализации. Обратите внимание, что функция имеет приставку___ init перед именем функции. Как сказано в гл. 2, «Исследовательский инструментарий», это означает, что функция загружается в память, освобождаемую по завершении процесса загрузки.

```
mm/slab.c
659  void  init kmem cache init (void)
660  {
661   size t left over;
662   struct cache_sizes *sizes;
663   struct cache names *names;

669   if (num_physpages > (32 « 20) » PAGE_SHIFT)
670    slab  break gfp order = BREAK GFP ORDER HI;
671
672
```

Строки 661-663

Переменные размеров и имен головы массива для выделенного массива kmalloc (общих кешей с геометрически рассчитываемым размером). В этой точке массивы находятся в области данных __init. Обратите внимание, что kmalloc () в этой точке не завершается; kmalloc () использует массив malloc_sizes, который к этому моменту уже должен быть заполнен. В этой точке у нас есть статически выделенный описатель cache_cache.

Строки 669-670

Этот блок кода определяет количество используемых секцией страниц. Количество страниц, которое может использовать секция, определяется количеством доступной памяти. Как на x86, так и на PPC переменная PAGE_SHIFT (include/asm/ page. h) устанавливается в 12. Поэтому мы проверяем, может ли num_physpahes хранить значение больше 8 Кб. Это верно, если компьютер оборудован более чем 32 Мб памяти. Если это так, мы можем вместить в каждую секцию BREAK_ GFP_ORDER_HI страниц. В противном случае для каждой секции выделяется одна страница.

```
mm/slab.c
690   init MUTEX(&cache chain sem) ;
691   INIT _ LIST_HEAD(&cache_chain);
692   list  add(&cache cache.next, &cache chain);
693   cache cache.array[smp processor id()] = &initarray cache.cache;
694
695   cache estimate(0, cache cache.objsize, 0,
696     &left_over, &cache_cache.num) ;
697   if ( !cache cache.num)
698    BUGO;
```

Строка 690

Эта строка инициализирует cache__chain семафора cache_chain_sem.

Строка 691

Инициализация списка cache_chain, где хранятся все описатели кешей.

Строка 692

Добавление описателя cache__cache в список cache_chain.

Строка 693

Создание кешей для каждого из процессоров. (Подробности выходят за рамки рассмотрения данной книги.)

Строки 695-698

Этот блок проверяет, чтобы хотя бы один из описателей кеша мог быть выделен в cache_cache. Кроме этого, он устанавливает поле num описателя cache_ cache и рассчитывает количество оставшегося пространства. Это используется для раскрашивания секции.

Раскрашивание секции (slab coloring) - это метод, с помощью которого ядро уменьшает возникновение ситуаций, связанных с выравниванием, которые сказываются на производительности.

mm/slab.c

```
705   sizes = malloc_sizes;
706   names = cache names;
707
708      while (sizes->cs size) {

714    sizes->cs_cachep = kmem_cache__create (
715     names->name, sizes->cs_size,
```

```
716      0, SLAB HWCACHE ALIGN, NULL, NULL);
717     if (!sizes->cs cachep)
718      BUG();

726    sizes->cs dmacachep = kmem cache create(
727      names->name dma, sizes->cs size,
728      0, SLAB CACHE DMA|SLAB HWCACHE ALIGN, NULL, NULL);
729     if (!sizes->cs_dmacachep)
730    BUG();
731
732     sizes++;
733     names++;
734    }
```

Строка 708

Эта строка проверяет, достигли ли мы конца массива размеров. Последний элемент массива размеров всегда равен 0. Следовательно, условие является истиной до тех пор, пока мы не достигнем последней ячейки массива.

Строки 714-718

Создание следующего, kmalloc_cache кеша для нормального выделения и проверка того, чтобы он не был пустым. (См. подраздел «kmem_cache_create ()».)

Строки 726-730

Этот блок создает кеши для выделения из DMA.

***Строки** 732-733*

Переход к следующему элементу в массивах размеров и имен.

Оставшаяся часть функции kmem_cache_init () обрабатывает перемещение временных данных загрузки для выделенных с помощью kmalloc данных. Мы пропустим объяснение этой части, так как она напрямую не связана с инициализацией описателей кеша.

4.5.2.2 kmem_cache_create()

Бывают моменты, когда регионов памяти, предоставляемых общим кешем, недостаточно. Эта функция вызывается, когда необходимо создать специализированный кеш. Шаги, необходимые для создания специализированного кеша, не особенно отличаются от шагов, необходимых для создания кеша общего назначения: создание, выделение и инициализация описателя кеша, выравнивание объектов, выравнивание описателя секции и добавление кеша в последовательность кешей. Эта функция не имеет приставки __ init перед своим именем, потому что во время ее вызова уже доступна постоянная память:

```
mm/slab.c
1027  kmem cache t *
1028  kmem_cache_create (const char *name, size_t size, size_t offset,
1029   unsigned long flags, void (*ctor)(void*, kmem cache  t *,
                                    unsigned long),
1030   void (*dtor)(void*, kmem_cache_t *, unsigned long))
1031  {
1032   const char *func  nm = KERN ERR Mkmem create:
1033   size t left  over, align, slab size;
1034   kmem_cache_t *cachep = NULL;
```

Давайте рассмотрим параметры kmem__cache_create.

name

Это имя, используемое для идентификации кеша. Оно сохраняется в поле name описателя кеша и отображается в /proc/slabinf o.

size

Этот параметр определяет размер (в байтах) объектов, содержащихся в кеше. Это значение сохраняется в поле obj size описателя кеша.

offset

Это значение определяет местоположение объектов в странице.

flags

Параметр flags связан с секцией. Описание поля flags описателя кеша вы можете найти в табл. 4.4.

ctor u dtor

ctor и dtor - это соответственно конструктор и деструктор, вызываемые при создании или уничтожении объектов в памяти региона.

Следующая функция выполняет отладку размеров и проверку того, что мы еще не рассматривали. Рассмотрим подробнее ее код.

```
mm/slab.c
1079   /* Получение объекта описателя кеша */
1080   cachep = (kmem__cache_t *) kmem_cache_alloc(&cache cache,
                                                   SLAB_KERNEL) ;
1081   if(!cachep)
1082    goto opps;
1083   memset (cachep, 0, sizeof (kmem__cache_t) ) ;

1144   do {
```

```
1145    unsigned int break flag = 0;
1146   cal_was tage:
1147    cache_.estimate(cachep->gfporder, size, flags,
1148         &left_over, &cachep->num);

1174                    } while (1);
1175
1176   if (!cachep->num) {
1177    printk( "kmem cache  create: couldn't create cache %s.\n", name);
1178    kmem cache free(&cache cache, cachep);
1179    cachep = NULL;
1180    goto opps;
1181   }
```

Строки 1079-1084

Здесь выделяется описатель кеша. Следом идет часть кода, ответственная за выравнивание объектов в секции. Мы оставляем эту часть кода за рамками нашего обсуждения.

Строки 1144-1174

Здесь определяется количество объектов в кеше. Основную работу выполняет cache_estimate (). Вспомните, что полученное значение будет храниться в поле num описателя кеша.

```
mm/slab.c
1201   cachep->flags = flags;
1202   cachep->gfpflags = 0;
1203   if (flags & SLAB CACHE DMA)
1204    cachep->gfpflags |= GFP_DMA;
1205   spin lock init(&cachep->spinlock);
1206   cachep->objsize = size;
1207   /* NUMA */
1208   INIT LIST HEAD(&cachep->lists.slabs full);
1209   INIT LIST  HEAD(&cachep->lists.slabs partial);
1210   INIT LIST HEAD(&cachep->lists.slabs free);
1211
1212   if (flags & CFLGS OFF SLAB)
1213    cachep->slabp cache = kmem find general  cachep(slab  size, 0) ;
1214   cachep->ctor = ctor;
1215   cachep->dtor = dtor;
1216   cachep->name = name;

1243   cachep->lists.next_reap = jiffies + REAPTIMEOUT__LIST3 +
1244       ((unsigned long)cachep)%REAPTIMEOUT_LIST3;
```

```
1245
1246   /* Для доступа к последовательности нужен семафор. */
1247   down(&cache chain sem);
1248   {
1249    struct list_head *p;
1250    mm segment t old fs;
1251
1252    old_fs = get_fs();
1253    set fs(KERNEL DS);
1254    list_for_each(p, &cache_chain) {
1255     kmem cache t *pc = list entry(p, kmem cache t, next) ;
1256     char tmp;

1265     if (!strcmp(pc->name,name)) {
1266       printk( "kmem  cache  create: duplicate cache %s\n",name);
1267       up (Sccache  chain  sem) ;
1268                 BUG();
1269     }
1270    }
1271    set_fs(old_fs);
1272    }
 1273
1274    /* настройка кеша завершена,  связывание кеша со  списком */
1275    list_add(&cachep->next, &cache_chain);
1276    up(&cache_chairL_sem) ;
1277   opps:
1278    return cachep;
1279   }
```

Сразу перед этим секция выравнивается по аппаратному кешу и раскрашивается. Заполняются поля описателя кеша color и color_of f.

Строки 1200-1217

Этот блок кода инициализирует поля описателя кеша практически аналогично тому что мы видели в kmem_cache_init ().

Строки 1243-1244

Установка времени следующего уничтожения.

Строки 1247-1276

Описатель кеша инициализируется, а вся связанная с кешем информация рассчитывается и сохраняется. Теперь мы можем добавить новый описатель кеша в список cache__chain.

4.5.3 Создание секции и cache_grow()

При создании кеша в нем не содержатся секции. На самом деле секции не выделяются до поступления запроса на объект, для которого нужна новая секция. Это происходит, когда поля описателя кеша lists . slabs_partial и lists . slabs__f гее пусты. Здесь мы не будем рассматривать, как запрос памяти преобразуется в запрос объекта внутри определенного кеша. Теперь мы будем считать, что это преобразование совершено, и сконцентрируемся на технической реализации внутри выделителя секции.

Секция внутри кеша создается с помощью cache_grow (). При создании секции мы не просто выделяем и инициализируем его описатель, но и выделяем настоящую память. Для этого нам необходим интерфейс для запроса страницы у системы близнецов. Это выполняется с помощью kmem_getpages () (mm/slab. c).

4.5.3.1 cache_grow()

Функция cache_grow() увеличивает количество секций в кеше на 1. Она вызывается только тогда, когда в кеше не осталось доступных объектов. Это происходит, когда lists . slabs_partial и lists . slabs_f гее являются пустыми.

```
mm/slab.c
1546  static int cache grow (kmem cache t * cachep, int flags)
1547   {
```

В функцию передаются следующие параметры:

- **cachep**. Это описатель для увеличиваемого кеша.
- **flags**. Флаги, связанные с созданием секции.

mm/slab.c

```
1572  check_irq_off();
1573  spin_lock(&cachep->spinlock) ;

1582   spin_unlock(&cachep->spinlock);
1583
1584   if (local flags &   GFP WAIT)
1585    local_irq_enable();
```

***Строки** 1572-1573*

Подготовка к манипуляции с полями описателя кеша с помощью блокировки прерываний и блокировки описателя.

Строки 1582-1585

Снятие блокировки с описателя кеша и повторная активизация прерываний.

mm/slab.c

```
1597    if (!(objp = kmem_getpages(cachep, flags)))
1598     goto failed;
1599
1600    /* Получение контроля над секцией. */
1601    if ('(slabp = alloc_slabmgmt (cachep, objp, offset, local_flags)))
1602     goto oppsl;

1605    i=1« cachep->gfporder;
1606    page = virt to page (objp);
1607    do (
1608      SET_PAGE_CACHE(page,  cachep);
1609      SET_PAGE_SLAB  (page,  siabp);
1610      SetPageSlab(page);
1611      inc_page_state(nr_slab);
1612      page++;
1613     }  while  (—i)  ;
1614
1615    cache__init__objs (cachep,  slabp,  ctor_flags) ;
```

Строки 1597-1598

Интерфейс с системой близнецов для получения страницы(страниц) для секции.

Строки 1601-1602

Место, в которое должен попасть описатель секции. Вспомните, что описатель секции может быть сохранен в самой секции или в первом кеше общего назначения.

Строки 1605-1613

Страницы необходимо ассоциировать с кешем и описателем секции.

Строка 1615

Инициализация объектов в секции.

```
mm/slab.c
1616    if (local flags &   GFP WAIT)
1617     local irq disable();
1618    check i rg o f f();
1619    spin  lock(&cachep->spinlock) ;
1620
```

```
1621    /* Сделать секцию активной. */
1622    list_add_tail(&slabp->list, &(list3_data(cachep)->slabs_free) ) ;
1623    STATS_INC_GROWN(cachep);
1624    list3_data(cachep)->free_objects += cachep->num;
1625    spin_unlock(&cachep->spinlock);
1626    return 1;
1627   oppsl:
1628    kmem_freepages(cachep, objp);
1629   failed:
1630    if (local_flags & __GFP_WAIT)
1631     local_irq_disable();
1632    return 0;
1633   }
```

Строки 1616-1619

Так как нам необходимо получить доступ и изменить поля описателя, нам нужно отключить прерывания и заблокировать данные.

Строки 1622-1624

Добавление нового описателя секции в поле lists. slabs_f ree описателя кеша. Обновление статистики, следящей за размером.

Строки 1625-1626

Разблокирование кольцевой блокировки и возвращение после удачного завершения.

Строки 1627-1628

Вызываются, если что-то во время обработки запроса страницы пошло не так. Обычно страница освобождается.

Строки 1629-1632

Включение прерываний обратно, чтобы они могли проходить и обрабатываться.

4.5.4 Уничтожение секции: возвращение памяти и kmem_cache_destroy()

И кеш и секция могут быть уничтожены. Кеши могут быть уменьшены или уничтожены для возвращения памяти в пул свободной памяти. Ядро вызывает эти функции, когда остается мало памяти. В любом случае секция уничтожается, а связанные с ней страницы возвращаются в систему близнецов для повторного использования. Функция kmem_cache_destroy() избавляется от кеша. Рассмотрим эту функцию подробнее. Кеши можно уничтожить и уменьшить с помощью kmem_cache__reap () (mm/ slab, c) и kmem_cache_shrunk() (mm/slab.c) соответственно. Для взаимодействия с системой близнецов существует функция kmem_f reepages () (mm/slab. c).

4.5.4.1 kmem_cache_destroy()

Существует несколько случаев, в которых необходимо убрать кеш. Динамически загружаемые модули (принимая во внимание, что постоянная память в загрузке и освобождении памяти не учувствует), создающие кеши, должны уничтожаться для освобождения памяти и для того, чтобы убедиться, что кеши не будут дублироваться при следующей загрузке модуля. Таким образом, специализированные кеши обычно уничтожаются именно таким образом.

Шаги для уничтожения кеша в обратном порядке повторяют шаги по его созданию. При уничтожении не нужно беспокоиться о выравнивании, а только об удалении описателя и освобождении памяти. Шаги по удалению кеша следующие:

1. Удаления кеша из последовательностей кешей.
2. Удаление описателя секции.
3. Удаление описателя кеша.

```
mm/slab.c
1421  int kmem cache destroy (kmem cache t * cachep)
1422  {
1423   int i;
1424
1425   if (!cachep || in  interrupt () )
1426    BUG();
1428    /* Поиск кеша в последовательности кешей. */
1429    down(&cache__chain_sem) ;
1430   /*
1431   * последовательность никогда не бывает пустой,
1432   * а cache_cache никогда не уничтожается  */
1433   list del(&cachep->next);
1434   up(&cache chain sem);
1435
1436   if( _ cache_shrink (cachep)) {
1437    slab error(cachep, "Can't free all objects");
1438    down(&cache  chain  sem) ;
1439    list_add(&cachep->next, &cache_chain);
1440    up(&cache chain sem);
1441    return 1;
1442   }

1450      kmem cache free(&cache cache, cachep);
1451
1452   return 0;
1453  }
```

Параметр функции cache является указателем на описатель кеша, который будет уничтожен.

Строки 1425-1426

Эта проверка существует для того, чтобы удостовериться, что прерывания неактивны и описатель кеша не равен NULL.

Строки 1429-1434

Получение семафора cache_chain, удаление кеша из очереди кешей и освобождение семафора последовательности кешей.

Строки 1436-1442

Здесь выполняется основная работа, связанная с освобождением места неиспользуемых секций. Если функция __ cache_shrink () возвращает true, это означает, что в кеше еще остались секции и поэтому он не может быть уничтожен. Поэтому мы отменяем предыдущий шаг и повторно вставляем описатель кеша в cache_chain после повторного получения семафора cache_chain и освобождаем, когда закончим.

Строка 1450

Заканчиваем освобождение описателя кеша.

4.6 Путь запроса памяти

До сих пор мы подходили к описанию выделителя секции так, как будто он не зависит от настоящих запросов к памяти. За исключением функции инициализации кеша мы не рассматривали, как связаны вызовы всех этих функций. Теперь мы рассмотрим поток управления, связанный с требованием памяти. Когда ядру нужно получить память группами с размером, указанным в байтах, используется функция kmalloc (), вызывающая, в свою очередь, kmem_getpages следующим образом:

kmalloc () -> __ cache__alloc () ->kmem_cache_grow() ->kmem_getpages ()

4.6.1 kmalloc()

Функция kmalloc () выделяет объекты памяти в ядре.

```
mm/slab.c
2098  void *  kmalloc (size_t size, int flags)
2099   {
2100    struct cache sizes *csizep = malloc sizes;
2101
2102    for (; csizep->cs_size; csizep++) {
2103     if (size > csizep->cs_size)
```

```
2104     continue;
2112     return __cache_alloc( flags & GFP__DMA ?
2113       csizep->cs  dmacachep : csizep->cs cachep, flags);
2114   }
2115   return NULL;
2116  }
```

4.6.1.1 size Количество

требуемых байтов.

4.6.1.2 flags

Означает тип требуемой памяти. Эти флаги передаются в систему близнецов для изменения поведения kmalloc (). Табл. 4.6 демонстрирует флаги, которые подробно описаны в подразделе «Система близнецов».

Строки 2102-2104

Поиск первого кеша с объектами больше требуемого размера.

Строки 2112-2113

Выделение объекта из зоны памяти, указанной в параметре flags.

4.6.2 kmem_cache_alloc()

Это функция-обертка для __cache_alloc (). Она не обладает никакой дополнительной функциональностью, а только передает параметры:

```
ram/slab.c
2070 void * kmem cache alloc (kmem cache t *cachep, int flags) 2071  {
2072   return   cache alloc(cachep, flags);
2073  }
```

4.6.2.1 cachep

Параметр cachep - это описатель кеша, из которого мы хотим выделить объект.

4.6.2.2 flags

Тип требуемой памяти. Передаются прямо, как указано в kmalloc ().

Для того чтобы освободить память байтового размера, выделенную с помощью kmalloc (), ядро предоставляет интерфейс kf ree (), получающую в качестве параметра указатель на память, возвращаемый kmalloc (). Рис. 4.9 демонстрирует передачу управления от kf ree к kmem_f reepages.

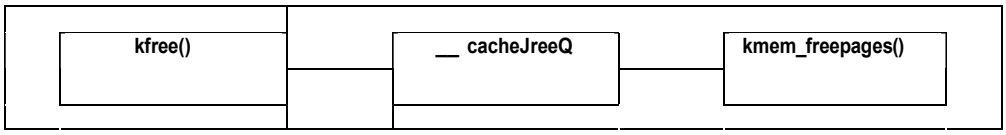


Рис. 4.9. Граф вызова kfree

4.7 Структуры памяти процесса в Linux

До сих пор мы описывали, как ядро распоряжается собственной памятью. Теперь мы обратим наше внимание на программы пользовательского пространства и на то, как ядро управляет памятью программ. Чудеса виртуальной памяти позволяют процессам пользовательского пространства работать с такой же эффективностью, как если бы им была доступна вся память. В реальности ядро управляет тем, что следует загрузить, как это загружать и что делать с загруженным дальше. Все, о чем мы рассуждали до сих пор в этой главе, связано с тем, как ядро управляет памятью и обеспечивает полную прозрачность для программ пользовательского пространства.

Во время создания процесса пользовательского пространства ему назначается виртуальное адресное пространство. Как говорилось ранее, виртуальное адресное пространство процесса представляет собой диапазон ^сегментированных линейных адресов, которые может использовать процесс. Размер диапазона определяется размером регистров системной архитектуры. Большинство систем обладает 32-битовым адресным пространством, с другой стороны, существуют системы, например G5, которые обладают 64-битовым адресным пространством.

Во время создания диапазон адресов процесса может расширяться или сужаться с помощью добавляемых или убираемых интервалов адресов (address intervals) соответственно. Интервал адресов (диапазон адресов) не определяется до тех пор, пока не будет вызван **регион памяти (memory region)** или **область памяти (memory area).** Это полезно при разделении диапазона адресов процесса на несколько зон разных типов. Эти разные типы обладают собственными схемами защиты или характеристиками. Схемы защиты памяти процесса связаны с контекстом процесса. Например, некоторые части программного кода помечаются как доступные только для чтения (текст), тогда как другие являются перезаписываемыми (переменные) или исполнимыми (инструкции). Кроме этого, отдельные процессы могут получать доступ только к тем областям памяти, которые им принадлежат.

Внутри ядра адресное пространство процесса вместе со всей связанной с ним информацией хранится в описателе iran_s truct. Вы можете вспомнить из гл. 3, «Процессы: принципиальная модель выполнения», что эти структуры связаны с task_struct процесса. Область памяти представляется описателем vm_area_struct. Каждый описатель области памяти описывает представляемый им последовательный интервал адресов. На протяжении этого раздела мы будем называть описатель интервала адресов описате-

лем области памяти или vms_area_struct. Теперь рассмотрим mm_struct и vm_area_struct. Рис. 4.10 иллюстрирует связь между этими двумя структурами.

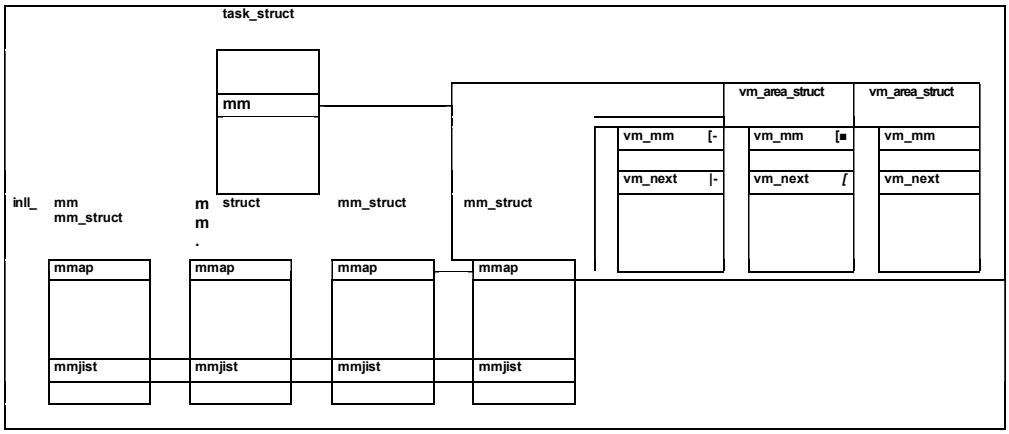


Рис. 4.10. Структуры памяти, связанные с процессом

4.7 A mm_struct

Каждая задача имеет структуру mm__struct (include/linux/sched.h), используемую ядром для представления диапазона адресов. Все описатели mm_struct хранятся в дву связном списке. Головой списка является структура mm_struct, соответствующая процессу 0, которым является процесс ожидания. Доступ к этому описателю можно получить через глобальную переменную init_mm:

```
include/linux/sched.h
185   struct ram_struct   {
186   struct vm__area_struct   *  mmap;
187   struct  rb_root mm_rb;
188   struct wi_area_struct   *  mmap_cache;
189   unsigned  long  free_area_cache;
190   pgd_t   *  pgd;
191   atomic_t mra_users;
192   atomic_t mm_count;
193   int map count;
194   struct rw seraaphore mmap sem;
195   spinlock t page table lock 196
197  struct list_head mmlist;

2 02  unsigned long start.jcode, end_code, start_data, end_data;
```

```
203  unsigned long start_brk, brk, start_stack;
2 04  unsigned long arg start, arg end, env start, env end;
2 05  unsigned long rss, total vm, locked vm;
2 06  unsigned long def flags,-
2 07  cpuma sk t cpu vm ma s k;
2 08  unsigned long swap_address;

228  };
```

4.7.1.1 mmap

Описатель области памяти (определенный в следующем подразделе), назначаемый процессу, помещается в список. Доступ к этому списку можно получить через поле mmap mm__struct. Перемещаться по списку можно с помощью поля vm_next каждой из структур vma_area_struct.

4.7.1.2 mm_rb

Простой связанный список позволяет легко перемещаться по всем описателям областей памяти, соответствующим определенным процессам. Тем не менее, если ядро ищет описатель области памяти конкретного процесса, простой список не обеспечивает достаточной скорости поиска. Поэтому структуры, связанные с диапазоном адресов конкретного процесса, также хранятся в красно-белом дереве, доступ к которому можно получить через поле mm_rb. Таким образом, увеличивается скорость поиска ядром конкретного описателя области памяти.

4.7.1.3 mmap_cache

mmap_cache - это указатель на последнюю область памяти, связанную с процессом. **Принцип локальности (principle of locality)** предполагает, что при обращении к области памяти, наиболее часто используемые области памяти расположены ближе. При этом высока вероятность, что требуемый в данный момент адрес принадлежит той же области памяти, которой принадлежит и предыдущий затребованный адрес. Вероятность того, что текущий адрес находится в запрашиваемой перед этим области, составляет примерно 35 процентов.

4.7.1.4 pgd

Поле pgd - это указатель на глобальную директорию страницы, которая хранит запись для этой области памяти. В mm_struct для ожидающего процесса (процесс 0) это поле указывает на swapper_pg_dir. (См. более подробную информацию в разд. 4.9 о том, на что указывают поля этой структуры.)

4.7.1.5 mm_user

Поле mm_struct хранит количество процессов, получивших доступ к этой области памяти. Легковесные процессы или потоки разделяют некоторые интервалы адресов и областей памяти. Поэтому mm_struct для потоков обычно имеют поле mm_users со значением больше 1. Этим полем можно управлять с помощью атомарных функций atomic_set() , atomic_dec_and__lock() , atomic_read() и atomic_inc ().

4.7.1.6 mm_count

mm_count - это количество использований mm_s truct. При определении возможности освобождения этой структуры делается проверка данного поля. Если оно содержит значение 0, то эта структура не используется процессами и, следовательно, ее можно освободить.

4.7.1.7 map_count

Поле map_count хранит количество областей памяти или описателей vma_area_ struct в адресном пространстве процесса. Каждый раз, когда в адресное пространство процесса добавляется новая область памяти, это поле увеличивается одновременно со вставкой vms_area_struct в список mmap и дерево mm_rb.

4.7.1.8 mmjist

Поле iran_list типа list_head хранит адреса соседних mm_struct в списке описателей памяти. Как говорилось ранее, голова списка указывает на глобальную переменную init_mm, являющуюся описателем процесса 0. При работе с этим списком mmlist_ lock защищает эту структуру от постороннего доступа.

Следующие 11 полей описывают работу с различными типами областей памяти, выделяемыми для процесса. Вместо того чтобы пускаться в подробное объяснение их отличий от ранее описанных структур, связанных с процессом, мы ограничимся общим описанием.

4.7.1.9 start_code и end_code

Поля start_code и end_code хранят начальный и конечный адреса блока кода региона процессорной памяти (т. е. выполняемый текстовый сегмент).

4.7.1.10 start_data и end_data

Поля start_data и end_data содержат начальный и конечный адреса инициализированных данных (которые находятся в части . data исполняемого файла).

4.7.1.11 stackj>rk и brk

Поля stack_brk и brk хранят начальный и конечные адреса кучи процесса.

4.7.1.12 stack_stack

stack_stack является начальным адресом стека процесса.

4.7.1.13 arg_start и arg_end

Поля arg_start и arg__end хранят начальный и конечный адреса аргументов, передаваемых процессу.

4.7.1.14 env_start и env_end

Поля env_start и env_end хранят начальный и конечный адреса раздела переменных окружения.

Это касается поля mm_struct, которое мы рассматриваем в этой главе. Теперь посмотрим на поля описателя области памяти, vm_area_struct.

4.7.2 vm_area_struct

Структура vm_area_s true t определяет регион виртуальной памяти. Процесс обладает несколькими регионами памяти, на каждый из которых приходится по одной структуре vm_area_strue t:

```
include/linux/mm.h
51  struct vm_area_struct {
52   struct mm  struct * vm mm;
53   unsigned long vm start;
54   unsigned long vm end;

57   struct vm_area_struct *vm__next;

60  unsigned long vm  flags;
61
62   struct rbjiode vm_rb;

72   struct vm_operations_struct * vm_ops;

};
```

4.7.2.1 vmjnm

Все регионы памяти принадлежат к адресному пространству, связанному с процессом и представляемому mm_struct. Структура vm_iran указывает на структуру типа mm_struct, описывающую адресное пространство, к которому принадлежит данная область памяти.

4.7.2.2 vm_start и vm_end

Регионы памяти связаны с интервалами адресов. В vm_area_struct этот интервал определяется таким образом, чтобы следить за начальным и конечным адресом интервала. В целях увеличения производительности начальный адрес региона памяти должен быть кратным размеру фрейма страницы. Ядро следит, чтобы фреймы страниц заполнялись данными из определенного региона памяти, размер которого также является кратным размеру фрейма страницы.

4.7.2.3 vrrwiext

Поле vm_next указывает на следующую vm_area_struct в связанном списке, содержащем все регионы в адресном пространстве процесса. Голова этого списка определена для адресного пространства в поле mmap структуры nim__struct.

4.7.2.4 vmjlags

Внутри этого интервала регион памяти связан с описывающими его характеристиками. Они хранятся в поле vm_f lags и применяются к страницам в регионе памяти. Табл. 4.6 описывает доступные флаги.

Таблица 4.6. Значения vm_area_struct->vmjlags

Флаг	Описание
VM_READ	Страницы в этом регионе доступны для чтения
VM_WRITE	Страницы в этом регионе доступны для записи
VM_EXEC	Страницы в этом регионе могут быть выполнены
VM_SHARED	Страницы в этом регионе могут быть доступны сразу для нескольких процессов
VM_GROWSDOWN	Линейные адреса добавляются начиная с нижнего
VM_GROWSUP	Линейные адреса добавляются начиная с верхнего
VM_DENYWRITE	Запись на страницы запрещена
VM_EXECUTABLE	Страницы в этом регионе состоят из исполнимого кода
VM_LOCKED	Страницы блокированы
VM_DONTCOPY	Страницы нельзя скопировать
VM DNTEXPAND[a]	Эта виртуальная область памяти не может быть расширена

a.В оригинальном тексте опечатка - должно быть VM_DONTEXPAND. *Примеч. науч. ред.*

4.7.2.5 vm_rb

vm_rb хранит узел красно-черного дерева, связанного с областью памяти.

4.7.2.6 vm_ops

vm_ops состоит из структуры указателей функций, обрабатывающих отдельные vm_area_struct. Эти функции включают открытие области памяти, закрытие и отмену отображения. Кроме этого, указатель функции на функцию вызывается каждый раз, когда возникает исключение об отсутствии страницы.

4.8 Размещение образа процесса и линейное адресное пространство

Когда в память загружается программа пользовательского пространства, она получает линейное адресное пространство, разделенное на области памяти (memory_areas), или сегменты. Эти сегменты выполняют различные функции при выполнении процесса. Функционально разделенные сегменты отображаются в адресное пространство процесса. С выполнением процесса связано 6 главных сегментов.

- TeKCT(text). Этот сегмент, также известный как сегмент кода, хранит исполнимые инструкции программы. Поэтому он обычно имеет атрибуты execute и read. В случае, когда из одной программы может быть загружено много процессов, загружать одни и те же инструкции несколько раз слишком расточительно. Linux позволяет нескольким процессам разделять в памяти текстовые сегменты. Поля start_code и end_code структуры mm_struct хранят адреса начала и конца текстового сегмента.
- Данные(cШа). Этот сегмент хранит все инициализированные данные. Инициализированные данные включают статически выделенные и глобальные инициализированные данные. Следующий фрагмент кода демонстрирует пример инициализированных данных.

```
example.c
int gvar = 10;
int main(){

}
```

- gvar. Глобальная переменная, которая инициализируется и хранится в сегменте данных. Эта секция имеет атрибуты чтения-записи, но не может быть разделена между несколькими процессами, выполняющими одну и ту же программу. Поля

start_data и end_data структуры mm_struct хранят адреса начала и конца сегмента данных.

- BSS. Эта секция хранит неинициализированные данные. Эти данные состоят из глобальных переменных, инициализируемых в 0 при запуске программы. Также эта секция называется секцией инициализируемых нулем данных. Следующий фрагмент кода демонстрирует неинициализируемые данные.

```
example2.c int
gvarl[10];
long gvar2;
int main(){

}
```

Объекты в этой секции имеют только атрибуты name и size.

- **Куча (Heap).** Используется для наращивания линейного адресного пространства процесса. Когда программа использует mall oc () для динамического получения памяти, эта память выделяется из кучи. Поля start_brk и brk структуры mm_struct хранят адреса начала и конца кучи. При вызове mall oc () для динамического получения памяти вызов системного вызова sys_brk () перемещает указатель brk на новую позицию, увеличивая при этом кучу.
- **Стек (Stack).** Хранит все выделяемые локальные переменные. Когда вызываются функции, локальные переменные для этих функций помещаются в стек. Как только функции завершаются, связанные с ними переменные извлекаются из стека. Остальная информация, включая адреса возврата и параметры, также размещается в стеке. Поле start_stack структуры mm_struct помечает начальный адрес стека процесса.

Несмотря на то что с выполняемым процессом связано 6 сегментов, они отображаются только в 3 области памяти адресного пространства. Эти области памяти называются text, data и stack. Сегмент data включает инициализированный выполняемый сегмент data, bss и кучу. Сегмент text включает исполнимый сегмент text. Рис. 4.11 показывает, как выглядит линейное адресное пространство и как mm_struct следит за этими сегментами.

Различные области памяти отображаются в файловую систему /proc. К отображенной для процесса памяти можно получить доступ через вывод /proc/<pid>/maps. Теперь рассмотрим пример программы и посмотрим список адресов памяти в адресном пространстве процесса. Код в example3 .c показывает отображенную в память программу.

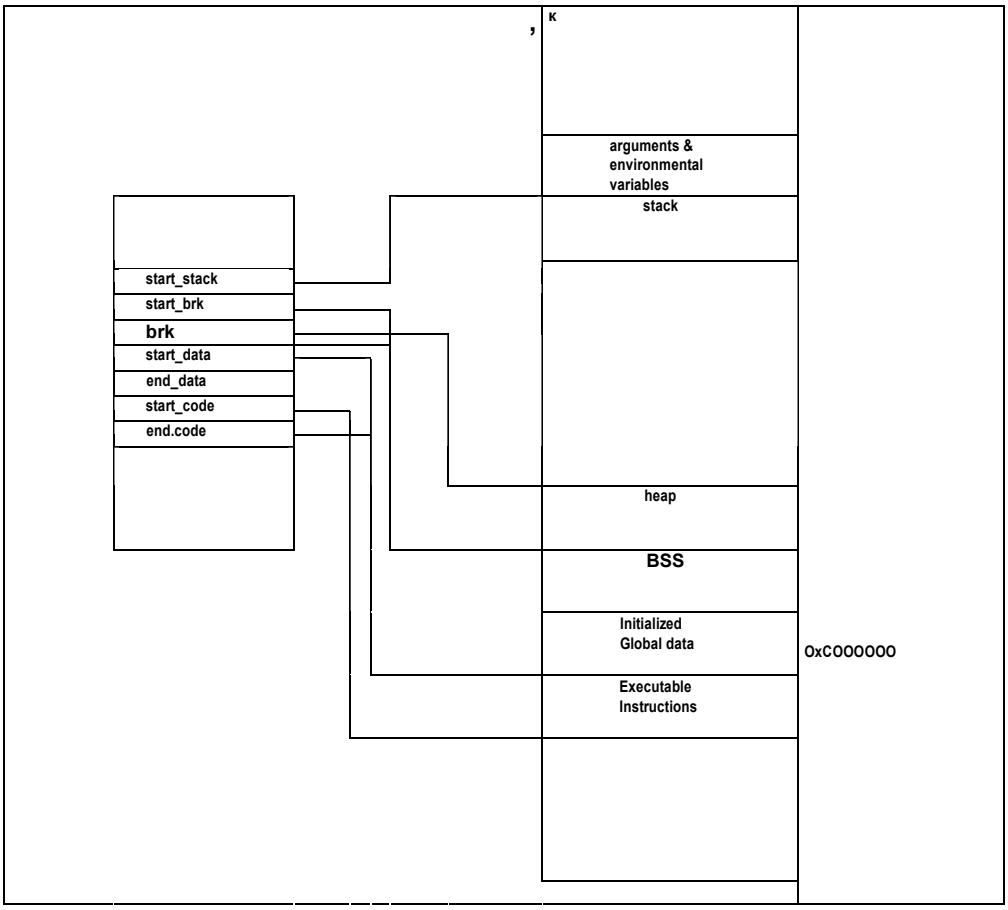


Рис. 4.11. Адресное пространство процесса

example3.c
```
#include <stdio.h>
int main(){
        while(1);
        return(O); }
```

Вывод /proc/<pid>/maps для этого **примера** содержит то, что представлено на рис. 4.12.

08048000-08049000	r-xp	00000000	03:05	1324039	/home/1 kp/example3
08049000-0804a000		00000000	03:05	1324039	/home/1 kp/example3
40000000-40015000	r-xp	00000000	03:05	767058	/lib/ld-2.3.2.so
40015000-40016000	rw-p	00014000	03:05	767058	/liMd-2.3.2.so
40016000-40017000	rw-p	00000000	00:00	0	
42000000-4212e000	r-xp	00000000	03:05	1011853	/lib/tls/libc-2.3.2.so
42126000-42131000	rw-p	00126000	03:05	1011853	/lib/tls/libc-2.3.2.so
42131000-42133000	rw-p	00000000	00:00	0	
bffeOOO-cOOOOOOO	rwxp	fffffOOO	00:00	0	

Рис. 4.12. /prod<pid>/maps

Самая левая колонка показывает диапазон сегментов памяти. Это начальные и конечные адреса отдельных сегментов. Следующая колонка показывает разрешение на доступ к этим сегментам. Эти флаги похожи на разрешения доступа к файлам: г означает чтение, w - запись, а x - возможность выполнения. Последним флагом может быть p, обозначающее частный (private) сегмент, или s, означающее разделяемый (shared) сегмент. (Частный сегмент не обязательно неразделяемый.); p указывает, что текущий сегмент не был разделен. Следующая колонка содержит отступ для сегмента. Четвертая колонка слева хранит два числа, разделенные двоеточием. Она означают максимальное и минимальное число файлов файловой системы, связанных с данным сегментом. (Некоторые сегменты не имеют ассоциированных с ними файлов, и для них эти значения будут равны 00:00.) Пятая колонка хранит inode файла, а самая правая колонка- имя файла. Для сегментов без имен файлов эта колонка остается пустой, а колонка inode равна 0.

В нашем примере первая строка содержит описание текстового сегмента нашей программы. Здесь можно увидеть устанавливаемые для исполнимых файлов флаги разрешений. Следующая колонка описывает сегмент данных нашей программы. Обратите внимание, что разрешения этой секции включают возможность записи.

Наша программа связывается динамически, что означает, что используемые ей функции принадлежат к загружаемой во время ее выполнения библиотеке. Эти функции необходимо отобразить в адресное пространство, чтобы к ним можно было получить доступ. Следующие 6 строк работают с динамически загружаемыми библиотеками. Следующие 3 строчки описывают text, data и bss библиотеки Id. За этими тремя строчками следует описание секций библиотеки test, data и bss в том же порядке.

Последняя строчка обладает разрешением на чтение, запись и выполнение, представляет стек процесса и расширяется до 0x0C0000000, т. е. до наибольшего адреса в памяти, доступного для процесса из пользовательского пространства.

4.9 Таблицы страниц

Память программы удобно представлять с помощью виртуальных адресов. Единственная возникающая проблема связана с тем, что, когда инструкции используются процессором, он ничего не может сделать с виртуальным адресом. Процессор оперирует физическими адресами. Связь между виртуальным и соответствующим ему физическим адресом обеспечивается ядром (с помощью аппаратного обеспечения) в таблицах страниц.

Таблицы страниц следят за памятью в элементах фреймов страниц. Они хранятся в оперативной памяти в течение всей жизни ядра. Linux использует так называемую трехуровневую схему подкачки. Трехуровневая подкачка необходима для того, чтобы даже 64-битовая архитектура смогла отобразить в виртуальную память все свои физические адреса. Как следует из названия схемы, трехуровневая система подкачки имеет 3 типа таблиц подкачки: таблицы верхнего уровня, называемые глобальной директорией страниц (Page Global Directory) (PGD), представляемой с помощью типа данных pgd_t; таблицы средней директории страниц (Page Middle Directory, PMD), представляемой с помощью типа данных pmd_t, и таблицы страницы (Page Table, PTE), представляемая в виде типа данных pte__t. Таблицы страниц изображены на рис. 4.13.

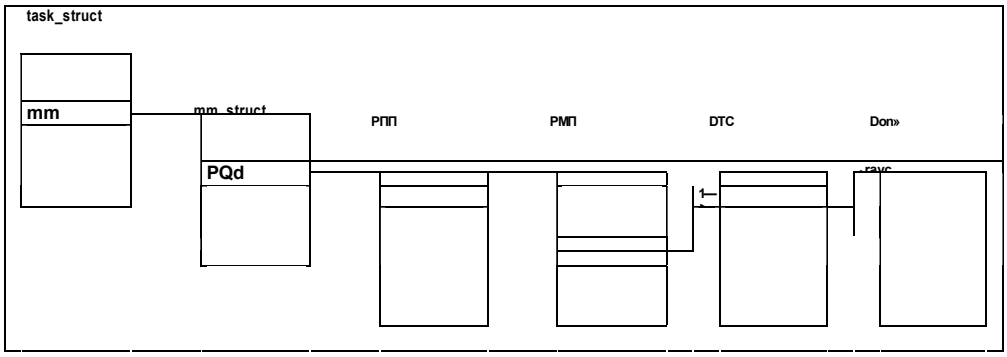


Рис. 4.13. Таблицы страниц в Linux

PGD хранит вхождения, связанные с PMD. PMD хранит вхождения, связанные с PTE, а PTE хранит вхождения ссылок на отдельные страницы. Каждая страница обладает собственным набором таблиц страниц. Поле mm_strue t->pgd хранит указатель на PGD процесса; 32- и 654-битовые виртуальные адреса разделяются (в зависимости от архитектуры) на поля отступа различного размера. Каждому полю соответствует отступ в PGD, PMD, PTE и в самой странице.

4.10 Ошибка страницы

На протяжении жизни процесса он может попытаться получить доступ к адресам, которые принадлежат адресному пространству, но не загружены в оперативную память. Вместо этого он может попытаться получить доступ к странице памяти, для которой у него нет разрешения на доступ (например, может попытаться произвести запись в область только для чтения). Когда это происходит, система генерирует **ошибку страницы (page fault).** Ошибка страницы - это обработчик исключения, обрабатывающий ошибки доступа программы к странице. Когда аппаратное обеспечение порождает исключение ошибки страницы, перехватываемое ядром, страница достается из хранилища. После этого ядро выделяет недостающую страницу.

На каждой архитектуре есть своя архитектурно-зависимая функция для обработки ошибки страницы. На x86 и PPC вызывается функция do__page_f ault (). Обработчик ошибки страницы x86 do_page__f ault (*regs, error_code) находится в /arch/ i3 8 6/mm/fault, c. Обработчик ошибки памяти PowerPC do_page_fault (*regs, address, error__code) находится в /arch/ppc/mm/fault. c. Они настолько похожи, что для того, чтобы понять функционирование версии для PowerPC, нам будет достаточно рассмотреть только вариант do_page__f ault () для x86.

Основная разница в обработке ошибки страницы этими двумя архитектурами проявляется на этапе сбора и сохранения информации об ошибке, предшествующем вызову do_page__f ault (). Для начала рассмотрим особенности обработки ошибки страницы на x86, а затем перейдем к функции do_page_fault (). После этого мы рассмотрим отличия версии для PowerPC.

4.10.1 Исключение ошибки страницы на x86

Обработчик ошибки страницы x86 do_page_f ault () вызывается в результате аппаратного прерывания 14. Это прерывание происходит, когда процессор обнаруживает, что верны следующие условия:

1. Подкачка включена, и в директории страницы очищен бит присутствия, или данный адрес нужен для элемента таблицы страницы.
2. Подкачка включена, и текущий уровень привилегий ниже, чем необходимо для доступа к затребованной странице.

При возникновении этого прерывания процессор сохраняет информацию двух видов:

1. Природу ошибки в нижних 4 битах слова, переданного в стек. [Бит 3 функцией do_page_f ault () не используется.] См. значения, соответствующие битам в табл. 4.7.
2. 32-битовый линейный адрес, породивший исключение в cr2.

Параметр regs функции do_page_fault () является структурой, содержащей системные регистры и параметр error_code, использующий 3-битовое поле для описания источника ошибки.

Таблица 4.7. error_code ошибки памяти

	Бит 2	Бит1	БитО
Value = 0	**Ядро**	**Чтение**	**Страница отсутствует**
Value = 1	**Пользователь**	**Запись**	**Ошибка защиты**

4.10.2 Обработчик ошибки страницы

На обеих архитектурах функция do_page__f ault () использует только что полученную информацию и выполняет одно из нескольких действий. Соответствующие фрагменты кода выполняют серию сложных проверок и заканчиваются одним из следующих случаев:

- с помощью handle_nim_f ault () находится адрес, вызвавший ошибку;
- выполняется дамп oops (no_context:) bad_page_f ault () для PowerPC;
- ошибка сегментации (bad_area:) bad_page_f ault () для PowerPC;
- вызывающей функции возвращается ошибка (f ixup).

```
arch/i38б/mm/fault.c
212  asmlinkage void do_page_fault{struct pt  regs *regs,
                                      unsigned long error  code)
213  {
214   struct task struct *tsk;
215   struct mm struct *mm;
216   struct vm_area_struct * vma;
217   unsigned long address;
218   unsigned long page;
219   int write;
220   siginfo t info;
221
222   /* получение адреса */
223   _asm_("movl %%cr2, %0" : "=r" (address));

232               tsk = current;
234         info.si_code = SEGV_MAPERR;
```

Строка 223

Адрес, по которому произошла ошибка страницы, сохраняется в управляющем регистре cr2. Выполняется чтение линейного адреса, и адрес присваивается локальной переменной.

Строка 232

Указатель на task_struct tsk устанавливается указывающим на текущую структуру task_struct.

Теперь мы готовы приступить к дальнейшему поиску адреса, где возникла ошибка страницы. Рис. 4.14 иллюстрирует работу следующих строк кода:

```
arch/i3 8б/mm/fault.c
246    if  (unlikely(address >= TASK_SIZE))  {
247      if  (!(error_code &  5 ) )
248      goto vmalloc_fault;

253    goto bad_area_.nosemaphore;
254    }

257    mm =  tsk->mm
```

Строки 246-248

Этот код проверяет, находится ли адрес, в котором произошла ошибка, в модуле ядра (т. е. в независимой области памяти). Адреса независимой области памяти обладают собственным линейным адресом >= TASK_SIZE. Если так, выполняется проверка того, что биты 0 и 2 error_code не установлены. Вспомните табл. 4.7, которая демонстрировала ошибки, возникающие при попытке доступа к несуществующей странице ядра. Если условие выполнилось, значит, в ядре произошла ошибка страницы и вызывается код в метке vmalloc__f ault:.

Строки 253

Если мы сюда попали, это означает, что, несмотря на то что доступ выполнялся к независимой области памяти, он произошел в пользовательском режиме и вызвал ошибку защиты или оба эти события. В этом случае мы переходим к метке bad_area__semaphore:.

Строки 257

Устанавливает локальную переменную mm таким образом, чтобы она указывала на описатель памяти текущей задачи. Если текущая задача является потоком ядра, ее значением будет NULL. Это происходит в следующих строчках кода.

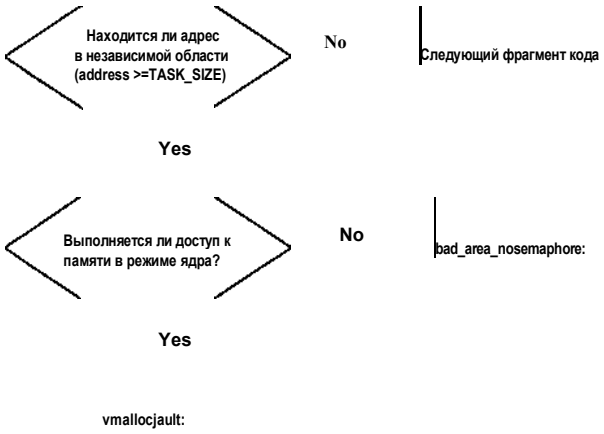


Рис. 4.14. Ошибка страницы 1

В этом месте мы определяем, что ошибка страницы не произошла в независимой области памяти. Рис. 4.15 иллюстрирует работу следующих строк кода:

arch/i386/mm/fault.c

```
262  if (in_atomic() || !ram)
2 63  goto bad__area_nosemaphore;

       down_read(&mm->mmap_sem) ;

       vma = find_vma(mm, address);
       if (!vma)
         goto bad area; if (vma-
       >vm_start <= address)
         goto good area; if (!(vma-
       >vm_flags & VM_GROWSDOWN))
         goto bad area; if
       (error_code & 4) {
264
265
266
267
268
269
270
271
272
273
274

       if (address + 32 < regs->esp)
         goto bad_area;

281
282
283  }
254  if (expand_stack(vma, address))
```

```
285     goto bad_area;
```

Строки 262-263

В этом блоке кода мы проверяем, не произошла ли ошибка во время обработки прерывания или в пространстве ядра. Если это так, мы переходим к метке bad_area__semaphore:.

Строка 265

В этой точке мы выполняем поиск в области памяти текущего процесса и поэтому блокируем для чтения семафор описателя памяти.

Строки 267-269

Добравшись сюда, мы знаем, что адрес, породивший ошибку страницы, не находится в потоке ядра или в обработчике прерывания и поэтому нам нужно выполнить поиск этого адреса в областях памяти других ближайших процессов. Если он там не обнаружится, мы переходим к метке bad__area:.

Строки 270-271

Если мы нашли верный регион внутри адресного пространства процесса, мы переходим к метке good__area:.

Строки 272-273

Если мы обнаружили, что регион неверен, мы проверяем, может ли ближайший регион быть увеличен, для того чтобы вместить страницу. Если нет, мы переходим к метке bad_area:.

Строки 274-284

В противном случае проблемный адрес является результатом стековой операции. Если увеличение стека не помогает, переходим к метке bad_area:.

Теперь мы перейдем к объяснению того, куда ведет каждая метка. Мы начнем с метки vmalloc_f ault, показанной на рис. 4.16.

```
arch/i38б/mm/fault.c 473
vmalloc_fault: {
    int index = pgd_index(address);
    pgd_t *pgd, *pgd_k;
    pmd_t * pmd, *pmd_k;
    pte_t *pte_k;
    asm(*movl %%cr3,%0":"=r"  (pgd));
    pgd = index + (pgd_t *)_____ va(pgd);
```

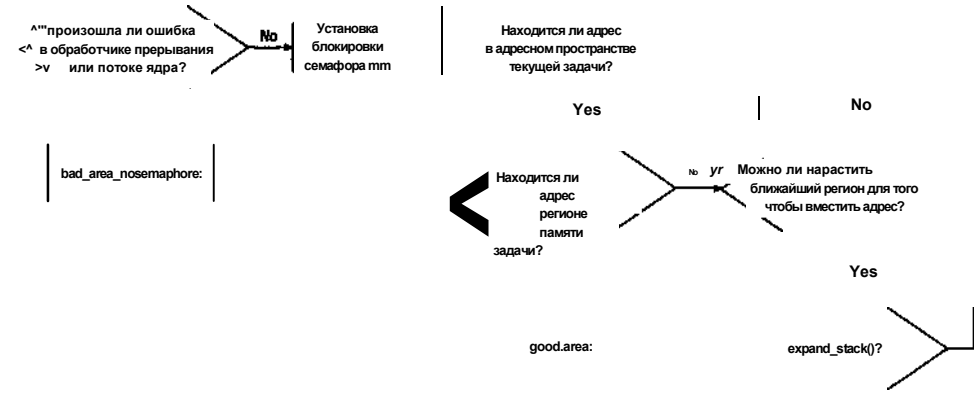


Рис. 4.15. Ошибка страницы 2

```
pgdjc = init__mm.pgd + index; 491      if
(!pgd_present(*pgd_k)) goto no_context; pmd =
pmd_offset(pgd,   address); pmd_k = pmd_offset
(pgd_k,   address); if   (!pmd_j?resent (*pmd_k) )
      goto no_context; set_prad(pmd,   *pmd_k); pte_k =
    pte_offset__kernel(pmd_k,   address)
506    if (!pte present(*pte k))
507     goto no  context;
508   return;
509   }
```

Строки 473-509

Глобальная директория страницы текущего процесса связана (с помощью сг3) и сохранена в переменную pgd, а глобальная директория страницы ядра связана с pgd_k (аналогично переменным pmd и pte). Если проблемный адрес неверен для системы подкачки ядра, код переходит к метке no_context:. В противном случае текущий процесс использует pgd ядра.

Теперь посмотрим на метку good__area:. В этой точке мы знаем, что область памяти, хранящая проблемный адрес, находится в адресном пространстве процесса. Теперь нам нужно убедиться, что разрешение доступа верно. Рис. 4.17 показывает диаграмму этой работы.

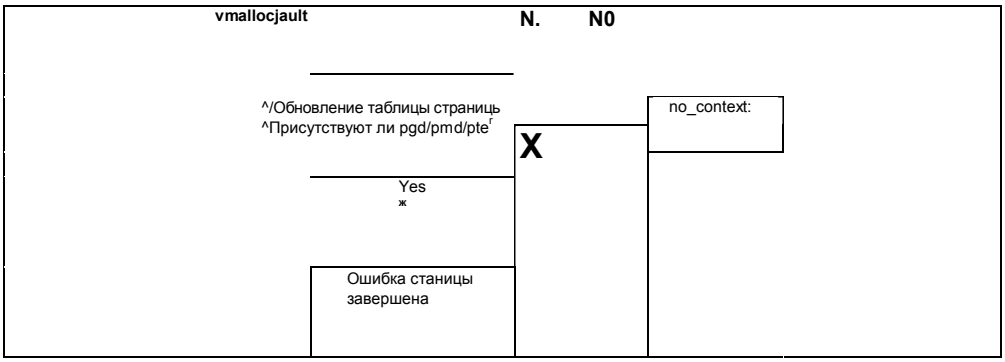


Рис. 4.16. Метка vmallocjault

```
arch/i386/mm/fault.c
290                 good area:
291         info.si code = SEGV ACCERR;
292                 write = 0;
293         switch (error_code & 3) {
294    default:  /* 3: write, present */

     /* fall through */
300     case 2;   /* write, not present */
301       if (!(vma>vm_flags & VM_WRITE))
302               goto bad area;
303                  write++;
304                   break;
305       case 1:   /* read, present */
306               goto bad area;
307         case 0:     /* read,  not present */
308if  (!(vma->vm_flags  &  (VM_READ  |  VM_EXEC)))
309                   goto bad_area;
310                        }
```

Строки 294-304

Если ошибка страницы произошла в записываемой области памяти (вспомните, что в этом случае самый левый бит, error__code, равен 1), мы проверяем, является ли область памяти доступной для записи. Если нет, мы получаем несоответствие разрешений и переходим к метке bad_area:. Если запись возможна, мы переходим к строке case и вызываем handle_mm_fault () с локальной переменной, установленной в 1.

Строки 305-309

Если ошибка памяти вызвана доступом для чтения или выполнения и страница присутствует, мы переходим к метке bad_area:, так как это явное нарушение прав доступа. Если страницы нет, мы проверяем, имеет ли область памяти разрешение за чтение или выполнение. Если нет, мы переходим к метке bad__area:, так как, если бы мы нашли страницы, разрешение не позволило бы выполнить операцию. Если нет, мы переходим к следующему случаю и выполняем handle_mm_fault O с локальной переменной, установленной в 0.

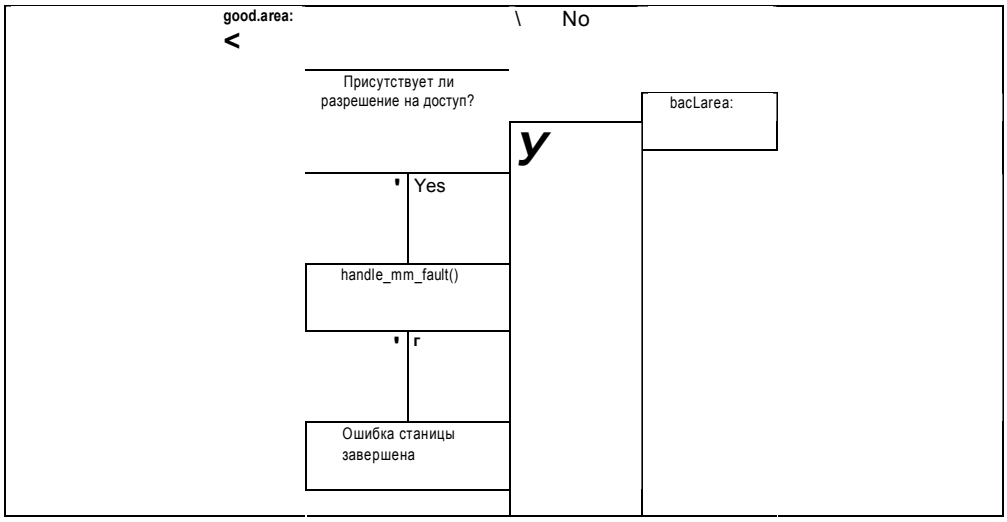


Рис. 4.17. Метка good area

Следующая метка помечает код, в который мы попадаем, когда проверка разрешения подтверждается. Этому случаю соответствует метка survive:.

```
arch/i38б/mm/fault.c
survive:
318  switch (handle  mm fault(mm, vma, address, write)) {
   case VM_FAULT_MINOR: tsk->min_flt++; break; case
   VM FAULT MAJOR: tsk->raaj flt++; break; case
   VM_FAULT_SIGBUS: goto do_sigbus;
```

```
case VM FAULT OOM: goto
out of memory; 329
default: BUG();
    }
```

Строки 318-329

Функция handle_mm_f ault () вызывается для текущего описателя памяти (mm), т. е. описателя области проблемного адреса независимо от того, доступна ли эта область для записи или для чтения-исполнения. Конструкция switch перехватывает управление, если мы не смогли обработать ошибку, для того чтобы функция смогла нормально завершиться.

Следующий блок кода описывает поток выполнения для метки bad_area и bad_area_no_semaphore. При переходе к этой точке нам известно следующее:

1. Породивший ошибку памяти адрес не является адресом пространства процесса, так как мы проверили его область памяти и не нашли там искомое.
2. Породивший ошибку адрес не находится в адресном пространстве процесса и регионе, до которого оно может наращиваться.
3. Адрес, породивший ошибку памяти, находится в адресном пространстве процесса, но не имеет разрешения для доступа к памяти, соответствующего выполнению желаемого действия.

Теперь нам нужно определить, произведен ли доступ из режима ядра. Следующий код и рис. 4.18 демонстрирует поток выполнения этой метки.

```
arch/i386/mm.fault.c
348   bad area:
349    up read (&mm- >mmap s em) ;
350
351             bad_area_nosemaphore:
3 52/* Доступ пользовательского режима порождает SIGSEGV */
3 53            if (error_code & 4) {
354        if (is prefetch(regs, address))
3 55                    return;
356
3 57          tsk->thread.cr2 = address;
3 58   t sk-> thread, err or  code = error code;
3 59          tsk->thread.trap_no = 14;
3 60          info.si signo = SIGSEGV;
3 61             info.si errno = 0;
3 62      /* info.si_code установлен выше */
```

```
3 63        info.si_addr = (void Maddress;
364         force_sig_info(SIGSEGV, &info, tsk);
3 65        return;
366     }
```

Строка 348

Функция up_jread () снимает блокировку чтения с семафора описателя памяти процесса. Обратите внимание, что мы просто перешли к метке bad_area после того, как поставили блокировку семафора описателя памяти и просмотрели его область памяти на предмет нахождения в адресном пространстве процесса нашего адреса. В противном случае мы переходим к метке bad_area_no_semaphore. Единственная разница между этими двумя случаями заключается в установке блокировки семафора.

Строки 351-353

Так как адрес не находится в адресном пространстве, мы проверяем, была ли ошибка сгенерирована в пользовательском режиме. Если вы вспомните табл. 4.7, значение error_code означает, что ошибка произошла в пользовательском режиме.

Строки 354-366

Мы определили, что ошибка произошла в пользовательском режиме, и поэтому нам нужно послать сигнал SIGSEGV (прерывание 14).

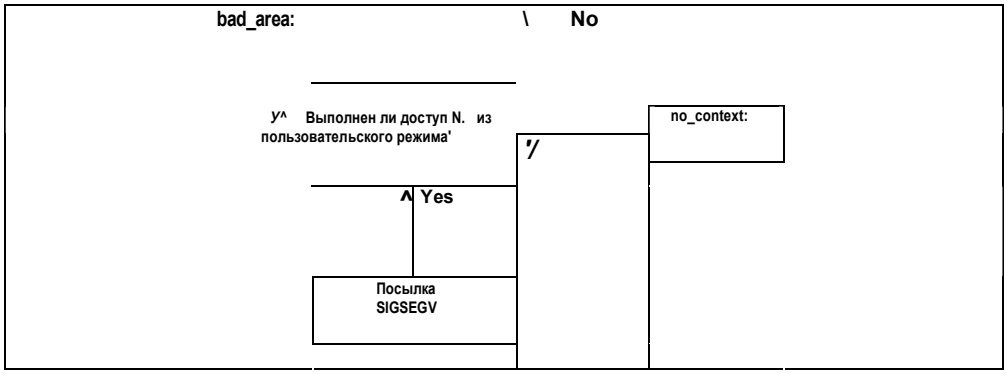


Рис. 4.18. Метка bad area

Следующий фрагмент кода описывает поток обработки метки no_context. Когда мы переходим в эту точку, мы знаем следующее:

- одна из таблиц страниц потеряна;
- доступ к памяти произведен в режиме ядра.

Рис. 4.19 иллюстрирует диаграмму потока метки no_context.

```
arch/ i3 8 б /mm/ fault. c
388     no_context:
390       if  (fixup_exception(regs))
      return; 432       die("Oops",   regs,
error_code);
   bust  spinlocks(0);
   do__exit (SIGKILL) ;
```

Строка 390

Функция f ixup_exception () использует eip, передаваемый для поиска в таблице исключений соответствующей инструкции. Если инструкция в таблице найдена, она уже должна быть откомпилирована с помощью «скрытого» встроенного кода обработки. Обработчик ошибки страницы do_page_fault () использует код обработки ошибки как адрес возврата и переходит по этому адресу. Затем код может выставить флаг ошибки.

Строка 432

Если в таблице исключений нет вхождения для соответствующей инструкции, код выполняет переход по метке no_context и завершается выводом на экран oops.

Рис. 4.19. Метка no context

4.10.3 Исключение ошибки памяти на PowerPC

Обработчик ошибок страниц на PowerPC do_page_f ault () вызывается в результате ошибки инструкции или сохранения данных. Из-за разницы между версиями процессоров PowerPC коды ошибок имеют немного различный формат, но несут одну и ту же информацию. Наибольший интерес представляют биты с проблемной операцией и биты,

означающие наступление ошибки защиты. Обработчик ошибки страницы PowerPC do_page_f ault () не инициирует ошибку oops.

На PowerPC код метки no_context совмещен с меткой bad_area и находится в функции bad__page_f ault (), завершающейся порождением ошибки сегментации. Эта функция также имеет функцию восстановления, которая просматривает exception_ table.

Резюме

В этой главе мы начали с обзора концепций, связанных с управлением памятью. Затем мы описали реализацию каждой из концепций. Первой рассмотренной концепцией были страницы, являющиеся для ядра базовой единицей управления памятью. Также мы рассмотрели слежение ядра за страницами. Затем мы обсудили зоны памяти, отделяющие память от аппаратных ограничений. Далее мы обсудили фреймы страниц и алгоритмы выделения и освобождения памяти в Linux, называемые системой близнецов.

После этого мы рассмотрели основы управления страницами и фреймами страниц, обсудили выделение памяти меньшего, чем у страницы, размера, реализуемое с помощью выделения выделителя секций. Таким образом, мы подошли к обсуждению kmalloc () и функциям выделения ядром памяти. Мы проследили выполнение функций вплоть до того, как они взаимодействуют с выделителем секций. На этом мы закончили обсуждение структур управления памятью.

После рассмотрения структур и алгоритмов управления памятью мы поговорили об управлении памятью процесса пользовательского пространства. Управление памятью процесса отличается от управления памятью в ядре. Мы обсудили расположение памяти для процесса и как ее отдельные части разделены и отображаются в память. Далее во время обсуждения процесса управления памятью мы представили вашему вниманию концепцию ошибок страниц обработчика прерываний, который занимается обработкой исчезнувших из памяти страниц.

Проект: отображение памяти процесса

Теперь мы рассмотрим, как память выглядит для нашей собственной программы. Этот проект состоит из объяснения программы пользовательского пространства, иллюстрирующей, как размещаются объекты в памяти. Для этого проекта мы создали простую разделяемую библиотеку и программу пользовательского пространства, применяющую ее функции. Из этой программы мы распечатаем расположение некоторых переменных и сравним его с отображением процесса для определения того, где размещаются переменные и функции.

Первым шагом является создание разделяемой библиотеки. Разделяемая библиотека может состоять из одной функции, которая будет вызываться из главной программы. Мы

хотим распечатывать адреса локальные переменных из этой функции. Ваша разделяемая библиотека должна выглядеть следующим образом:

```
lkpsinglefoo.c
mylibfoo() {
  int libvar;
  printf("variable libvar \t location: Ox%x\n", fclibvar); }
```

Откомпилируем и скомпонуем singlef oo. c в разделяемую библиотеку:

```
#lkp>gcc -c lkpsinglefoo.c
#lkp>gcc lkpsinglefoo.c -o liblkpsinglefoo.so -shared -lc
```

-shared и -lc - это флаги опций сборщика. Опция -shared требует, чтобы был создан разделяемый объект, который может быть объединен с другими объектами. Флаг -1c указывает, что при сборке будет произведен поиск библиотеки C[1].

Эти команды генерируют файл 1 inklks ingle foo. so. Для того чтобы его использовать, нужно скопировать его в /lib.

Следующее главное приложение вызывает собранную нами библиотеку:

```
lkpmem.c
#include <fcntl.h>
int globalvarl; int
globalvar2 = 3; void
mylocalfoo() {
  int functionvar;
  printf("variable functionvar \t location: Ox%x\n", fcfunctionvar) ;
}
int mainO
{
  void *localvarl = (void *)malloc(2048)
  printf("variable globalvarl \t location: Ox%x\n", &globalvarl);
  printf("variable globalvar2 \t location: Ox%x\n", &globalvar2);
  printf("variable localvarl \t location: Ox%x\n", blocalvarl);
  mylibfoo();
  mylocalfoof);
  while(1);
  return(0);
```

[1] Библиотеки libc. *Примеч. науч. ред.*

```
}
```

Откомпилируем lkmem. с следующим образом:

```
#lkp>gcc  -o  lkpmem lkpmem.c  -llkplibsinglefoo
```

Во время выполнения lkmem вы можете увидеть информацию о размещении в памяти различных переменных. Функция блокируется с помощью while (1) и не возвращает значения. Это позволит вам получить PIII процесса и получить карту памяти. Для этого нужно ввести следующую команду:

```
#lkp>./lkpmem
#lkp> ps  aux  |  grep  lkpmem
#lkp>  cat  /proc/<pid>/maps
```

Облачает сегмент памяти, в котором находится переменная.

Упражнения

1. Почему программы, реализуемые из одного исполнимого файла, не могут разделять сегменты памяти?
2. Как будет выглядеть стек следующей функции после трех итераций?

   ```
   foo(){ int
       a; foo()
   }
   ```

   Если она продолжится, какая может возникнуть проблема?
3. Заполните значениями описатель vm_area_s true t, соответствующий карте памяти, приведенной на рис. 4.11.
4. Как связаны между собой страницы и секции?
5. Тридцатидвухбитовые системы Linux при загрузке не используют среднюю директорию страниц. Вместо этого эффективно используется двухуровневая таблица страниц. Первые 10 бит виртуального адреса соответствуют отступу в глобальной директории страниц (PGD). Вторые 10 бит соответствуют отступу в таблице страниц (PTE). Оставшиеся 12 бит соответствуют отступу страницы.

   Какой размер в Linux имеет страница? К скольким страницам может обратиться задача? К какому объему памяти?
6. Как связаны зоны памяти и страницы?
7. Чем на аппаратном уровне «реальные» адреса отличаются от «виртуальных»?

Глава 5

Ввод-Вывод

В этой главе:

Ядро Linux представляет собой набор кода, который выполняется на одном или нескольких процессорах. Для остальной системы с помощью аппаратной поддержки предоставляется процессорный интерфейс. На самом нижнем, машинно-зависимом уровне ядро общается с этими устройствами с помощью простых ассемблерных инструкций. Эта глава раскрывает взаимоотношения ядра с окружающим оборудованием, с акцентом на файловый ввод-вывод и аппаратные устройства. Мы продемонстрируем, как ядро Linux связывает вместе аппаратную и программную часть начиная с наивысшего уровня виртуальной системы до нижних уровней физической записи битов информации.

Глава начинается с обзора ядра компьютера, процессора и подключения к ядру остальной аппаратуры. Также обсуждается концепция шин, включая то, как они связывают процессор с другими элементами системы (такими, как память). Также мы представим вашему вниманию устройства и контроллеры, используемые на платформах x86 и PowerPC.

Имея базовое понимания компонентов системы и их связей, мы сможем приступить к анализу слоев программного обеспечения начиная с приложений для операционной системы и кончая специфическими блочными устройствами, используемыми для хранения, - жесткими дисками и их контроллерами. Несмотря на то что концепция файловой системы не раскрывается до следующей главы, мы рассмотрим достаточно для того, чтобы опуститься до уровня обобщенного блочного устройства и, что самое главное, методов связи блочных устройств, очереди запросов.

Мы обсудим важные связи между механическим устройством (жестким диском) и программной частью системы при рассмотрении концепции планировщика ввода-вывода. Понимая физическую геометрию жесткого диска и того, как операционная система разбивает диск, мы сможем понять синхронизацию программного и аппаратного обеспечения.

Переходя к аппаратуре, мы увидим, как интерфейсы обобщенного блочного драйвера связаны со специфическим блочным драйвером, что позволяет программному обеспечению управлять различными аппаратными устройствами. И наконец, в нашем путешествии с уровня приложений на уровень ввода-вывода мы коснемся аппаратного ввода-вывода, необходимого для контроллера диска и продемонстрируем вашему вниманию другие примеры ввода-вывода и драйверов устройств из этой книги.

Затем мы обсудим другой важный тип устройств - символьные устройства и чем они отличаются от блочных и сетевых устройств. Также мы рассмотрим важность других устройств - контроллера DMA, таймера и терминального устройства.

5.1 Как работает оборудование: шины, мосты, порты и интерфейсы

Процессор общается с окружающими устройствами через набор электрических связей, называемых **линиями (lines). Шины (busses)** - это группы линий с похожими функциями. Наиболее простой тип шины идет к процессору и от него и использует адресацию устройства; для посылки, получения данных (data) от устройств и для передачи управляющей информации, такой, как устройство-зависимая инициализация и характеристики. Поэтому мы можем сказать, что основными методами общения процессора с устройствами (и наоборот) являются общение через адресную шину, шину данных и управляющую шину.

Самая главная функция процессора в системе заключается в выполнении инструкций. Объединения этих инструкций называются **компьютерными программами (computer programs)** или программным обеспечением (software). Программы размещаются на устройствах (или группах устройств), известных как память. Процессор связан с памятью с помощью адресов, данных и управляющей шины. При выполнении программы процессор выбирает расположение инструкции в памяти с помощью адресной шины и перемещает инструкцию с помощью шины данных. Управляющая шина обрабатывает направление (в процессор или из него) и тип (в данном случае памяти) передачи. В эту терминологию добавляется некоторое количество путаницы, когда речь идет об определенной шине, такой, как **управляющая шина (front side bus),** или шина PCI (PCI bus), мы имеем в виду сразу и шину адресов, и данных, и управления.

Задача выполнения программного обеспечения на системе требует широкого набора периферийных устройств. Современные компьютерные системы имеют два основных вида периферийных устройств (также известных как контроллеры), сгруппированные на **северном мосту (Northbridge)** и на **южном мосту (Southbridge).** Традиционно термин **мост (bridge)** описывает аппаратное устройство, связывающее две шины. Рис. 5.1 иллюстрирует, как южный и северный мосты взаимодействуют с устройствами. Вместе эти устройства образуют **чипсет (chipset)** системы.

Северный мост соединяет высокоскоростные, высокопроизводительные периферийные устройства, такие, как контроллер памяти и контроллер PCI. Несмотря на то что существуют решения с интегрированными в северный мост графическими контроллерами, обычно присутствуют специальные высокопроизводительные шины, такие, как ускоренный графический порт (Accelerated Graphics Port, AGP) или PCI Express для взаимодействия с отдельными графическими адаптерами. Для получения скорости и хорошей производительности северный мост включает в себя управляющую шину[1] и в зависимости от дизайна чипсета шину PCI и/или шину памяти.

[1] **На некоторых PowerPC-системах управляющая шина эквивалентна процессорной локальной шине.**

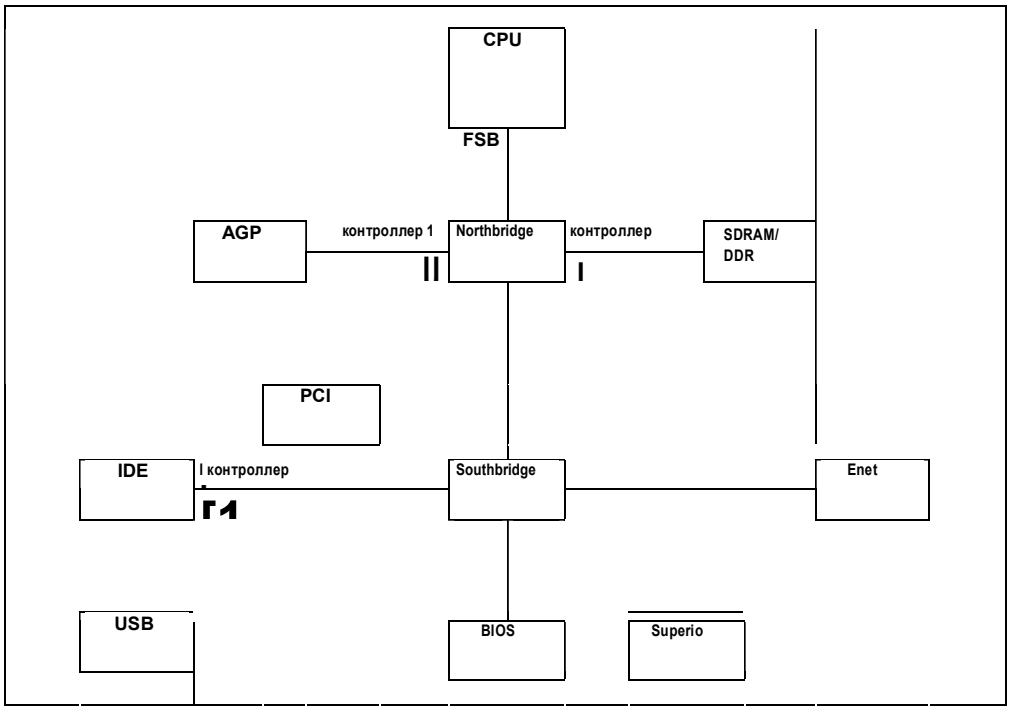


Рис. 5.1. Старый дизайн Intel

Южный мост, связанный с северным мостом, также подключен к комбинации низкопроизводительных устройств. Например, южный мост Intel PIDC4 подключен к PCI-ISA-мосту, IDE-контроллеру, USB, таймеру реального времени, двойному контроллеру прерываний 82C59 (описанному в гл. 3, «Процессы: принципиальная модель выполнения»), таймеру 82C54, двойному контроллеру DMA и подерживает ввод-вывод APIC.

В ранних персональных компьютерах x86 связь между базовой периферией, такой, как клавиатура, последовательный и параллельный порты, была выполнена через **шину ввода-вывода (I/O bus).** Шина ввода-вывода была типом управляющей шины. Шина ввода-вывода представляет собой довольно медленный способ связи с управляемой периферией. На x86 существуют специальные инструкции, такие, как **inb** (read **in byte)** (прочитать в байт) и **outb** (write **out** a **byte)** (записать из байта), для работы с шиной ввода-вывода. Шина ввода-вывода реализована с помощью разделения адресов процессора и линий данных. Управляющие линии активизируются, только когда используются специальные инструкции ввода-вывода, позволяющие устройствам ввода-вывода не теряться в памяти. Архитектура PowerPC обладает другим методом управления периферийными устройствами, известным как **отображение ввода-вывода в память (memory-mapped I/O).** При

отображении ввода-вывода в память устройствам назначаются регионы адресного пространства для связи и управления.

Например, на архитектуре x86 регистры данных первого параллельного порта находятся в **порту ввода-вывода (I/O port)** 0x378, тогда как на PPC в зависимости от реализации он может находиться в памяти по адресу OxГОООО3ОО. Для чтения из регистра данных первого последовательного порта на x86 мы исполняем ассемблерную инструкцию in al, 0x378. В этом случае мы активизируем линию управления для контроллера параллельного порта. Для шины это означает, что 0x378 - это не адрес в памяти, а порт ввода-вывода. Для чтения регистра данных первого последовательного порта на PPC мы выполняем ассемблерную инструкцию lbz г3, 0 (Oxf 0000300). Контроллер параллельного порта следит за шиной адресов[1] и отвечает только на запросы от определенного диапазона адресов, при которых OxГОООО3ОО будет неудачным.

С развитием персональных компьютеров все больше дискретных устройств ввода-вывода объединялось в единственный интегрированный **Superio** чипсет. Функции Superio обычно берет на себя южный мост (как в ALI M1543C). В качестве примера типичной функциональности, которую можно обнаружить в Superio-устройстве, рассмотрим SMSC FDC37C932. Он включает контроллер клавиатуры, таймер реального времени, устройство управления питанием, контролер гибких дисков, контроллер последовательного порта, параллельный порт, интерфейс IDE и ввод-вывод общего назначения. Другие южные мосты содержат интегрированный контроллер локальной сети, PCI Express, аудиоконтроллер и т. д.

Новая архитектура систем Intel перешла к концепции **хабов (hubs).** Северный мост стал называться хабом контроллеров графики и памяти (Graphics and Memory Controller Hub, GMCH). Он поддерживает высокопроизводительные AGR- и DDR-контроллеры. С появлением PCI Express чипсеты Intel превратились в хаб-контроллер памяти (Memory Controller Hub) (MCH) для контроллеров графики и памяти DDR2. Южный мост стал называться хабом контроллера ввода-вывода (I/O Controller Hub, ICH). Эти хабы связаны через проприетарную шину точка-точка, называемую хабом архитектуры Intel (Intel Hub Architecture, ША). Более подробную информацию можно найти в описании чипсетов Intel 865G[2] и 925XE[3]. Рис. 5.2 иллюстрирует ICH.

AMD перешла от старого стиля Intel с северным и южным мостами к технологии упаковки **HyperTransport** между основными компонентами чипсета. Для операционной системы HyperTransport является PCI-совместимым[4]. (См. описание чипсета AMD для серии чипсетов 8000.) Рис. 5.3 иллюстрирует технологию HyperTransport.

[1] Наблюдение за шиной адресов также часто связывают с декодированием шины адресов.

[2] http://www.intel.com/design/chipsets/datashts/25251405 .pdf.

[3] http://www.intel.com/design/chipsets/datashts/30146403.pdf.

[4] См. описание чипсетов AMD серии 8000: http://www.amd.com/us-en/Processors/ProductInformation/0,30_118_6291_4886,00.html.

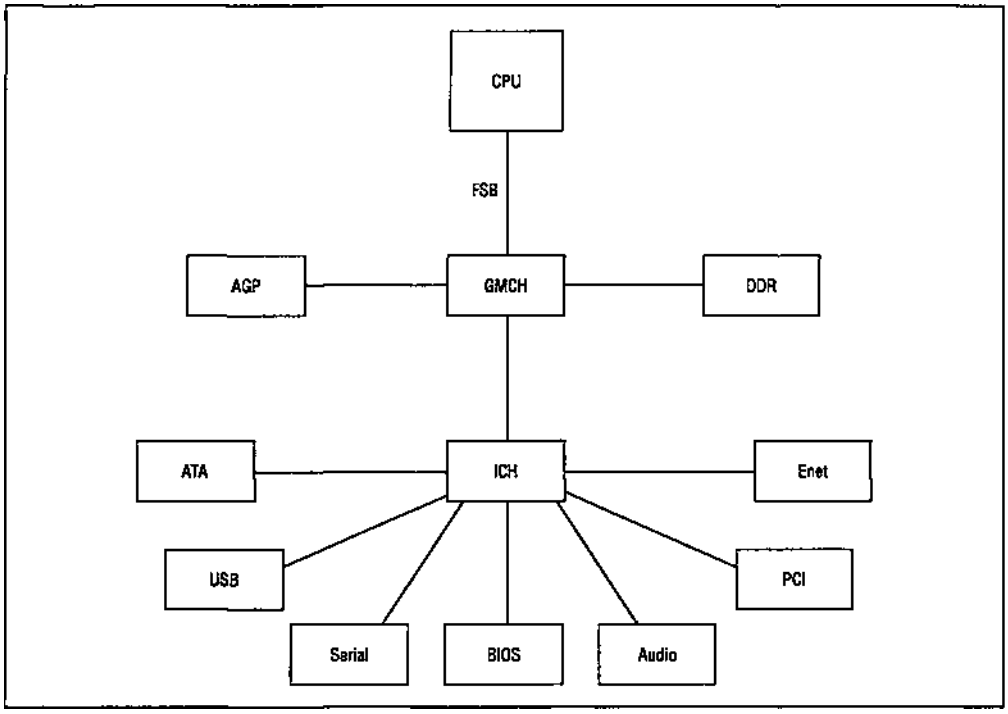


Рис. 5.2. Новый хаб Intel

Apple, в PowerPC, использует проприетарный дизайн, называемый универсальной архитектурой материнской платы (Universal Motherboard Architecture, UMA). Целью UMA является использование одинаковых чипсетов на всех Mac-системах.

Чипсет G4 включает «UniNorth-контроллер памяти и мост шины PCI» в качестве северного моста и «Key Largo I/O и контроллер дисковых устройств» в качестве южного моста. UniNorth поддерживает SDRAM, Ethernet и AGP. Южный мост Key Largo связан с UniNorht с помощью моста PCI-to-PCI, поддерживает шины ATA, USB, беспроводную локальную сеть (WLAN) и звук.

Чипсет G5 включает системный контроллер программно-специфическую интегрированную цепь (Application Specific Intergated Circuit, ASIC), поддерживающую AGP и память DDR. Он связан с системным контроллером через шину HyperTransport с контроллером PCI-X и высокопроизводительным устройством ввода-вывода. Более подробно об этой архитектуре можно прочитать на страницах Apple для разработчиков.

Имея этот базовый обзор основных системных архитектур, мы можем теперь сфокусироваться на интерфейсах, предоставляемых устройствами ядру. В гл. 1, «Обзор»,

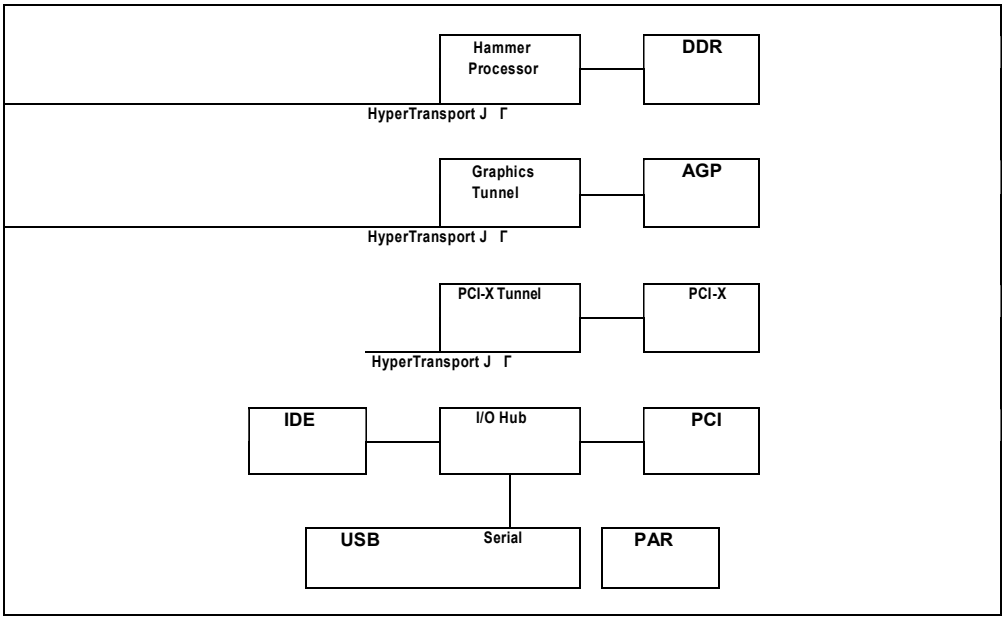


Рис. 5.3. AMD HyperTransport

говорилось, что устройства представлены файлами в файловой системе. Разрешение файла, режимы и связанные с файловой системой системные вызовы, такие, как open () и read () применяются к этим специальным файлам точно так же, как и к обычным. Значение каждого вызова отличается в зависимости от обрабатываемого устройства и изменяется для обработчиков каждого типа устройств. Тем не менее детали обработки устройства сделаны прозрачными для программиста приложений и скрыты в ядре. Стоит сказать, что, когда процесс применяет системный вызов к файлу устройства, он приводится к одному из типов функций обработки устройства. Эти функции-обработчики определяются в драйвере устройства. Рассмотрим основные типы устройств.

5.2 Устройства

Существует два типа файлов устройств: файлы блочных устройств и файлы символьных устройств. Блочные устройства передают данные порциями, а символьные устройства (как следует из названия) передают данные по символу за один раз. Третий тип устройств, сетевые устройства, является специальным случаем, наследующим свойства как блочных, так и символьных устройств. При этом сетевые устройства не представляются файлами.

Старый метод назначения устройствам номеров, когда старшие номера обычно связывались с драйверами устройств или контроллерами, а младшие номера были отдельными устройствами внутри контроллера, уступил место новому, динамическому методу, называемому devf s. Исторически эти старшие и младшие номера были 8-битовыми, что позволяло иметь немногим больше 200 статически выделенных главных устройств на всей планете. Блочные и символьные устройства представлялись списками по 256 вхождений. (Вы можете найти официальный список выделяемых чисел для основных и дополнительных устройств в /Documentation/devices . txt.)

Файловая система устройств Linux (Linux Device Filesystem, devfs) присутствовала до версии ядра 2.3.46; devfs не включена по умолчанию в сборку ядра 2.6.7, но может быть включена через файл настройки с помощью CONFIG_DEVFS=Y. При включении devfs модуль может регистрировать устройство по его имени, а не по паре старшего и младшего номеров. Для совместимости devfs позволяет использовать старые старшие и младшие номера или генерировать уникальные 16-битовые номера устройств на любой конкретной системе.

5.2.1 Обзор блочных устройств

Как говорилось ранее, операционная система Linux рассматривает все устройства в качестве файлов. Любые полученные элементы от блочного устройства могут быть связаны случайным образом. Хорошим примером блочного устройства является дисковый привод. Файловая система для диска IDE называется /dev/hda. С /dev/hda. связан старший номер 3 и младший номер 0. Сам дисковый привод обычно обладает контроллером и по своей сути является электромеханическим устройством (т. е. имеющим движущиеся части). Раздел «Общая концепция файловых систем» в гл. 6, «Файловые системы», рассматривает основы конструкции жесткого диска.

5.2.1.1 Обобщенный слой блочных устройств

Драйвер устройства регистрируется во время инициализации драйверов. При этом драйвер добавляется в таблицу драйверов ядра (driver table), а номер драйвера отображается в структуру block_device_operations. Структура block_device__operations хранит функции для запуска и остановки данного блочного устройства в системе:

```
include/linux/fs.h
7 60  struct block device operations {
761    int (*open) (struct inode *, struct file *);
762    int (*release) (struct inode *, struct file *);
763    int (*ioctl) (struct inode *, struct file *,
                     unsigned, unsigned long);
764    int (*media changed) (struct gendisk *);
765    int (*revalidate_disk) (struct gendisk *);
```

```
766    struct module *owner;
767    };
```

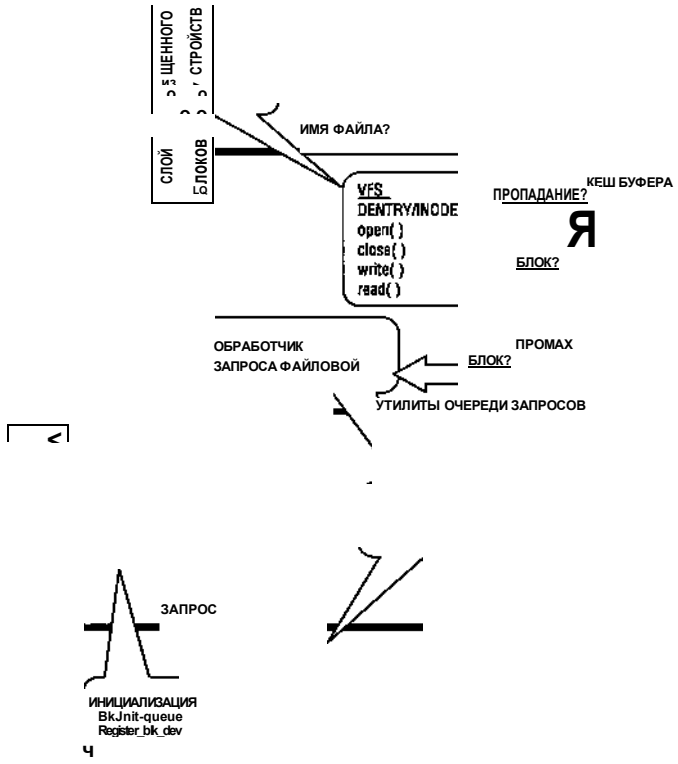
Интерфейс блочных устройств похож на интерфейсы других устройств. Функции open () (строка 761) и release () (строка 762) - **синхронные** (т. е. запускаются сразу после завершения вызова). Наиболее важные функции, read () и write (), реализованы для блочных устройств особым способом из-за их механической натуры. Рассмотрим доступ к блоку данных с дискового привода. Время, требуемое на позиционирование головки над соответствующей дорожкой, и то, чтобы диск повернулся на требуемый блок, с точки зрения процессора является достаточно долгим. Эта **задержка (latency)** привела к идее создания **системной очереди запросов (system request queue).** Когда файловой системе требуется один или несколько блоков данных и они не находятся в локальном **кеше страниц (page cache),** она помещает запрос в очередь запросов и передает очередь на слой обобщенного блочного устройства. Слой обобщенного блочного устройства определяет наиболее эффективный способ механического получения (или сохранения) информации и передает ее драйверу жесткого диска.

Очень важно, что в момент инициализации драйвер блочного устройства регистрирует обработчик очереди запросов в ядре (с помощью специального менеджера блочного устройства) для выполнения операций чтения-записи блочного устройства. Слой обобщенного блочного устройства работает как интерфейс между файловой системой и интерфейсом уровня регистров и позволяет с помощью оптимизации в очереди запросов на чтение и запись наилучшим образом использовать новые и более разумные устройства. Это достигается с помощью вспомогательных утилит запросов. Например, если устройство для данной очереди поддерживает команды очередей, операции чтения и записи оптимизируются для использования на данном оборудовании с помощью перераспределения запросов. В качестве примера оптимизации очереди может служить возможность установки того, сколько запросов могут находиться в очереди ожидания. Связь уровня приложений, слоя файловой системы, слоя обобщенного блочного устройства и драйвера устройства показана на рис. 5.4. Файл biodoc. txt в /Documentation/block содержит дополнительную информацию об этом слое и информацию об изменениях со времен старых версий ядра.

5.2.2 Очереди запросов и планировщик ввода-вывода

Когда запросы на чтение и запись передаются по слоям через VFS, они проходят драйверы файловой системы и кеш страниц[1] и заканчиваются входом в драйвер блокового устройства для выполнения настоящих операций ввода-вывода на устройстве, хранящем требуемые данные.

[1] **Этот переход описывается в гл. 6**


ЩЕННОГО
СТРОЙСТВ
СЛОЙ БЛОКОВ
ИМЯ ФАЙЛА?
VFS
DENTRY/INODE
open( )
close( )
write( )
read( )
КЕШ БУФЕРА
ПРОПАДАНИЕ?
Я
БЛОК?
ОБРАБОТЧИК
ЗАПРОСА ФАЙЛОВОЙ
ПРОМАХ
БЛОК?
УТИЛИТЫ ОЧЕРЕДИ ЗАПРОСОВ
ЗАПРОС
ИНИЦИАЛИЗАЦИЯ
BkJnit-queue
Register_blk_dev

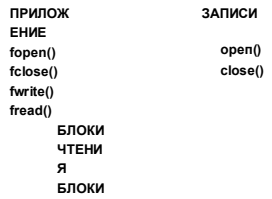


Рис. 5.4. Чтение-запись блоков

Как упоминалось ранее, драйвер блочного устройства создает и инициализирует очередь запросов во время инициализации. Также во время инициализации определяется алгоритм планировки ввода-вывода, используемого для чтения и записи с обслуживаемого блочного устройства. Алгоритм планирования ввода-вывода также называется **алгоритмом лифта (elevator algorithm).**

По умолчанию алгоритм планировки ввода-вывода определяется ядром во время загрузки и по умолчанию выбирается **предварительный планировщик ввода-вывода**

(anticipatory I/O scheduler)[1]**.** Установкой параметра ядра elevator вы можете изменять тип планировщика ввода-вывода:

- **deadline.** Предельный планировщик ввода-вывода.
- **noop.** Безоперационный планировщик ввода-вывода.
- **as.** Преждевременный планировщик ввода-вывода.

На момент написания этой книги существует патч, делающий планировщик ввода-вывода полностью модульным. Применяя modprobe, пользователь может загружать модули и переключаться между ними на лету[2]. С этим патчем как минимум один планировщик должен быть откомпилирован с самим ядром.

Перед тем как приступить к описанию работы планировщика ввода-вывода, нам нужно коснуться основ очередей запросов.

Блочные устройства используют очереди запросов для упорядочения запросов ввода-вывода на блоки устройства. Некоторые блочные устройства, такие, как виртуальный диск в памяти, не испытывают большой нужды в упорядочении запросов на ввод-вывод, так как это только тормозит работу. Другие блочные устройства, такие, как жесткие диски, нуждаются в сортировке запросов, так как для них операции чтения и записи обладают большой задержкой. Как говорилось ранее, головка жесткого диска должна переместиться на нужную дорожку, что с точки зрения процессора происходит слишком медленно.

Очереди запросов решают эту проблему, пытаясь организовать последовательность запросов чтения и записи блоков с помощью отсрочки запросов. Простой и понятной аналогией планировщика ввода-вывода является работа лифта[3]. Если приказ на остановку лифта получен через отданный приказ, вы получите лифт, эффективно перемещающийся с этажа на этаж; он может спуститься с верхнего этажа на нижний без промежуточных остановок. Отвечая на запросы, поступающие с той стороны, куда движется лифт, он увеличивает свою эффективность и удовлетворенность пассажиров. Аналогично запросы ввода-вывода к жестокому диску группируются вместе, чтобы избежать лишних задержек при перемещении головки взад-вперед. Все упомянутые планировщики ввода-вывода (безоперационный, предельный и преждевременный) реализуют эту базовую функциональность лифта. Следующий раздел рассматривает лифты подробнее.

[1] **Некоторые драйверы блочных устройств могут изменять свой планировщик ввода-вывода во время работы, если они отображаются в sysfs.**

[2] **Более подробную информацию можно найти в сети по запросам «Jens Axboe» и «Modular 10 Shedulen>.**

[3] **Именно благодаря этой аналогии планировщики ввода-вывода связаны с лифтами.**

5.2.2.1 Безоперационный планировщик ввода-вывода

Безоперационный планировщик ввода-вывода[1] получает запросы и сканирует очередь, определяя, можно ли объединить их с уже существующими запросами. Это возможно, если новый запрос близок к существующему. Если новый запрос необходим для блока перед тем, для которого уже есть запрос, он добавляется в начало существующего запроса. Если новый запрос существует для блока после того, для которого уже есть запрос, он добавляется в конец существующего запроса. При нормальном вводе-выводе мы читаем файл с начала до конца, и поэтому большинство запросов сливаются с уже существующими запросами.

Если безоперационный планировщик обнаруживает, что новый запрос нельзя слить с уже существующим, так как он находится недостаточно близко, планировщик ищет запрос между этими двумя позициями. Если новый запрос вызывает операции ввода-вывода для секторов между существующими запросами, он вставляется в очередь на найденную позицию. Если он не может быть помещен между существующими запросами, он добавляется в конец.

5.2.2.2 Предельный планировщик ввода-вывода

Безоперационный планировщик[2] страдает одним недостатком: при достаточно близких запросах новый запрос никогда не будет обработан. Многие новые запросы, близкие к существующим, будут слиты или вставлены между существующими элементами, а новый запрос будет отброшен и помещен в конец очереди ожидания. Предельный планировщик пытается решить эту проблему с помощью назначения каждому запросу предельного времени и, кроме того, использует две дополнительные очереди для эффективной обработки времени, а в остальном он похож по эффективности на безоперационный алгоритм для работы с диском.

Когда приложение делает запрос на чтение, оно обычно ожидает выполнения запроса перед продолжением выполнения. Запрос на запись, наоборот, обычно не заставляет приложение ожидать; запись можно выполнить в фоновом режиме, когда приложение займется другими своими делами. Предельный планировщик ввода-вывода использует эту информацию для предпочтения операций чтения операциям записи. Очереди чтения и записи хранятся отдельно и сортируются по близости секторов. В очередях чтения и записи запросы сортируются по времени (FIFO).

При поступлении нового запроса он помещается в очередь безоперационного планировщика. Затем запрос помещается в очередь чтения или записи в зависимости от типа операции ввода-вывода. Затем запрос обрабатывает предельный планировщик ввода-вывода, проверяя сперва предельное время обработки головы очереди чтения. Аналогично если голова очереди чтения не достигла своего предела, планировщик проверяет

[1] Код безоперационного планировщика находится в drivers/block/noop-iosched. c.

[2] Код предельного планировщика ввода-вывода находится в drivers/block/deadline-iosched.c.

голову очереди записи; если предел достигнут, голова очереди обрабатывается. Стандартная очередь проверяется, только когда нет достигнувших предела элементов очередей чтения и записи, и обрабатывается как и для безоперационного алгоритма.

Запросы на чтение истекают быстрее, чем запросы на запись: S с против 5 с по умолчанию. Эта разница в достижении предела операциями чтения и записи может привести к тому, что множество операций чтения могут вызвать голод обработки операций записи. Поэтому предельному планировщику с помощью параметров указывается максимальное количество операций чтения, которые можно обработать до обработки операции записи; по умолчанию это 2, но, так как последовательные запросы можно трактовать как один запрос, может произойти 32 операции чтения перед тем, как запрос на запись будет считаться испытывающим голод[1].

5.2.2.3 Предварительный планировщик ввода-вывода

Главной проблемой предварительного планировщика ввода-вывода является интенсивное поступление операций записи. Так как он нацелен на максимизацию эффективности чтения, запрос на запись может предваряться чтением, из-за чего головка диска перейдет на новую позицию, а затем для выполнения операции записи будет возвращаться назад, в начальную позицию. Предварительный планировщик ввода-вывода[2] пытается предупредить следующую операцию и таким образом повысить производительность ввода-вывода.

Структурно предварительный планировщик ввода-вывода похож на предельный планировщик ввода-вывода. У него есть очереди чтения и записи, организованные по принципу FIFO, и очередь по умолчанию, упорядоченная по близости секторов. Основная разница заключается в том, что после запроса на чтение планировщик не начинает сразу обрабатывать другие запросы. В течение 6 мс он ничего не делает, выполняя дополнительное предварительное чтение. Если поступает новый запрос к прилежащей области, он сразу обрабатывается. После периода предсказания планировщик возвращается к обычным операциям, как описано для предельного планировщика ввода-вывода.

Этот период предсказания позволяет минимизировать задержку ввода-вывода с помощью переноса головки диска от одного сектора к другому.

Как и для предельного планировщика ввода-вывода, алгоритм предварительного планировщика ввода-вывода настраивается несколькими параметрами. По умолчанию время ожидания для запроса на чтение равно 1/8 с, а время ожидания для запроса на запись -1/4 с. Два параметра управляют тем, когда следует выполнять проверки переключения между потоками чтения и записи[3]. Поток чтения проверяет истекшие запросы на запись каждые 1/4 секунды, а поток записи проверяет запросы на чтение каждые 1/16 секунды.

[1] См. параметры функции в строках 24-27 deadline-iosched. c.

[2] Код предварительного планировщика ввода-вывода находится в drivers/block/as-iosched. c.

[3] См. определение параметров в строках 30-60 as-iosched. c.

По умолчанию планировщиком ввода-вывода является предварительный планировщик ввода-вывода, потому что он оптимизирован для большинства приложений и блочных устройств. Предельный планировщик ввода-вывода иногда лучше подходит для работы с базами данных или приложений, требующих высокой производительности при работе с жестким диском. Безоперационный планировщик ввода-вывода обычно используется в системах, где важно время поиска, таких, как встроенные системы, работающие с операционной памятью.

Теперь перенесем свое внимание с различных планировщиков ввода-вывода в ядре Linux на саму очередь запросов и способ, которым блочные устройства инициализируют очереди прерываний.

5.2.2.4 Очередь запросов

В Linux 2.6 каждое блочное устройство имеет собственную очередь запросов, обрабатывающую запросы ввода-вывода к устройству. Процесс может обновлять очередь запросов устройства только в том случае, если заблокирует очередь запросов. Давайте рассмотрим структуру request__queue.

```
include/linux/blkdev.h
270 struct reguest queue
271 {
272   /*
273   * Объединение с головой очереди для разделения кеша
274   */
275   struct list_head queue__head;
276   struct request *last_merge;
277   elevator t  elevator;
278
279   /*
280   * очередь запрашивает свободный список для записи и для чтения
281   */
282   struct request_list   rq;
```

Строка 275

В этой строке описывается указатель на голову очереди запросов.

Строка 276

Это последний запрос, помещенный в очередь ожидания.

Строка 277

Функция планировщика (elevator) используется для обработки очереди запросов. Это может быть стандартный планировщик ввода-вывода (безоперационный,

предельный или предварительный) или новый тип планировщика, специально разработанный для данного блочного устройства.

Строка 282

request_list - это структура, состоящая из двух wait_queue: одной для очереди чтения блочного устройства и одной для очереди записи.

```
include/linux/blkdev.h 283
284    request fn proc   *request fn;
285    merge request fn  *back     merge fn;
286    merge request fn *front merge  fn;
287    merge requests fn  *merge requests fn;
288    make request fn  *make request fn;
289   prep rq fn    *prep rq fn;
2 90   unplug fn   *unplug fn;
2 91   merge  bvec fn  *merge bvec  fn;
292    activity_fn    *activity_fn;
293
```

Строки 283-293

Специальные функции планировщика (лифта) могут быть определены для управления тем, как обрабатываются запросы для блочных устройств.

```
include/linux/blkdev.h
294    /*
2 95   *  Условие автоматического отключения
296    */
297    struct timer list unplug timer;
298    int    unplug_thresh; /* После такого количества запросов */
299    unsigned long  unplug_delay; /* После такой задержки*/
300    struct work struct , unplug  work;
301
3 02   struct backing dev info backing dev info;
303
```

Строки 294-303

Эти функции используются для изъятия функции планировщика ввода-вывода, используемых для блочного устройства. **Подключение (plugging)** связано с ожиданием, пока очередь запросов заполнится и можно будет выбрать алгоритм планировщика для упорядочения и сортировки запросов ввода-вывода, оптимизируя время, необходимое для выполнения запросов ввода-вывода. Например, жесткий диск под-

ключает несколько запросов на чтение в ожидании того, что после поступления еще нескольких запросов придется меньше перемещать головку. Скорее всего, чтение удастся отсортировать последовательно или даже кластеризовать в одну более длинную операцию чтения. **Отключение (unplugging)** связано с методом, когда устройство решает больше не ожидать и обрабатывать запросы, в зависимости от возможных будущих оптимизаций. (См. более подобную информацию в Documentation/block/biodoc. txt.)

```
include/linux/blkdev. h
304    /* Владелец очереди может использовать эту информацию
305    *  по своему усмотрению.
306    *  Это не касается ll_rw_blk.
307    */
3 08   void     *queuedata;
309
310    void     *activity_data;
311
```

Строки 304-311

Как следует из комментариев, эти строки обрабатывают очередь запросов с помощью определенного устройства и/или драйвера устройства.

```
include/linux/blkdev. h
312    /*
313    * Для страниц за этими пределами очереди нужны страницы раскачки.
314    */-.-.
315    unsigned long   bounce__pfn;
316    int     bounce_gfp; 317
```

Строки 312-317

Подпрыгивание (bouncing) означает принятый в ядре перенос буфера ввода-вывода из верхней памяти в буфер в нижней памяти. В Linux 2.6 ядро позволяет устройству самостоятельно обрабатывать необходимый ему буфер в буфере в верхней памяти. Сейчас подпрыгивание обычно используется только в тех случаях, когда устройство не может обработать буфер в верхней памяти.

include/linux/blkdev.h

```
318    /*
319    * см. ниже в QUEUE * различные флаги очереди
320    */
321    unsigned long  queue flags;
322
```

Строки 318-321

Переменная queue_flags хранит один или несколько флагов, приведенных в табл. 5.1 (см. include/linux/blkdev.h, строки 368-375).

Таблица 5.1. queue Jlags

Флаг	Функция
QUEUE_FLAG_CLUSTER	Несколько сегментов кластера в 1
QUEUE__FLAG__QUEUED	Используется обобщенный код очереди
QUEUE__FLAG_STOPPED	Очередь остановлена
QUEUE_FLAG_READFULL	Очередь чтения заполнена
QUEUE_FLAG_WRITEFULL	Очередь записи заполнена
QUEUE_FLAG_DEAD	Очередь отменена
QUEUE_FLAG__REENTER	Избегание повторных вхождений
QUEUE_FLAG_PLUGGED	Подключение очереди

```
   include/linux/blkdev.h
323 /*
324 *       защита структур очереди от повторных вхождений
325 */
326 spinlock_t  *queue_lock;
327
328 /*
329 *       kobject очереди
330 */
331 struct kobject kobj;
332
333
334 * настройки очереди
335 */
336 unsigned long  nr__requests; /* Максимальный номер запроса */
```

```
337     unsigned int  nr_congestion_on;
338     unsigned int  nr_congestion_off; 339
340     unsigned short  max_sectors;
341     unsigned short  max_phys_segments;
342     unsigned short  max_Jiw_segments;
343     unsigned short  hardsect_size;
344     unsigned int  max_segment_size; 345
346     unsigned long  seg_boundary_mask;
347     unsigned int  dma_alignment; 348
349    struct blk_queue_tag *queue_tags;
350
351    atomic_t   refcnt;
352
353    unsigned int   in_flight;
354
355     /*
356     * sg stuff
357    */
3 58   unsigned int      sg_timeout;
359     unsigned int      sg__reserved_size;
360    };
```

Строки 323-360

Эти переменные определяют управляемые ресурсы для очереди запросов, такие, как блокировка (строка 326) и объекты ядра (строка 331). Также присутствуют специальные настройки очереди, такие, как максимальное число запросов (строка 336) и физическое содержимое блочного устройства (строки 340-347). Кроме этого, можно определить атрибуты SCSI (строки 355-359), если они применимы к данному блочному устройству. Если вы хотите использовать управляемые командные очереди, используйте структуру queue_tags (строка 349). Поля refcnt и in_f light (строки 351 и 353) считают количество обращений к очереди (обычно используются для блокировки) и количество запросов, которые обрабатываются «на лету».

Очереди запросов, используемые блочными устройствами, инициализируются в ядре Linux 2.6 простым вызовом следующей функции в функции__ init устройства. Внутри этой функции мы можем увидеть анатомию очереди запроса и связанные с ней вспомогательные функции. В ядре Linux 2.6 каждое блочное устройство управляет собственной блокировкой в отличие от более ранних версий Linux и передает циклическую блокировку в качестве второго аргумента. Первый аргумент - это функция запроса, предоставляемая драйвером блочного устройства.

```
drivers /block/ ll rw  blk. c
1397  request_queue_t *blk_init_queue (request  fn proc *rfn,
                                       spinlock t *lock)
1398  {
13 99              request  queue_t   *q;
1400      static  int  printed;
1401
1402   q = blk alloc queue(GFP KERNEL);
1403   if (!q)
1404   return NULL;
1405
1406   if (blk init free_list(q))
1407    goto out init;
1408
1409   if (Iprinted) {
1410    printed = 1;
1411    printk("Using %s io scheduler\n",
               chosen__elevator->elevator_name) ;
1412   }
1413
1414   if (elevator init(q, chosen__elevator))
1415    goto out elv;
1416
1417   q->request fn   = rfn;
1418   q->back  merge fn  = ll back merge fn;
1419   q->front_merge_fn  = ll_front_merge_fn;
1420   q->merge requests fn = ll merge requests fn;
1421   q->prep rq fn   = NULL;
1422   q->unplug fn  = generic unplug  device;
1423   q->queue flags   = (1 « QUEUE_FLAG_CLUSTER) ;
1424   q->queue lock  = lock;
1425
1426    blk queue segment boundary(q, Oxffffffff);
1427
1428   blk queue make request(q,    make request);
1429   blk  queue max segment size(q/ MAX SEGMENT  SIZE) ;
1430
1431   blk queue max hw segments (q, MAX HW SEGMENTS) ;
1432   blk queue max phys segments (q, MAX PHYS SEGMENTS) ;
1433
1434   return q;
1435  out elv:
1436   b1k c1eanup queue(q);
1437  out_init:
```

```
1438    kmem_cache_free(requestq_cachep, q);
1439    return NULL;
1440  }
```

Строка 1402

Выделение очереди из памяти ядра и обнуление ее содержимого.

Строка 1406

Инициализация списка запросов, содержащего очередь чтения и очередь записи.

Строка 1414

Связь выбранного листа с очередью и его инициализация.

Строки 1417-1424

Связь лифтозависимых функций с данной очередью.

Строка 1426

Эта функция устанавливает границу объединяемого сегмента и проверяет, чтобы он был не меньше минимального размера.

Строка 1428

Эта функция устанавливает функцию, используемую при изъятии запроса из очереди драйвером. Позволяет использовать для обработки очереди альтернативную функцию.

Строка 1429

Инициализирует верхний предел размера комбинируемых сегментов.

Строка 1431

Инициализация максимального количества сегментов, которые может обработать драйвер.

Строка 1432

Инициализация максимального количества физических сегментов за один запрос. Значения для строк 1429-1432 устанавливаются в include/linux/kerne 1 .h.

Строка 1434

Возвращение инициализированной очереди.

Строки 1435-1439

Вспомогательная функция для очистки памяти в случае возникновения ошибки.

Теперь наши запросы находятся на своих местах и инициализированы.

Перед тем как мы рассмотрим слой обобщенного устройства и обобщенный блочный драйвер, давайте коротко пройдемся по программному слою и посмотрим на манипуляции с вводом-выводом блочного устройства (см. рис. 5.4).

На уровне приложения приложение инициализирует файловую операцию с помощью f read (). При вызове f read () этот запрос передается в слой виртуальной файловой системы (VFS) (описываемой в гл. 4[1]), где хранится структура файла dentry, и через структуру inode. Слой VFS пытается найти запрашиваемую страницу в буфере кеша, и, если она там отсутствует вызывается **обработчик файловой системы (filesystem handler)** для получения требуемого физического блока; inode связан с обработчиком файловой системы, который связан с соответствующей файловой системой. Обработчик файловой системы вызывает **утилиты очереди запросов (request queue utilites),** являющиеся частью **слоя обобщенного блочного устройства (generic block device layer),** для создания корректного запроса для физических блока и устройства. Запрос помещается в **очередь запросов,** поддерживаемую **слоем обобщенного блочного устройства.**

5.2.3 Пример: «обобщенное» блочное устройство

Рассмотрим слой обобщенного блочного устройства. В соответствии с рис. 5.4 он находится выше слоя физического устройства и сразу под слоем файловой системы. Основная задача слоя обобщенного устройства - это поддержание очереди запросов и связанные с ней операции.

Сначала мы регистрируем наше устройство с помощью register_blkdev (maj or, dev_name, fops). Эта функция получает старший номер запроса, имя блочного устройства (появляющееся в директории /dev) и указатель на структуру файловой операции. В случае удачи возвращается желаемый старший номер.

Далее мы создаем структуру gendisk.

Функция alloc__disk(in t minors) в include/linux/genhd. h получает номер раздела и возвращает указатель на структуру gendisk. Теперь посмотрим на структуру gendisk:

```
include/linux/genhd.h
081  struct gendisk {
82    int major;    /* старший номер драйвера */
83    int first minor;
84    int minors;
85    char disk name[16];      /* имя старшего драйвера */
86    struct hd_struct **part;  /* [индекс младшего] */
87    struct block device operations *fops;
88    struct request queue *queue;
89    void *private_data;
```

[1] В гл. 6 (опечатка в оригинале). *Примеч. науч. ред.*

```
090  sector_t capacity;
091
92    int flags;
93    char devfs_name[64];   /* devfs crap */
94    int number;    /* еще то же самое */
95    struct device *driverfs_dev;
96    struct kobject kob j ;
097
98    struct timer  rand_state * random;
99    int policy;
100
101   unsigned sync_io;   /*  RAID  */
102   unsigned long stamp, stamp idle;
103   int in_flight;
104  #ifdef CONFIG_SMP
105   struct disk  stats *dkstats;
106  #else
107   struct disk stats dkstats;
108  #endif
109  };
```

Строка 82

Поле ma j or__num заполняется на основе результата regis ter_blkdev ().

Строка 83

Блочное устройство для жесткого диска может обрабатывать несколько физических устройств. Несмотря на то что это зависит от драйвера, младший номер обычно соответствует каждому физическому приводу. Поле f irst_minor является первым физическим устройством.

Строка 85

Имя disk__name, такое, как hda или sdb, является именем всего диска. (Разделы диска именуются hdal, hda2 и т. д. Они являются *логическими* дисками *внутри* физического диска.)

Строка 87

Поле fops в block_device_operations инициализирует структуру файловой операции. Структура файловой операции содержит указатель на вспомогательную функцию в низкоуровневом драйвере устройства. Эти функции являются драйверозависимыми и не обязательно реализованы во всех драйверах. Обычно реализуются файловые операции open, close, read и write. В гл. 4, «Управление памятью»[1] обсуждается структура файловой операции.

Очевидно, имеется в виду гл. 6, «Файловые системы». *Примеч. науч. ред.*

Строка 88

Поле queue указывает на список запрашиваемых операций, которые должен выполнить драйвер. (Мы еще обсудим инициализацию очереди запросов.)

Строка 89

Поле private_data хранит драйверозависимые данные.

Строка 90

Поле capacity устанавливается в соответствии с размером устройства (в секторах по 512 кб). Вызов set_capacity () должен получать это значение.

Строка 92

Поле flags означает атрибуты устройства. В случае дискового привода это тип носителя, т. е. CD, съемный привод и т. д.

Теперь мы посмотрим, с чем связана инициализация очереди запросов. Когда очередь запросов уже определена, мы вызываем blk_init_queue (request_fn_proc, spinlock__t). Эта функция получает в качестве первого параметра функцию передачи, которая будет вызываться в интересах файловой системы. Функция blk_init_ queue () выделяет очередь с помощью blk__alloc__queue () и затем инициализирует структуру очереди. Второй параметр, blk_init__queue (), - это связанная с очередью блокировка всех операций.

И наконец, для того чтобы сделать блочное устройство видимым для ядра, драйвер должен вызвать add__disk():

```
Drivers/block/genhd.c
193  void add disk(struct gendisk *disk)
194  {
195   disk->flags |= GENHD FL UP;
196   blk register region(MKDEV(disk->major, disk->f irst minor) ,
197     disk->minors, NULL, exact match, exact lock, disk);
198   register disk(disk);
199   blk register queue(disk);
200  }
```

Строка 196

Устройство отображается в ядро на основе своего размера и количества разделов.

Вызов blk_register__region () имеет несколько параметров:

1. В этот параметр упакованы старший номер диска и *первый* младший номер.
2. Это диапазон младших номеров, следующих за первым (если этот драйвер обрабатывает несколько младших номеров).

3. Это загружаемый модуль, содержащий драйвер (если он есть).
4. exact__match - это функция для поиска соответствующего диска.
5. exact_lock - это функция блокировки кода, после того как exact_match найдет нужный диск.
6. disk - это обработчик, используемый exact_match и exact_lock для идентификации нужного диска.

Строка 198
register__disk проверяет раздел и добавляет его в файловую систему.

Строка 199
Регистрация очереди запросов для определенного региона.

5.2.4 Операции с устройством

Базовое обобщенное блочное устройство имеет open, close (освобождение), ioctl и, что самое главное, функцию request. По крайней мере функции open и close могут быть простыми счетчиками. Интерфейс ioctl() может использоваться для отладки и выполнения измерений при прохождении через различные слои программного обеспечения. Функция request, вызываемая, когда запрос помещается в очередь файловой системой, извлекает структуру запроса и обрабатывает его содержимое. В зависимости от того, является ли запрос запросом на чтение или на запись, устройство выполняет соответствующее действие.

К очереди запросов нельзя получить прямой доступ, а только через вспомогательные функции. (Их можно найти в driver /block/elevator, с и include /linux/ blkdev.h.) Для сохранения совместимости с базовой моделью устройства мы можем включить возможность взаимодействия со следующим запросом в нашу функцию request:

drivers/block/elevator.c
186 struct request *elv_next_request (request__queue_t *q)

Эта вспомогательная функция возвращает указатель на следующую структуру запроса. Проверяя ее элементы, драйвер может получить всю информацию, необходимую для определения размера, направления и других дополнительных операций, связанных с данной очередью.

Когда драйвер завершает запрос, он сообщает об этом ядру с помощью вспомогательной функции end_request ():

```
drivers/block/ll_rw_blk.c
2599  void end_request(struct request *req, int uptodate)
```

```
2600    {
2601    if  (!end_that_request_first (req,   uptodate,req->hard_.cur_sectors))   {
2602    add_disk_randomness (req->rq_disk) ;
2603    blkdev_dequeue_request(req);
2604    end_that_request_last(req) ;
2605      }
2606    }
```

Строка 2599

Передает очередь запроса, полученную из elev_next_request ().

Строка 2601

end__that_request_first () передает соответствующее количество секторов [если секторы находятся близко, просто возвращается end_request ()].

Строка 2602

Добавляет в систему пул энтропии. Пул энтропии - это способ генерации случайных номеров в системе из функции, достаточно быстрой для вызова во время обработки прерываний. Базовая идея заключается в том, чтобы собрать байты из данных разных драйверов в системе и сгенерировать на их основе случайное число. Это обсуждается в гл. 10, «Добавление вашего кода в ядро». Еще одно объяснение находится в конце /drivers/char/random.c.

Строка 2603

Удаление структуры запроса из очереди.

Строка 2604

Сбор статистики и приготовление структуры к освобождению.

В этой точке обобщенный драйвер обслуживает запросы до их освобождения.

В соответствии с рис. 5.4 мы имеем слой обобщенного блочного устройства, создающего и поддерживающего очереди запросов. Последним слоем в системе блочного ввода-вывода является аппаратный (или специальный) драйвер устройства. Аппаратный драйвер устройства использует вспомогательные функции очереди запросов из обобщенного слоя для обслуживания запросов из зарегистрированной очереди запросов и передает уведомление, когда запрос завершен.

Аппаратный драйвер устройства обладает знаниями о нижестоящем оборудовании относительно их регистров, ввода-вывода, таймера, прерываний и DMA (обсуждается в подразд. 5.2.9, «Прямой доступ к памяти (DMA)»). (Подробности реализации драйверов для ШЕ и SCSI лежат за пределами рассмотрения этой главы. Мы рассмотрим подробнее аппаратные драйверы устройств в гл. 10, а серия проектов поможет вам написать каркас для собственного драйвера.)

5.2.5 Обзор символьных устройств

В отличие от блочных устройств символьные устройства посылают поток данных. Все последовательные устройства являются символьными. Когда мы используем классический пример контроллера клавиатуры или последовательного терминала в качестве символьного устройства, становится интуитивно понятным, что мы не можем (и не хотим) получать данные от устройства не по порядку. Так мы подходим к серой области пакетной передачи данных. Сеть Ethernet на физическом уровне является последовательным устройством, но на уровне шины используется DMA для передачи в память и из памяти больших порций данных.

Как разработчики драйвера устройства мы можем сделать с устройством что угодно, но на практике мы редко будем получать случайный доступ к аудиопотоку или писать поток на IDE-диск. Несмотря на то что оба примера звучат заманчиво, мы должны придерживаться двух простых правил:

- все устройства ввода-вывода в Linux основаны на файлах;
- все устройства ввода-вывода в Linux являются либо символьными, либо блочными.

Драйвер параллельного порта в конце этой главы - символьный драйвер устройства. Символьные и блочные устройства схожи между собой интерфейсом, основанным на файловом вводе-выводе. Извне оба типа используют файловые операций, такие, как open, close, read и write. Внутри самое главное различие между символьными драйверами устройств и блочными драйверами устройств заключается в том, что символьные устройства не обладают системой блокировки для очередей запросов операций на чтение и на запись (как было сказано ранее). Зачастую для не имеющих буфера символьных устройств прерывание выполняется для каждого полученного элемента (символа). Для блочных устройств, наоборот, получается целая порция (порции) данных и затем для этой порции вызывается прерывание.

5.2.6 Замечание о сетевых устройствах

Сетевые устройства имеют атрибуты как блочных, так и символьных устройств и зачастую рассматриваются как особый класс устройств. Подобно символьным устройствам, на физическом уровне данные передаются последовательно. При этом данные упаковываются в пакеты и передаются на и сетевой контроллер с него с помощью прямого доступа к памяти (обсуждается в подразд. 5.2.9) как для блочных устройств.

Сетевые устройства только упоминаются в этой главе, но из-за своей сложности они выходят за пределы рассмотрения этой книги.

5.2.7 Устройства таймера

Таймер - это устройство ввода-вывода, считающее *сердцебиение* системы. Без концепции прошедшего времени Linux вообще не смог бы функционировать. Гл. 7, «Планирование и синхронизация ядра», описывает системный таймер и таймер реального времени.

5.2.8 Терминальные устройства

Ранние терминалы были телетайпными машинами (отсюда и произошло имя tty для драйвера последовательного порта). Консольное устройство было разработано в середине прошлого века с целью отправки и приема текста по телеграфным сетям. В ранних 60-х телетайп превратился в ранний стандарт RS-232 и стал использоваться во множестве появляющихся микрокомпьютеров. В терминалах 70-х телетайп использовался для связи компьютеров. Настоящие терминалы стали редкостью. Популярные на мейнфреймах и мини-компьютерах в 70-х, терминалы были заменены на компьютерах 80-х программными эмуляторами терминалов. Сам терминал (зачастую называемый «глупым» терминалом) представлял собой обычные монитор и клавиатуру, подключенные по последовательной линии к мейнфрейму. В отличие от ПК они достаточно умны для отправки и получения текстовых данных.

Главная консоль (настраиваемая при загрузке) является первым терминалом, появившимся в системе Linux. Обычно после нее запускается графический интерфейс, а далее при необходимости используется оконный эмулятор терминала.

5.2.9 Прямой доступ к памяти (DMA)

Контроллер DMA является аппаратным устройством, расположенным между устройством ввода-вывода и (обычно) высокопроизводительной шиной системы. Назначение контроллера DMA заключается в перемещении большого массива данных без вмешательства процессора. Контроллер DMA без задействования процессора может быть запрограммирован на перемещение блоков данных в основную память и из нее. На уровне регистров контроллер DMA получает адреса источника и назначения и длину, необходимые для выполнения задачи. Затем, пока основной процессор бездействует, контроллер может посылать порцию данных из устройства в память, из памяти в память и из памяти на устройство.

Многие контроллеры (дисковый, сетевой и графический) имеют встроенный DMA-движок, позволяющий передавать большие объемы данных без участия процессора.

Резюме

Эта глава описывает, как ядро Linux обрабатывает ввод и вывод. То есть мы рассмотрели следующие вопросы:

- Выполнили обзор аппаратуры, используемой ядром Linux для выполнения низкоуровневых операций ввода и вывода, таких, как мосты и шины.
- Рассмотрели, как Linux представляет интерфейсы для блочных устройств.
- Мы рассмотрели различные планировщики Linux и очереди запросов: безоперационный, предельный и предварительный.

Проект: сборка драйвера параллельного порта

Этот проект представляет вашему вниманию основы контроллера параллельного порта и во что сливаются ранее описанные функции ввода-вывода. Параллельный порт обычно интегрирует в Superio часть чипсета и является хорошим примером для написания основы драйвера символьного устройства. Этот драйвер, или динамически загружаемый модуль (module), не особенно полезен, хотя и годится для дальнейшего усовершенствования. Так как мы адресуем устройство на уровне регистров, этот модель может использоваться на системах PowerPC для доступа к вводу-выводу, как описано в документации по отображению в память.

Наш параллельный драйвер устройства использует стандартные open (), close () и, что самое главное, интерфейс ioctl () для иллюстрации архитектуры и внутренней работы драйвера устройства. Мы не будем использовать в этом проекте функции read () и write (), так как функция ioctl () может возвращать значения регистров. (Так как наш драйвер устройства является загружаемым модулем, мы будем называть его просто модулем.)

Мы начнем с краткого описания того, как общаться с параллельным портом, а затем перейдем к рассмотрению основных операций нашего драйвера устройства. Мы используем интерфейс ioctl () для обращения к отдельным регистрам в устройстве и создадим приложение для взаимодействия с нашим модулем.

Аппаратное обеспечение параллельного порта

Любой поиск в сети о параллельном порте выдает огромный массив информации. Так как нашей целью в этой главе является описание модулей Linux, мы коснемся только основ этого устройства.

В этом проекте мы будем экспериментировать на x86-системе. Структуру драйвера легко портировать на PowerPC; для этого нужно просто обратиться к другому устройству на уровне ввода-вывода. Несмотря на то что параллельный порт существует на многих встроенных реализациях PowerPC, он слабо распространен на десктопах (таких, как G4 и G5).

Для настоящего общения с регистрами параллельного порта мы используем inb () и outb(). Мы легко можем использовать readbO и writebO, доступные в io.h на обеих архитектурах - x86 и PPC. Макросы readb () и writeb () являются хорошим выбором для аппаратно-независимой реализации, так как они обращаются к низкоуровневым функциям ввода-вывода, используемым на x86 и PPC.

Параллельный порт на системах x86 обычно является частью устройства Superio или может быть отдельной (PCI) картой, добавляемой в систему. Если вы перейдете на страницу настройки BIOS, мы увидите, что параллельный порт (порты) отображается в системное пространство ввода-вывода. Для систем x86 параллельный порт может располагаться по адресам 0x278,0x378 или Ox3Ьc и использует IRQ 7. Это базовый адрес устройства. У параллельного порта есть три 8-битовых регистра, начинающихся с базового адреса, которые показаны в табл. 5.2. Для примера мы будем использовать базовый адрес 0x378.

Таблица 5.2. Регистры параллельного порта

Бит									Адрес порта ввода-вывода
Регистр данных (вывод)	D7	D6	D5	D4	D3	D2	Dl	DO	0x378 (base+0)
Регистр состояния (ввод)	Busy*	АСК	Paper end	Select	Error				0x379 (base+1)
Управляющий регистр (вывод)					Select[8]	Init	Auto feed[8]	Strobe*	0x379 (base+2)

Низкий активный сигнал.

Регистр данных содержит 8 бит для записи со штырьков разъема.

Регистр состояния содержит входные сигналы с разъема.

Управляющий регистр посылает специфические управляющие сигналы на разъем. Разъем параллельного порта имеет 25-пиновый D-порт (DB-25). Табл. 5.3 демонстрирует, как эти сигналы передаются на отдельные штырьки разъема.

Таблица 5.3. Набор сигналов на штырьках параллельного разъема

Имя сигнала	Номер штырька
Строб (Strobe)	1
DO	
Dl	2
D2	
D3	3
D4	
D5	4
	5
	6
	7

Таблица 5.3. Набор сигналов на штырьках параллельного разъема (Окончание)

D6	**8**
D7	**9**
Подтверждение (Acknowledge)	**10**
Занят (Busy)	**11**
Бумага закончилась (Paper End)	**12**
Внутренний выбор (Select In)	**13**
Автонасыщение (Auto feed)	**14**
Ошибка (Error)	**15**
Инициализация (Initialize)	**16**
Выбор (Select)	**17**
Земля (Ground)	**18-25**

ВНИМАНИЕ! Параллельный порт является чувствительным к статическому электричеству и перегрузкам. Не используйте свой интегрированный (встроенный в материнскую плату) параллельный порт:

- если вы точно уверены в своем умении обращаться с оборудованием;
- если вас не смущает вероятность выхода из строя параллельного порта или всей материнской платы.

Мы настойчиво рекомендуем использовать карту адаптера параллельного порта для этого и других экспериментов

Для операций ввода мы используем переключатель D7 (штырек 9), для **подтверждения** (штырек 10) и D6 (штырек 8), для **занято** (штырек 11) с резисторами по 470 Ом. Для мониторинга вывода мы будем использовать индикаторы LED с данными штырьков с DO по D4 с резисторами **максимального ограничения** 470 Ом. Для этой цели можно использовать старый кабель от принтера или 25-пиновый разъем D-Shell «nana» из ближайшего магазина электроники.

ПРИМЕЧАНИЕ. Хороший программист уровня регистров всегда должен знать как можно больше об аппаратном обеспечении, с которым он работает. Сюда входит отыскание перечня данных для вашего драйвера параллельного порта. В этом перечне данных вы можете найти текущие ограничения/утечки драйвера. На многих сайтах в сети выложены интерфейсные решения для работы с параллельным портом, включая изолированные, расширяемые системы сигналов и резисторов усиления и ослабления. И несмотря на то что они находятся за пределами рассмотрения данной книги, вам стоит с ними ознакомиться самостоятельно.

Этот модуль адресует параллельный порт с помощью функций outb () и inb (). Вспомните гл. 2, «Исследовательский инструментарий», в которой описано, что в зависимости от платформы компиляции эти функции корректно реализуют инструкции in и out для x86 и инструкции lbz и stb для отображаемого в память ввода-вывода на PowerPC. Этот встроенный код можно найти в файле /io. h соответствующей платформы.

Программное обеспечение параллельного порта

Нижеследующее обсуждение посвящено соответствующей функции для этого проекта. Полный листинг программы для parll. c вместе с файлом parll. h приведен в конце этой книги.

1) Настройка файловых операций (fops)

Как говорилось ранее, этот модуль использует open (), close () и ioctl (), как и описанные ранее init и cleanup.

Первым шагом является настройка структуры файловых операций. Эта структура определена в /linux/f s .h, перечисляющем все функции, которые можно реализовать в нашем модуле. Нам не нужно использовать все операции, достаточно только самых необходимых. Поиск в сети по запросу C99 или linux module даст вам больше информации об этих методах. Используя эту структуру, мы сообщаем ядру о местонахождении наших реализаций (или точках вхождения) open, release и iotcl.

```
par11.c
struct file operations parlport fops = {
       .open =  parlport open, .ioctl =
       parlport ioctl, .release =
       parlport_close };
```

Далее мы создаем функции open () и close (). Данные функции-пустышки используются для сигнализации об открытии и закрытии:

```
parll.c
static int parlport_open(struct inode *ino, struct file *filp)
{
  printk("\n parlport open function");
  return 0; } static int parlport close(struct inode *ino, struct
file *filp) {
  printk("\n parlport close function");
  return 0;
}
```

Создадим функцию ioctl (). Обратите внимание, что определение функции делается в начале parll. c:

```
#define MODULEJNAME    "parll"
static  int base  =  0x378;
parll.c
static int parlport ioctl(struct inode *ino, struct file *filp,
     unsigned int ioctl_cmd, unsigned long parm)
{
 printk("\n parlport ioctl function");
 if(_IOC_TYPE(ioctl_cmd) != IOCTL_TYPE) {
  printk("\n%s wrong ioctl type",MODULE_NAME);
  return -1; >
  switch(ioctl cmd)
  { case DATA OUT:
    printk("\n%s ioctl data out=%x", MODULE NAME,(unsigned int)parm);
    outb(parm & Oxff, base+0);
    return (parm & Oxff);
   case GET.JSTATUS:
    parm = inb(base+l);
    printk("\n%s ioctl get status=%x",MODULE_NAME,(unsigned int)parm) ;
    return parm;
   case CTRL OUT:
    printk("\n%s  ioctl Ctrl out=%x",MODULE NAME,(unsigned int)parm);
    outbfparm && Oxff, base+2);
 return 0; }  //end
switch return 0; }
//end ioctl
```

Функция ioctl () делает возможной обработку любых определенных пользователем команд. В нашем модуле мы используем три регистра, связанные с параллельным портом пользователя. Команда DATA_OUT посылает значение из регистра data, команда GET_STATUS читает из регистра status, и, наконец, команда CTRL_OUT позволяет установить сигнал для порта. Несмотря на то что лучше было бы скрыть специфические для устройства операции внутри функций read() и write (), этот модуль, работоспособен, так как служит только для экспериментов с вводом-выводом, а не для реального применения.

Эти три команды определены в заголовочном файле parll. h. Они создаются с применением вспомогательных функций IOCTL для проверки типов. Вместо использования целого для представления функции IOCTL мы используем IOCTL-макрос проверки типов 10 (*type, number),* где параметр type определен как ***p*** (для параллельного порта), а **number** как текущий номер IOCTL, используемый в выражении. В начале parlport_ ioctl () мы проверяем тип, которым должен быть **p.** Так как код **приложения** использует тот же заголовочный файл, что и драйвер, интерфейс будет согласованным.

2) Настройка функции инициализации модуля

Модуль инициализации используется для связи модуля с операционной системой. Он может применяться для ранней инициализации необходимых структур данных. Так как драйверу параллельного порта не требуется сложных структур данных, мы просто регистрируем модуль.

```
parll.c
static int parll_init (void)
{
  int retval;
  retval= register_chrdev(Major, MODULE_NAME, &parlport_fops) ;
  if(retval < 0) {
  printk( "\n%s: can't register",MODULE NAME);
  return retval; }
  else
  {
   Maj or=retval;
   printk("\n%s:registered, Major=%d",MODULE_NAME,Major);
   if(request region(base,3,MODULE NAME)) printk("\n%s:I/0
   region busy.", MODULE_NAME); }
   return 0;
}
```

Функция init_module() отвечает за регистрацию модуля в ядре. Функция regis ter_chrdev () получает старший номер запроса (описывается в разд. 5.2 и далее в гл. 10; если 0, ядро назначает ее модулю). Вспомните, что старший номер хранится в структуре inode, на которую указывает структура dentry, на которую указывает структура файла. Вторым параметром является имя устройства, отображаемое в /proc/ devices. Третьим параметром является только что описанная структура операций.

В случае успешной регистрации функция init вызывает request_region() с базовым адресом параллельного порта и длины диапазона (в байтах) вставляемых регистров.

Функция init_module () возвращает отрицательное число в случае неудачи.

3) Настройка функции очистки модуля

Функция cleanup__module () отвечает за отмену регистрации модуля и освобождение запрошенного ранее диапазона ввода-вывода:

```
parll.c
static void parll_cleanup( void )
{
   printk("\n%s:cleanup ",MODULE_NAME);
   release_region(base,3);
   unregis ter_chrdev (Maj or, MODULE_NAME) ; }
```

И наконец, мы помещаем запрашиваемый init и точку очистки:

```
parll.c
module_init (parll_init) ;
module_exit(parll_cleanup);
```

4) Вставка модуля

Теперь мы вставляем наш модуль в ядро, как в предыдущем проекте, с помощью

Lkp:~# **insmod parll.ko**

Загляните в /var/ log/messages, где отображается вывод нашей функции init () и уделите особое внимание отображающимся там **старшим** возвращаемым номерам.

Как в предыдущем проекте, мы просто вставляем наш модуль в ядро и удаляем его оттуда. Теперь нам нужно связать наш модуль с файловой системой с помощью команды mknod. Введите в командной строке следующее:

Lkp:~# **znknode /dev/parll c <XXX> O**

Параметры:

- **c**. Создается символьный специальный файл (в отличие от блочного).
- **/dev/parll**. Путь к нашему устройству (для открытого вызова).
- **XXX.** Старший номер, возвращаемый во время init (из var/log/messages).

- 0. Младший номер нашего устройства (в данном примере не используется).

Например, если вы увидите старший номер 254 в /var/log/messages, команда будет выглядеть следующим образом:

Lkp:~# mknode /dev/parll c 254 O

5) Код приложения

Мы создаем простое приложение, которое открывает наш модуль и начинает бинарный отчет на штырьках с DO до D7.

Этот код компилируется с помощью дсс app. c. По умолчанию программа собирается в a. out.

```
app. c
000  //Приложение, использующее драйвер параллельного порта

tinclude <fcntl.h>
tinclude <linux/ioctl.h>
004  #include "parll.h"

main() {
  int fptr;
  int i,retval,parm =0;
  printf(H\nopening driver now"); 012   if((fptr
= open(-/dev/parll-,0_WR0NLY))<0)
  {
   printf ("\nopen failed, returned=?%d-, fptr);
   exit(l);
  }

018   for(i=0;i<0xff;i++)
  {
20    system("sleep .2");
21    retval=ioctl(fptr,DATA OUT,parm);
22    retval=ioctl(fptr,GET_STATUS,parm);

024   if(!(retval & 0x80))
    printf(H\nBusy signal count=%xH,parm);
   if(retval & 0x40)
027     printf(-\nAck signal count=%x",parm);
028  //   if(retval & 0x20)
//   printf("\nPaper end signal count=%x",parm);
//   if(retval & 0x10)
//   printf(-\nSelect signal count=%x-,parm);
//   if(retval & 0x08)
```

```
033    //        printf("\nError signal  count=%x",parm);
    parm++ ; }
038     close(fptr);
}
```

Строка 4

Общий для приложения и драйвера заголовочный файл, содержащий главные макросы IOCTL для проверки типов.

Строка 12

Открытие драйвера для получения файлового описателя нашего модуля.

Строка 18

Вход в цикл.

Строка 20

Замедление цикла, чтобы мы могли увидеть огоньки и отсчет.

Строка 21

Используя файловый указатель, посылаем команду DATA_OUT в модуль, который, в свою очередь, использует outb () для записи последних значащих 8 бит параметров для порта данных.

Строка 22

Чтение байта состояния с помощью ioctl с команды GET__STATUS.

Строки 24-27

Смотрим интересующие нас биты. Обратите внимание, что Busy — это низкий активный сигнал, поэтому, когда ввода-вывода нет, мы читаем его как true.

Строки 28-33

Эти строки вы сможете раскомментировать, когда захотите усовершенствовать дизайн.

Строка 38

Закрытие модуля

Если вы собрали разъем, как показано на рис. 5.5, сигналы busy и аск посылаются, когда два значащих бита счетчика включены. Код приложения считывает эти биты и производит соответствующий вывод.

Мы осветили только основные элементы драйвера символьного устройства. Зная эти функции, легко проследить работу кода или создать собственный драйвер. Добавление

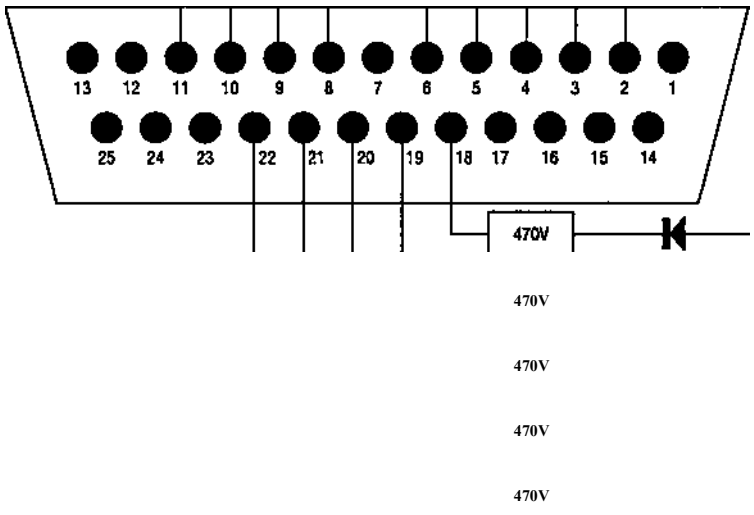


Рис. 5.5. Собранный разъем

в этот модуль обработчика прерывания породит вызов к request_irq () и передачу номера желаемого IRQ и имени обработчика. Его нужно добавить в init_module (). Вот несколько возможных способов усовершенствования драйвера:

- **Заставить модуль параллельного порта обрабатывать прерывания таймера в качестве ввода.**
- **Как можно превратить 8 бит ввода-вывода в 16,32, 64? Чем мы жертвуем?**
- **Посылать символы из параллельного порта с помощью функций записи модуля.**
- **Добавить функцию прерывания для использования сигнала аск.**

Упражнения

1. **Загрузите модуль. В качестве какого файла модуль отобразится в файловой системе?**
2. **Найдите старший и младший номера файла загруженного устройства.**
3. **В каких случаях стоит использовать предельный планировщик ввода-вывода вместо предварительного планировщика ввода-вывода?**
4. **В каких случаях стоит использовать безоперационный планировщик ввода-вывода вместо предварительного планировщика ввода-вывода?**
5. **Какими характеристиками обладают контроллер северного моста и контроллер южного моста?**
6. **В чем заключается преимущество встраивания функции в чип Superio?**
7. **Почему мы до сих пор не видим интегрированных в Superio-чип графических и сетевых решений?**
8. **В чем заключается главное различие и преимущества *журналируемых* файловых систем, таких, как ext3, над стандартными файловыми системами как ext2?**
9. **В чем заключается основа теории предварительного планировщика ввода-вывода? Для чего эта технология подходит лучше всего - для дискового привода или для диска в оперативной памяти?**
10. **В чем заключается основное различие между *блочными* и *символьными* устройствами?**
11. **Что такое DMA? Благодаря чему этот способ обеспечивает эффективную передачу данных?**
12. **Для чего изначально использовались телетайпные машины?**

Глава 6

Файловые системы

В этой главе:

Вычислительная техника занимается хранением, выдачей и обработкой информации. В гл. 3, «Процессы: принципиальная модель выполнения», мы говорили о том, что процессы являются базовыми единицами выполнения, и рассматривали, как процессы обрабатывают информацию, сохраняя ее в своем адресном пространстве. При этом адресное пространство ограничивалось тем, что существовало только во время жизни процесса и хранилось преимущественно в системной памяти. Файловая система решает проблему необходимости большей вместимости, долговременного (non-volatile) хранения информации на носителе, отличном от регистров памяти. Долговременная информация - это данные, которые продолжают существовать после завершения обрабатывающего их процесса или завершения работы операционной системы.

Хранение информации на внешнем носителе порождает проблему, как эту информацию представлять. Базовой единицей хранения информации является файл. Файловая система или подсистема работы с файлами является компонентом операционной системы, отвечающей за структуру файлов, их обработку и защиту. Эта гла-I ва раскрывает темы, связанные с реализацией файловой системы Linux.

6.1 Общая концепция файловой системы

Мы начнем с описания концепции, лежащей в основе файловой системы Linux. Многим из вас знакомы эти концепции из-за их связи с использованием Linux и программированием приложений пользовательского пространства. Если вы хорошо разбираетесь в общих концепциях файловой системы, вы можете пропустить этот раздел и перейти сразу к разд. 6.2, «Виртуальная файловая система Linux».

6.1.1 Файл и имена файлов

Слово *фат* берет свое начало из терминологии реального мира. Информация хранилась в файлах еще до появления электровакуумных ламп. Файлы реального мира состоят из одного или нескольких бумажных страниц определенного размера. Сами файлы обычно хранятся в шкафах.

В Linux файл представляет собой линейную последовательность байтов. Файловую систему не интересует значение этих байт (так же как шкаф совершенно не связан с содержащимися в нем файлами), но они чрезвычайно важны для пользователя. Файловая система предоставляет пользователю интерфейс для хранения данных и прозрачной манипуляции с физическими данными на внешних устройствах.

Файл в Linux имеет множество атрибутов и характеристик. Наиболее знакомый пользователям атрибут - это имя файла. Обычно в имени файла находит отражение его содержимое. Имя файла может иметь расширение (filename extension), являющееся дополнением к имени файла, записываемым в его конце через точку. Расширение предоставляет

дополнительную возможность для разделения содержимого приложений пользовательского пространства. Например, все рассматриваемые нами в примерах файлы имеют расширение .h или .c. Программы пользовательского пространства, такие, как компиляторы и сборщики, благодаря этим индикаторам понимают, что имеют дело с заголовочным или исходным файлом соответственно.

Несмотря на то что расширение может быть важным для пользовательских приложений, таких, как компилятор, для операционной системы оно безразлично, так как она имеет дело с файлом только как с контейнером байтов вне зависимости от его содержимого и назначения.

6.1.2 Типы файлов

Linux поддерживает множество типов файлов, включая обычные файлы, директории, ссылки, файлы устройств, сокеты и каналы. **Обычные файлы (regular files)** включают в себя бинарные файлы и ANSI-файлы. ANSI-файл - это просто последовательность текста, которая может быть отображена и понята пользователем без предварительной интерпретации программой. Некоторые ANSI-файлы являются исполнимыми и называются **сценариями (scripts).** Такие файлы выполняются программами, называемыми интерпретаторами. Оболочка в своей основе тоже является интерпретатором. Исполнимые файлы являются не ANSI-файлами и содержат на первый взгляд бессмысленный набор данных. Эти файлы имеют внутренний формат, который интерпретируется ядром при выполнении программы. Формат известен как **объектный формат файла,** и каждая операционная система интерпретирует собственный формат объектного файла. Гл. 9, «Построение ядра Linux», описывает объектный формат файла подробнее.

В Linux файлы организованы в иерархическую систему директорий, наподобие показанной на рис. 6.1. **Директория (directory)** содержит файлы и необходима для поддержания структуры файловой системы. Следующие разделы более подробно описывают директории и файловые структуры Linux.

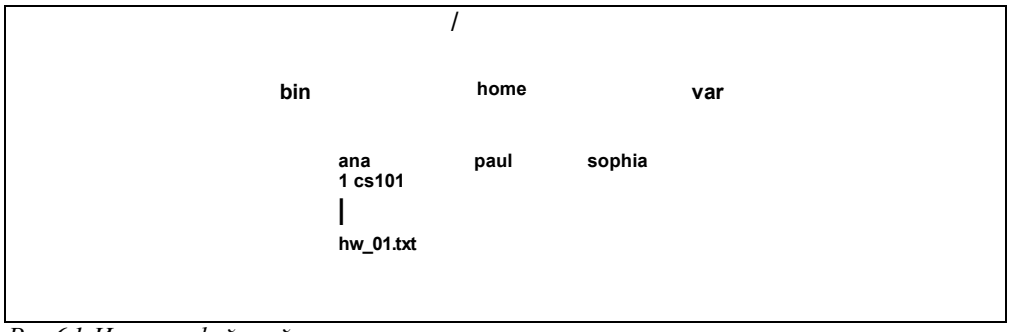


Рис. 6.1. Иерархия файловой системы

Ссылка (link) - это файл, указывающий на другой файл, т. е. файловый указатель. Такие файлы просто содержат информацию, необходимую для доступа к другому файлу.

Файл устройства - это представление устройства ввода-вывода, используемое для доступа к аппаратному устройству. Программа, которой требуется доступ к устройству ввода-вывода, может использовать те же атрибуты, которые применяются к файлам для получения тех же результатов при работе с устройством. Существует два основных типа устройств: **блочные устройства,** передающие блоки данных, и **символьные устройства,** передающие данные посимвольно. Гл. 5, «Ввод-вывод», более подробно описывает устройства ввода-вывода.

Сокеты и **каналы** являются формами межпроцессной коммуникации (Interprocess Communication, IPC). Эти файлы поддерживают направленную передачу данных между процессами. Мы не будем рассматривать эти специальные файлы.

6.1.3 Дополнительные атрибуты файла

Файлы имеют еще несколько атрибутов помимо имени, типа и данных. Операционная система ассоциирует с каждым файлом дополнительную информацию, такую как разрешения для доступа к файлу. Защита файла становится чрезвычайно важной на многопользовательских системах, таких, как Linux. Пользователи разделяются на три категории:

- пользователь, или владелец, файла;
- группа, пользователи, принадлежащие к группе, которой принадлежит файл;
- все остальные, куда включаются все пользователи системы, не принадлежащие к группе файла.

Для каждого из этих пользователей определен специальный набор разрешений. Несмотря на то что с файлом можно проделать множество операций, Linux обобщает разрешения на выполнение трех файловых операций: чтение, запись и выполнение. Так как каждый класс операций применяется ко всем трем категориям пользователей, каждый файл имеет набор из девяти связанных с ним разрешений.

Другие атрибуты файла включают размер файла, метку времени создания, время последнего изменения, которые отображаются утилитой Is. Когда мы рассмотрим реализацию файлов в ядре, мы увидим, что многие другие атрибуты файла являются невидимыми для пользователя.

6.1.4 Директории и пути к файлам

Директория - это файл, поддерживающий иерархическую структуру файловой системы. Директория следит за содержащимися в ней файлами, внутренними директориями и информацией о самой себе. В Linux каждый пользователь обладает собственной «домашней директорией» (home directory), в которой хранятся его файлы и создается его собственное дерево директорий. На рис. 6.1 мы можем увидеть пример организации древовидной структуры файловой системы.

При организации файловой системы в древовидную структуру отдельное имя файла не позволяет обнаружить файл; нам нужно знать, где в дереве этот файл находится. **Путь к файлу (pathname)** описывает местонахождение файла. Путь к файлу можно описать относительно корня дерева. Такой путь будет называться **абсолютным (absolute pathname).** Абсолютный путь начинается с корневой директории, обозначаемой /. Имя узла директории, т. е. имя директории, завершается /, например: bin/. Таким образом, абсолютный путь к файлу представляет собой набор всех узлов директорий, которые нужно посетить, чтобы добраться до файла. На рис. 6.1 абсолютным путем к файлу hwl. txt является /hime/ana/cslOl /hwl. txt. Кроме этого? путь может быть представлен **относительным путем (relative pathname).** Это зависит от **рабочей директории (working directory)** связанного с файлом процесса. Рабочая директория, или текущая директория, - это директория, связанная с выполнением процесса. Таким образом, если рабочей директорией нашего процесса будет /hime/ana/, мы можем ссылаться на файл как на cslOl/hwl. txt.

В Linux директории содержат файлы, выполняющие различные задачи во время работы операционной системы. Например, разделяемые файлы хранятся в /usr и /opt, а неразделяемые в /etc и /boot. Таким же образом нестатические файлы, содержимое которых изменяется системными программами, хранятся в директории vcertain в /var. Информацию о стандарте на структуру каталогов фаловой системы (file system hierarchy standard) можно найти по адресу http: //www.pathname. com/fhs[1].

В Linux с каждой директорией связаны две сущности: . (читается как «дот») и .. (читается как «дот дот»). Означает текущую директорию, а вторая - родительскую директорию. Для корневой директории . и .. означают текущую директорию. (Другими словами, корневая директория является собственной родительской директорией.) Соответственно меняется и запись относительного пути. В нашем предыдущем примере рабочей директорией была /home/ana и относительным путем к нашему файлу cswlOl/ hwl. txt. Относительным путем для hwl. txt в директории paul из нашей рабочей директории будет . . /paul/cal 01 /hwl. txt, так как вначале нам нужно подняться на уровень выше.

6.1.5 Файловые операции

Файловые операции включают в себя все операции, которые система разрешает проделывать с файлами. В целом файлы можно создавать и уничтожать, открывать и закрывать, читать и перезаписывать. Помимо этого, файлы можно переименовывать и изменять их атрибуты. Файловая система предоставляет системным вызовам интерфейсы для этих операций, и они, в свою очередь, окружены несколькими оболочками для доступа из пользовательских приложений с помощью связываемых библиотек. Мы рассмотрим некоторые из этих операций и их реализацию в файловой системе Linux.

[1] Русский перевод можно найти по адресу http://rus-linux.net/MyLDP/file-sys/fhs-2.2-rus/. ***Примеч. науч. ред.***

6.1.6 Файловые описатели

Файловые описатели имеют тип int, используемый системой для идентификации открытых файлов. Системный вызов open () возвращает описатель файла, который можно использовать в дальнейших операциях с этим файлом. В следующих разделах мы рассмотрим, как описатель файла описывается в терминах ядра.

Каждый процесс хранит массив файловых описателей. Когда мы обсуждали структуры ядра для поддержки файловой системы, мы видели, как эта информация организуется в массив. Существует соглашение, согласно которому первый элемент массива (файловый описатель 0) ассоциирован со стандартным вводом, второй элемент (файловый описатель 1) - со стандартным выводом, а третий (файловый описатель 2) - со стандартным выводом ошибок. Они позволяют приложениям открывать файлы стандартного ввода, вывода и ошибок. Рис. 6.2 иллюстрирует массив файловых описателей, связанных с процессом.

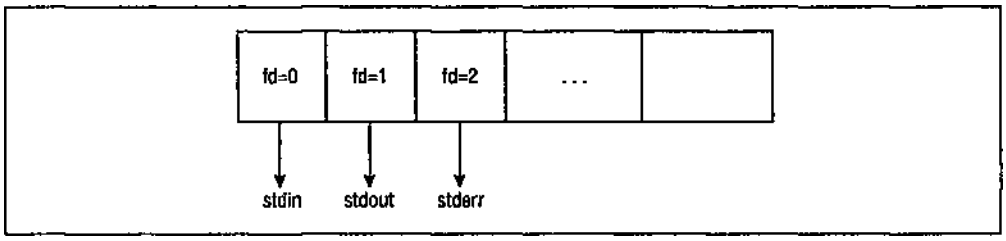


Рис. 6.2. Массив файловых описателей

Файловые описатели назначаются на основе «наименьшего доступного индекса». Поэтому, если процесс открывает много файлов, назначаемые файловые описатели будут увеличиваться по сравнению с предыдущими, до тех пор пока первые открытые файлы не будут закрыты. Чтобы в этом убедиться, мы рассмотрим, как системные вызовы открытия и закрытия манипулируют с файловыми описателями. При этом за время своей жизни процесс может открыть два различных файла и назначить им один и тот же описатель, если первый файл будет закрыт перед открытием второго. И наоборот, по отдельности два разных описателя файла могут указывать на один и тот же файл.

6.1.7 Блоки диска, разделы и реализация

Для понимания концепции реализации файловой системы нам нужно понимать некоторые базовые концепции работы жесткого диска. Жесткий диск записывает данные с помощью магнитной записи. Жесткий диск содержит несколько вращающихся дисков, на которые записываются данные. Головка (head), смонтированная на механической руке, движется над поверхностью диска, читает и записывает данные вдоль радиуса диска, прямо как гвоздь по вращающемуся столу. Сам диск вращается наподобие поворотного стола. Каждый диск разбивается на множество концентрических кругов, называемых

дорожками (tracks). Дорожки начинаются снаружи и идут к центру диска. Группы одинаковых дорожек (на всех дисках) называются **цилиндрами.** Каждая дорожка, в свою очередь, разделяется на (обычно) секторы по 512 кб. Цилиндры, дорожки и головки образуют **геометрию (geometry)** диска.

Пустой диск перед использованием необходимо **отформатировать (format).** Форматирование создает дорожки, блоки и разделы на диске. Раздел - это логический диск **(logical disk)** и то, как операционная система выделяет и использует геометрию жесткого диска. Разделы позволяют разделить один диск таким образом, чтобы казалось, что присутствует несколько дисков. Это позволяет использовать несколько файловых систем на одном диске. Каждый раздел разделяется на дорожки и блоки. Создание дорожек и блоков на диске выполняется с помощью программ наподобие f df ormat[1], а создание логических дисков выполняется с помощью программ, подобных f disk.

Дерево файлов Linux предоставляет доступ более чем к одной файловой системе. Это означает, что, если у вас есть диск с несколькими разделами, каждый со своей файловой системой, вы можете видеть в одном логическом пространстве имен всю файловую систему целиком. Это возможно благодаря тому, что каждая система присоединяется к дереву файловой системы Linux с помощью команды mount. Мы говорим, что файловая система смонтирована для обозначения того факта, что файловая система присоединена к главному дереву и доступна через него. Файловые системы монтируются в директории[2]. Директории, куда монтируются файловые системы, называются **точками монтирования (mount point).**

Одной из главных сложностей в реализации файловых систем является определение того, как операционной системе следить за последовательностью байтов, образующей файл. Как упоминалось ранее, пространство раздела диска разделяется на порции пространства, называемые блоками. Размер блока зависит от реализации. Управление блоками определяет скорость доступа к файлу и уровень фрагментации[3], а также теряемое пространство. Например, если размер нашего блока равен 1024 байтам и размер файла равен 1567 байт, файл занимает два блока. Операционная система следит за блоками, принадлежащими конкретному файлу, храня необходимую информацию в структуре, называемой **индексным узлом (index node, inode).**

6.1.8 Производительность

Существует множество способов увеличения производительности файловой системы. Одним из способов является поддержание внутренней инфраструктуры в ядре, обеспечи-

[1] **fdformat используется для низкоуровневого форматирования (создания дорожек и секторов) дискет. Формати рование дисков IDE и SCSI обычно производится в заводских условиях.**

[2] **В терминах деревьев можно сказать, что вы добавляете поддерево к узлу основного дерева.**

[3] **Мы касались фрагментации в гл. 4, «Управление памятью», и видели, как появляются бесполезные дыры в памяти. Аналогичная проблема существует и при хранении информации на жестких дисках.**

вающей быстрый доступ к inode, соответствующему данному пути к файлу. При рассмотрении реализации файловой системы мы посмотрим, как это сделано в ядре.

Кеш страниц - это еще один способ повышения производительности файловой системы. Кеш страниц - это набор страниц^памяти. Он разработан таким образом, чтобы кешировать множество типов страндаЄ принадлежащих файлам на диске, файлам в памяти или другим страничным объектам, к которым имеет доступ ядро. Такой механизм кеширования уменьшает необходимость доступа к диску и благодаря этому увеличивает производительность системы. Эта глава показывает, как кеш страниц работает с доступом к диску при работе с файлами.

6.2 Виртуальная файловая система Linux

Реализация файловой системы различается от системы к системе. Например, в Windows реализация размещения файла на блоках диска отличается от размещения файла на блоках диска в файловой системе UNIX. На самом деле у Microsoft существует множество реализаций файловых систем, связанных с различными операционными системами: MS_DOS для DOS и Win 3.x, VFAT для Windows 9x и NTFS для Windows NT. Операционная система UNIX также имеет различные реализации, такие, как SYSV и MINIX. Сам Linux использует такие файловые системы, как ext2, ext3 и ReiserFS.

Поддержка множества файловых систем является большим преимуществом Linux. Вы не только можете просматривать файлы с родных файловых систем (ext2, ext3 и ReiserFS), но и с других файловых систем, принадлежащих к другим операционным системам. В единственной системе Linux вы можете получать доступ к файлам различных форматов. Табл. 6.1 иллюстрирует поддерживаемые на данный момент файловые системы. Для пользователя между этими файловыми системами нет никакой разницы; он может беспрепятственно монтировать любую файловую систему к оригинальному пространству имен дерева.

Linux поддерживает несколько файловых систем на одном диске. Кроме этого, поддерживаются монтируемые по сети файловые системы и специальные файловые системы, используемые для приводов, не применяющих для записи намагниченные диски. Например, procf s является псевдофайловой системой. Эта виртуальная файловая система отвечает за хранение информации об особенностях вашей системы. Файловая система proofs не занимает места на диске, а помещаемые в нее файлы создаются на лету. Другой подобной файловой системой является devf s[1], предоставляющей интерфейс для драйвера устройства.

[1] В Linux 2.6 devf s заменила udev, до сих пор поддерживаемую на минимальном уровне. Более подробную информацию о udev можно найти по адресу http://www.kernel.org/pub/linux/utils/kernel/ hotplug/udev-FAQ.

Таблица 6.1. Некоторые поддерживаемые Linux файловые системы

Файловая система	Описание
ext2	Вторая расширенная файловая система
ext3	Журналируемая файловая система ext3
Reiserfs	Журналируемая файловая система
JFS	Журналируемая файловая система IBM
XFS	Высокопроизводительная SGI Irix-журналируемая файловая система
MINIX	Оригинальная файловая система Linux, файловая система ОС minix
ISO9660	Файловая система CD-ROM
JOILET	Расширение файловой системы CD-ROM от Microsoft
UDF	Альтернативная файловая система CDROM-, DVD-дисков
MSDOS	Дисковая операционная система Microsoft
VFAT	Таблица виртуального выделения файлов Windows 95
NTFS	Файловая система Windows NT, 2000, XP, 2003
ADSF	Дисковая файловая система Acorn
HFS	Файловая система Apple Macintosh
BEFS	Файловая система BeOs
FreeVxfs	Поддержка Veritas Vxfs
HPFS	Поддержка OS/2
SysVfs	Поддержка файловой системы System V
NFS	Поддержка сетевой файловой системы
AFS	Файловая система Anrew (также сетевая)
UFS	Поддержка файловой системы BSD
NCP	Файловая система NetWare
SMB	Samba

Linux добивается такого «маскарада» физических аспектов файловых систем с помощью специального уровня абстракции между пользовательским пространством и физической файловой системой. Этот слой называется виртуальной файловой системой (virtual filesystem, VFS). Она разделяет специфическую структуру файловой системы и функ-

ции самого ядра. VFS управляет связанными с файловой системой системными вызовами и преобразует их в функции, соответствующие типу файловой системы. На рис. 6.3 продемонстрирована структура управления файловой системой.

Пользовательские приложения получают доступ к VFS через системные вызовы. Реализация поддержки каждой файловой системы должна содержать набор функций, выполняющих поддерживаемые VFS операции (например, открытие, чтение, запись и закрытие). VFS следит за поддерживаемыми файловыми системами и функциями, выполняющими каждую из операций. Из гл. 5 вы знаете, что слой обобщенного блочного устройства находится между файловой системой и настоящим драйвером устройства. Такой слой абстрагирования позволяет реализовывать специфический для файловой системы код независимо от специфического устройства, на котором эта файловая истема располагается.

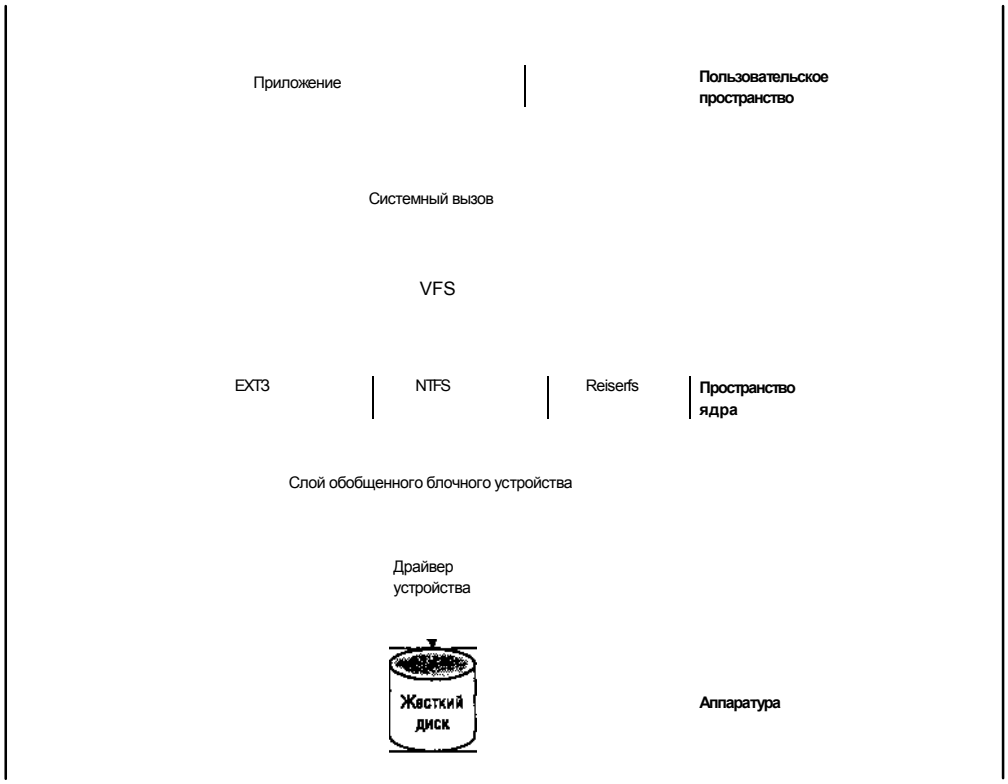


Рис. 6.3. VFS Linux

6.2.1 Структуры данных VFS

VFS опирается на структуры данных, хранящие обобщенное представление файловой системы.

Вот эти структуры данных:

- **superblock. Хранит информацию, связанную со смонтированной файловой системой.**
- **inode. Хранит связанную с файлом информацию.**
- **file. Хранит информацию, связанную с открытым файлом процессом.**
- **dentry. Хранит информацию, связанную с путем к файлу и указывающую на файл.**

В дополнение к этим структурам VFS использует дополнительные структуры, такие, как vf smount и nameidata, хранящие информацию о монтировании и о пути к файлу соответственно. Мы увидим, как эти две структуры связаны с главными, описанными выше, по мере того как будем рассматривать их по отдельности.

Составляющие VFS структуры связаны с действиями, которые можно производить с представляемыми этими структурами объектами. Эти действия определены в таблице операций для каждого объекта. Таблица операций представляет собой список указателей на функции. Мы определяем таблицу операций для каждого объекта при его описании. Рассмотрим эти структуры поближе. (Обратите внимание, что мы не задерживаемся на механизме блокировки для простоты и ясности.)

6.2.1.1 Структура superblock

Когда файловая система смонтирована, вся связанная с ней информация хранится в структуре super__block. Для каждой файловой системы существует отдельная структура superblock. Мы рассмотрим определение этой структуры, а затем разъясним назначение наиболее важных полей.

```
include/linux/fs.h
666    struct super block {
667    struct list_heads_list;
668    dev t s dev;
669    unsigned longs blocksize;
670    unsigned longs_old_blocksize;
671    unsigned chars blocksize  bits;
672    unsigned chars dirt;
673    unsigned long longs_maxbytes;
674    struct file  system type*s  type;
675    struct super  operations*s op;
676    struct dquot operations*dq op;
677    struct quotactl_ops*s_qcop;
678    struct export_operations*s__export_op;
```

```
679   unsigned longs_flags;
680   unsigned longs_magic;
681   struct  dentry*s_root;
682   struct  rw__semaphore s_umount
683   struct  semaphores_lock;
684   int       s_count;
685   int       s_syncing;
686   int       s_need_sync_f s ;
687  atomic. _t  s_active;
688   void      *s_security;
689
690   struct  list_head  s_dirty;
691   struct  list_head  s_io;
692   struct  hlist_head s_anon;
693   struct  list_head  s_files;
694
695   struct  block_device  *s__bdev;
696   struct  list_head  s_instance
697   struct  quota_info  s__dquot;
698
699   char    s_id[32];
700
701   struct  kobject  kobj;
702    void   *s_fs__info;

708   struct  semaphore s_vfs_renam

709 };
```

Строка 667

Поле s_list имеет тип lis^head[1] и указывает на следующий и предыдущий элементы циклического двусвязного списка, куда входит данный super__block. Как и многие другие структуры в ядре Linux, структура super_block хранится в циклическом дву связном списке. Тип данных list_head содержит указатель на две другие структуры list_head: list_head следующего объекта superblock и list_head предыдущего объекта superblock. [Глобальная переменная super_blocks (fs/super. c) указывает на первый элемент списка.]

Строка 672

Для файловых систем на дисках структура superblock заполняется информацией, изначально расположенной в специальных секторах диска и загружаемой в структуру superblock. Так как VFS позволяет редактирование полей структуры su-

[1] **Гл. 2, «Исследовательский инструментарий», описывает тип данных list_head подробнее.**

perblock, содержащаяся в структуре superblock информация может оказаться рассинхронизированной с данными на диске. Это поле указывает, что структура superblock была изменена и ее необходимо синхронизировать с диском.

Строка 673

Это поле типа unsigned long определяет максимальный допустимый размер файла в файловой системе.

Строка 674

Структура siperblock содержит общую информацию о файловой системе. При этом ее необходимо связать со специфической информацией о файловой системе (например, MSDOS, ext2, MINIX и NFS). Структура f ile_system_type хранит специфическую для файловой системы информацию, по одной структуре, для каждого типа сконфигурированной в ядре файловой системе. Это поле указывает на соответствующую специфическую для файловой системы структуру и то, как VFS управляет преобразованием обобщенных запросов в специфические операции файловой системы.

Рис. 6.4 показывает связь между superblock и структурой f ile_system__type. Мы показывали, как поле superblock->s_type указывает на соответствующую структуру f ile_system__type в списке file_systems. (В подразд. 6.2.2, «Глобальные и локальные списки связей», мы покажем, что такое список f ile_systems.)

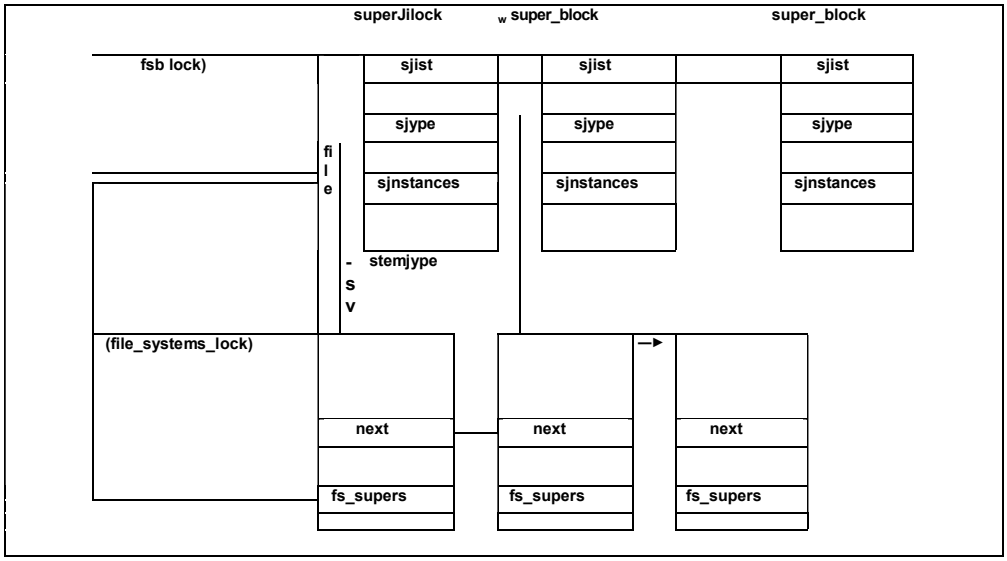


Рис. 6.4. Связь между superblock и file_system Jype

Строка 675

Поле, являющееся указателем на структуру super__operations. Этот тип данных хранит таблицу операций superblock. Сама структура super_operations хранит указатели на функции, которые инициализируются конкретными операциями файловой системы superblock.

Строка 681

Это поле является указателем на структуру dentry. Структура dentry хранит путь к файлу. Отдельный объект dentry ассоциируется с директорией монтирования, к которой принадлежит superblock.

Строка 690

Поле s__dirty (не путайте с s_dirt) - это структура list__head, указывающая на первый и последний элементы списка неочищенных inode, принадлежащих файловой системе.

Строка 693

Поле s__f iles является структурой list_Jiead, указывающей на первый элемент списка структур файлов, которые используются и назначены superblock. Это один из трех списков, в которых можно обнаружить структуру файла.

Строка 696

Поле s_instance является структурой list_head, указывающей на прилегающие элементы superblock в списке superblocks того же типа файловой системы. Голова этого списка хранится в поле fs__supers структуры file_ system__type.

Строка 702

Это указатель типа void* на дополнительный информационный superblock, специфичный для каждой файловой системы (например, ext3__sb_inf o). Он действует как объединяющий все данные superblock о диске с определенной файловой системой, не абстрагируемые в рамках концепции superblock виртуальной файловой системы.

6.2.1.2 Операции superblock

Поле superblock s__op указывает на таблицу операций, которую может выполнить superblock файловой системы. Этот список специфичен для каждой файловой системы, так как он работает с реализаций файловой системы напрямую. Таблица операций хранится в структуре типа superoperations:

```
include/linux/fs.h struct
super_operations {
```

```
        struct inode *(*alloc inode)(struct super block *sb);
        void (*destroy_inode)(struct inode *);
        void (*read_inode) (struct inode *) ;
        void (*dirty inode) (struct inode *);
        void (*write_inode) (struct inode *, int);
        void (*put inode) (struct inode *);
        void (*drop_inode) (struct inode *);
        void (*delete inode) (struct inode *) ;
        void (*put_super) (struct super_block *);
        void (*write super) (struct super  block *);
        int (*sync_fs)(struct super_block *sb, int wait);
        void (*write super lockfs) (struct super  block *);
        void (*unlockfs) (struct super_block *);
        int (*statfs) (struct super_block *, struct kstatfs *);
        int (*remount fs) (struct super block *, int *, char *);
        void (*clear_inode) (struct inode *);
        void (*umount begin) (struct super block *);
        int (*show_options)(struct seq__file *, struct vfsmount *);
};
```

При инициализации superblock файловой системы полю s_op назначается значение указателя на соответствующую таблицу операций. Далее в этой главе мы покажем, как эта таблица операций реализована для файловой системы ext2. Табл. 6.2 демонстрирует список операций superblock. Некоторые из этих функций являются опциональными и заполняются только подмножеством поддерживаемых файловых систем. Для неподдерживаемых в конкретной файловой системе полей в структуре operations устанавливается значение NULL.

Таблица 6.2. Операции superblock

Имя операции superblock	Описание
alloc_inode	Нововведение в 2.6. Выделяет и инициализирует inode vf s в суперблоке. Специфика инициализации ложится на конкретную файловую систему. Выделение выполняется с помощью kmem_cache_ create () или kmem_cache_alloc () (см. гл. 4) в кеше inode
destroy_inode	Нововведение в 2.6. Освобождает принадлежащий суперблоку inode. Освобождение выполняется с помощью вызова kmem__cache_f ree ()
read__inode	Чтение inode, указанного в поле inode->i_ino. Поля inode обновляются данными с диска. Исключительно важно поле inode->i_op

Таблица 6.2. Операции superblock (Окончание)

dirty_inode	Помещает inode в список неочищенных inode суперблока. Голова и хвост циклического двусвязного списка хранится в поле superblock->s_dirty. Рис. 6.5 иллюстрирует список неочищенных inode суперблока
write_inode	Записывает информацию inode на диск.
put_inode	Освобождает inode из кеша inode. Вызывается iput ()
drop_inode	Вызывается при завершении последнего доступа к inode
delete__inode	Удаляет inode с диска. Используется с больше не нужным inode. Вызывается из generic_delete_inode ()
put_super	Освобождение суперблока (например, при размонтировании файловой системы)
write_super	Записывает информацию из суперблока на диск
sync__f s	На данный момент используется ext3, Reiserfs, XFS и JFS. Эта функция записывает неочищенную структуру superblock на диск
write_super_lockf s	Используется ext3, Reiserfs, XFS и JFS. Эта функция блокирует изменения в файловой системе. Затем она обновляется из суперблока на диске
unlockf s	Изменяет набор блоков с помощью функции write__super_lockf s ()
stat_f s	Вызывается, для получения статистики файловой системы
remount__f s	Вызывается когда файловая система размонтируется для обновления опций монтирования
clear_inode	Освобождает inode и все связанные с ней страницы
umount_begin	Вызывается во время прерывания операции монтирования
show_options	Используется для получения информации о файловой системе из монтируемой файловой системы

На этом заканчивается представление структуры superblock и их операций. Теперь мы рассмотрим подробнее структуру inode.

6.2.1.3 Структура inode

Мы упоминали что inode - это структуры, следящие за информацией о файле, начиная с указателей и заканчивая блоками, содержащими данные файла. Вспомните, что дирек-

тории, устройства и каналы (например) также представляются в ядре как файлы, поэтому они также представляются с помощью inode. Объект inode существует в течение всего времени жизни файла и содержит информацию, поддерживаемую на диске.

Inode хранятся в списке в упорядоченном виде. Один список представляет собой хеш-таблицу, уменьшающую время поиска конкретного inode. Inode может находиться в одном из трех двусвязных списков. Табл. 6.3 иллюстрирует три типа списков. Рис. 6.5 демонстрирует связь между структурой superblock и списком неочищенных inode.

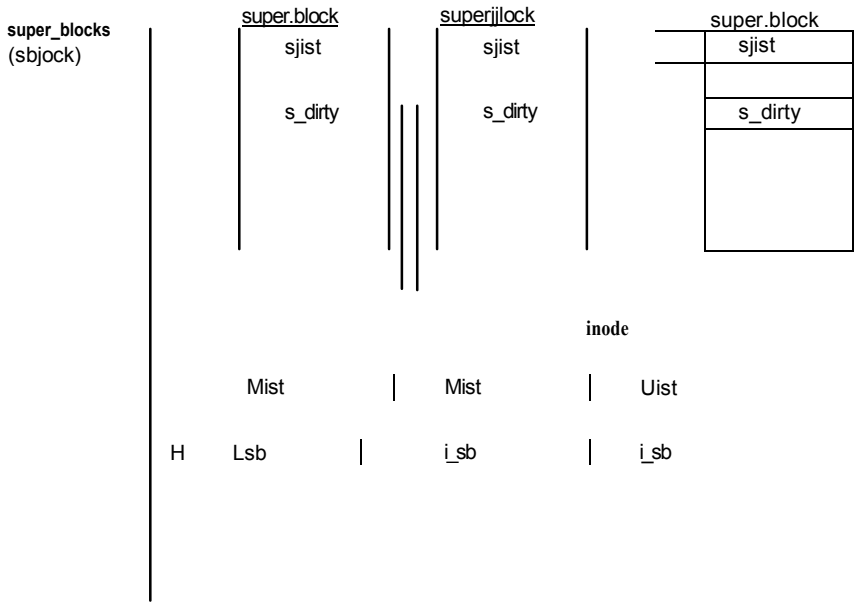


Рис. 6.5. Связь между superblock и inode

Структура inode очень большая и содержит множество полей. Далее приведена небольшая часть полей структуры inode:

```
   include/linux/fs.h
368 struct inode { struct
369 hlist node struct      i hash;
370 list head struct list i list;
371 head                  i__dentry;
372 unsigned long   i ino;
373 atomic_t    i_count;
```

```
390   struct inode_operations  *i_op;

392   struct super_block  *i__sb;

407   unsigned long   instate;

421     };
```

Строка 369

Поле i_hash имеет тип hlist^node[1]. Содержит указатель на хеш-список, использующийся для ускорения поиска. Хеш-список inode представлен в глобальной переменной inode__hashtable.

Строка 370

Это поле связывает соседние структуры в списке inode; inode могут находиться в одном из трех связанных списков.

Таблица 6.3. Списки inode

Список	Lcount	Неочищенные	Указатели
Верный, используется	i_count = 0	Очищены	inode__unused (глобальный)
Верный, не используется	i_count>0	Очищены	inode_in_use (глобальный)
Неочищенные inode	i_count > 0	Не очищены	Поле s__dirty суперблока

Строка 371

Это поле указывает на список структур dentry, соответствующих файлу. Структура dentry содержит имя файла, представленного в inode. Файл может иметь несколько структур dentry, если имеет несколько псевдонимов.

Строка 372

Это поле хранит уникальный номер inode. Когда inode выделяется в конкретном суперблоке, это число автоматически увеличивается по сравнению с назначенным перед этим ID inode. При вызове операции суперблока read_JLnode () inode, указанный в этом поле, будет прочитан с диска.

[1] hlist_node имеет тип указателя на двусвязный список, подобный list_head. Разница заключается в том, что голова списка (тип hlist_head) содержит единственный указатель, указывающий на первый элемент вместо двух (где второй указатель является концом хвоста). При этом уменьшаются расходы на работу с хеш-таблицей.

Строка 373

Поле i_count является счетчиком, увеличиваемым при каждом использовании inode. Значение 0 означает, что inode не используется, а положительное значение - что используется.

Строка 392

Это поле хранит указатель на суперблок файловой системы, в котором находится файл. Рис. 6.5 демонстрирует, что поля i_sb для всех inode из списка неочищенных inode суперблока указывают на текущий суперблок.

Строка 407

Это поле соответствует флагу состояния inode. В табл. 6.4 перечислены все возможные значения.

Таблица 6.4. Состояния inode

Флаг inode	Описание
I_DIRTY_SYNC	См. описание I_DIRTY
I_DIRTY_DATASYNC	См. описание I__DIRTY
I_DIRTY_PAGES I	См. описание I_DIRTY
DIRTY	Этот макрос совместим с любым из флагов I_DIRTY_*. Он включает в себя проверку каждого из этих флагов. Флаг I_DIRTY_* означает, что содержимое inode было перезаписано и его нужно синхронизировать
I LOCK	Устанавливается, когда inode блокируется, и снимается, когда inode разблокируется; inode блокируется во время создания и во время участия в операции ввода-вывода
I_FREEING	Устанавливается при удалении inode. Этот флаг служит для обозначения удаляемых inode, чтобы никто не смог к ним обратиться
I_CLEAR	Означает, что inode больше не используется
I NEW	Устанавливается при создании inode. Флаг убирается после того, как inode разблокируется

inode с установленными флагами I_LOCK и I_DIRTY находится в списке inode__in_use. При отсутствии этих флагов он добавляется в список inode_unused.

6.2.1.4 Структура dentry

Структура dentry представляет собой директорию, которую VFS использует для слежения за связями на основе имени директории, организации и логического размещения фай-

лов. Каждый объект dentry объединен с путем и связан с другими описывающими его структурами. Например, в пути /home/lkp/Chapter0б. txt существуют dentry, созданные для /, home, lkp и chapter0б . txt. Каждый dentry связан со своими inode, суперблоком и дополнительной информацией. Рис. 6.6 иллюстрирует связь между структурами superblock, inode и dentry.

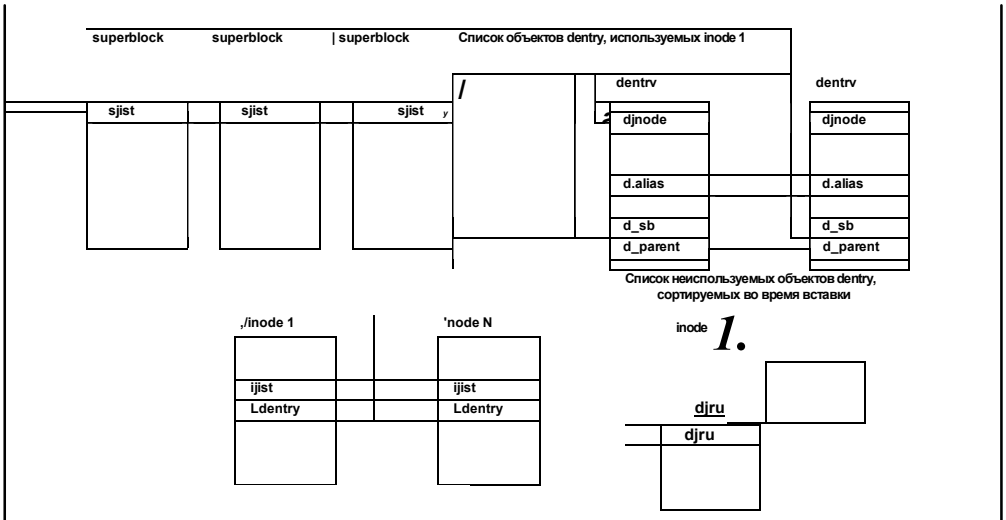


Рис. 6.6. Связь между superblock, inode и dentry

Рассмотрим некоторые поля структуры dentry.

```
include/linux/dcache.h
81  struct dentry {

85    struct inode  * d inode;
86    struct list head d lru;
87    struct list_head d_child;      /*  ребенок списка родителя */
88    struct list head d subdirs;    /*  наш ребенок */
89    struct list head d  alias;
90    unsigned long d time;         /*   используется d_revalidate */
91    struct dentry operations  *d op;
92    struct super block  * d sb;

100   struct dentry * d_parent;
```

```
105  } ____ cacheline_aligned;
```

Строка 85

Поле d_inode указывает на inode, соответствующий файлу, связанному с dentry. В случае, если компонент пути, связанный с dentry, не имеет связанного inode, устанавливается в NULL.

Строки 85-88

Это указатели на ближайшие элементы списка dentry. Объект dentry может находиться в одном из списков, перечисленных в табл. 6.5.

Таблица 6.5. Список dentry

Имя списка	Указатель на список	Описание
Используемые dentry	d_alias	inode, с которым ассоциированы эти denrty, указывает на голову списка через поле i__dentry
Неиспользуемые dentry	d_lru	Эти dentry больше не используются, но хранятся на тот случай, если с путем будет ассоциирован тот же самый компонент

Строка 91

Поле d__op указывает на таблицу операций dentry.

Строка 92

Этот указатель на суперблок ассоциируется с компонентом, представляемым denrty. На рис. 6.6 можно увидеть, как denrty связан со структурой superblock.

Строка 100

Это поле хранит указатель на родительский dentry или на dentry, соответствующий родительскому компоненту пути. Например, в пути /home/paul поле d__parent dentry для paul указывает на dentry для home, а поле d_parent этого dentry, в свою очередь, указывает на dentry для /.

6.2.1.5 Структура file

Еще одной структурой, используемой VFS, является file. Когда процесс работает с файлом, структура данных VFS file используется для хранения информации, описывающей связь между процессом и файлом. В отличие от других структур в этой структуре не содержатся оригинальные данные с диска; структура file создается на лету во время системного вызова open () и уничтожается во время системного вызова close (). Из гл. 3 мы

знаем, что на протяжении жизни процесса, структура file представляет открытые процессом файлы, связывая их с описателем процесса (task_struct). Рис. 6.7 иллюстрирует связь структуры file с другими структурами VFS; task_struct указывает на таблицу описателей файла, хранящую список указателей на все описатели файлов, открытые процессом. Вспомните, что первые три вхождения в таблице описателей соответствуют описателям файлов для stdin, stdout и stderr соответственно.

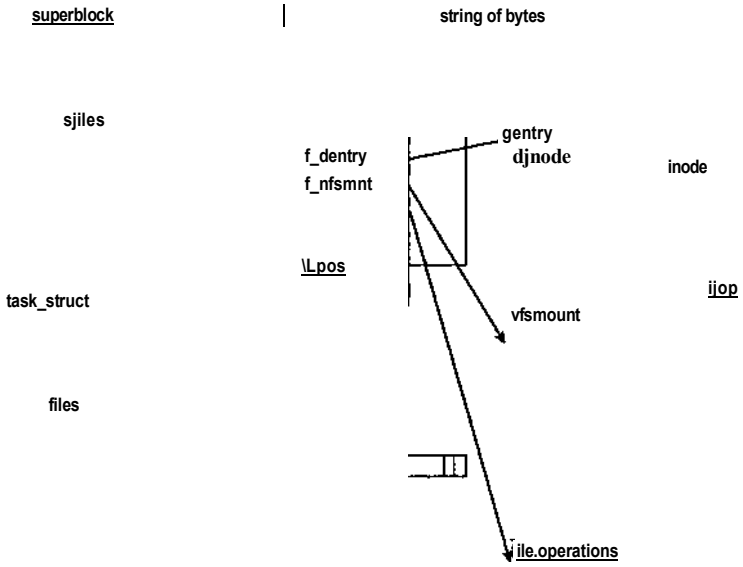


Рис. 6.7. Объекты file

Ядро следит за структурами file в циклическом двусвязном списке. Существует три списка, в которых могут находиться структуры файлов в зависимости от их использования и назначения. В табл. 6.6 перечислены все три списка.

Ядро создает структуру file с помощью get__empty_f lip (). Эта функция возвращает указатель на структуру файла или NULL, если свободных систем не осталось или если в системе мало свободной памяти.

Теперь мы рассмотрим наиболее важные поля структуры file.

```
include/linux/fs.h
506  struct file {
507   struct list head  f list;
508   struct dentry  *f dentry;
509   struct vfsmount   *f  vfsmnt;
510   struct file_operations   *f__op;
511   atomic t   f count;
512   unsigned int  f flags;
513   mode  t   f mode;
514   loff t   f pos;
515   struct fown_struct   f_owner;
516   unsigned int  f uid, f gid;
517   struct    file_ra_state   f__ra;

527  struct address_jspace   *f_mapping;

529  };
```

Строка 507

Поле f_list типа list_head хранит указатель на ближайшие структуры file в списке.

Таблица 6.6. Списки файлов

Имя	Связанный указатель на голову списка	Описание
Список пустых объектов file	Глобальная переменная f ree_JList	Двусвязный список, составленный из всех доступных объектов. Размер списка всегда не меньше чем **NR_RESERVED_FILES**
Список используемых, но не назначенных объектов file	Глобальная переменная anon_list	Двусвязный список, составленный из всех доступных объектов, используемых, но не назначенных суперблокам

Таблица 6.6. Списки файлов (Окончание)

Список объектов суперблока файла	Поле суперблока s_files	Двусвязный список, составленный из всех объектов file, связанных с суперблоками

Строка 508

Указатель на структуру dentry, связанную с file.

Строка 509

Это указатель на структуру vfsmount, связанную с монтируемой файловой системой, к которой принадлежит файл. Файловые системы смонтированы в структуру vfsmount, хранящую соответствующую информацию. Рис. 6.8 иллюстрирует структуру данных, связанную со структурами vfsmount.

Строка 510

Указатель на структуру f ile_operations, хранящую таблицу операций файла, применимых к файлу. (Поле inodes i_f op указывает на ту же самую структуру.) Рис. 6.7 иллюстрирует эту связь.

Строка 511

К файлу могут конкурентно иметь доступ сразу несколько процессов. Поле f _count устанавливается в 0, когда структура file не используется (и поэтому доступна для использования). Поле f _count устанавливается в 1, когда ассоциируется с файлом, и увеличивается на единицу каждый раз, когда с файлом начинает работать еще один процесс, обрабатывающий файл. Поэтому если объект file, представляющий файл, открыт сразу четырьмя процессами, поле f_count будет хранить 5.

Строка 512

Поле f _f lags содержит флаги, передаваемые через системный вызов open (). Мы опишем это подробнее в подразд. 6.5.1.

Строка 514

Поле f _pos хранит отступ файла. Это неотъемлемый указатель чтения-записи, используемый некоторыми методами файловых операций для обозначения текущей позиции в файле.

Строка 516

Нам нужно знать, кто владеет процессом для определения разрешений для доступа к файлу при работе с этим файлом. Эти поля соответствуют uid и gid пользователя, запустившего процесс и открывшего структуру file.

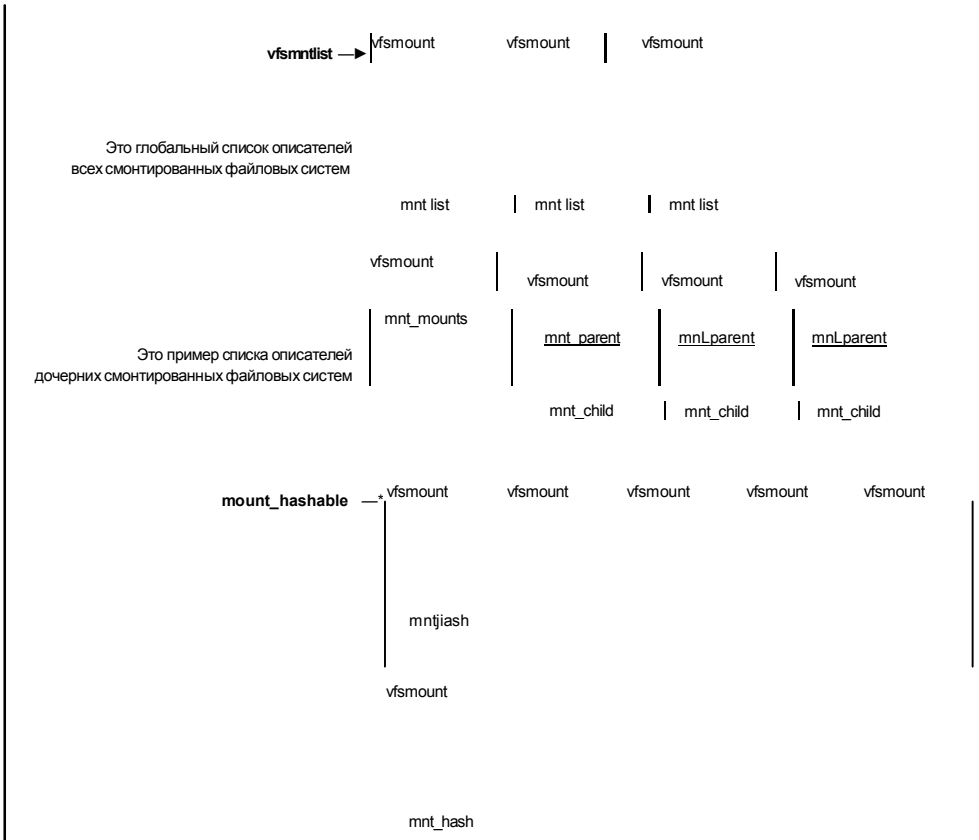


Рис. 6.8. Объекты vfsmount

Строка 517

Файл может читать страницы из кеша страниц, являющегося набором страниц в памяти. Оптимизация для чтения включает в себя предварительное чтение последовательных страниц файла для уменьшения стоимости доступа к ним при запросе. Поле f_ra хранит структуру типа file_ra_state, содержащую всю информацию, связанную с состоянием предварительного чтения.

Строка 527

Это поле указывает на структуру address_space, связанную с механизмом кеширования страниц для данного файла. Оно обсуждается подробнее в подразделе «Кеш страниц».

6.2.2 Глобальные и локальные списки связей

Ядро Linux использует глобальные переменные, хранящие указатели на связанный список ранее упоминавшихся структур. Все структуры хранятся в двусвязном списке. Ядро хранит указатель на голову списка в качестве точки входа в список. Все структуры имеют поля типа list_head[1], используемые для указания на предыдущий и следующий элементы списка. Табл. 6.7 суммирует глобальные переменные, хранимые ядром и указывающие на списки различных типов.

Структуры super_block, f ile___system_type, dentry и vf smount хранятся в собственном списке; inode могут хранится либо в inode_in_use, либо в inode__unused или в локальном списке суперблока, к которому они принадлежат. Рис. 6.9 демонстрирует некоторые связи между структурами.

Таблица 6.7. Глобальные переменные, связанные с VFS

Глобальная переменная	Тип структуры
super_blocks	super_block
file_systems	f ile_system__type
dentry__unused	dentry
vfsmntlist	vfsmount
inode__in_use	inode
inode_unused	inode

Переменная super_blocks указывает на голову списка суперблоков с элементами, указывающими на предыдущий и следующий элементы списка с помощью поля s_list. Поле s_dirty структуры суперблока, в свою очередь, указывает на имеющиеся у него inode, которые необходимо синхронизировать с диском; inode, не находящиеся в локальном списке суперблока, находятся в списках inode_in_use или inode_unused. Все inode указывают на следующий и предыдущие элементы списка с помощью поля i_lis t.

Кроме этого, суперблок указывает на голову списка, содержащую структуры файла, назначенные суперблоку с помощью списка s_f iles. Неназначенные структуры файлов помещаются в один из списков free_list списка anon_list. Оба списка имеют

[1] Структура inode тоже имеет такое поле hlist_node, как мы видели с подразд. 6.2.1.3, «Структура inode».

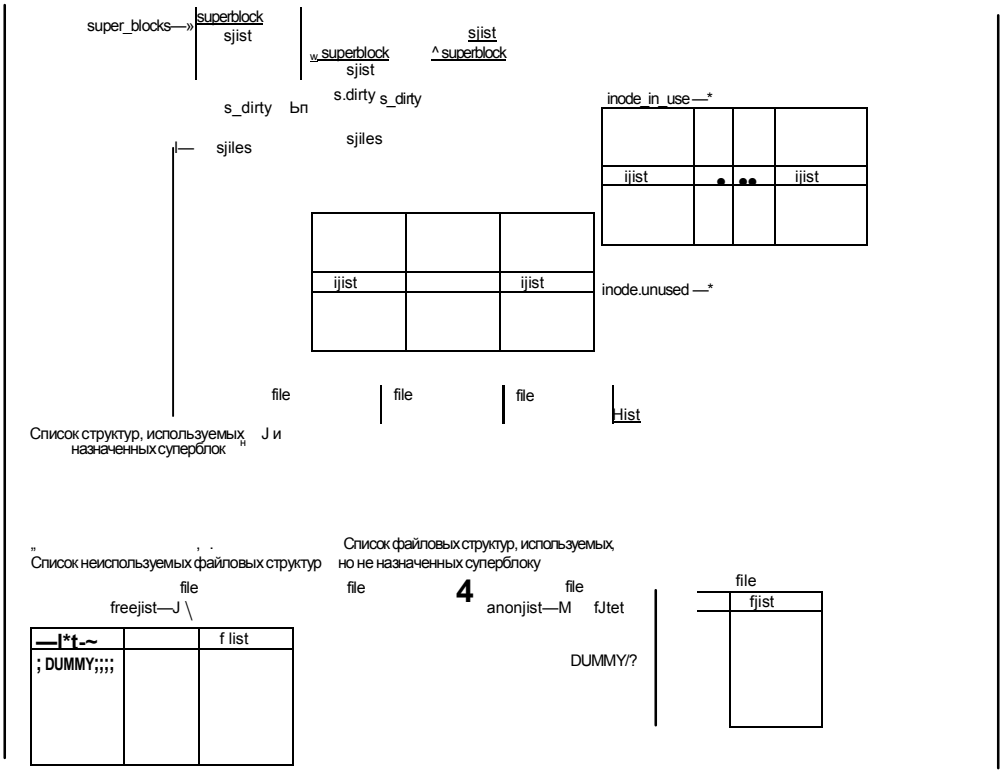


Рис. 6.9. Глобальные переменные, связанные с VFS

структуру файла-пустышку в качестве головы списка. Все структуры файлов указывают на следующий и предыдущий элементы своего списка с помощью поля f_list.

На рис. 6.6 вы можете увидеть, как inode указывает на список структур dentry с помощью поля i_dentry.

6.3 Структуры, связанные с VFS

Помимо основных структур VFS, с VFS взаимодействует еще несколько структур: fs_struct, files_struct, namespace и fd_set. Структуры fs_struct, f iles_struct и namespace являются связанными с процессом объектами, хранящими связанные с файлами данные. Рис. 6.10 показывает, как описатель процесса ассоциируется со связанными с файлами структурами. Теперь рассмотрим дополнительные структуры.

6.3.1 Структура fs_struct

В Linux множество процессов может указывать на один файл. В результате VFS Linux должна хранить информацию о том, как взаимодействуют процессы и файлы. Например, процессы, начатые разными пользователями с учетом разрешений на файловые операции. Структура f s_struct хранит всю информацию, связывающую конкретный процесс с файлом. Перед рассмотрением структуры f iles_struct нам нужно рассмотреть структуру fs_struct, потому что files_struct использует тип данных f s__struct.

f s_s true t может быть связана со множеством описателей процессов; следовательно, несложно предположить, что f s_struct, представляющая файл, связана со множеством описателей task__struct:

```
include/linux/fs struct.h
 7  struct fs struct {
 8   atomic_t count;
 9   rwlock_t lock;
10   int umask;
11   struct dentry * root, * pwd, * altroot;
12   struct vfsmount * rootmnt, * pwdmnt, * altrootmnt;
13  };
```

6.3.1.1 count

Поле count хранит количество описателей процессов, связанных с конкретной fs__struct.

6.3.1.2 umask

Поле umask хранит маску, представляющую набор разрешений для открытого файла.

6.3.1.3 root, pwd и altroot

Поля root и pwd указывают на объект dentry, связанный с корневой директорией процесса и текущей рабочей директорией соответственно; altroot - это указатель на структуру dentry, альтернативную корневой директории. Это поле используется для эмуляции окружения.

6.3.1.4 rootmnt, pwdmnt и altrootmnt

Поля rootmnt, pwdmnt и altrootmnt указывают на объект смонтированной файловой системы корня процесса, текущую директорию и альтернативную корневую директорию соответственно.

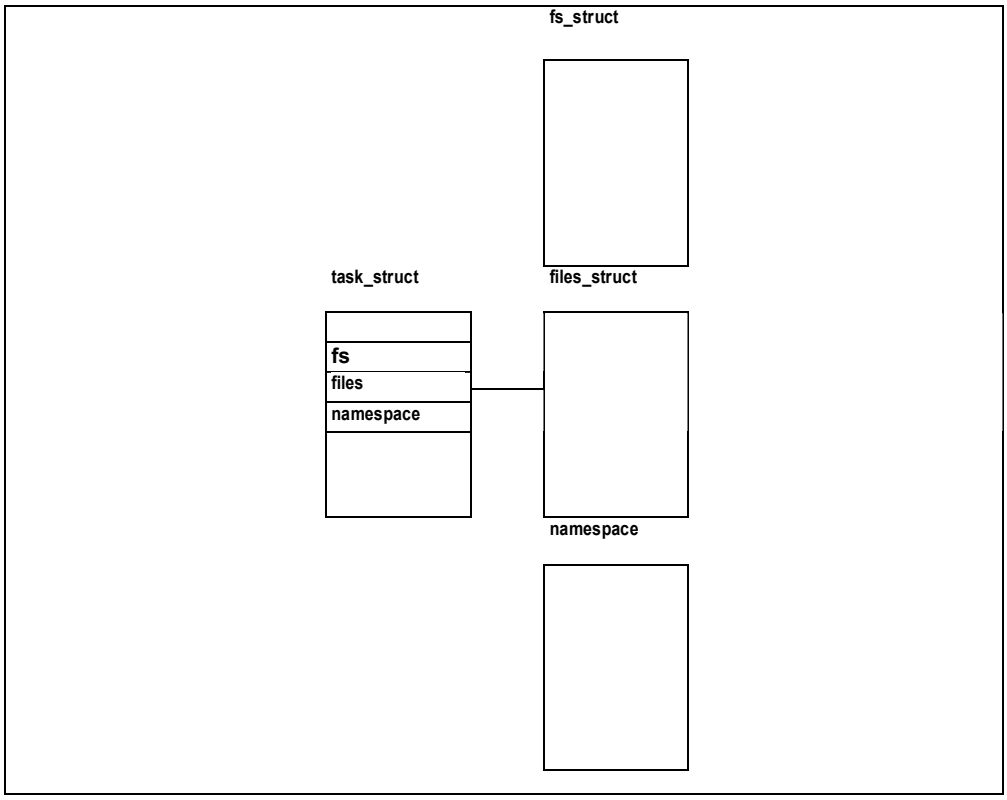


Рис. 6.10. Связанные с процессом объекты

6.3.2 Структура files_struct

f iles_struct содержит информацию, связанную с открытыми файлами и их описателями. В нашем вступлении мы упоминали, что описатели файлов представляют собой уникальный тип данных int, связанных с открытым файлом. В терминах ядра описатель файла представляет собой индекс в массиве f d объектов файлов task_struct текущей задачи или current->files->fd. Рис. 6.7 демонстрирует массив fd задачи task_struct и то, как он указывает на структуру file файла.

Linux может ассоциировать набор описателей файлов с разделяемыми свойствами, такими, как только чтение или только запись. Структура f d_struct представляет набор описателей файла; f iles_s truct использует этот набор для группировки своих описателей файла:

```
include/linux/file.h
22  struct files struct {
23    atomic t count;
24   spinlock t file lock
25    int max_fds;
26    int max  fdset;
27    int next_fd;
28    struct file ** fd;
29   fd_set *close__on_exec;
30   fd set *open fds;
31    fd_set close__on_exec_init;
32    fd jset open fds init;
33    struct file * fd_array[NR_OPEN_DEFAULT];
34   };
```

Строка 23

Поле count существует для того, чтобы files__struct могла быть связана со множеством описателей процессов, практически как f d_struct. Это поле увеличивается в функции ядра f get () и уменьшается в функции ядра f put (). Эти функции вызываются во время процесса закрытия файла.

Строка 25

Поле max__f ds отслеживает максимальное количество файлов, которые может открыть процесс. По умолчанию max_f ds равно 32. Это значение связано с размером массива f d_array - NR__OPEN_DEFAULT. Если файл захочет открыть более 32 файлов, это значение будет увеличиваться.

Строка 26

Поле следит за максимальным количеством описателей файла. Подобно шах_ f ds, это поле увеличивается, если максимальное количество файлов, открытое процессом, превышено.

Строка 27

Поле next_f d хранит значение следующего назначенного описателя. Мы увидим, что это поле используется при открытии и закрытии файла, но нам нужно понять одну вещь: описатель файла назначается с инкрементированием за исключением случая, когда предыдущий назначенный описатель, связанный с файлом, закрыт. В этом случае поле next_f d устанавливается в это значение. Таким образом, файловые дескрипторы назначаются в наименьшее возможное значение.

Строка 28

Массив f d указывает на объект массива открытого файла. По умолчанию это массив f d_array, хранящий 32 описателя файла. Когда поступает запрос на более чем 32 описателя файла, он указывает на новый сгенерированный массив.

Строки 30-32

Все поля: close__on_exec, open_fds/ close__on_exec_init и open_fds_ init, имеют тип fd_set. Мы упоминали, что структура fd_set хранит набор описателей файла. Перед объяснением каждого поля по отдельности мы рассмотрим структуру f d_set.

f d_set_datatype можно отследить до структуры, хранящей массив unsigned long, каждое из которых хранит описатель файла:

include/linux/types.h fd_set;
22 typedef ____kernel_fd_set

Тип данных f d_set - это определение типа __ _kernel_ _fd _set Эта с
хранит массив unsigned long: _ .

include/linux/posix_types.h 3 6 [.LONGS '*
typedef struct { ___ FDSET]
37 unsigned long fds_bits _
38 } __kernel_fd_set;

_ FDSET__LONG хранит значение 32 на 32-битовой системе и 16 на 64-битовой системе, для того чтобы f d_set всегда был размера 1024. Вот где определяется значение __FDSET_LONG:

```
include/linux/posix_types.h
 6  #undef   NFDBITS
 7  #define  _NFDBITS  (8 * sizeof(unsigned long))
 8
 9  #undef    FD SETSIZE
10  #define __ FD_SETSIZE   1024
11
12  #undef    FDSET LONGS
13  tdefine  _ FDSET_LONGS  ( _ FD_SETSIZE/__NFDBITS)
```

Для манипуляции с этими наборами описателей есть 4 макроса (см. табл. 6.8).

Таблица 6.8. Макросы наборов описателей файла

Макрос	Описание
FD_SET	Помещение файлового описателя в набор
FD_CLR	Удаление описателя файла из набора
FD_ZERO	Очистка набора описателей файлов
FD_ISSET	Возврат если описатель файла установлен

Теперь рассмотрим отдельные поля.

6.3.2.1 close_on_exec

Поле close_on_exec является указателем на набор описателей файлов, помеченных к закрытию при exec (). Изначально (и обычно) указывает на поле close_on_exec_ init. Изменяется, если количество описателей файлов, отмеченных для открытия во время exec (), выходит за пределы размера битового поля close_on_exec_init.

6.3.2.2 openjds

Поле open_f ds указывает на набор описателей файлов, отмеченных для открытия. Как и close_on_exec, оно изначально указывает на поле open_f ds__in.it и изменяется, если количество помеченных описателей файлов выходит за пределы размера битового поля open_f ds_init.

6.3.2.3 close_on_exec

Поле close__pn_exec__init хранит битовое поле, следящее за описателями файлов, закрываемых во время exec ().

6.3.2.4 openjdsjnit

Поле open_f ds_init хранит битовое поле, следящее за описателями открытых файлов.

6.3.2.5 fd_array

Массив указателей f d_array указывает на первые 32 описателя открытых файлов.

Структура f s_j5truct инициализируется макросом INIT_FILES.

```
include/linux/init_task.h
 6  #define INIT FILES \
 7  {
 8     .count    = ATOMIC__INIT(1),
```

```
 9    .file lock   = SPIN LOCK UNLOCKED,
10    .max fds   = NR_OPEN_DEFAULT,
11    .max fdset   =   FD SETSIZE,
12    .next fd   = 0,
13    .fd    = &init files.fd array[0],
14    .close on exec  = &init_files.close on exec init,
15    .open fds  = &init f iles. open  fds init,
16    .close__on exec init = { { 0, } },
17    .open fds init = { { 0, } },
18    .fd_array   = { NULL, }
19   }
```

Рис. 6.11 иллюстрирует то, как f s_struct выглядит после инициализации.

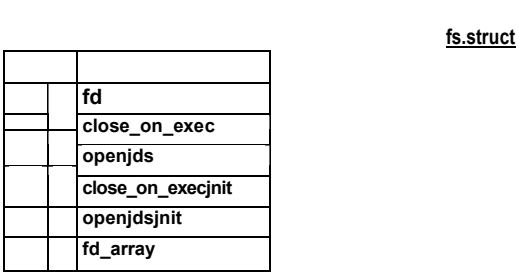


Рис. 6.11. Инициализация fs_struct

```
include/linux/file.h
6  #define NR_OPEN_DEFAULT BITS_PER_LONG
```

Глобально определенный NR_OPEN_DEFAULT устанавливается в BITS_PER_LONG, равный 32 на 32-битовых системах и 64 на 64-битовых системах.

6.4 Кеш страниц

Во вступительном разделе мы упоминали, что кеш страниц- это набор страниц в памяти. При частом доступе к данным важно иметь быстрый доступ к этим данным. Когда данные дублируются и синхронизируются на двух устройствах, одно из которых обычно меньше по объему, но обеспечивает значительно более быстрый доступ, чем другое, мы говорим, что используется кеширование. Кеш страниц - это способ, в соответствии с ко-

торым операционная система хранит часть информации с жесткого диска в памяти для быстрого доступа. Теперь мы рассмотрим, как он работает и реализуется.

Когда вы выполняете запись в файл на жестком диске, файл разбивается на порции, называемые страницами, которые отображаются в оперативную память. Операционная система обновляет страницы в памяти, и немного позднее страницы записываются на диск.

Если страница копируется с жесткого диска в память (это называется подкачкой или свопингом), она становится чистой или неочищенной. Неочищенная страница изменяется в памяти, но изменения на диск не записываются. Чистая страница существует в памяти в том же состоянии, что и на диске.

В Linux память разделяется на зоны[1]. Каждая зона имеет список активных и неактивных страниц. Когда страница является неактивной на протяжении некоторого времени, она свопируется (записывается обратно на диск) в свободную память. Каждая страница в списке зон имеет указатель на address__space. Каждая address_space имеет указатель на структуру address_space__operations. Страницы помечаются как неочищенные с помощью вызова функции set_dirty__page () структуры address_space_operation. Рис. 6.12 иллюстрирует эту зависимость.

Рис. 6.12. Кеш страниц и зоны

[1] См. подробности в гл. 4.

6.4.1 Структура address_space

Ядром кеша страниц является объект address_space. Рассмотрим его поближе.

```
include/linux/fs.h
32 6 struct address space {
327   struct inode  *host;    ./* владелец: inode, block device */
328   struct radix_tree_root page_tree;  /* корневое дерево для всех
                                                  страниц*/
329   spinlock_t   tree_lock; /* и защищающие его циклические
                                    блокировки */
33 0  unsigned long  nrpages;        /* общее количество страниц */
331   pgoff t writeback  index;  /* отсюда начинается обратная запись */
332   struct address_space_operations *a_ops;  /* методы */
333   struct prio_tree_root i_mmap; /* дерево частного отображения
                                         в память */
334   unsigned int  i_mmap_writable;/* счетчик отображения VM_SHARED */
33 5  struct list head i mmap nonlinear;   /*список отображения
                                              VM NONLINEAR */ 33 6
spinlock t i mmap lock;  /* защищенное дерево, счетчик, список */
337   atomic__t  truncate_count; /* исключает появление
                                    соревновательных состояний */
338   unsigned long   flags;           /* биты ошибки/маска gfp */
33 9   struct backing  dev info *backing dev info; /* предварительное
                                       чтение с устройства и т. д. */
340   spinlock t    private lock;   /* для нужд address space */
341   struct list head  private list;   /* то же */
342   struct address  space   *assoc mapping; /* то же */
343 };
```

Комментарии к коду структуры достаточно информативны. Некоторые дополнительные пояснения позволят понять работу операций кеша страниц.

Обычно address_space связан с inode и поле host указывает на этот inode. Тем не менее для выполнения основного назначения кеша страниц и структуры адресного пространства это поле необязательно. Оно может быть NULL, если address_space связан с объектом ядра, не являющимся inode.

Структура address__space имеет поле, которое уже должно быть вам знакомо: address_space_operations. Как и структура файла f ile_operations, address_space_operations содержит информацию об операциях, корректных для address_space.

```
include/linux/fs.h
297 struct address_space_operations {
```

```
298   int (*writepage)(struct page *page, struct writeback control *wbc);
299   int (*readpage)(struct file *, struct page *);
300   int (*sync page)(struct page *); 301
3 02 /* Запись обратно некоторых грязных страниц, отображенных в память */
3 03              int (*writepages) (struct address__space *,
              struct writeback control *); 304
3 05             /* Установка страницы в грязное состояние */
3 06          int (*set page dirty)(struct page *page);
307
3 08                  int (*readpages)(struct file *filp,
              struct address _space *mapping,
309              struct list head *pages, unsigned nr pages);
310
311    /*
312    * ext3 требует, чтобы за удачным вызовом prepare write() следовал
313    * вызов commit_write()- он должен быть сбалансирован
314    */
315    int (*prepare write)(struct file *, struct page *, unsigned,
               unsigned);
316    int (*commit  write)(struct file *, struct page *,
               unsigned, unsigned);
317    /* Этот ляп нужен только для FIBMAP. Не используйте его */
318    sector t (*bmap)(struct address space *, sector t);
319    int (*invalidatepage) (struct page *, unsigned long);
320    int (*releasepage) (struct page *, int);
321    ssize t (*direct IO)(int, struct kiocb *, const struct iovec *iov,
322         loff t offset, unsigned long nr segs);
323  };
```

Эти функции достаточно понятны: readpage () и wri tepage () читают и записывают страницы, связанные с адресным пространством соответственно. Несколько страниц можно прочитать и записать с помощью readpages () и writepages (). Журналируе-мые файловые системы, такие, как ext3, могут предоставлять функции для prepare., write () и commit_write ().

Когда ядро проверяет кеш страниц для некоторой страницы, проверка должна выполняться быстро. Поэтому каждое адресное пространство имеет radix_tree, выполняющий быстрый поиск для определения того, находится ли страница в кеше страниц или нет.

Рис. 6.13 иллюстрирует, как файлы, inode, адресные пространства и страницы связаны друг с другом; этот рисунок полезен для анализа кода кеша страниц.

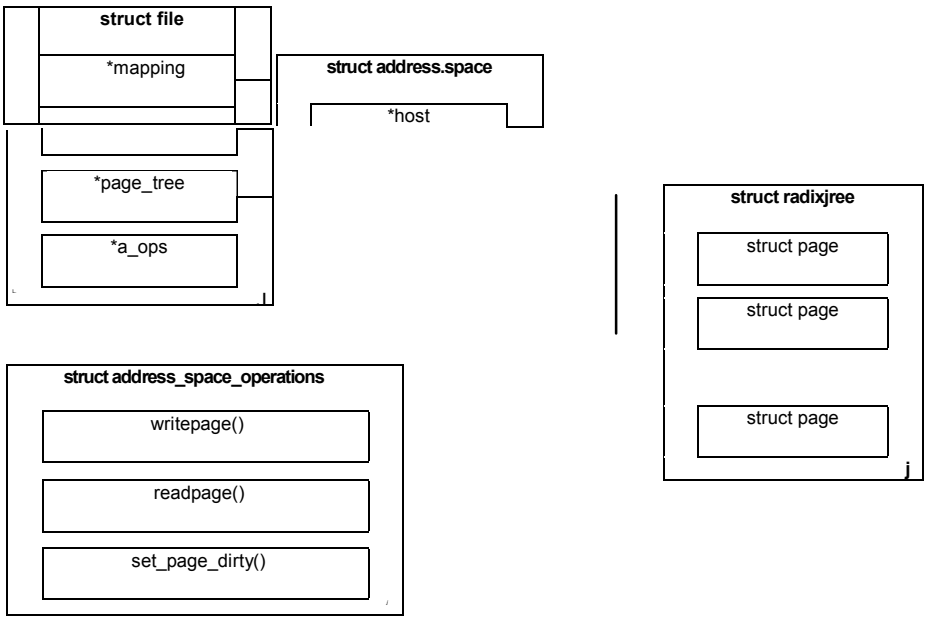


Рис. 6. 13. Файлы, inode, адресные пространства и страницы

6.4.2 Структура bufferjiead

Каждый сектор блочного устройства представляется в ядре Linux с помощью структуры buf fer_head, которая хранит всю информацию, необходимую для отображения физического сектора в памяти в физическую оперативную память. Структура buf f er_head изображена на рис. 6.14.

```
include/linux/buffer_head.h
47 struct buffer head {
48  /* Первая строка кеша: */
49  unsigned long b  state; /* битовая карта состояния буфера (см. ниже) */
50  atomic t b count;      /* пользователи употребляют этот блок */
51  struct buffer_head *b_this_page;  /* циклический список буферов
                                         страниц */
52  struct page *b page;  /* страница, куда отображается данный
                              заголовок буфера */ 53
54  sector t b blocknr;     /* номер блока */
55  u32 b_size;     /* размер блока */
```

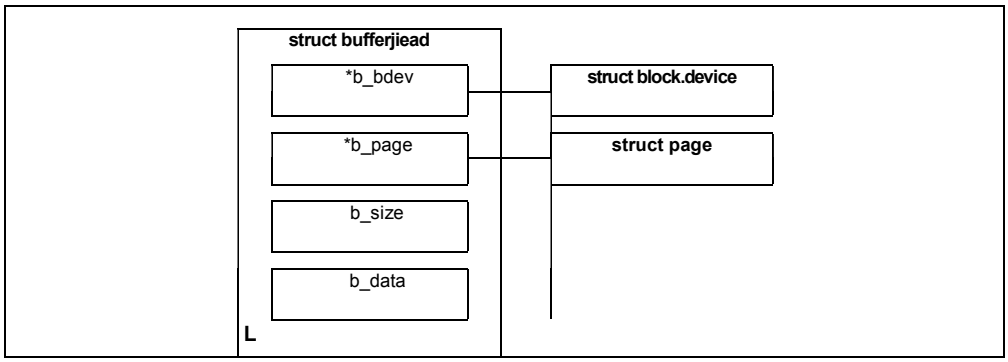


Рис. 6.14. Структура bufferjiead

```
56   char *b_data;      /* указатель на блок данных */ 57
58   struct block_device *b_bdev;
59   bh end io t *b end io;   /* завершение ввода-вывода */
60   void *b private;        /* зарезервировано для b end io */
61   struct list_head b_assoc_buffers; /* связь с другим отображением
                                          в память */
62   };
```

Физический сектор, к которому относится структура buf f er_head, - это логический блок b_blocknr устройства b_dev.

Физическая память, к которой обращается структура buf f er_head, - это блок памяти, начинающийся в b_data длиной b_size байт. Блок памяти находится в физической странице b_page.

Другое определение структуры buf f er_head применяется для использования задачи управления тем, какие физические секторы отображаются в физическую память. (Так как это отступление касается структуры bio, а не buf fer_head, обратитесь к mpage. c для получения более подробной информации о структуре buf f er_head.)

Как упоминалось в гл. 4, каждая страница физической памяти в ядре Linux представляется структурой страницы. Страница состоит из нескольких блоков ввода-вывода. Так как каждый блок ввода-вывода не может быть больше страницы (хотя и может быть меньше), страница состоит из одного или более блоков ввода-вывода.

В старых версиях Linux блоки ввода-вывода выполнялись только с помощью буферов, но в версии 2.6 разработан новый способ, использующий структуру bio. Новый способ позволяет ядру Linux группировать блоки ввода-вывода вместе более управляемым способом.

Представьте, что мы записываем порцию данных в начало текстового файла и в его конец. Это обновление требует двух структур buf fer_head для передачи данных:

одной, указывающей на начало, и одной, указывающей на конец. Структура bio позволяет файловой операции группировать отдельные порции вместе в единую структуру. Это альтернативный способ рассмотрения буферов и страниц в виде последовательности сегментов буфера. Структура bio_vec представляет последовательность сегментов в буфере. Структура bio_vec изображена на рис. 6.15.

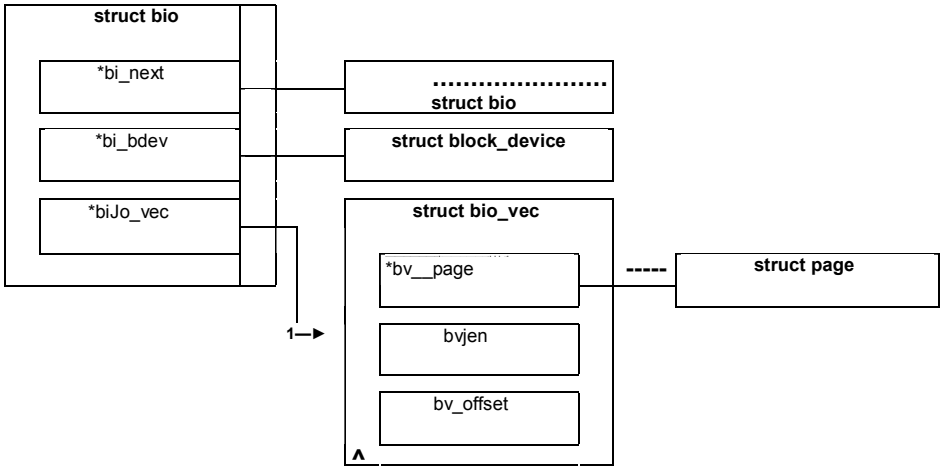


Рис. 6.15. Структура bio

```
include/linux/bio.h
47 struct bio_vec {
48   struct page  *bv  page;
49   unsigned int bv_len;
50   unsigned int bv offset;
51 };
```

Структура bio__vec хранит указатель на страницу, длину сегмента и отступ сегмента в страницу.

Структура bio состоит из массива структур bio_vec (вместе с другими полями). Поэтому структура bio представляет собой количество последовательных сегментов памяти из одного или нескольких буферов на одной или нескольких страницах[1].

[1] См. более подробную информацию о структуре bio в файле include/linux/bio.h.

6.5 Системные вызовы VFS и слой файловой системы

До этой точки мы рассмотрели все структуры, связанные с VFS и кешем страниц. Теперь мы сфокусируемся на двух системных вызовах, используемых для работы с файлами, и отследим их выполнение до уровня ядра. Мы увидим, что системные вызовы open (), close(), read () и write () используют ранее описанные структуры.

Мы упоминали, что в VFS файлы трактуются абстрактно. Вы можете открывать, читать, записывать и закрывать файлы, а то, что происходит на физическом уровне, слой VFS не волнует. Эта специфика отражена в гл. 5.

В VFS интегрирован специфичный для файловой системы слой, преобразующий файловый ввод-вывод VFS в страницы и блоки. Так как на компьютере может присутствовать несколько типов файловых систем, таких, как отформатированный в ext2 жесткий диск niso9660 cdroni, слой файловой системы делится на два основных раздела: обобщенные и специфические операции файловой системы (см. рис. 6.3).

Следуя методу рассмотрения сверху вниз, этот раздел отслеживает запросы на чтение и запись начиная с вызовов VFS read () или write () через слой файловой системы до специфических блочных запросов ввода-вывода, обрабатываемых драйвером блочного устройства. В нашем путешествии мы перейдем от обобщенной файловой системы к специфическому слою файловой системы. В качестве примера специфического слоя файловой системы мы будем использовать драйвер файловой системы ext2. При этом имейте в виду, что в зависимости от файла, к которому происходит обращение, могут быть использованы драйверы различных файловых систем. Далее мы встретимся с кешем страниц, встроенным в Linux и располагающемся в слое обобщенной файловой системы. В старых версиях Linux буфер кеша и кеш страниц присутствовали, тогда как в версии 2.6 кеш страниц взял на себя всю функциональность буфера кеша.

6.5.1 open()

Когда процесс желает работать с содержимым файла, он использует системный вызов open():

Синтаксис

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char  *pathname, int  flags);
int open(const char  *pathname, int  flags,  mode__t mode);
int creat(const char  *pathname, xnode_t mode);
```

Системный вызов open получает в качестве аргументов путь к файлу, флаги для обозначения режима доступа к открываемому файлу и битовую маска разрешений (если файл

создается); open () возвращает описатель файла для открываемого файла (если вызов закончился удачно) или код ошибки (если он завершился неудачно).

Параметр flags формируется с помощью исключающего ИЛИ одной или нескольких констант, определенных в include/linux/f cntl. h. В табл. 6.9 перечислены флаги для open () и соответствующие значения констант. Может быть указан только один флаг 0_RDONLY, 0_WRONLY или 0_RDWR. Дополнительные файлы являются опциональными.

Таблица 6.9. Флаги openQ

Флаг	Значение	Описание
0_RDONLY	0	Открытие файла для чтения
0_WRONLY	1	Открытие файла для записи
0_RDWR	2	Открытие файла для чтения и записи
O CREAT	100	Означает, что, если файл не существует, его следует создать. Функция с г eat () является эквивалентом функции open () с этим установленным флагом
0_EXCL	200	Используется в комбинации с 0_CREAT, означающего что open () окончится неудачей, если файл не существует
0_NOTCTTY	400	В случае, если путь указывает на терминальное устройство, процесс может не считать его управляющим терминалом
0_TRUNC	0x1000	Если файл существует, он урезается до 0 байт
0_APPEND	0x2000	Запись производится в конец файла
0_NONBLOCK	0x4000	Открытие файла в деблокирующем режиме
0_NDELAY	0x4000	То же, что и 0_NONBLOCK
0_SYNC	0x10000	Запись в файл ожидает завершения физического ввода-вывода. Применяются к файлам блочных устройств
0_DIRECT	0x20000	Минимизирует буферизацию кеша для ввода-вывода в файл
O_LARGEFILE	Ox 100000	Большая файловая система позволяет файлам иметь размер, представляемый более чем 31 битом. Этот флаг указывается, для того чтобы быть уверенным в их открытии
0_DIRECTORY	0x200000	Если путь не означает директорию, открытие завершается неудачей
O_NOFOLLOW	**0x400000**	Если путь представляет собой символическую ссылку, открытие завершается неудачей

Рассмотрим системный вызов.

```
fs/open.c
927  asmlinkage long sys_open (const char _____user * filename, int flags,
                                    int mode)
928  {
929   char * tmp;
93 0   int fd, error;
931
932 #if BITS_PER_LONG != 32
933   flags |= 0 LARGEFILE;
934  #endif
935   tmp = getname(filename);
93 6   fd = PTR ERR(tmp);
937   if (!IS ERR(tmp)) {
938   fd = get unused fd();
939   if (fd >= 0) {
940    struct file *f = filp open(tmp, flags, mode);
941    error = PTR ERR(f);
942    if (IS ERR(f))
943      goto out error;
944    fd_install(fd, f) ;
945   }
946  out:
947    putname(tmp);
948   }
949   return fd;
950
951  out_error:
952   put_unus ed f d (f d) ;
953   fd = error;
954   goto out;
955     }
```

Строки 932-934

Проверка системы на 32-битовость. Если это не так, включается флаг поддержки большой файловой системы 0_LARGEFILE. Это позволяет функции открывать файлы большего размера, чем можно представить 31 битом.

Строка 935

Функция getname () копирует имя файла из пользовательского пространства в пространство ядра с применением strncpy__f rom__user ().

Строка 938

Функция get__unused_f d () возвращает первый из доступных описателей файлов (или индекс в массиве f d: current->f iles->f d) и отмечает, что он занят. Это значение получает локальная переменная f d.

Строка 940

Функция filp__open() выполняет основную работу системного вызова open и возвращает структуру файла, связывающую процесс с файлом. Рассмотрим функцию f ilp_open () поближе.

```
fs/open.c
740  struct file *filp open(const char * filename, int flags, int mode)
741  {
742   int namei flags, error;
743   struct nameidata nd;
744
745   namei  flags = flags;
746   if ((namei flags+l) & 0 ACCMODE)
747    namei flags++;
748   if (namei flags & 0_TRUNC)
749    namei flags |= 2;
750
751   error = open namei (filename, namei flags, mode, &nd) ;
752   if (!error)
753    return dentry open(nd.dentry, nd.mnt, flags);
754
755              return ERR__PTR(error) ;
```

Строки 745-749

Функции поиска пути, такие, как open_namei (), ожидают флаги режима доступа, закодированные в специфическом формате, отличном от формата, используемого системным вызовом open. Эти строки копируют флаги режима доступа в переменную namei__flags и формируют флаг режима доступа, которым может интерпретироваться open_namei ().

Главное различие заключается в том, что для поиска пути не требуется указывать режимы доступа для чтения и записи. Такой режим доступа «без разрешений» не имеет смысла при открытии файла и поэтому не включен в состав флагов для системного вызова open. «Без разрешений» обозначается значением 00. Разрешение на чтение обозначается установкой значения нижнего бита в 1, а на запись - установкой верхнего бита в 1. Эквивалентами флагов для системного вызова open 0_RDONLY, 0_WRONLY и 0_RDWR являются соответственно 00, 01 и 02, как можно видеть в include/asm/f cntl .h.

Переменная namei_f lags может извлекать режим доступа с помощью логического И над переменной 0_ACCMODE. Эта переменная хранит значение 3 и равняется true, если переменная, с которой выполняется И, хранит значения 1, 2 или 3. Если флаг системного вызова был установлен в 0_RDONLY, 0_WRONLY и 0__RDWR, добавление 1 к этому значению переводит его в формат поиска пути и дает true при выполнении операции И с 0__ACMODE. Вторая проверка убеждается, что установленный флаг системного вызова open позволяет усечение файла; установленный бит высшего порядка обозначает доступ для чтения.

Строка 751

Функция open_namei () выполняет поиск пути, генерирует связанную структуру nameidata и извлекает соответствующий inode.

Строка 753

dentry_open () представляет собой обертку для dentry_open_it (), создающую и инициализирующую структуру файла. Она создает структуру файла с помощью вызова к функции ядра get_empty_filp(). Эта функция возвращает ENFILE, если files_stat.nr_files больше либо равно files_stat.max_files. Этот случай демонстрирует, что системный предел общего количества открытых файлов достигнут.

Рассмотрим функцию dentry_open_it ().

```
fs/open.c
844  struct file *dentry open it(struct dentry *dentry,
845           struct vfsmount *mnt, int flags, struct lookup  intent *it)
846  {
847   struct file * f;
848   struct inode *inode;
849   int error;
850
851   error = -ENFILE;
852   f = get_empty_filp();

855   f->f flags = flags;
856   f->f mode = (flags+1) & 0 ACCMODE;
857   f->f it = it;
858   inode = dentry->d  inode;
859   if (f->f_mode & FMODE_WRITE) {
860    error = get write access(inode);
861    if (error)
862      goto cleanup file;
863   }
```

```
866    f->f dentry = dentry;
867    f->f vfsmnt = mnt;
868    f->f pos = 0;
869    f->f_op = fops_get(inode->i_fop);
870    file move(f, &inode->i sb->s files);
871
872    if (f->f_op && f->f_op->open) {
873      error = f->f op->open(inode,f);
874      if (error)
875       goto cleanup_all;
876      intent release(it) ;
877    }

891  return f;

907  }
```

Строка 852

Структура файла назначается с помощью системного вызова get_empty_f ilp ().

Строки 855-856

Флаг f _f lags структуры файла устанавливается на основе значений, переданных в системный вызов open. Поле f _mode устанавливается в режим доступа, переданный в системный вызов open, но уже в формате, ожидаемом функцией поиска пути.

Строки 866-869

Поле структуры файла f_dentry устанавливается указывающей на структуру dentry, связанную с путем к файлу. Поле f_vsmnt устанавливается указывающим на структуру vmf smount файловой системы; f_pos устанавливается в 0, что означает начальную позицию f ile_of f set в начале файла. Поле f_op устанавливается указывающим на таблицу операций, на которую указывает inode файла.

Строка 870

Функция file_move() вызывается для вставки структуры файла в список суперблоков файловой системы структуры файла, представляющей файлы.

***Строки** 872-777*

Здесь появляется следующий уровень функции open. Он вызывается, если у файла есть специфическая функциональность, требующаяся для его открытия. Также она вызывается, если таблица файловых операций для файла содержит функции открытия.

На этом мы заканчиваем рассмотрение функции dentry_open_it ().

В конце f ilp_open () у нас есть выделенная файловая структура, вставленная в голову поля суперблока s_files, с f_dentry, указывающим на объект den try, f_vf smount, указывающим на объект vf smount, f _op, указывающий на таблицу файловых операций inode i_f op, f_f lags с установленными флагами доступа и f_jmode, установленный в режим разрешений для вызова open ().

Строка 944

Функция f deinstall () устанавливает массив f d, указывающий на адрес объекта файла, возвращенного filp_open(). Поэтому он устанавливается в current-> files->fd[fd].

Строка 947

Функция put name () освобождает пространство ядра, выделенное для хранения имени файла.

Строка 949

Возвращается описатель файла f d.

Строка 952

Функция put_unused_fd() очищает выделенный описатель файла. Она вызывается, когда создать объект файла невозможно.

Подводя итог, иерархически системный вызов open () выглядит следующим образом:

sys__open ():

- **getname** (). Перемещает имя файла в пространство ядра.
- **get_unsused_f d** (). Получает следующий доступный описатель файла.
- **f ilp_open** (). Создает структуру nameidata.
- **open_namei** (). Инициализирует структуру nameidata.
- **dentry_open** (). Создает и инициализирует объект файла.
- **f deinstall** (). Устанавливает current->f iles->f d[f d] в объект файла.
- **putname** (). Освобождение пространства ядра для имени файла.

Рис. 6.16 иллюстрирует структуры, которые инициализируют, устанавливают и настраивают функции, где все это выполняется.

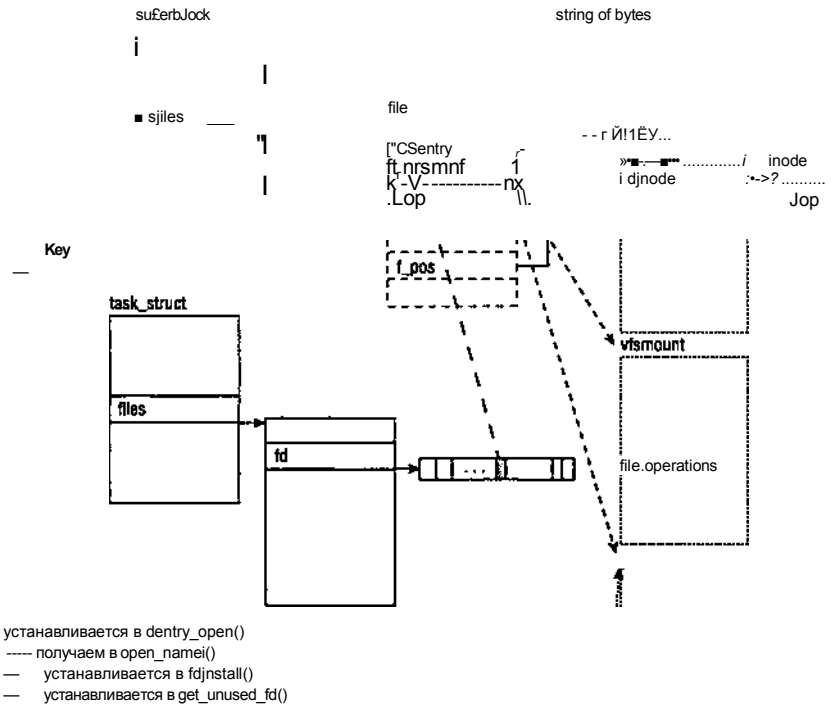


Рис. 6.16. Структуры файловой системы

Табл. 6.10 иллюстрирует некоторые ошибки sys_open () и функции ядра, где они возникают.

Таблица 6.10. Ошибки sysopenQ

Код ошибки	Описание	Функция, возвращающая ошибку
ENAMETOOLONG	Путь слишком длинный	getname()
ENOENT	Файл не существует (и флаг 0_CREAT не установлен)	getname ()
EMFILE	Процесс открыл максимальное количество файлов	get_unused_fd( ]

Таблица 6.10. Ошибки sys_openQ (Окончание)

ENFILE	Система открыла максимальное количество файлов	get_unused_fd()

6.5.2 closeQ

После того как процесс закончит работу с файлом, он выполняет системный вызов close():

```
Синтаксис
#include <unistd.h>
int close(int fd);
```

Системный вызов close получает в качестве параметра описатель файла для закрываемого файла. В стандартных С-программах этот вызов производится незадолго до завершения программы. Давайте углубимся в код sys_close ():

```
fs/open.c
1020  asmlinkage long sys close(unsigned int fd)
1021  {
1022   struct file * filp;
1023   struct files struct *files = current->files;
1024
1025   spin lock(&files->file lock) ;
1026   if (fd >= files->max_fds)
1027    goto out unlock;
1028   filp = files->fd[fd];
1029   if Ufilp)
1030    goto out unlock;
1031   files->fd[fd] = NULL;
1032   FD CLR(fd, files->close on exec);
1033   _ put_unused_fd(files, fd) ;
1034   spin unlock(&files->file lock);
1035   return filp close(filp, files);
1036
1037  out unlock:
1038   spin unlock(&files->file lock);
103 9   return -EBADF;
1040  }
```

Строка 1023

Поле текущего файла task_struct указывает на структуру f iles_struct, соответствующую нашему файлу.

Строки 1025-1030

Эти строки начинаются с блокировки файла во избежание проблем с синхронизацией. Далее мы проверяем корректность файлового описателя. Если число файлового описателя больше чем наибольший доступный для файла номер, мы убираем блокировку и возвращаем ошибку EBADF. В противном случае мы получаем адрес файловой структуры. Если индекс описателя файла не возвращает файловой структуры, мы также снимаем блокировку и возвращаем ошибку, так как нам нечего закрывать.

Строки 1031-1032

Здесь мы устанавливаем current->f iles->f d [f d] в NULL, удаляя указатель на объект файла. Также мы очищаем бит файлового описателя в файловом описателе, установленном для f iles->close_on_exec. Так как описатель файла закрыт, процессу не нужно беспокоиться о его содержимом при вызове exec ().

Строка 1033

Функция ядра __ put_unused_f d () очищает бит файлового описателя в описате ле файла, установленном files->open_fds, так как он больше не является открытым. Также мы выполняем действия, необходимые для назначения «наимень шего доступного индекса» для описателя файла.

```
fs/open.c
897  static inline void  _put_unused_fd(struct files_struct *files,
                                        unsigned int fd)
898   {
899     FD CLR(fd, files->open fds);
890   if (fd < files->next_fd)
891    files->next fd = fd;
892   }
```

Строки 890-891

Поле next_f d хранит значение следующего назначаемого описателя файла. Если текущее значение описателя файла меньше, чем хранится в f iles->next_f d, то это поле устанавливается равным значению текущего описателя файла. Таким образом мы проверяем, чтобы назначаемый описатель файла имел наименьшее доступное значение.

Строки 1034-1035

Здесь снимается блок с файла, а управление передается в функцию f ilp_close (), изменяющую значение, возвращаемое в системный вызов close. Функция f ilp_close () выполняет основную работу системного вызова close. Давайте рассмотрим f ilp_close () подробнее.

```
fs/open.c
987  int filp_close(struct file *filp, fl_owner_t id)
988  {
989  int retval;
990  /* Выдача и очистка всех накопившихся ошибок */
991  retval = filp->f_error;
992  if (retval)
993   filp->f_error = 0;
994
995  if (!file_count(filp)) {
996   printk(KERN_ERR "VFS: Close: file count is 0\n");
997   return retval;
998   }
999
1000  if (filp->f_op && filp->f_op->flush) {
1001   int err = filp->f_op->flush(filp);
1002   if (!retval)
1003     retval = err;
1004

}
1005
1006  dnotify_flush(filp, id);
1007  locks_remove_posix(filp, id);
1008  fput(filp);
1009  return retval;
1010  }
```

Строки 991-993

Эти строки очищают любые возникшие ошибки.

Строки 995-997

Это простая проверка необходимых для закрытия файла условий. Файл, для которого f ile_count равно 0> уже должен быть закрыт. Поэтому в этом случае f ilp_close возвращает ошибку.

Строки 1000-1001

Инициирует файловую операцию flush () (если она определена). Выполняемые ею действия зависят от конкретной файловой системы.

Строка 1008

Для освобождений структуры file вызывается fput (). Выполняемые этой функцией действия включают вызов файловой операции release (), удаляющей указатели на объекты dentry и vsmount и, наконец, освоббждающей файловый объект.

Иерархия вызовов из системного вызова close () выглядит следующим образом:

sys_close():

- **put unused f** d (). Возвращение описателя файла в доступный пул.
- **f ilp__close** (). Подготовка объекта к очистке.
- **fput** (). Очищает файловый объект.

Табл. 6.11 демонстрирует ошибку, возвращаемую sys_close () в использующую ее функцию ядра.

Таблица 6.11. Одна из ошибок sys_close()

Ошибка	Функция	Описание
EBADF	Sys_close ()	Неверный описатель файла

6.5.3 read()

Когда программа пользовательского уровня вызывает read (), Linux преобразует его в системный вызов sys_read ():

```
fs/read_write.c
272 asmlinkage ssize_t sys_read(unsigned int fd, char   user * buf,
                                size t count)
273 {
274  struct file *file;
275  size_t ret = -EBADF;
276  int fput needed;
277
278  file = fget light(fd, &fput needed);
279  if (file) {
280    ret = vfs read(file, buf, count, &file->f pos);
281    fput_light(file, fput__needed) ;
282  }
283
284  return ret;
285 }
```

Строка 272

sys_read () получает описатель файла, указатель на буфер в пользовательском пространстве и количество байтов, которые следует прочитать в этот буфер.

Строки 273-282

Поиск файла осуществляется с помощью преобразования описателя файла в файловый указатель с помощью fget_light (). Далее мы вызываем vf s_read(), выполняющую основную работу. Каждый вызов fget_light() нужно совмещать с f put_light (), что мы и сделаем после завершения vf s_read ().

Системный вызов sys_read() передает управление в vfs_read(), куда мы за ним и последуем.

```
fs/read write.c
2 00 ssize  t vfs read(struct file *file, char _ user *buf, size_t count,
                       loff t *pos)
201 {
2 02  struct inode *inode = file->f dentry->d inode;
203  ssize t ret; 204
205  if (!(file->f mode & FMODE READ))
206    return -EBADF;
207  if (!file->f op || (!file->f_op->read && \ ! file->f_op->aio_read) )
208    return -EINVAL;
209
210  ret = locks_verify_area(FLOCK VERIFY READ/ inode,
                               file, *pos, count);
211  if (!ret) {
212    ret = security file  permission (file, MAY READ);
213    if (!ret) {
214      if (file->f op->read)
215        ret = file->f op->read(file, buf, count, pos);
216         else
217           ret = do sync read(file, buf, count, pos);
218         if (ret > 0)
219           dnotify parent(file->f dentry, DN ACCESS);
220     }
221  }
222
223  return ret;
224 }
```

Строка 200

Первые три параметра передаются или преобразуются из ориганальных параметров sys_read (). Четвертый параметр - это отступ (offset) в файле, откуда следует начинать чтение. Он не может быть равен нулю, если vf spread () вызывается непосредственно, так как эта функция должна вызываться из пространства ядра.

Строка 202

Сохранение указателя на inode файла.

Строки 205-208

Выполнение базовой проверки структуры файловых операций, для того чтобы мы могли быть уверенными, что там определены операции чтения или асинхронного чтения. Если операции чтения не определены или если таблицу операций не удается найти, функция возвращает в этой точке ошибку EINVAL. Эта ошибка означает, что описатель файла, прикрепленный к структуре, не может быть использован для чтения.

Строки 210-214

Мы проверяем, чтобы читаемая область не была заблокированной и файл был авторизован для чтения. Если это не так, мы уведомляем родителя файла (в строках 218-219).

Строки 215-217

Это самое главное место vf s_read (). Если операция чтения определена, мы ее вызываем; в противном случае мы вызываем do_sync_read ().

Во время нашего исследования мы проследили только стандартную операцию чтения и не касались функции do_sync_read (). Далее нам станет ясно, что оба эти вызова доходят до одной и той же точки.

6.5.3.1 Переход от общего к частному

Это наша первая встреча со множеством абстракций, в которые мы погрузились во время движения от слоя обобщенной файловой системы к слою специальной файловой системы. Рис. 6.17 иллюстрирует то, как структура файла указывает на таблицу операций специфической файловой системы. Вспомните, что при вызове read_inode (), inode заполняется информацией, включающей поле fop, указывающее на соответствующую таблицу операций, определенную в реализации специфической файловой системы (например, ext2).

При создании файла или монтировании слой специфической файловой системы инициализирует свою структуру файловых операций. Так как теперь мы работаем с файловой системой ext2, структура файловых операций будет следующей:

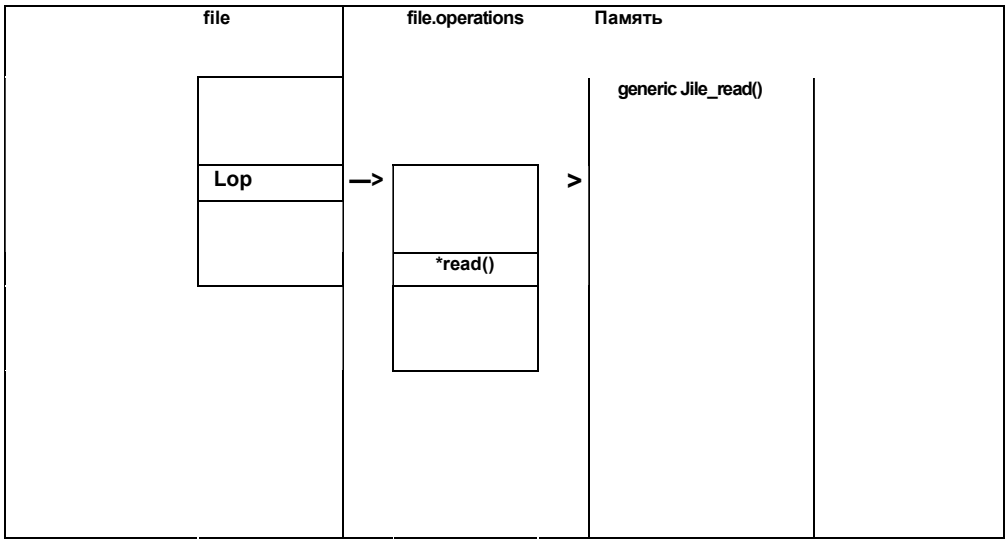


Рис. 6.17. Файловые операции

fs/ext2/file.c

```
42 struct file__operations ext2_file_operations
43     .llseek     = generic_file_llseek,
44     .read       = generic_file_read,
45     .write      = generic__file_write,
46     .aio_read   = generic_file_aio_read,
47     .aio write  = generic file aio write,
48     .ioctl      = ext2 ioctl,
49     .ramap  = generic  f ile mmap/
50     .open   = generic file open/
51     .release    *= ext2L release f ile,
52     .fsync  = ext2 sync_file,
53     .readv  = generic file readv,
54     .writev  = generic file writev/
55     .sendfile  = generic file sendfile/
56 };
```

Вы можете заметить, что практически в каждой файловой операции файловая система ext2 решает, использовать ли решения Linux по умолчанию. Отсюда возникает вопрос: когда файловая система может захотеть использовать собственную реализацию файловых операций? Когда файловая система существенно отличается от используемых в UNIX, могут потребоваться дополнительные шаги для того, чтобы Linux смогла с ней

работать. Например, для DOS- или FAT-файловых систем необходимо реализовать собственную операцию записи, но тем не менее использовать стандартную функцию чтения[1]. Рассмотрим, как слой специфической файловой системы ext2 передает управление слою обобщенной файловой системы, и начнем с рассмотрения generic_file_read().

```
mm/filemap.c
924 ssize t
925 generic_file_read(struct file *filp, char   user *buf,
                     size^t count, loff_t *ppos)
926 {
927   struct iovec Iocal iov = { . iov base = buf, .iov len = count };
928   struct kiocb kiocb;
929   ssize  t ret;
930
931   init_sync_Jciocb(&kiocb,  filp);
932   ret =  ___ generic_r_file_aio_read(&kiocb,  &Iocal_iov,  1,  ppos);
933   if  (-EIOCBQUEUED =- ret)
934      ret  = wait_on_sync_Jciocb(&kiocb);
935   return ret;
936 }
937
938 EXPORTjSYMB0L(generic_file_read) ;
```

Строки 924-925

Обратите внимание, что параметры просто передаются из функции чтения более высокого уровня. У нас есть filp, файловый указатель; buf, указатель на буфер в памяти, куда читается файл; count, количество символов для чтения, и ppos, позиция в файле, откуда начинается чтение.

Строка 927

Создается структура iovec, содержащая адреса и длину буфера пользовательского пространства, в который будут сохранены результаты чтения.

Строки 928 и 931

Инициализируется структура kiocb с помощью файлового указателя. [Kiokcb расшифровывается как kernel I/O control block (блок ядра управления вводом-выводом).]

Строка 932

Основная работа выполняется обобщенной функцией асинхронного чтения.

[1] Более подробную информацию см. в файле fs/fat/f ile.c.

Асинхронные операции ввода-вывода

Типы данных kiocb и iovec отвечают за операции асинхронного ввода-вывода в ядре Linux.

Асинхронный ввод-вывод полезен, когда процесс желает выполнить операцию ввода или вывода без ожидания поступления результата операции. Он особенно полезен для высокоскоростного ввода-вывода, так как предоставляет устройству возможность сортировать и планировать запросы ввода-вывода вместо самого процесса.

В Linux вектор ввода-вывода (iovec) предоставляет диапазон адресов памяти и определен как

include/linux/uio.h
```
20  struct  iovec
21  {
22   void  _ user *iov_base; /* BSD использует caddr t
                                (1003.1g требует void *) */
23     kernel size t iov len; /* Должна быть size t (1003.lg) */
24  };
```

Это простой указатель на раздел памяти и длину памяти.

Блок ядра управления вводом-выводом (kiocb) - это структура, необходимая для определения того, когда и как обрабатывать вектор асинхронного ввода-вывода.

Функция __ generic_f ile_aio_read () использует структуры kiocb и iovec для прямого чтения page_cache.

Строки 933-935

После того как мы запустили чтение, мы ожидаем, пока чтение закончится, и затем возвращаем результат в операцию чтения.

Вспомните путь do_sync_read () в vf s_read (); здесь эта функция будет запус каться по другому пути. Продолжим рассмотрение пути файлового ввода-вывода и рас смотрим __ generic_f ile_aio__read ():

```
mm/filemap.c
835 ssize t
83 6 __generic file aio read(struct kiocb *iocb,
        const struct iovec *iov,
837     unsigned long nr segs, loff t *ppos)
838 {
83 9   struct file *filp = iocb->ki filp;
840   ssize t retval;
841   unsigned long seg;
842   size t count;
843
```

```
844   count = 0;
845   for (seg = 0; seg < nr_segs; seg++) {
846     const struct iovec *iv = &iov[seg];

852     count += iv->iov len;
853     if (unlikely((ssize t)(count|iv->iov len) < 0))
854      return -EINVAL;
855     if (access_ok(VERIFY_WRITE, iv->iov_base, iv->iov_len) )
856      continue;
857     if (seg == 0)
858       return -EFAULT;
859     nr segs = seg;
860     count -= iv->iov len
861     break;
862   }
```

Строки 835-842

Вспомните, что nr_segs устанавливается в 1 в нашем вызове и что iocb и iov хранят файловый указатель и информацию о буфере. Сразу после этого мы извлекаем файловый указатель из iocb.

Строки 845-862

Этот цикл for проверяет передаваемую структуру iovec, состоящую их корректных сегментов. Вспомните, что она содержит информацию о буфере пользовательского пространства.

mm/filemap.c

```
863
864  /* объединение iovecs и пересылка direct-to-BIO для 0 DIRECT */
865  if (filp->f_flags & 0_DIRECT) {
866    loff t pos = *ppos, size;
867    struct address space *mapping;
868    struct inode *inode;
869
870    mapping = filp->f_mapping;
871    inode = mapping->host;
872    retval = 0;
873    if (!count)
874      goto out; /* skip atime */
875    size = i size read(inode);
876    if (pos < size) {
877     retval = generic_file_direct_IO(READ, iocb,
878           iov, pos, nr_segs);
```

```
879      if (retval >= 0 && !is jsync kiocb(iocb))
880        retval = -EIOCBQUEUED;
881      if (retval > 0)
882        *ppos = pos + retval;
883     }
884     f ile  accessed(filp) ;
885     goto out;
886   }
```

Строки 863-886

Этот фрагмент кода просто входит в чтение из прямого ввода-вывода. Прямой ввод-вывод передает кеш страниц и используется при работе с блочными устройствами. Для наших целей нам тем не менее вообще не нужно заходить в этот фрагмент кода. Основная масса файлового ввода-вывода происходит в файловом кеше, который мы вскоре опишем, что значительно быстрее, чем непосредственная работа с блочным устройством.

mm/filemap.c

```
887
888    retval = 0;
889    if (count) {
890      for (seg = 0; seg < nr segs; seg++) {
891      read descriptor t desc;
892
893      desc.written = 0;
894      desc.buf = iov[seg].iov base;
895      desc.count = iov[seg].iov len;
896      if (desc.count == 0)
897        continue;
898      desc.error = 0;
899      do generic f ile read(filp,ppos, &desc, f ile read actor) ;
900      retval += desc.written;
901      if (!retval) {
902        retval = desc.error;
903        break;
904      }
905     }
906    }
907  out:
908    return retval;
909  }
```

Строки 889-890

Так как наша iovec корректна, нам нужен только один сегмент, над которым мы выполняем этот цикл for единственный раз.

Строки 891-898

Мы преобразуем структуру iovec в структуру read_descriptor_t. Структура read_descriptor_t следит за состоянием чтения. Вот описание структуры read_descriptor_t:

```
include/linux/fs.h
837  typedef struct {
83 8  size t written;
83 9  size_t count;
840    char __user * buf;
841    int error;
842   } read_descriptor_t;
```

Строка 838

Поле written следит за количеством переданных байтов.

Строка 839

Поле count следит за количеством байтов, которые осталось передать.

Строка 840

Поле buf хранит текущую позицию в буфере.

Строка 841

Поле error хранит код ошибки, возникающей в процессе операции чтения.

Строка 899

Мы передаем нашу новую структуру типа read_descriptor_t desc в do_generic_f ile_read() вместе с нашим файловым указателем f ilp и нашей позицией ppos. f ile_read_actor () - это функция, копирующая страницу в буфер пользовательского пространства, содержащуюся в desc.[1]

Строки 900-909

Количество считанного подсчитывается и возвращается вызвавшему функцию коду.

В этой точке функции read () мы получаем доступ к странице кеша[2] и определяем, можем ли мы выполнять чтение из оперативной памяти, не используя прямого доступа к блочному устройству.

[1] **f ile_read_actor () можно найти на строке 794 в mm/ f ilemap. c.**

[2] **Кеш страниц описывается в разд. 6.4 «Кеш страниц».**

6.5.3.2 Отслеживание кеша страниц

Вспомните, что в последнюю встреченную нами функцию do_generic_file__ read() передавались файловый указатель filp, отступ ppos, read_descriptor_t desc и функция f ile_read__actor.

```
include/linux/fs.h
1420 static inline void do_generic_file_read(struct file * filp,
1421           loff t *ppos, read descriptor t * desc,
1422           read actor t actor)
1423 {
1424  do generic mapping read(filp->f ..mapping,
1425        &filp->f ra,
1426        filp,
1427        ppos,
1428        desc,
1429        actor);
1430 }
```

Строки 1420-1430

do_generic_f ile_read() - это просто обертка для do_generic_mapping_ read(); f ilp->f_mapping - указатель на объект address_space, a f ilp-> f _ra - структура, хранящая адреса состояний предварительного чтения из файлов[1].

Итак, мы преобразовали наше чтение из файла в чтение из кеша страниц с помощью объекта address_space в нашем файловом указателе. Так как do_generic__ mapping_read () является чрезвычайно длинной функцией, обрабатывающей множество различных случаев, мы попытаемся проанализировать ее код как можно короче.

```
mm/filemap.c
645 void do  generic jnapping read(struct address space *mapping,
646        struct file ra state * ra,
647        struct file * filp,
648        loff_t *ppos,
649        read descriptor t * desc,
650        read actor t actor)
651 {
652   struct inode *inode = mapping->host;
653   unsigned long index, offset;
654   struct page *cached page;
655   int error;
```

[1] **Более подробно это поле для оптимизации чтения описывается в подразделе «Структура file».**

```
656   struct  file_ra_state ra =  *_ra;
657
658    cached  page = NULL;
659    index = *ppos » PAGE_CACHE_SHIFT;
660    offset = *ppos & ~PAGE_CACHE_MASK;
```

Строка 652

Мы извлекаем inode читаемого файла из address_space.

Строки 658-660

Инициализируем cached_page в NULL перед тем, как определить, существует ли она в кеше страниц. Также мы рассчитываем index и offset на основе содержимого кеша страниц. Индекс соответствует количеству страниц в кеше страниц, а отступ соответствует размещению внутри этой страницы. Когда размер страницы равен 4096 байт, правильный бит, сдвинутый на 12, представляет файловый указатель, соответствующий индексу страницы.

«Кеш страниц может состоять из больших, чем размер одной страницы, частей, так как это обеспечивает большую скорость передачи» (linux/pagemap.h). PAGE_CACHE_ SHIFT PAGE_CACHE__MASK - это настройки, управляющие структурой и размером кеша страниц:

```
ram/filemap.c
661
662    for (;;) {
663      struct page *page;
664      unsigned long end_index, nr, ret;
665      loff t isize = i size  read( inode) ;
666
667   end_index = isize » PAGE_CACHE_SHTFT;
668
669      if (index > end index)
670        break;
671      nr = PAGE_CACHE_SIZE;
672      if (index == end_index) {
673        nr= isize & ~PAGE_CACHE_MASK;
674        if (nr <= offset)
675          break;
676      }
677
678     cond_resched();
679     page cache readahead(mapping, &ra, filp, index);
680     nr = nr - offset;
```

Строки 662-680

Этот блок кода проходит по всему кешу страниц и выбирает достаточно страниц, необходимых для выполнения запроса команды read.

```
mm/filemap.c
682 find_page:
683     page = find_get_page(mapping, index);
684     if (unlikely(page == NULL)) {
685       handle_ra_miss(mapping, &ra, index);
686       goto no_cached_page;
687     }
688     if (!PageUptodate(page))
689       goto page_not_up_to_date;
```

Строки 682-689

Мы пытаемся найти первую запрашиваемую страницу. Если страница не находится в кеше страниц, мы переходим к метке no__cache_page. Если страница не обновлена, мы переходим к метке page_not__up_to_date; f ind_get_page () использует дерево оптимизации адресного пространства для поиска страницы в index, означающей отступ.

```
mm/filemap.c
690 page_ok:
691   /* Если пользователи могут записывать в эту страницу с помощью
692    * случайных виртуальных адресов, вам придется позаботиться о по-
693    * тенциальном связывании перед чтением страницы со стороны ядра.
694    */
695     if (mapping_writably_mapped(mapping))
696       flush_dcache_page(page);
697
698     /* Отметка читаемой страницы, если мы начинаем чтение. */
700     if (loffset)
701       mark_page_accessed(page);

714     ret = actor(desc, page, offset, nr);
715     offset += ret;
716     index += offset >> PAGE_CACHE_SHIFT;
717     offset &= ~PAGE_CACHE_MASK;
718
719     page_cache_release(page);
720     if (ret == nr && desc->count)
721       continue;
```

```
722         break;
723
```

Строки 690-723

Комментарии в коде достаточно подробны. Если нам нужно получить больше страниц, мы сразу возвращаемся в начало цикла, где манипуляции с индексом и отступом в строках 714-716 позволяют выбрать следующую получаемую страницу. Если больше не нужно читать никаких страниц, мы прерываем цикл for.

```
mm/filemap.c
724 page not up  to date:
725       /* Получение эксклюзивного доступа к страницам ... */
726       lock  page(page) ;
727
728       /* Отменили ли мы хеширование перед снятием блока? */
729       if (!page->mapping) { 73 0
          unlock_page(page);
731         page_cache_release(page);
732         continue;
734
735       /* Заполнил ли ее кто-либо? */
736       if (PageUptodate(page)) {
737         unlock page(page);
738         goto page  ok;
739       }
```

***Строки** 724-739*

Если страница не обновлена, мы проверяем ее снова, и возвращается на метку page_ok, если она обновлена. В противном случае мы пытаемся получить к ней эксклюзивный доступ; для этого нам необходимо подождать, пока мы его получим. Как только мы получили эксклюзивный доступ, мы смотрим, хочет ли страница удалить себя из кеша страниц; если да, мы выбрасываем ее перед возвращением в начало цикла for. Если она до сих пор осталась и обновлена, мы разблокируем страницу и переходим на метку page_ok.

```
mm/filemap.c
741 readpage:
742 /* Итак, начинаем настоящее чтение. Чтение разблокирует страницу. */
743       error = mapping->a_ops->readpage(filp, page);
744
745     if (!error) {
```

```
746       if (PageUptodate(page))
747         goto page ok;
748       wait on page locked(page);
749       if (PageUptodate(page))
750         goto page ok;
751       error = -E0;
752      }
753
754   /* Так!.. Возникла ошибка асинхронного чтения. Сообщаем об этом */
755       desc->error = error;
756       page_cache_release(page);
757       break;
```

Строки 741-743

Если страница не обновлена, мы можем вернуться на предыдущую метку с установленным блоком страницы. Настоящая операция чтения mapping->a_ops-> readpage (f ilp, page) разблокирует страницу. [Мы отследим readpage () немного позднее, когда закончим наше объяснение.]

Строки 746-750

Если чтение страницы завершилось успешно, мы проверяем, чтобы она была обновлена, и переходим на page_ok, если это так.

Строки 751-757

Если возникает ошибка асинхронного чтения, мы записываем ошибку в desc, освобождаем страницу из кеша страниц и разрываем цикл for.

```
mm/filemap.c
759 no cached page:
760     /*
761     * Итак, она не кеширована, поэтому нам нужно создать
762     * новую страницу...
763     */
764     if (!cached_page) {
765       cached page = раде cache alloc cold(mapping) ;
766       if (!cached page) {
767         desc->error = -ENOMEM;
7 68        break;
769       }
770     }
771     error = add to page cache lru(cached  page, mapping,
772             index, GFP_KERNEL);
773     if (error) {
774       if (error == -EEXIST)
```

```
775          goto find_page;
776        desc->error = error;
777        break;
778      }
779        page = cached_page;
780      cached_page = NULL;
781      goto readpage;
782    }
```

Строки 698-772

Если читаемая страница не кешируется, мы выделяем новую страницу в адресном пространстве и добавляем ее в кеш последних использованных (LRU) и кеш страниц.

***Строки** 773-775*

Если произошла ошибка при добавлении страницы в кеш из-за того, что она уже там находится, мы переходим к метке f ind_page и делаем еще одну попытку. Такая ситуация может возникнуть, если несколько процессов пытаются прочитать одну и ту же некешируемую страницу; как только выделение завершится успешно, другие операции выделения попытаются выделиться и обнаружат, что выделение уже выполнено.

***Строки** 776-777*

Если при добавлении страницы в кеш произошла ошибка из-за того, что она уже там находится, мы протоколируем ошибку и разрываем цикл for.

Строки 779-781

После того как мы успешно выделили и добавили страницу в кеш страниц и кеш LRU, мы устанавливаем наш указатель страницы на новый кеш и пытаемся прочесть его, перейдя на метку readpage.

ram/filemap.c

```
784                    *_ra = ra;
785
786    *ppos = ((loff_t) index « PAGE_CACHE_SHIFT) + offset;
787    if (cached_page)
788      page cache release(cached page);
789    file_accessed(filp);
790  }
```

***Строка** 786*

Мы рассчитываем реальный отступ на основе индекса нашей страницы кеша и отступа.

Строки 787-788

Если мы выделяем новую страницу и корректно добавили ее в кеш страниц, мы ее удаляем.

Строка 789

Мы обновляем время последнего обновления файла через inode.

Логика, заключенная в этой функции, представляет собой ядро кеша страниц. Обратите внимание, что кеш страниц не касается никаких специфических данных файловой системы. Это позволяет ядру Linux использовать кеш страниц независимо от нижележащих структур файловой системы. Благодаря этому кеш страниц может содержать страницы из MINIX, ext2 и MSDOS одновременно.

Кеш страниц поддерживает свой агностицизм слоя специфической файловой системы с помощью функции readpage () из адресного пространства. Для каждой специфической файловой системы реализована своя функция readpage (). Поэтому, когда слой обобщенной файловой системы вызывает mapping->a_ops->readpge (), она в свою очередь вызывает специфическую readpage () из драйвера файловой системы структуры addrerss_space_operations. Для файловой системы ext2, readpage () определена следующим образом:

```
fs/ext2/inode.c
676 struct address space operations ext2 aops = {
677     .readpage  = ext2 readpage,
678     .readpages  = ext2 readpages,
679     .writepage  = ext2 writepage,
68 0    .sync_page  = block_sync_page,
681     .prepare  write = ext2 prepare write,
682     .commit write  = generic commit write,
683     .bmap    = ext2_bmap,
684     .direct IO   = ext2 direct IO,
685     .writepages   = ext2 writepages,
686 };
```

Поэтому readpage () вызывает в свою очередь ext2_readpage ():

```
fs/ext2/inode.c
616 static int ext2 readpage(struct file *file, struct page *page)
617 {
618   return mpage_readpage(page, ext2_get_block);
619 }
```

ext2_readpage () вызывает mpage_readpage (), являющуюся вызовом слоя обобщенной файловой системы и передает управление функции слоя специфической файловой системы ext2_get_block().

Функция обобщенной файловой системы mpage_readpage () ожидает функцию get_block () в качестве своего второго аргумента. Каждая файловая система реализует несколько функций ввода-вывода, специфичных для данной файловой системы; get_block () является одной из них. Функция get_block () файловой системы отображает логические блоки в страницах address_space в реальные страницы блочного устройства специфической файловой системы. Давайте рассмотрим специфику mpage_readpage ().

```
fs/mpage.c
358 int mpage readpage(struct page *page, get block t get block)
359 {
3 60   struct bio *bio = NULL;
3 61   sector_t last_block_in_bio = 0;
362
3 63  bio = do mpage readpage(bio, page, 1,
3 64       &last block in bio, get block) ;
365   if (bio)
366     mpage bio submit(READ, bio);
3 67   return 0;
368 }
```

Строки 360-361

Мы выделяем пространство для управления структурой bio, используемой адресным пространством для управления страницами, которые мы пытаемся считать с устройства.

Строки 363-364

Вызывается do_mpage_readpage (), преобразующая логические страницы в структуру bio, составленную из реальных страниц блочного устройства. Структура bio отслеживает информацию, связанную с блочным вводом-выводом.

Строки 365-367

Мы передаем новую созданную структуру bio в функцию mpage_bio_submit () и возвращаемся.

Давайте немного вернемся назад и просмотрим еще раз (на более высоком уровне) поток функции чтения:

1. Используя описатель файла из вызова read(), мы получаем указатель на файл, из которого мы получаем inode.

2. Слой файловой системы проверяет нахождение страницы или страниц в кеше страниц в оперативной памяти, соответствующих данному inode.
3. Если страница не найдена, слой файловой системы использует драйвер специфической файловой системы для преобразования запроса к файлу в ввод-вывод с блочного устройства.
4. Мы выделяем пространство для страниц в кеше страниц address_space и создаем структуру bio, связывающую новые страницы с секторами на блочном устройстве.

mpage_readpage () - это функция, создающая структуру и связывающая новые выделенные страницы со структурой bio. При этом в страницах не содержится никаких данных. Для заполнения слою файловой системы необходимо выполнить операции с устройством с помощью драйвера блочного устройства. Они выполняются с помощью функции submit_bio () внутри mpage_bio__submi t ():

```
fs/mpage.c
90 struct bio *mpage bio submit (int rw, struct bio *bio)
91 {
92   bio->bi_end_io = mpage_end__io_read;
93   if (rw == WRITE)
94     bio->bi end  io = mpage end io  write;
95   submit bio (rw, bio);
96   return NULL;
97 }
```

Строка 90

Первое, на что нужно обратить внимание, - это mpage_bio_submit (), работающая как для чтения, так и для записи через параметр rw. Она заполняет структуру bio, которая в случае чтения является пустой и должна быть заполнена. В случае записи структура bio заполнена и драйвер блочного устройства копирует ее содержимое на диск.

Строки 92-94

Если мы производим чтение или запись, мы устанавливаем соответствующую функцию, вызываемую по завершении ввода-вывода.

Строки 95-96

Мы вызываем submit_bio() и возвращаем NULL. Вспомните, что mpage_ readpageO сама ничего не делает с возвращаемым mpage_bio_submit () значением.

submit_bio () является частью драйвера слоя обобщенного блочного устройства ядра Linux.

```
drivers/block/ll rw blk.c
2433 void submit bio(int rw, struct bio *bio)
2434 {
2435   int count = bio_sectors(bio);
2436
2437   BIO BUG ON(!bio->bi size);
2438   BIO BUG ON(!bio->bi io vec);
2439   bio->bi rw = rw;
2440   if (rw & WRITE)
2441     mod page state(pgpgout, count);
2442   else
2443     mod page state(pgpgin, count);
2444
2445   if (unlikely(block dump)) {
2446     char b[BDEVNAME SIZE];
2447     printk(KERN_DEBUG H%s(%d): %s block %Lu on %s\n''/
2448       current->coram/ current->pid/
2449       (rw & WRITE) ? "WRITE" : "READ",
2450       (unsigned long long)bio->bi sector,
2451       bdevname(bio->bi_bdev,b));
2452   }
2453
2454   generic make request(bio);
2455 }
```

Строки 2433-2443

Эти вызовы включают отладку: устанавливают атрибуты чтения-записи структуры bio и выполняют заполнение состояния страницы.

Строки 2445-2452

Эти строки обрабатывают редкие случаи, возникающие при выводе информации о блоке. Выдается отладочное сообщение.

Строка 2454

generic_make_request () скрывает основную функциональность и использует очередь запросов драйвера специфического блочного устройства для обработки блочных операций ввода-вывода.

Комментарии к этой части generic_make_request () являются самодостаточными.

drivers/block/ll_rw_blk.c
```
233 6  /* Вызывающий generic_make_request должен удостовериться,  что
2337  * bi_io_vec установлена  для  описания  буфера  в  памяти и что bi_dev
233 8  * и bi_sector установлены для  описания адресов драйвера и
233 9  * bi_end_io и  опционально bi_private устанавливается для описания
2340  *  сигнализации  сообщением о  завершении.  */
```

На этом этапе мы конструируем структуру bio, и далее структура bio_vec отображается в буфер памяти, упомянутый в строке 2337, структура bio инициализируется параметрами адресов устройства. Если вы хотите углубиться дальше в драйвер блочного устройства, обратитесь к подразд. 5.2.1, «Обзор блочных устройств», описывающему то, как драйвер блочного устройства обрабатывает очередь запросов и специальные аппаратные константы, находящиеся в этом устройстве. Рис. 6.18 иллюстрирует, как системный вызов read () проходит по различным слоям функциональности ядра.

После этого драйвер блочного устройства прочитает настоящие данные и поместит их в структуру bio, в чем легко убедиться, если за ним проследить. Новые выделенные страницы в кеше страниц заполняются, а ссылки на них передаются обратно на слой VFS и копируются в раздел, определенный в пользовательском пространстве не так давно в оригинальном вызове read ().

Мы ожидаем, что вы спросите: «Это же только одна половина истории. А что насчет записи вместо чтения?»

Мы надеемся, что наше объяснение прояснило для вас системный вызов read (), проходящий в ядре Linux по тому же пути, по которому проходит что и вызов write (). Тем не менее существует несколько различий, которые мы сейчас рассмотрим.

6.5.4 write()

Вызов write () отображает в память sys_write () и затем vf s_write () тем же способом, которым это проделывает и вызов read ().

```
fs/read write.c
244 ssize_t vfs_write(struct file *file, const char   user *buf,
                      size_t count, loff_t *pos)
245 {

259      ret = file->f_op->write(file, buf, count, pos);

268 }
```

vf s_write () использует обобщенную функцию записи f ile_operations для определения того, какой слой специфической файловой системы используется для чтения.

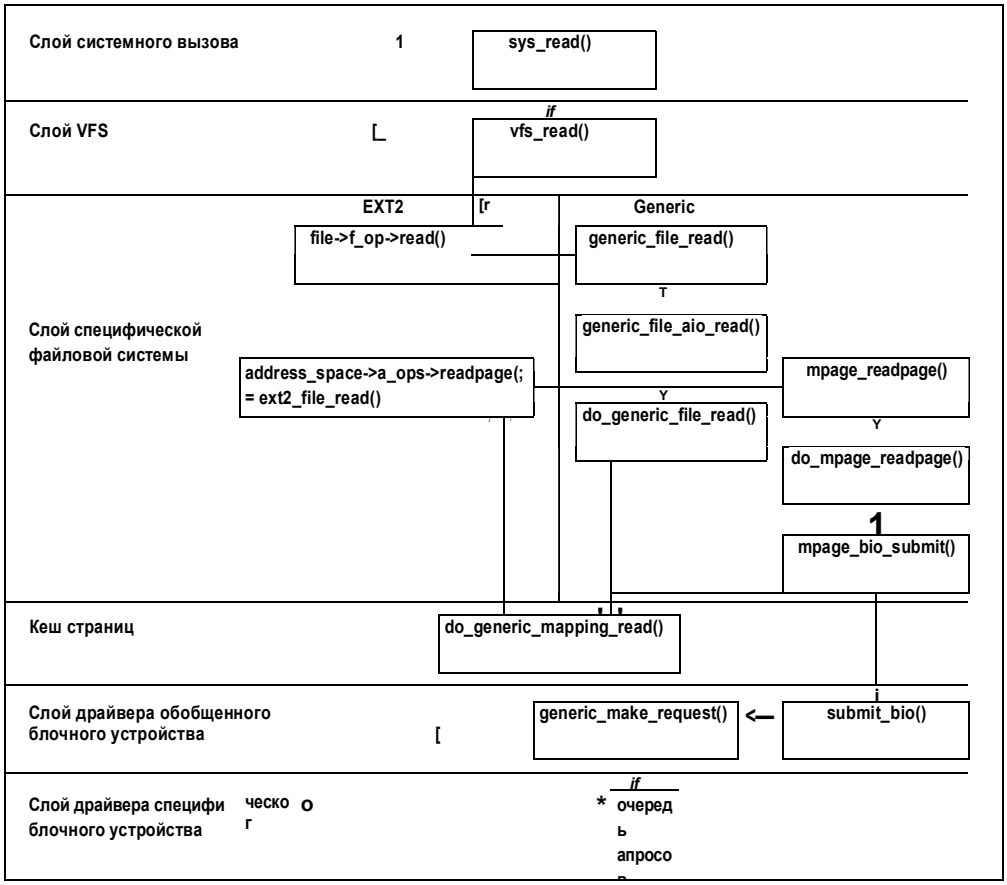


Рис. 6. 18. Прохождение readQ сверху вниз

В нашем примере с ext2 он преобразуется с помощью структуры ext2_file_operations:

```
fs/ext2/file.c
42 struct file operations ext2 file operations = {
43    .llseek  = generic f ilejlseek,
44    .read  = generic file read,
45    .write  = generic_file_write,

56 };
```

Строки 44-45

Вместо вызова generic_file_read() мы вызываем generic_file_ write ().

generic_f ile_write () блокирует файл для предотвращения повторной записи в один и тот же файл и вызывает generic_f ile_write_nolock(); generic_ f ile_write_nolock() преобразует файловый указатель и буферы для параметров kiocb и iovec и вызывает функцию записи в кеш страниц generic_file_ aio_write_nolock ().

Здесь появляется отличие от чтения. Если записываемая страница не находится в кеше, запись не обращается к самому устройству. Вместо этого она помещает страницу в кеш страниц и выполняет чтение. Страницы в кеше страниц записываются на диск не сразу; вместо этого они помечаются как грязные и периодически сбрасываются на диск.

В этом заключается главное отличие от функции read () readpage (). Внутри generic_f ile_aio_write_nolock() указатель address__space_operations получает доступ к prepare_write () и commit_write (), различающиеся от одной файловой системы к другой. Вспомните ext2_aops - и мы увидим, что драйвер ext2 использует собственную функцию, ext2_jpreapre_write (), и обобщенную функцию generic_commit__write ().

```
fs/ext2/inode.c
628 static int
629 ext2 prepare write(struct file *file, struct page *page,
630       unsigned from, unsigned to)
631  {
632     return block_prepare_write(page,from,to, ext2_get_block);
633  }
```

Строка 632

ext2_prepare_write () - это простая обертка для обобщенной функции файловой системы block_prepare_write (), передаваемой в специфическую для файловой системы ext2 функцию get_block().

block_prepare_write () выделяет любые новые буферы, которые нужно записать. Например, если в файл добавляются новые данные, создаются новые буферы и связываются со страницами для сохранения новых данных.

generic_commit_wri te () получает передаваемую страницу и проходит по всем буферам, маркируя каждый из них как грязный. Разделы prepare и coiranit для записи разделяются для предотвращения раздробления сбрасываемых из кеша страниц на блочное устройство.

6.5.4.1 Сброс грязных страниц

Вызов write () возвращается после вставки и помечает грязными все записываемые страницы. В Linux имеется демон pdf lush, который записывает грязные страницы из кеша страниц на блочное устройство в двух случаях:

* **Свободная память в системе приближается к пороговому значению.** Страницы из кеша страниц сбрасываются на диск для освобождения памяти.
- **Грязные страницы достигли определенного возраста.** Страницы, которые не были сброшены на диск на протяжении некоторого времени, записываются на свое блочное устройство.

Демон pdflush вызывает специфическую для файловой системы функцию wri tepages (), когда он оказывается готов к записи страниц на диск. Поэтому для нашего примера нам придется вспомнить структуру ext2_f i 1 e_operations, которая заменяет wri tepages () функцией ext2_writepages () \

```
fs/ext2/inode.c
670 static int
671 ext2_writepages(struct address_space *mapping,
                    struct writeback control *wbc)
672 {
673   return mpage writepages(mapping, wbc, ext2 get block) ;
674 }
```

Как и другие специфические реализации обобщенной функции файловой системы, ext2_writepages () просто вызывает функцию обобщенной файловой системы mpage_writepages () с помощью специфической для файловой системы функции ext2_get_block ().

mpage_writepages () циклически проходит по всем грязным страницам и вызывает для всех грязных страниц функцию mpage_writepages (). Подобно mpage_ readpages (), mpage_writepages () возвращает структуру bio, отображающую данные с физического диска в физическую оперативную память. Далее mpage_ wri tepages () вызывает submi t_bio () для пересылки новой структуры bio драйверу блочного устройства для передачи данных на само устройство.

[1] Демон pdflush нас мало интересует для отслеживания записи, и мы его можем игнорировать. Тем не менее, если для вас интересны подробности, то вам стоит ознакомиться с кодами iran/pdf lush. c, mm/ fs-writeback . c и mm/page-writeback.c.

Резюме

Эта глава началась с обзора структур и глобальных переменных, образующих модель файла. Среди этих структур - superblock, inode, dentry и структуры файла. Далее мы рассмотрели связанные с VFS структуры. Мы увидели, как в VFS работает поддержка различных файловых систем.

Далее мы рассмотрели связанные с VFS системные вызовы открытия и закрытия для иллюстрации их совместной работы. Мы отследили вызовы пользовательского пространства read() и write () и их прохождение через слой обобщенной файловой системы и слой специфической файловой системы. Используя драйвер файловой системы ext2 в качестве примера слоя специфической файловой системы, мы показали, как ядро использует вызовы функций драйвера специфической файловой системы и обобщенной файловой системы. Это привело нас к обсуждению кеша страниц - разделу памяти, хранящему страницы блочного устройства, присоединенного к системе, к которым недавно обращались.

Упражнения

1. В каких случаях используется поле inode i_hash вместо i_list? Почему в одной и той же структуре содержится как хеш-список, так и линейный список?
2. Среди всех рассмотренных файловых структур назовите структуры, соответствующие жесткому диску.
3. Для каких типов операций используется объект dentry? Почему недостаточно обычных inode?
4. Как связаны описатель файла и структура файла? Как один к одному? Много к одному? Один ко многим?
5. Для чего используется структура f d_set?
6. Какой тип структуры данных отвечает за обеспечение максимальной скорости работы с кешем?
7. Представьте, что вам нужно написать новый драйвер файловой системы? Вы заменяете драйвер файловой системы ext2 новым драйвером (media_f s), оптимизированным для ввода-вывода при работе с мультимедийными файлами. Где вам нужно произвести изменения, чтобы быть уверенным, что ядро Linux будет использовать ваш драйвер вместо драйвера ext2?
8. Каким образом страница становится грязной? Как грязные страницы записываются на диск?

Глава 7

Планировщик и синхронизация ядра

В этой главе:

Ядро Linux является многозадачным, что означает, что в системе может выполняться множество процессов таким образом, как будто выполняется только один процесс. Способ, которым операционная система выбирает, какому процессу в данный момент разрешено получить доступ к системному процессору (процессорам), называется **планировщиком.**

Планировщик отвечает за перемену доступа к процессору между различными процессами и выбирает порядок, в котором они получают доступ к процессору. Linux, как и большинство операционных систем, включает планировщик с помощью таймера прерывания. Когда таймер заканчивается, ядру необходимо решить, передать ли процессор другому процессу, и если да, то какому процессу позволить получить процессор следующим. Количество времени между прерываниями таймера называется **временным срезом (timeslice).**

Системные процессы имеют тенденцию разделяться на два типа: интерактивные и неинтерактивные. Интерактивные процессы сильно зависят от ввода-вывода и в результате не всегда используют выданный им временной срез и вместо этого передают его другому процессу. Неинтерактивные процессы, наоборот, сильно зависят от процессорного времени и чаще всего используют свои временные срезы полностью. Планировщику приходится балансировать между требованиями процессов этих двух типов и следить за тем, чтобы каждому процессу досталось достаточно времени без видимого влияния на выполнение других процессов.

Linux, как и другие планировщики, различает еще один тип процессов: процессы реального времени. Процессы реального времени должны выполняться в реальном времени. Linux поддерживает процессы реального времени, но они выходят за пределы логики планировщика. Проще говоря, планировщик Linux считает процессом реального времени любой процесс, для которого установлен более высокий приоритет, чем для остальных. Поэтому следить за тем, чтобы процессы не съедали все остальное процессорное время во время работы процесса в реальном времени приходится разработчику.

Обычно планировщики используют несколько типов очереди процессов для управления выполнением процессов в системе. В Linux эта очередь процессов называется **очередью выполнения.** Очередь выполнения подробно описана в гл. 3, «Процессы: принципиальная модель выполнения»[1], но давайте вспомним некоторые фундаментальные принципы связи между планировщиком и очередью выполнения.

В Linux очередь выполнения состоит из двух массивов приоритетов:

- **активных:** хранят процессы, еще не использовавшие свой временной срез;
- **истекших:** хранят процессы, которые уже использовали свой временной срез.

[1] **Разд. 3.6 описывает очередь выполнения.**

На самом верхнем уровне работа планировщика в Linux заключается в выборе процесса с наибольшим приоритетом, передаче ему процессора для выполнения и помещении его в массив истекших, когда его временной срез завершается. Не забывая это назначение наивысшего уровня, давайте рассмотрим операции планировщика Linux.

7.1 Планировщик Linux

В ядре Linux 2.6 представлен полностью новый планировщик, называемый также планировщиком 0(1). Это значит, что планировщик выполняет планирование выполнения задач за константное время[1]. Гл. 3 описывает базовые структуры планировщика, их инициализацию при создании процессов. Этот раздел описывает, как задачи выполняются на единственном процессоре. Мы будем упоминать о планировщике для многопроцессорной системы (SMP), но в общем процессы внутри всех планировщиков совпадают. Далее мы опишем, как планировщик переключает текущий выполняющийся процесс, определяя, какой контекст будет вызван следующим, и затем мы коснемся других значительных изменений в ядре 2.6 - приоритетных прерываний обслуживания.

На наивысшем уровне планировщик просто группирует функции, работающие с этими структурами данных. Почти весь код, связанный с планировщиком, можно найти в kernel/sched.c и include/linux/sched.h. Важно не забывать, что, как мы упоминали ранее, в коде планировщика термины *задача* и *процесс* обозначают одно и то же. Задача или процесс в планировщике представляют собой набор структур данных и поток управления. Код планировщика также связан с task_struct - структурой данных, используемой Linux для слежения за процессами[2].

7.1.1 Выбор следующей задачи

После того как процесс инициализирован и размещен в очереди выполнения, в какой то момент он должен получить доступ к процессору для выполнения. За передачу управления процессором другому процессу отвечают две функции: schedule () и scheduler_tick(); scheduler_tick() - это системный таймер, который периодически вызывается ядром и помечает процессы, выполнение которых нужно распланировать. При наступлении события таймера текущий процесс замирает и ядро Linux берет управление процессором на себя. Когда сообщение таймера завершается, ядро обычно передает управление процессором процессу, который замер. При этом, если приостановленный процесс был помечен как планируемый, ядро вызывает schedule () и выбирает процесс, которому передается управления вместо того, который выполнялся перед тем, как ядро заняло процессор. Процесс, выполнявшийся перед тем, как ядро перехватило управление, называется **текущим процессом.** В некоторых особо сложных

[1] 0(1) в нотации большого О означает константное время. [2]- В гл. 3 подробно описывается структура task_struct.

случаях ядро может перехватить управление у самого себя; это называется **приоритетным прерыванием обслуживания ядра.** В следующем разделе мы будем считать, что планировщик выбирает, какому из двух процессов пользовательского пространства нужно передать управление процессором.

Рис. 7.1 иллюстрирует, как процессор с течением времени передается между различными процессами. Мы видим, что *Process A* контролирует процессор и выполняется. Системный таймер scheduler_tick() завершается, получает управление процессором от А и помечает А как требующий перепланирования. Ядро Linux вызывает schedule (), выбирающую *Process B* и передающую ему управление процессором.

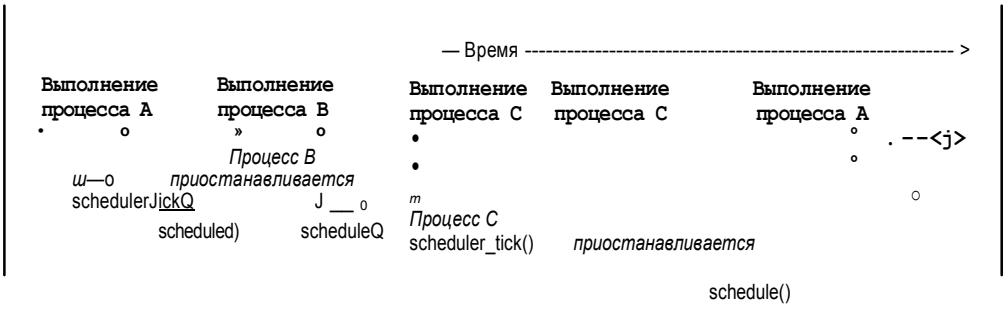


Рис. 7.1. Планирование процессов

Process B выполняется некоторое время, которое ему выделено для работы с процессором. Обычно это происходит, когда процесс ожидает некоторых ресурсов. *B* вызывает schedule (), выбирающую в качестве следующего *Process* С.

Process C выполняется до возникновения scheduler_tick (), которая не помечает С как требующий перепланировки. В результате внутри вызова schedule () у *Process C* не забирается управление процессором.

Process C приостанавливается вызовом schedule (), определяющей, что *Process A* требуется передать управление процессором, и запускающей *A* на выполнение.

Сначала мы рассмотрим schedule (), с помощью которой ядро Linux решает, какой процесс нужно выполнять следующим, а затем мы рассмотрим scheduler_tick(), в которой ядро решает, какому процессу можно занять процессор. Комбинированный эффект этих двух функций демонстрирует поток управления внутри планировщика:

```
kernel/sched.c
2184  asmlinkage void schedule(void)
2185  {
2186  long *switch_count;
2187  task_t *prev, *next;
2188  runqueue_t *rq;
2189  prio_array_t *array;
```

```
2190 struct list head *queue;
2191 unsigned long long now;
2192 unsigned long run_time;
2193 int idx;
2194
2195 /*
2196 * Тест на атомарность.Так как do exit() необходимо вызвать
2197 * schedule() атомарно, мы игнорируем этот путь.
2198 * В противном случае жалуемся, что мы вызываем планировщик,
2199 * хотя это и не нужно. */
2200 if (likely(! (current->state & (TASK DEAD | TASK ZOMBIE) ) ) ) {
2201 if (unlikely(in atomic())) {
2202 printk(KERN_ERR "bad: scheduling while atomic!\n H);
2203 dump_stack();
2204 }
2205 }
 2206
2207 need resched:
2208 preempt disable () ;
2209 prev = current;
2210 rq = this rq();
2211
2212 release_Jcernel_lock(prev);
2213 now = sched_clock();
2214 if (likely(now - prev->timestamp < NS MAX SLEEP AVG))
2215 run time = now - pr ev-> times tamp;
2216 else
2217 run time = NS MAX SLEEP AVG;
2218
2219 /*
2220 * Задача с интерактивным поведением получает меньше времени
2221 * благодаря высокому sleep avg для отсрочки потери ими их
2222 * интерактивного статуса
2223 */
2224 if (HIGH CREDIT(prev))
2225 run time /= (CURRENT BONUS(prev) ? : 1);
```

Строки 2213-2218

Мы подсчитываем длительность времени, в течение которого процесс будет активен в планировщике. Если процесс будет активен дольше чем среднее максимальное время сна (NS_MAX_SLEEP_AVG), мы устанавливаем его время выполнения равным среднему максимальному времени сна.

Таким образом, код ядра Linux вызывает другой блок кода на время временного среза. **Временной срез (timeslice)** связан как со временем между прерываниями планировщика, так и с длительностью времени, в течение которого процесс использует процессор. Если процесс израсходовал свой временной срез, процесс становится исчерпанным и неактивным. **Временная отметка (timerstamp)** - это абсолютное значение, определяющее, как долго процесс использует процессор. Процессор использует временную отметку для определения временного среза процесса, использующего процессор.

Например, представим, что Process A имеет временной срез длительностью 50 циклов таймера. Он использует процессор в течение 5 циклов и затем передает управление процессором другому процессу. Ядро использует временную отметку для определения того, что у Process A осталось 45 циклов временного среза.

Строки 2224-2225

Интерактивные процессы - это процессы, тратящие большинство отведенного им времени на ожидание ввода. Хорошим примером интерактивного процесса является контроллер клавиатуры - большинство времени он ожидает ввода, но, когда он случается, пользователь ожидает, что он получит высокий приоритет.

Интерактивные процессы, для которых кредит интерактивности больше 100 (значение по умолчанию), получают свое эффективное run_time, деленное на [sleep_avg/max_sleep_avg*MAX_BONUS (10)]*.

```
kernel/sched.c
2226
2227 spin_lock_irq(&rq->lock);
2228
2229 /*
223 0 * как только мы входим в приоритетное прерывание обслуживания,
2231  * сразу переходим к выбору новой задачи.
2232  */
2233  switch_count = &prev->nivcsw;
2234  if (prev->state && ! (preempt_count () & PREEMPT_ACTIVE) ) {
223 5 switch_count = &prev->nvcsw;
2236 if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
2237 unlikely(signal_pending(prev))))
2238 prev->state = TASK_RUNNING;
2239 else
2240 deactivate_task(prev, rq) ;
2241 }
```

[h] **Бонусы - это модификаторы планировщика для повышения приоритета.**

Строка 2227

Функция выполняет блокировку очереди выполнения, так как мы хотим ее изменить.

Строки 2233-2241

Если мы вошли в schedule () с предыдущим процессом, являющимся приоритетным прерыванием обслуживания, то мы покидаем предыдущий запущенный процесс, если ожидается сигнал. Это значит, что ядро приоритетно прерывает обслуживание обычного процесса быстрым завершением; соответственно код содержится в двух операторах unlikely () \ Если приоритетных прерываний обслуживания больше нет, мы убираем прервавшие процессы из очереди выполнения и продолжаем выбирать следующий процесс для выполнения.

```
kernel/sched.c
2243 cpu = smp_processor_id();
2244 if (unlikely(!rq->nr running)) {
2245 idle balance(cpu, rq) ;
2246 if (!rq->nr running) {
2247 next = rq->idle;
2248 rq->expired timestamp = 0;
2249 wake sleeping dependent(cpu, rq);
2250 goto switch_tasks;
2251 }
2252 }
2253
2254 array = rq->active;
2255 if (unlikely(!array->nr active)) {
2256 /*
2257 * Переключение между активным и истекшим массивами.
2258 */
2259 rq->active = rq->expired;
2260 rq->expired = array;
22 61 array = rq->active;
2262 rq->expired timestamp = 0;
22 63 rq->best expired prio = MAX PRIO;
2264 }
```

Строка 2243

Мы получаем идентификатор текущего процессора с помощью smp_processor id O.

[1] Более подробную информацию о функции unlikely см. в гл. 2, «Исследовательский инструментарий».

Строки 2244-2252

Если очередь выполнения не имеет в себе процессов, мы устанавливаем следующим процессом процесс простоя и сбрасываем временную отметку очереди выполнения в 0. На многопроцессорных системах мы сначала проверяем, выполняются ли на других процессорах процессы, которые можно выполнить на этом процессоре. В результате простаивающие процессы равномерно распределяются по всем процессорам системы. Только если не остается процессов, которые можно переместить на другие процессоры, следующим процессом в очереди выполнения мы устанавливаем процесс простоя и сбрасываем временную отметку истекших процессов.

Строки 2255-2264

Если очередь выполнения активных процессов пуста, мы переключаемся между активным и истекшим массивами указателей перед выбором нового процесса для выполнения.

```
kernel/sched.c
2266 idx = sched find first bit(array->bitmap);
2267 queue = array->queue + idx;
2268 next = list entry(queue->next, task t, run list);
2269
2270 if (dependent  sleeper(cpu, rq, next)) {
2271   next = rq->idle;
2272   goto switch__tasks;
2273
}
2274
2275 if (!rt_task(next) && next->activated > 0) {
2276   unsigned long long delta = now - next->timestamp;
2277
2278   if (next->activated == 1)
2279   delta = delta * (ON RUNQUEUE WEIGHT * 128 / 100) / 128;
2280
2281   array = next->array;
2282   dequeue task(next, array);
2283   recalc task  prio(next, next->timestamp + delta);
2284   enqueue_task(next, array);
2285 }
228 6 next->activated = 0;
```

Строки 2266-2268

Планировщик ищет процесс с наивысшим приоритетом для запуска с помощью sched_f ind_f irst_bit () и затем устанавливает queue в указатель на список,

хранящийся в массиве приоритетов в специальном месте; next инициализируется первым процессом из queue.

Строки 2270-2273

Если активируемые процессы зависят от спящих сестринских процессов, мы выбираем новый процесс для активации и переходим в switch_task для продолжения функции планировщика.

Предположим, что у нас есть Process A, порожденный Process B для чтения с устройства, и Process A ожидает завершения Process B, чтобы продолжить свое выполнение. Если планировщик выберет для активации Process A, этот блок кода dependent_ sleeper () определит, что Process A ожидает Process B, и выберет более новый процесс для активации.

Строки 2275-2285

Если атрибут активации процесса больше 0 и следующий процесс не является задачей реального времени, мы удаляем его из queue, пересчитываем его приоритет и снова помешаем его в очередь.

Строка 2286

Мы устанавливаем атрибут активации процесса в 0 и затем выполняем его.

```
kernel/sched.c
2287 switch tasks:
2288   prefetch(next);
2289   clear_tsk_need_resched(prev);
2290   RCU qsctr(task cpu(prev))++;
2291
2292   prev->sleep avg -= run time;
2293   if ((long)prev->sleep_avg <= 0) {
2294     prev->sleep avg = 0;
2295     if (! (HIGH CREDIT(prev) || LOW CREDIT(prev) ) )
2296       prev->interactive credit--;
2297     }
2298     prev->timestamp = now;
2299
2300   if (likely(prev != next)) {
2301     next->timestamp = now;
2302     rq->nr switches++;
2303          rq->curr = next;
2304          ++*switch_count;
2305
2306   prepare_arch_switch(rq, next);
2307prev = context_switch(rq, prev, next);
```

```
23 08                    barrier();
2309
2310        finish_task_switch(prev);
2311    } else
2312    spin_unlock_irq(&rq->lock);
2313
2314    reacquire_kernel_lock(current);
2315    preempt_enable_no_resched();
2316    if (test_thread_flag(TIF_NEED_RESCHED))
2317    goto need_resched;
2318 }
```

Строка 2288

Мы пытаемся поместить память структуры задачи нового процесса в кеш процессора первого уровня (L1). (См. более подробную информацию в include/linus/ prefetch, h.)

Строка 2290

Так как мы выполняем переключение контекста, нам нужно проинформировать об этом текущий процессор. Это позволяет многопроцессорному устройству получить доступ к разделяемому между несколькими процессорами ресурсу в эксклюзивном режиме. Этот процесс называется обновлением копирующего чтения. (Более подробную информацию см. в http://lse.sourceforge.net/locking/ rcupdate. html.)

Строки 2292-2298

Мы уменьшаем атрибут sleep_avg предыдущего процесса на количество времени, в течение которого он выполнялся, корректируя отрицательные значения. Если процесс не является ни интерактивным, ни неинтерактивным, а его значение интерактивности находится между наименьшим и наибольшим значениями, мы увеличиваем его значение интерактивности, так как он обладает низким значением среднего сна. Мы обновляем его временную отметку в значение текущего времени. Эта операция помогает планировщику следить за тем, сколько процессорного времени тратит текущий процесс, и оценить, сколько процессорного времени он потребует в будущем.

Строки 2300-2304

Если вы не выбрали тот же самый процесс, мы устанавливаем временную отметку процесса, увеличиваем счетчики очереди выполнения и устанавливаем в качестве текущего процесса новый процесс.

Строки 2306-2308

Эти строки описывают context_switch() на языке ассемблера. Задержимся на несколько абзацев до тех пор, пока мы погрузимся в объяснение переключения контекста в следующем разделе.

Строки 2314-2318

Мы перепоручаем блокировку ядра, включаем приоритетное прерывание обслуживания и смотрим, нужно ли нам производить немедленную перепланировку; если да, мы возвращаемся в начало schedule ().

Может случиться, что после выполнения context_switch () нам потребуется выполнить перепланировку. Возможно, что scheduler_tick() пометит новый процесс как нуждающийся в перепланировке или, когда включено приоритетное прерывание обслуживания, он будет помечен. Мы продолжаем процесс перепланировки (и затем переключаем контекст) до тех пор, пока не найдем первый процесс, не требующий перепланировки. Процесс, завершающий schedule (), становится новым процессом, выполняемым на данном процессоре.

7.1.2 Переключение контекста

Функция context_switch (), вызываемая из schedule () в /kernel/sched. c, выполняет специфическую для системы работу по переключению окружения памяти и состояния процесса. На абстрактном уровне context_switch переключает текущую задачу и следующую задачу. Функция context_switch () начинает выполнение следующей задачи и возвращает указатель на структуру задачи, которая выполнялась до вызова:

```
kernel/sched.c
1048 /*
1049 * context switch - переключение на новый ММ и новое
1050 * состояние регистра потока.
1051 */
1052 static inline
1053 task t * context switch(runqueue t *rq, task  t *prev, task t *next)
1054 {
1055     struct mm_struct  *mm =  next->mm;
1056     struct mm_struct  *oldmm = prev->active_mm;

1063     switch_irati(oldnim/ mm,  next) ;

1072    switch to(prev/ next, prev);
1073
1074    return prev;
1075 }
```

Здесь мы описываем две задачи context_switch: переключение виртуального отображения памяти и переключение структуры задачи/потока. За выполнение первой задачи отвечает switch_mm() с применением множества аппаратно-зависимых управляющих памятью структур и регистров:

```
/include/asm-i386/mmu_context.h
026  static  inline void switch_mm(struct ram_struct  *prev,
27         struct mm_struct   *next,
28         struct  task_struct   *tsk)
029  {
030   int cpu = smp processor id();
031
32    if (likely(prev != next)) {
33    /* остановка сброса ipis для предыдущего mm */
34    cpu clear(cpu, prev->cpu vm mask);
035  #ifdef CONFIG SMP
36    cpu tlbstate[cpu].state = TLBSTATE OK;
37    cpu tlbstate[cpu].active mm = next;
038  #endif
039   cpu set(cpu, next->cpu vm jriask) ;
040
41    /* Перезагрузка таблицы страниц */
42    load cr3(next->pgd);
043
44    /*
45    * загрузка LDT, если LDT отличается:
46    */
047  if (unlikely(prev->context.ldt != next->context.ldt))
048     load_LDT_nolock(&next->context, cpu);
49   }
50   #ifdef CONFIG SMP
051   else {
```

Строка 39

Связывает новую задачу с текущим процессором.

Строка 42

Код переключения контекста памяти использует аппаратный x86-регистр cr3, хранящий базовый адрес всех виртуальных операций для данного процесса. Новый глобальный описатель страницы загружается сюда из next->pgd.

Строка 47

Большинство процессов разделяют один и тот же LDT. Если процессу требуется другой LDT, он загружается сюда из новой структуры next->context.

Другая половина функции context_switch() находится в /kernel/sched. c, где вызывается макрос switch_to (), который вызывает С-функцию ____ switch_to (). Архитектурные ограничения независимы от архитектурных зависимостей для x86- и PPC-макросов switch_to ().

7.1.2.1 Отслеживание x86 switchJo()

Код x86 более компактен, чем код PPC. Далее приведен архитектурно-зависимый код для __ switch_to (); task_struct (не tread_struct) передается в _____ switch_to (). Код, обсуждаемый далее, - встроенный ассемблерный код для вызова С-функции __ switch_to() (строка 23) с соответствующей структуры task_struct в качестве параметра.

switch_to получает три указателя на задачи: prev, next и las t. Дополнительно существует текущий указатель.

Давайте объясним на самом верхнем уровне, что происходит, когда вызывается switch_to (), и как указатель на задачу изменяется после вызова switch_to ().

Рис. 7.2 демонстрирует три вызова switch_to () с использованием трех процессов А, В иС.

Мы хотим переключиться с А на В. Перед **первым вызовом** мы имеем:

- Текущий -* А
- Предыдущий -> А, следующий -► В

После **первого вызова:**

- Текущий -► В
- Последний -► А

Теперь мы хотим переключиться с В на С. Перед **вторым вызовом** мы имеем:

- Текущий -► В
- Предыдущий -♦ В, следующий -* С

После **второго вызова:**

- Текущий -♦ С
- Последний -+ В

После возвращения из второго вызова текущий указывает на задачу (С) и последний указывает на (В).

Далее метод продолжает работать с задачей (А), переключая ее еще раз, и т. д.

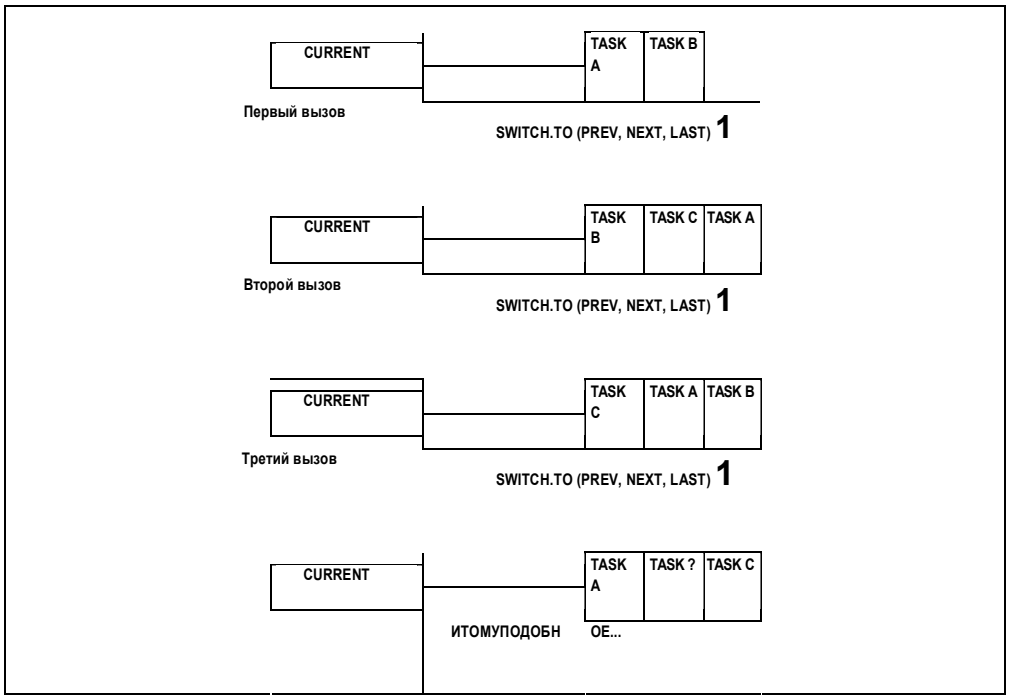


Рис. 7.2. Вызовы switch_to

Ассемблерная вставка функции switch_to() является отличным примером ассемблерной магии в ядре. Кроме этого, она является отличным примером С-расширения дсс. (См. в гл. 2, «Исследовательский инструментарий», обучающий пример по этой функции.) Теперь мы пройдемся по коду этого блока.

```
/include/asm-i386/system.h
012 extern struct task struct * FAST/CALL (  _switch_to(struct task_struct
*prev, struct task_struct *next));

015 tdefine                                          \
016 switch_to(prev,next,last) do {
017 unsigned long esi,edi; asm
018                       volatile
019     "pushl %%ebp\n\t"   ("pushfl
020     "movl %%esp,%0\n\t" \n\t"
021     "movl %5,%%esp\n\t" \
022     "movl $lf,%l\n\t"   /*                              сохранение ESP */       N
        "pushl %6\n\t"      /*                              восстановление ESP     */
                              /* сохранение EIP */     \
                              /* восстановление EIP   */
```

```
023    "jmp ___ switch_to\n"          \
23      -l:\t"              \
24      "popl %%ebp\n\t"          \
025    "popfl"              \
26      :"=m" (prev->thread.esp),"=m" (prev->thread.eip) ,  \
27      "=a" (last),"=S" (esi),"=D" (edi)      \
28      :"m" (next->thread.esp),"m" (next->thread.eip) ,  \
29      "2" (prev), "d" (next));        \
030 } while (0)
```

Строка 12

Макрос FASTCALL обращается к _____ attribute_regparm(3), принудительно передающему параметры в регистры вместо стека.

Строки 15-16

Конструкция do { } while (0) позволяет (помимо прочего) иметь макросу локальные переменные esi и edi. Помните, что это просто локальные переменные с такими именами.

Текущая задача и структура задачи

Как мы видели в ядре, в том случае, когда нам нужно получить или сохранить информацию в задачу (процесс), которая является *текущей* для данного процессора, мы используем глобальную переменную current для получения соответствующей структуры задачи. Например, current->pid хранит ID процесса. Linux предоставляет быстрый (и оригинальный) метод получения структуры текущей задачи.

Каждому процессу назначается 8 Кб последовательной памяти при создании. (В версии 2.6 при компиляции имеется возможность изменить это значение на 4 Кб вместо 8 Кб.) Этот блок в 8 Кб занят структурой задачи и стеком данного процесса. Во время создания процесса Linux помещает структуру задачи в нижний конец 8 Кб памяти, а указатель на стек устанавливается в верхний конец этого блока. Указатель на стек ядра (особенно для x86 и PPC rl) увеличивается по мере накопления данных в стеке. Так как эта область 8 Кб выровнена по страницам, ее начальный адрес (в шестнадцатеричной нотации) всегда кончается на 0x000 (множитель 4 Кб).

Как вы можете догадаться, оригинальным методом, с помощью которого Linux определяет структуру текущей задачи, является AND содержимого указателя на стек и 0xffff_f000. Последние версии Linux на PPC пошли немного далее и используют для хранения указателя на текущий процесс регистр общего назначения 2.

Строки 17и 30

Конструкция asm volatileO[1] включает блок встроенного ассемблерного кода, а ключевое слово volatile подразумевает, что компилятор не будет изменять (оптимизировать) данную функцию.

Строки 17-18

Помещает регистры flags и ebp в стек. (Обратите внимание, что мы используем стек, связанный с задачей prev.)

Строка 19

Эта строка сохраняет указатель текущей задачи esp в структуру задачи prev.

Строка 20

Перемещает указатель стека со следующей структуры задачи в esp текущего процессора.

ОБРАТИТЕ ВНИМАНИЕ: по определению мы просто желаем переключения контекста.

Теперь у нас есть новый стек ядра, и поэтому все ссылки к текущему процессу будут производиться к структуре задачи next.

Строка 21

Сохранение адреса возврата в prev в структуру задачи. Отсюда задача prev продолжает свое выполнение после повторного запуска.

Строка 22

Помещение адреса возврата [откуда мы возвращаемся из ___switch_to ()] в стек. Это eis из задачи next.

Строка 23

Переход в С-функцию__ switch_to () для обновления следующей информации:

- структуры следующего потока указателем на стек ядра;
- локального описателя хранилища потока для данного процессора;
- при необходимости f s и gs для prev и next;
- при необходимости регистров отладки;
- при необходимости битовых карт ввода-вывода.

Далее__ switch_to () возвращается и обновляет структуру задачи prev.

***• Более подробную информацию о volatile см. в гл. 2**

Строки 24-25

Извлечение базового указателя и регистра flags из стека ядра новой (следующей задачи).

Строки 26-29

Это параметры ввода-вывода для функции ассемблерной вставки. (См. разд. 2.4 для получения более подробной информации о *принудительной* передаче этих параметров.)

Строка 29

С помощью магии ассемблера prev возвращается в eax, являющийся третьим по счету параметром. Другими словами, входной параметр prev передается из макроса switch_to () как последний выходной параметр.

Так как switch_to() является макросом, он выполняется внутри кода, который вызывает context_switch (). Обычно он не возвращает функций.

Для простоты запомните, что switch_to () передает prev обратно в регистр eax, а выполнение продолжается в context_switch (), где следующей инструкцией является return prev (строка 1074 в kernel/sched.c). Это позволяет context_swi ten () передавать указатель обратно в последнюю запущенную задачу.

7.1.2.2 Отслеживание context_switch() на PPC

Код context_switch () для PPC делает немного больше работы для получения того же результата. В отличие от регистра сг3 на архитектуре x86, PPC использует функцию хеширования для указания на окружение контекста. Следующий код для switch_mm() касается этой функции, но в гл. 4, «Управление памятью», находится его более подробное описание.

Здесь приведена функция switch_mm(), которая, в свою очередь, вызывает функцию set_context ().

```
/include/asm-ppc/mmu context.h
155 static inline void switch mm(struct mm struct *prev,
       struct mm struct *next,struct task struct *tsk)
156   {
157    tsk->thread.pgdir = next->pgd;
158    get mmu context(next);
159    set context(next->context, next->pgd);
160   }
```

Строка 157

Глобальная директория страницы (регистр сегмента) для нового потока устанавливается указывающей на указатель next->pgd.

Строка 158

Поле context структуры mm_struct (next->context) передается в switch__iran() и обновляется значением соответствующего контекста. Эта информация получается из глобальной ссылки на переменную context_map[ ], которая содержит набор полей битовых карт.

Строка 159

Это вызов ассемблерной функции set_context. После самого кода приводится обсуждение этой функции. Во время выполнения инструкции Ыг на строке 1468 код возвращается в функцию switch_mm.

```
/arch/ppc/kernel/head.S
1437   GLOBAL(set context)
1438  mulli r3,r3,897  /* умножение контекста на фактор ассимметрии */
1439  rlwinm  r3,r3,4,8,27    /* VSID = (context & Oxfffff) « 4 */
1440  addis   r3,r3,0x6000     /* установка битов Ks, Ku */
1441  li  r0,NUM_USER_SEGMENTS
1442  mtctr  rO

1457  3:  isync

1461  mtsrin r3,r4
1462  addi  r3,r3,0x111          /* следующий VSID */
1463  rlwinm  r3,r3,0,8,3        /* очистка переполнения поля VSID */
1464  addis   r4,r4,0x1000      /* адрес следующего сегмента */
1465  bdnz   3b
1466  sync
1467  isync
1468  blr
```

Строки 1437-1440

Поле context структуры mm_struct (next->context) передается в set_ context () через г3 и устанавливает хеш-функцию для сегментации PPC.

Строки 1461-1465

Поле структуры mm_struct (next->pgd) передается в set_context () через г4 и указывает на регистр сегмента.

Сегментация - это основа управления памятью на PPC (см. гл. 4). Во время возвращения из set_context (), инициализируется mm__struct для соответствующей области памяти и выполняется возвращение в switch_mm ().

7.1.2.3 Отслеживание прохождения switch_to() на PPC

Результат реализации switch_to () на PPC не обязательно идентичен вызову x86; он берет указатели на current и next задачи и возвращает указатель на предыдущую выполнявшуюся задачу:

```
include/asm-ppc/system.h
88 extern struct task_struct * _ switch_to(struct task_struct *,
89   struct task struct *);
90 #define switch to(prev, next, last)
((last) =  _ switch_to((prev), (next)))
91
92 struct thread struct;
93 extern struct task_struct *_switch(struct thread_struct *prev,
94         struct thread_struct *next);
```

В строке 88 __switch_to () получает параметр типа task_struct и на строке 93 получает в качестве параметра thread_struct. Это необходимо для того, чтобы вхождение в task_struct содержало архитектурно-зависимую информацию о регистрах процессора, необходимую данному процессу. Теперь давайте рассмотрим реализацию __switch_to ().

```
/arch/ppc/kernel/process.c
200  struct task_struct * _ switch_to(struct task__struct *prev,
                                      struct task struct *new)
201  {
202   struct thread__struct *new_thread, *old_thread;
203   unsigned long s;
204   struct task struct *last;
205   local_irq__save(s) ;

247   new thread = &new->thread;
248   old thread = &current->thread;
249   last = „switchtold^hread, new  thread) ;
250   local irq restore(s);
251   return last;
252  }
```

Строка 205

Отключение прерываний перед переключением контекста.

Строки 247-248

Выполнение еще продолжается в контексте старого потока, а указатель на структуру потока передается в функцию _switch ().

Строка 249

Ассемблерная функция _switch() вызывается для выполнения работы по переключению двух структур потоков (см. следующий раздел).

Строка 250

Включение прерываний после переключения контекста.

Для лучшего понимания того, что нужно обменять в потоке PPC, нам нужно рассмотреть структуру thread_struct, передаваемую в строке 249.

Вы можете вспомнить из описания переключения контекста x86, что переключение официально не происходит до тех пор, пока мы не указываем на новый стек ядра. Это происходит в _swi tch ().

Отслеживание кода PPC для switch 0

По соглашению параметры С-функции PPC (слева направо) хранятся в г3, г4, г5,..., г12. При вхождении в switch () г3 указывает на thread_struct для текущей задачи, а г4 указывает на thread_struct для новой задачи:

```
/arch/ppc/kernel/entry.S
437   GLOBAL( switch)
438    stwu  rl,-INT FRAME SIZE(rl)
43 9   mflr  rO
440    stw   rO,INT FRAME SIZE+4(rl)
441    /* r3-rl2 сохраняются вызывающим -- Cort */
442    SAVE NVGPRS(r1)
443    stw  rO,_NIP(rl)  /* возвращение в вызов переключателя */
444    mfmsr    rll

458   1:  stw  rll, MSR(rl)
459    mfcr  rlO
460    stw  rlO, CCR(rl)
461    stw rl,KSP(r3)   /* Установка старого указателя на стек */
462
463    tophys(r0,r4)
464    CLR TOP32(rO)
465    mtspr SPRG3,rO /* обновление физического адреса текущего THREAD */
466    lwz   rl,KSP(r4)   /* Загрузка нового указателя на стек */
467    /* сохранение старого текущего "последнего"
          для возвращаемого значения */
468    mr  r3,r2
```

```
469  addi  r2,r4,-THREAD  /* Обновление текущего */

478   lwz  rO,_CCR(rl)
479   mtcrf  OxFF,rO
480   REST_NVGPRS(rl)
481
482   lwz r4, NIP(rl) /* Возвращение в вызывающий  switch код
                          в новой задаче */
483   mtlr r4
484   addi  rl,rl,INT FRAME SIZE
485   blr
```

Побайтовый механизм замены предыдущей thread_struct новой мы оставляем на ваше самостоятельное изучение. Это не сложно, а основы использования rl, r2, г3, SPRG3 и г4 вы можете увидеть в _switch ().

Строки 438-460

Окружение сохраняется в текущий стек с сохранением указателя на текущий стек вг1.

Строка 461

Далее окружение полностью сохраняется в текущую thread_struct, передающуюся через указатель в г3.

Строки 463-465

SPRG3 обновляется для указания на структуру потока для новой задачи.

Строка 466

KSP - это отступ в структуре задачи (г4) в указателе на новый стек ядра задачи. Указатель на стек rl обновляется этим значением. (В этой точке выполняется переключение контекста PPC.)

Строка 468

Текущий указатель на предыдущую задачу возвращается из _switch() в г3. Он представляет последнюю задачу.

Строка 469

Текущий указатель (г2) обновляется указателем на новую структуру задачи (г4).

Строки 478-486

Восстановление оставшегося окружения из нового стека и возвращение в вызывающую функцию с предыдущей структурой задачи в г3.

На этом мы заканчиваем объяснение context_switch (). В этой точке процессор обменял два процесса: prev и next, вызванные context_switch в schedule ().

kernel/sched.c
1709 prev = context_switch(rq, prev, next);

Теперь prev указывает на процесс, который мы только что переключили, а next указывает на текущий процесс.

Теперь, когда мы обсудили планирование процессов в ядре Linux, мы можем разобраться в том, как планируются задачи, а именно, что вызывает schedule () и как процесс передает процессор другому процессу.

7.1.3 Занятие процессора

Процессы могут занимать процессор простым вызовом функции schedule (). Она обычно используется в коде ядра и драйвером устройства, которое хочет заснуть или дождаться поступления сигнала[1]. Другие задачи тоже постоянно хотят использовать процессор, и системный таймер должен сообщать им, когда они смогут выполниться. Ядро Linux периодически захватывает процессор, при этом активные процессы останавливаются, и затем выполняет несколько зависящих от времени задач. Одна из этих задач, scheduler_tick (), позволяет ядру заставить процесс приостановиться. Если процесс выполняется слишком долго, ядро не возвращает этому процессу управление и вместо него выбирает другой процесс. Теперь мы изучим, как scheduler_tick () определяет текущий процесс, который должен занять процессор.

```
kernel/sched.c
1981 void scheduler tick(int user ticks, int sys ticks)
1982 {
1983   int cpu = smp processor id();
1984   struct cpu_usage_stat *cpustat = &kstat_this_cpu.cpustat;
1985   runqueue t *rq = this_rq();
1986   task t *p = current;
1987
1988   rq->timestamp last tick = sched clock();
1989
1990   if (rcu pending(cpu))
1991    rcu_check_callbacks(cpu, user_ticks);
```

Строки 1981-1986

Этот блок кода инициализирует структуры данных, необходимые функции scheduler_tick(); cpu, cpu_usage_stat и rq получают значения иденти-

[1] Соглашение Linux утверждает, что вы никогда не должны вызывать schedule во время циклической блокировки, так как это может завести систему в тупик. Это действительно хороший совет!

фикатора процессора, статус процессора и очередь выполнения для текущего процессора; р - это указатель на текущий выполняемый на ери процесс.

Строка 1988

Последний тик очереди выполнения устанавливается в текущее время в наносекундах.

Строки 1990-1991

На SMP-системах нам нужно проверить наличие требующих выполнения просроченных обновлений чтения-записи (RCU). Если это так, мы выполняем их с помощью rcu__check_callback ().

```
kernel/sched.c
1993    /* обратите внимание: контекст irq этого таймера также
           должен учитываться циклом for */
1994    if (hardirq count() - HARDIRQ OFFSET) {
1995      cpustat->irq += sys ticks;
1996      sys ticks = 0;
1997    } else if (softirq count()) {
1998      cpustat->softirq += sys ticks;
1999      sys_ticks = 0;
2000    }
2001
2002          if (p == rq->idle) {
2 003  if (atomic read(&rq->nr iowait) > 0)
2 004     cpustat->iowait += sys_ticks;
2005      else
2006        cpustat->idle += sys ticks;
2007      if (wake priority sleeper(rq))
2008        goto out;
2009      rebalance tick(cpu, rq, IDLE);
2010      return;
2011    }
2012    if (TASK NICE(p) > 0)
2013      cpustat->nice += user_ticks;
2014    else
2 015           cpustat->user  += user_ticks;
2 016          cpustat->system +=  sys_ticks;
```

Строки 1994-2000

cpustat следит за статистикой ядра, и мы обновляем статистику об аппаратных и программных прерываниях по количеству наступивших системных тиков.

Строки 2002-2011

Если текущего выполняемого процесса нет, мы автоматически проверяем наличие процессов, ожидающих ввода-вывода. Если это так, статистика ввода-вывода

процессора увеличивается; в противном случае увеличивается статистика ожидающего процессора. В однопроцессорных системах rebalance_tick () не делает ничего, а на многопроцессорных системах rebalance_tick() старается сбалансированно загрузить текущий процессор, так как он простаивает.

Строки 2012-2016

В этом блоке кода собирается дополнительная статистика о процессоре. Если текущий процесс был niced, мы увеличиваем счетчик nice для процессора; в противном случае увеличивается пользовательский счетчик тиков. И наконец, мы увеличиваем системный счетчик тиков процессора.

```
kernel/sched.c
2019    if (p->array != rq->active) {
2020      set tsk need resched(p);
2021      goto out;
2022    }
2023    spin lock(&rq->lock);
```

Строка 2019-2022

Здесь мы видим, почему мы сохраняем указатель на массив приоритетов в task_struct процесса. Планировщик проверяет текущий процесс и смотрит, не является ли он больше активным. Если процесс завершился, планировщик устанавливает флаг перепланировки процесса и переходит в конец функции scheduler., tick(). В этой точке (строки 2092-2093) планировщик пытается сбалансированно загрузить процессор, так как активных задач нет. Этот случай наступает, когда планировщик перехватывает управление процессором перед тем, как текущий процесс сможет себя перепланировать или очиститься после удачного выполнения.

Строка 2023

В этой точке мы знаем, что текущий процесс был запущен и, не завершился и существует. Теперь планировщик хочет передать управление процессором другому процессу; первое, что он должен сделать, - это заблокировать очередь выполнения.

```
kernel/sched.c
2024    /*
2 025    * Задача, выполняющаяся во время этого тика, обновляет счетчик
2026    * временного среза. Обратите внимание: мы не обновляем приоритета
2027    * потока до тех пор, пока он не засыпает или не расходует свой
2 028   * временной срез. Это позволяет интерактивным задачам использовать
2 029    * свои временные срезы на наивысшем уровне приоритета levels.
2030    */
2031    if (unlikely(rt_task(p))) {
```

```
2032                        /*
2 033    * RR-задаче требуется специальная форма управления временным
         * срезом.
2034     * FIFO-задача не имеет временных срезов.
2035     */
2036     if ((p->policy == SCHED_RR) && !--p->time_slice) {
2037       p->time slice = task timeslice(p);
2038       p->first_time_slice = 0;
2039       set tsk need resched(p);
2040
2041       /* помещение в конец очереди */
2042       dequeue task(p, rq->active);
2043       enqueue_task(p, rq->active);
2044     }
2045     goto out unlock;
2046  }
```

Строки 2031-2046

Простейший случай для планировщика наступает, когда текущий процесс является задачей реального времени. Задачи реального времени всегда имеют наивысший приоритет по сравнению с другими задачами. Если задача является FIFO и запущена, она продолжает свои операции, а мы переходим в конец функции и снимаем блокировку очереди выполнения. Если текущий процесс является циклической задачей реального времени, мы уменьшаем его временной срез. Если у задачи не осталось временного среза, наступает время перепланировать следующую циклическую задачу реального времени. Текущая задача получает новый временной срез, рассчитываемый task_timeslice (). Далее задача сбрасывает свой первый временной срез. Затем задача помечается как требующая перепланировки и, наконец, помещается в конец списка циклических задач реального времени удалением ее из активного массива очереди выполнения и помещением в его конец. После этого планировщик переходит в конец функции и снимает блокировку с очереди выполнения.

```
kernel/sched.c
2047     if (!—p->time slice) {
2048       dequeue task(p, rq->active);
2049       set tsk need resched(p);
2050       p->prio = effective prio(p);
2051       p->time slice = task timeslice(p);
2052       p->first time slice = 0;
2053
2054       if (!rq->expired timestamp)
2055         rq->expired_timestamp = jiffies;
```

```
2056     if (!TASK INTERACTIVE(p) || EXPIRED STARVING(rq) ) {
2 057      enqueue task(p, rq->expired) ;
2 058      if (p->static_prio < rq->best_expired_prio)
2059         rq->best expired  prio = p->static prio;
2060      } else
2061        enqueue task(p, rq->active);
2062    } else {
```

Строки 2047-2061

В этой точке планировщик знает, что текущий процесс не является процессом реального времени. Он увеличивает временной срез процесса а также, в этом же блоке, временной срез, который истек и достиг 0. Планировщик удаляет задачу из активного массива и устанавливает флаг перепланировки процесса. Приоритет задачи пересчитывается, а временной срез сбрасывается. Обе эти операции производятся с учетом предыдущей активности процесса[1]. Если временная отметка истекшей очереди ожидания достигла 0, что обычно происходит, когда в массиве активной очереди выполнения не остается процессов, мы присваеваем ему значение текущего момента.

Моменты

Моменты (jiffies) - это 32-битовые переменные, отсчитывающие количество тиков с момента загрузки системы. На процессоре с частотой 100 Гц эти переменные переполнятся и будут обнулены примерно через 497 дней. Макрос в строке 20 представляет собой метод для доступа к этому значению в качестве и64. Кроме этого, в include/jiffies.h существует макрос для определения переполнения моментов.

include/linux/jiffies.h
```
017     extern unsigned long volatile  jiffies;
020     u64  get_jiffies_64(void);
```

Обычно мы поощряем интерактивные задачи, заменяя их в активном массиве приоритетов очереди выполнения; случай else в строке 2060. Тем не менее мы не хотим тормозить истекшие задачи. Для определения того, не ожидает ли истекшая задача передачи ей процессора слишком долго, мы используем EXPIRED_STARVING () (см. EXPIRED_STARVING в строке 1968).

Функция возвращает true, если первая истекшая задача ожидает «неоправданно» долгое время или если массив истекших задач содержит задачу, имеющую более высокий приоритет, чем у текущего процесса. Неоправданность ожидания зависит от загрузки

[к] См. ef f ective__prio () и task_timeslice ().

и количества обменов массивов активных и истекших задач и увеличивается с ростом количества запущенных задач.

Если задача не является интерактивной или если истекшая задача тормозится, планировщик берет текущий процесс и включает его в массив приоритетов истекшей очереди выполнения. Если статический приоритет текущего процесса больше, чем наибольший приоритет задачи из истекшей очереди выполнения, мы обновляем очередь выполнения для отражения факта, что приоритет истекшего массива увеличился. [Помните, что задачи с большим приоритетом в Linux имеют меньшие номера и поэтому в коде производится проверка (<).]

```
kernel/sched.c
2062    } else {
2063      /*
2 064      * Предотвращение слишком долгих временных срезов, позволяющих
2065     * монополизировать процессор. Мы делаем это, разбивая временные
2066      * срезы на меньшие порции.
2 068      * ОБРАТИТЕ ВНИМАНИЕ: это не значит, что временные срезы задачи
2069     * истекли или были удалены другим путем, их обслуживание просто
2070      * приоритетно прерываются другой задачей с эквивалентным
2071      * приоритетом.(Задача с большим приоритетом может приоритетно
2072     * прервать выполнение данной задачи.) Мы переносим эту задачу в
2073      * конец списка уровней приоритетов, представляющий собой
2 074      * карусельную структуру задач с одинаковым приоритетом.
2075      *
2076      * Применяется только к интерактивным задачам
2 077                      * с диапазоном не менее TIMESLICE GRANULARITY.
2078      */
2079      if (TASK INTERACTIVE(p) && !((task timeslice(p) -
2080        p->time slice) % TIMESLICE GRANULARITY(p)) &&
2081         (p->time slice >= TIMESLICE GRANULARITY(p) ) &&
2082         (p->array == rq->active)) {
2083
2 084                                          dequeue_task(p, rq->active);
2085        set tsk need resched(p);
2086        p->prio = effective prio(p); 2
087         enqueue_task(p, rq->active);
2088      }
2089    }
2 090                                                        out unlock:
2 091                                                spin unlock(&rq->lock);
2092  out:
2093    rebalance tick(cpu, rq, NOT IDLE);
2094  }
```

Строки 2079-2089

Последний случай перед вызовом планировщика - это когда текущий процесс является запущенным и у него еще остался временной срез. Планировщику нужно удостовериться, что процесс с большим временным срезом не заблокирует процессор. Если задача интерактивна, имеет временной срез большей длины, чем TIMESLICE_GRANUARITY, и активна, планировщик убирает ее из активной очереди. После этого устанавливается флаг перепланировки задачи, пересчитывается ее приоритет и она помещается обратно в активный массив очереди выполнения. Это позволяет быть уверенным, что процесс с определенным приоритетом и с большим временным срезом не застопорит другой процесс с аналогичным приоритетом.

Строки 2090-2094

Планировщик завершает перераспределение очереди выполнения и разблокирует ее; если выполнение происходит на SMP-системе, производится попытка сбалансированной нагрузки.

Связь того, как процесс помечается для перепланировки с помощью scheduler_tick () и как процесс планируется с помощью schedule (), иллюстрирует работу планировщика в ядре Linux версии 2.6. Теперь мы углубимся в детали того, что называется в планировщике «приоритетом».

7.1.3.1 Динамический расчет приоритета

В предыдущем подразделе мы касались специфики динамического расчета приоритетов задач. Приоритет задачи основан на ее поведении в прошлом, а также на определенном пользователем значении nice. Функцией, динамически определяющей новый приоритет задачи, является recalc__task__prio ():

kernel/sched.c

```
3 81        static void recalc_task_prio(task_t   *p,   unsigned long  long now)
382                                     {
3 83    unsigned long long     sleep_time = now - p->timestamp;
3 84          unsigned long sleep_time;
385
3 86    if (   sleep_time > NS_MAX_SLEEP_AVG)
3 87                  sleep_time = NS_MAX_SLEEP_AVG;
388                                 else
3 89                   sleep_time = (unsigned long)    sleep_time;
390
391    if (likely(sleep_time > 0)) {
392       /*
3 93       * Слишком долго спавшая задача категоризируется как
3 94 * простаивающая и получает статус интерактивной для того, чтобы
```

```
3 95    * оставаться в активном состоянии, предотвращать блокировки
396      * процессора и простой других процессов.
397      */
398      if (p->ram && p->activated != -1 &&
399        sleep  time > INTERACTIVE SLEEP(p) ) {
400         p->sleep avg = JIFFIES TO NS(MAX SLEEP AVG -
401              AVG TIMESLICE);
402         if (!HIGH CREDIT(p))
403           p->interactive credit++;
404       } else {
405         /*
406         * Чем меньше sleep avg задачи,
407         * тем чаще увеличивается ее время сна.
408         */
409         sleep_time *= (MAX_BONUS - CURRENT_BONUS(p)) ? : 1; 410
411         /*
412         * Задача с низким значением интерактивности ограничивается
413         * одним временным срезом в размере бонуса sleep avg.
414         */
415         if (LOW CREDIT(p) &&
416          sleep time > JIFFIES TO NS(task timeslice(p)))
417           sleep time = JIFFIES TO NS(task timeslice(p)) ;
418
419         /*
420         * Задачи без high_credit, пробуждающиеся от беспрерывного
421         * сна, ограничиваются в росте sleep avg, так как они
422         * смогут заблокировать процессор, ожидая ввода-вывода
423         */
424         if (p->activated == -1 && !HIGH CREDIT(p) && p->ram) {
425           if (p->sleep avg >= INTERACTIVE SLEEP(p))
426             sleep time = 0;
427           else if (p->sleep avg + sleep time >=
428               INTERACTIVE SLEEP(p)) {
429            p->sleep avg = INTERACTIVE SLEEP(p);
43 0            sleep time = 0;
431           }
432         }
433
434                                    /*
43 5            * Этот код награждает бонусом интерактивные задачи.
436         *
437         * Это награждение работает с помощью обновления значения
438         * 'average sleep time' на основе ->timestamp. Чем больше 43 9   *
времени задача тратит на сон, тем больше average -440       * и выше
получаемый приоритет.
```

```
441         */
442         p->sleep avg += sleep time;
443
444         if (p->sleep avg > NS MAX SLEEP AVG) {
445           p->sleep avg = NS MAX SLEEP AVG;
446           if (!HIGH CREDIT(p))
447           p->interactive credit++;
448         }
449       }
450     }
452
452    p->prio = effective prio(p);
453 }
```

Строки 386-389

На основе времени now мы рассчитываем длительность времени, в течение которого будет спать процесс p, и назначаем sleep_time максимальное значение NS_MAX_SLEEP_AVG. (По умолчанию NS__MAX_SLEEP_AVG равняется 10 мс.)

Строки 391-404

Если процесс p спит, мы сначала проверяем, спит ли он достаточно для того, чтобы задачу можно было классифицировать как интерактивную. Если это так, мы изменяем среднее время сна процесса с помощью sleep_tiirie->INTERACTIVE_ SLEEP (p), а если p классифицируется не как интерактивный процесс, мы увеличиваем interactive_credit для p.

Строки 405-410

Задача с меньшим временем сна получает большее время сна.

Строки 411-418

Если задача интенсивно нагружает процессор и она классифицирована не как интерактивная, мы ограничиваем процесс как минимум одним временным срезом, равным бонусу в виде среднего времени сна.

Строки 419-432

Задачи, еще не классифицированные как интерактивные (без HIGH_CREDIT), разбуженные от беспрерывного сна, и получают среднее время сна INTERACTIVE ().

Строки 434-450

Мы добавляем новое рассчитанное sleep_time к среднему времени сна процесса, проверяя, чтобы оно не превысило NS_MAX_SLEEP_AVG. Если процессы не признаны интерактивными, но спали максимальное время, мы увеличиваем их значение интерактивности.

Строка 452

Наконец, приоритет устанавливается с помощью ef fective_prio () с учетом нового рассчитанного поля sleep_avg для p. Это делается с помощью масштабирования среднего времени от 0 до MAXSLEEPAVG в диапазоне от -5 до +5. Поэтому процесс со статическим приоритетом 70 может иметь динамический приоритет в пределах от 65 до 85, в зависимости от своего поведения в прошлом.

И еще одна финальная черта: процесс, не являющийся процессом реального времени, попадает в диапазон от 101 до 140. Процессы, обрабатываемые с большим приоритетом, попадают в диапазон 105 и ниже, хотя и не могут перевалить за барьер процессов реального времени. Поэтому интерактивный процесс с большим приоритетом может никогда не иметь динамического приоритета ниже 101. (В конфигурации по умолчанию процессы реального времени покрывают диапазон от 0 до 100.)

7.1.3.2 Вывод из активного состояния

Мы уже обсуждали, как задача попадает в планировщик после вызова fork и как задача перемещается из массивов активных приоритетов в истекшие в очереди выполнения на процессоре. Но когда же задача удаляется из очереди выполнения?

Задача может быть удалена из очереди выполнения двумя основными способами:

- задача прерывается ядром и изменяет состояние на незапущенное, после чего в задачу перестают поступать сигналы (см. строку 2240 в kernel/sched. c);
- на SMP-машинах задача может быть удалена из очереди выполнения и помещена в другую очередь выполнения (см. строку 3384 в kernel /sched. c).

Первый случай происходит, когда schedule () вызывается после того, как процесс погружается в сон в очереди ожидания. Задача помечает себя как невыполняемая (TASK_INTERRUPTIBLE, TASK_UNINTERRUPTIBLE, TASK_STOPPED и т. д.), и ядро перестает предоставлять ей доступ к процессору, удаляя ее из очереди выполнения.

Случай, когда процесс перемещается в другую очередь выполнения, обрабатывается в SMP-разделе ядра Linux, который мы здесь не рассматриваем.

Теперь мы проследим, как процесс удаляется из очереди выполнения с помощью deactivate_task ().

```
kernel/sched.c
507 static void deactivate task(struct task struct *p, runqueue t *rq)
508 {
509   rq->nr running—;
510   if (p->state == TASK UNINTERRUPTIBLE)
511     rq->nr_uninterruptible++;
512   dequeue_task(p, p->array);
```

```
513    p->array = NULL;
514  }
```

Строка 509

Сначала планировщик уменьшает количество запущенных процессов, так как p больше не является запущенной.

Строки 510-511

Если задача непрерывна, мы увеличиваем количество непрерывных задач в очереди выполнения. Соответствующие операции декрементации вызываются, когда непрерывный процесс просыпается [см. строку 824 в kernel/sched.c в функции try_to__wake___up ()].

Строки 512-513

Статистика нашей очереди выполнения обновляется, как только мы удаляем процесс из очереди выполнения. Ядро использует поле p->array для проверки того, является ли процесс запущенным и находится ли он в очереди выполнения. Так как больше ни одно из этих условий не выполняется, мы устанавливаем его в NULL.

Необходимо сделать еще кое-что с очередью выполнения; давайте рассмотрим специфику dequeue__task ().

```
kerne1/sched.c
3 03 static void dequeue task(struct task struct *p, prio array  t
*array)
304  {
305    array->nr_active—;
3 06    list del(&p->run list);
307     if  (list_empty(array->queue  + p->prio))
308           clear   bit (p->prio/  array->bitmap) ;
309  }
```

Строка 305

Мы изменяем количество активных задач в массиве приоритетов, в котором находится процесс p; не важно, активный ли это массив или истекший.

Строки 306-308

Мы удаляем процесс из списка процессов в массиве приоритетов с приоритетом p. Если результирующий список будет пустым, нам нужно очистить бит в битовой карте массива приоритетов для того, чтобы указать, что в p->prio () не осталось процессов.

list_.de 1 () выполняет всю работу по удалению за один шаг, так как p-> run_list представляет собой структуру list_head и поэтому содержит указатели на предыдущее и следующее вхождения в списке.

Мы достигли точки, где процесс удаляется из очереди выполнения и становится полностью неактивным. Если этот процесс находится в состоянии TASK_INTERRUPTIBLE или TASK_UNINTERRUPTIBLE, он может быть разбужен и помещен обратно в очередь выполнения. Если процесс находится в состоянии TASK_STOPPED, TASK_ZOMBIE или TASK_DEAD, все его структуры удаляются и сбрасываются.

7.2 Приоритетное прерывание обслуживания

Приоритетное прерывание обслуживания - это переключение одной задачи на другую. Мы упоминали как schedule () и scheduler_tick() решают, на какую задачу переключиться далее, но мы еще не описали, как Linux решает, когда выполнить переключение. В ядре 2.6 представлено приоритетное прерывание обслуживания, означающее, что как программы пользовательского пространства, так и программы пространства ядра могут быть переключены в одно и то же время. Так как в Linux 2.6 приоритетное прерывание обслуживания является стандартным, мы опишем, как работает приоритетное прерывание обслуживания для ядра и пользовательских задач в Linux.

7.2.1 Явное приоритетное прерывание обслуживания в ядре

Проще всего понять явное приоритетное прерывание обслуживания в ядре. Оно происходит в пространстве ядра, когда код ядра вызывает schedule (). Код ядра может вызывать schedule () двумя способами: либо с помощью прямого вызова schedule О, либо с помощью блокировки.

Когда происходит явное приоритетное прерывание обслуживания в ядре, как, например, в драйвере устройства, ожидающем в wait_queue, управление просто передается планировщику и для выполнения выбирается новая задача.

7.2.2 Неявное пользовательское приоритетное прерывание обслуживания

Когда ядро завершает обработку задачи пользовательского пространства и готово передать управление задаче пользовательского пространства, оно сперва проверяет, какой задаче можно передать управление. Это не должна быть задача пользовательского пространства, которая передает управление ядру. Например, если задача А порождает системный вызов, после завершения системного вызова ядро передает управление системой задаче В.

Каждая задача в системе имеет флаг «необходимости перепланировки», который устанавливается, когда задачу следует перепланировать:

```
include/linux/sched.h
988 static inline void set tsk need resched(struct task struct *tsk)
989 {
990    set_tsk_thread_flag(tsk,TIFJSTEED_RESCHED) ;
991

}
992
993 static inline void clear tsk need resched(struct task struct *tsk)
994 {
995   clear_tsk_thread_flag(tsk,TIF_NEED_RESCHED) ;  }

1003 static inline int need__resched(void)
1004 {
1005   return unlikely (test_thread_f lag (TIF_NEED_RESCHED) ) ;
1006 }
```

Строки 988-996

Интерфейсы set__tsk_need_resched и clear_tsk_need_resched позволяют устанавливать архитектурно-специфический флаг TIF_NEED_RESCHED.

Строки 1003-1006

need_resched проверяет, установлен ли TIF_NEED_RESCHED во флаге текущего процесса.

Когда ядро возвращается в пользовательское пространство, оно выбирает процесс, которому нужно передать управление, как описано в schedule () и scheduler_ tick(). Тогда как scheduler_tick() может пометить задачу как требующую перепланировки, только schedule () может манипулировать этим знанием; schedule () циклически выбирает новую задачу для выполнения до тех пор, пока новая выбранная задача не потребует перепланировки. После завершения schedule () новая задача получает управление процессором.

Поэтому, пока процесс запущен, системный таймер порождает прерывания, запускающие scheduler_tick(); scheduler_tick() может помечать задачу как требующую перепланировки и перемешает ее в массив истекших. При завершении операций ядра за scheduler_tick() могут следовать другие прерывания и ядро будет передавать управление процессором, а schedule () вызываться для выбора следующей запускаемой задачи. Поэтому scheduler_tick() отмечает процессы и сортирует очереди, но schedule () выбирает следующую задачу и передает управление процессором.

7.2.2 Неявное приоритетное прерывание обслуживания ядра

В Linux 2.6 реализовано новое неявное приоритетное прерывание обслуживания ядра. Когда задача ядра получает управление процессором, его обслуживание может приори-

тетно прерываться только другой задачей ядра, если она не содержит никаких блокировок. Каждая задача имеет поле preempt_count, помечающее задачу как приоритетно прерываемую. Счетчик увеличивается каждый раз, когда задача блокируется, и уменьшается, когда разблокируется. Функция schedule () отключает приоритетное прерывание обслуживания, когда определяет, какую задачу запустить следующей.

Существует две возможности для неявного приоритетного прерывания обслуживания ядра: либо код ядра вызывается из блока кода, для которого отключено приоритетное прерывание обслуживания, или процесс возвращается в код ядра из прерывания. Если управление возвращается в пространство ядра из прерывания, прерывание вызывает schedule () и новая задача вызывается тем же, описанным выше способом.

Если код ядра вызывается из блока кода с отключенным приоритетным прерыванием обслуживания, включение приоритетного прерывания обслуживания может привести к приоритетному прерыванию обслуживания текущей задачи:

```
include/linux/preempt.h
46 #define preempt  enable() \
47 do { \
48    preempt_enable_no_resched(); \
49    preempt_check__resched() ; \
50 } while (0)
```

Строки 46-50

preempt__enable () вызывает preempt_enable_no_resched(), уменьшающую preempt_count текущей задачи на единицу, и затем вызывает preempt_check_resched ().

```
include/linux/preempt.h
40 #define preempt check resched() \
41 do { \
42   if (unlikely(test thread flag(TIF NEED RESCHED) ) ) \
43     preempt schedule(); \
44 } while (0)
```

Строки 40-44

Preempt_check_resched () смотрит, помечена ли текущая задача как требующая перепланировки; если это так, вызывается preempt_schedule ().

```
kernel/sched.c
2328 asmlinkage void   sched preempt schedule(void)
2329 {
```

```
2330            struct thread_info *ti = current_thread_info();
2331
2332    /*
2333     * Если отключено ненулевое количество preempt_count или
233 4 * прерываний, мы не хотим приоритетно прерывать обслуживание
233 5           * текущей задачи. Просто возвращаемся. */
2336      if (unlikely(ti->preempt_count || irqs_disabled() ) )
233 7                           return;
2338
233 9                       need_resched:
2340    ti->preempt_count = PREEMPT_ACTIVE;
2341    schedule();
2342    ti->preempt_count = 0;
2343
2344    /* мы можем потерять возможность приоритетного прерывания
           обслуживания между планировщиком и текущим моментом */
2345     barrier();
2346     if (unlikely (test_thread_flag(TIF_NEED_RESCHED) ) )
2347       goto need_resched;
2348  }
```

Строки 2336-2337

Если текущая задача все еще имеет положительное значение preempt_count, как при рекурсивной команде preempt_disable (), или если для текущей задачи отключены прерывания, мы возвращаем управление процессором текущей задаче.

Строки 2340-2347

Текущая задача не имеет блокировок, так как preempt_count равно 0, а IRQ включены. Поэтому мы устанавливаем preempt_count текущей задачи для обозначения входящего приоритетного прерывания обслуживания и вызываем schedule (), выбирающую другую задачу.

Если задача запущена из кода, требующего перепланировки, ядру нужно удостовериться, что у текущей задачи можно забрать управление процессором. Ядро проверяет значение preempt_count задачи. Если preempt_count равно 0 и поэтому текущая задача не содержит блокировок, вызывается schedule () и для выполнения выбирается новая задача. Если preempt_count не равно 0, передавать управление другой задаче небезопасно, а управление возвращается в текущую задачу до тех пор, пока с нее не будут сняты все блокировки. Когда текущая задача освобождает блокировку, проверяют, требует ли задача перепланировки. Когда текущая задача освобождает последнюю блокировку и preempt_count достигает 0, производится немедленное перепланирование.

7.3 Циклическая блокировка и семафоры

Когда два или более процесса требуют специального доступа к разделяемому ресурсу, они вынуждены устроить состязание за управление данным блоком кода. Базовой формой блокировки в Linux является циклическая блокировка.

Циклическая блокировка (spinlock) получила свое имя благодаря тому факту, что она выполняется циклически или крутится *(spin),* ожидая наступления блока. Благодаря такой работе циклической блокировки желательно не вставлять в циклически блокируемый код никаких повторных блокировок. Иначе может произойти ступор системы.

Перед применением циклической блокировки структура spin_lock должна быть инициализирована. Это делается с помощью вызова spin_lock_init ():

include/linux/spinlock.h

```
63 #define spin lock init(x) \
64   do { \
65      (x)->magic = SPINLOCK MAGIC; \
66      (x)->lock = 0; \
67      (x)->babble = 5; \
68      (x)->module =   FILE  ; \
69      (x)->owner = NULL; \
70      (x)->oline = 0; \
71    } while (0)
```

Этот блок кода устанавливает spin_lock в «разблокированное» состояние или в 0 в строке 66 и инициализирует другие переменные структуры. Здесь мы коснемся переменной (x) ->1оск.

После инициализации spin_lock ее можно получить с помощью spin_lock() или spin_lock_irqsave (). Функция spin_lock_irqsave () отключает прерывания перед блокировкой, а spin_lock () - нет. Если вы используете spin_lock (), процесс может быть прерван в заблокированном разделе кода.

Для освобождения spin_lock после выполнения критической секции кода вам нужно вызвать spin_unlock() или spin_unlock_irqrestore(); spin_unlock_irqrestore() восстанавливает состояние регистров прерывания до уровня, в котором они были до вызова spin_lock_irq ().

Давайте рассмотрим вызовы spin_lock_irqsave () и spin_unlock_ irqre-store().

```
include/linux/spinlock.h
258 #define spin_lock_irqsave(lock, flags) \
259 do { \
260   local_irq_save(flags); \
```

```
261   preempt disable(); \
262    raw spin lock flags(lock, flags); \
263 } while (0)

321 #define spin_unlock_irqrestore(lock, flags) \
322 do { \
323   _raw_spin_unlock(lock); \
324   local irq restore(flags); \
325   preempt_enable(); \
326 } while (0)
```

Обратите внимание, как во время блокировки отключается приоритетное прерывание обслуживания. После этого можно быть уверенным, что операции в критической секции не будут прерваны. Флаг IRQ сохраняется в строке 260 и восстанавливается в строке 324.

Недостатком циклической блокировки является бесполезный цикл, ожидающий снятия блокировки. Ее лучше использовать для критических секций кода, которые не требуют много времени на выполнение. Для долго выполняющихся блоков кода лучше использовать другой механизм блокировки в Linux - семафор.

Семафоры отличаются от циклической блокировки тем, что при возникновении конкурентного обращения к ресурсу задача засыпает вместо того, чтобы быть в бесконечном ожидании. Достоинством семафора является безопасность блокировки; они безопасны при выполнении на SMP и при возникновении прерываний.

```
include/asm-i3 8 б/semaphore.h
44 struct semaphore {
45   atomic_t count;
46   int sleepers;
47   wait queue head t wait;
48 #ifdef WAITQUEUE_DEBUG
49   long  _ magic;
50 #endif
51 };
```

```
include/asm-ppc/semaphore.h
24 struct semaphore {
25   /*
26   * Обратите внимание, что отрицательное значение счетчика
27   * эквивалентно 0, а также дополнительно означает, что процесс
28   * (процессы) должен спать или ожидать.
29   */
30   atomic_t count;
```

```
31   wait queue head t wait;
32 #ifdef WAITQUEUE DEBUG
33   long _ magic;
3 4 #endif
35 };
```

Реализации на обеих архитектурах предоставляет указатель на wait_queue и счетчик. Счетчик хранит количество процессов, которые может хранить семафор в каждый из промежутков времени. Применяя семафор, мы можем иметь несколько процессов, одновременно вошедших в код критической секции. Если счетчик инициализирован в 1, только один процесс может войти в код критической секции; семафор со счетчиком, равным 1, называется **мьютексом (mutex).**

Семафоры инициализируются с помощью sema_init O, а их блокировка и разблокирование производятся с помощью вызовов down () и up () соответственно. Помимо этого, существует down_interruptible (), которая возвращает 0, если семафор получен, и EINTR, если процесс был прерван при блокировке.

Когда процесс вызывает down () или down_interrruptible (), поле счетчика в семафоре уменьшается. Если значение поля меньше 0, вызывающий down () процесс блокируется и добавляется в wait_queue семафора. Если поле больше либо равно 0, процесс продолжает свою работу.

После выполнения кода критической секции процесс должен вызвать up () для сообщения семафору о завершении работы с критической секцией. С помощью вызова up () процесс увеличивает значение поля count в семафоре и, если счетчик больше или равен O, пробуждает процесс, ожидающий в wait_queue семафора.

7.4 Системные часы: прошедшее время и таймеры

В целях планирования ядро использует системные часы для определения того, как долго выполняется задача. Мы уже рассматривали системные часы в гл. 5 и использовали их в качестве примера для обсуждения прерываний. Здесь мы рассмотрим часы реального времени, их применение и реализацию, но для начала давайте определимся с основными понятиями часов.

Часы - это периодические сигналы, возникающие в процессоре, позволяющие ему обрабатывать кванты времени. Процессор в соответствии с сигналом часов узнает, когда ему нужно выполнить следующую операцию, такую, как сложение двух целых чисел или извлечение значения из памяти. Скорость этого сигнала часов (1.4 ГГц, 2 ГГц и т. д.) исторически применяется для сравнения скорости процессора системы на рынке электроники.

На данный момент ваша система содержит несколько запущенных часов/таймеров. Простыми примерами могут служить время и дата, отображаемые внизу вашего экрана

(также известные как настенные часы), курсор, настойчиво пульсирующий на десктопе, экранная заставка, включившаяся на вашем ноутбуке, после того как вы не прикасались к нему некоторое время. Более сложные примеры включают в себя воспроизведение аудио и видео, повторное нажатие клавиш (когда клавиша зажимается), скорость передачи коммуникационных портов и обсуждаемое ранее измерение длительности выполнения задачи.

7.4.1 Часы реального времени: что это такое

В Linux интерфейс к *таймеру настенных часов* предоставляется через функцию ioctl () драйвера устройства /dev/rtc. Устройство этого драйвера называется Real Time Clock (RTC)[1]. RTC[2] предоставляет функцию для работы со 114-битовым значением в NVRAM. На входе этого устройства установлен осциллятор с частотой 32768 КГц, подсоединенный к резервной батарее. Некоторые дискретные модели RTC имеют встроенные осциллятор и батарею, тогда как другие RTC встраиваются прямо в контроллер периферийной шины (например, южный мост) чипсета процессора. RTC возвращает не только время суток, но, помимо прочего, является и программируемым таймером, имеющим возможность посылать системные прерывания. Частота прерываний варьируется от 2 до 8192 Гц. Также RTC может посылать прерывания ежедневно, наподобие будильника. Далее мы рассмотрим код RTC.

```
/include/linux/rtc.h
/*
 * ioctl вызывает приоритетное прерывание обслуживания для /dev/rtc
 * interface, если включен любой из RTC-драйверов.
 */
70  #define RTC_AIE_ON _IO('p', 0x01) /* Включение прерывания звонка */
71  #define RTC_AIE__OFF _IO('p', 0x02) /* ... отключение */

/* Включение прерывания обновления */
72  #define RTC_UIE_ON _IO(•p■, 0x03)
73  #define RTC_UIE_OFF _IO('p', 0x04)   /* ... отключение  */

/* Включение периодического прерывания */
74  #define RTC_PIE__ON JOCp', 0x05)
75  #define RTC_PIE_OFF „ЮС'р', 0x06)     /* ... отключение  */

/* Включение сторожевого прерывания */
76  #define RTC WIE ON „ЮСр', OxOf)
77  #define RTC_WIE_OFF _IO('p', 0x10)  /* ... отключение  */
```

[1] Часы реального времени. *Примеч. пер.*

[2] Производится несколькими поставщиками. Наиболее распространенным является вариант tc146818 от Motorola. (Данный RTC больше не производится, и его место занял Dallas DS12885 или его эквиваленты.)

```
/* Установка времени звонка */
78  #define RTC_ALM_SET _IOW('p'/ 0x07, struct rtc_time)
/* Чтение времени звонка*/
79  #define RTC_ALM_READ  _IOR('p'    0x08, struct rtc_time)
/* Чтение времени RTC  */
80  tdefine RTC RD TIME   _IOR('p'    0x09/ struct rtc_time)
/* Установка времени RTC */
81  #define RTC_SET_TIME    _IOWCp\       0x0a, struct rtc_time)
/* Чтение частоты IRQ */
82  #define RTC_IRQP_READ  _IOR('p• ,  0x0b, unsigned long)
/* Установка частоты IRQ */
83  #define RTC_IRQP_SET  _IOW('p\   0x0c, unsigned long)
/* Чтение epoch */
84  tdefine RTC_EPOCH_READ _IORCp\       OxOd, unsigned long)
/* Установка epoch */
85  #define RTC_EPOCH_SET  _IOW('p\  OxOe, unsigned long)
86 /* Установка сигнала на
пробуждение */                        struct rtc_wkalrm) struct
87  #define RTC_WKALM_SET  _IOW('p\
OxOf
                                     rtc wkalrm) struct
/* Получение сигнала на пробуждение 4
88 #def ine RTC__WKAIM_RD  _IOR ( ' p • ,  0x10,
                                     rtc_pll_info) 0x12, struct
89  /* Получение корректировки PLL
*/
90  #define RTC__PLL_GET   „JORCp',   rtc_pll_info)
0x11
/* Установка корректировки PLL */
91  #define RTC_PLL_SET    _IOW('p'
```

Функция управления ioctl () находится в include/linux/rtch. Не все из перечисленных там вызовов ioctl () для RTC реализованы для PPC-архитектуры. Эти управляющие вызовы вызывают, в свою очередь, аппаратно-специфические функции более низкого уровня (если они реализованы). Пример в этом подразделе использует функцию RTC_RD_TIME.

Следующий пример вызова ioctl () используется для получения времени суток. Программа просто открывает драйвер, запрашивает у аппаратуры RTC текущую дату и время и помещает эту информацию в stderr. Обратите внимание, что единовременно только один пользователь может обращаться к драйверу RTC. Проверяющий это код демонстрируется при обсуждении драйвера.

```
Documentation/rtc.txt
/*  Укороченная версия кода в /Documentation/rtc.txt  */

int main(void) {

int fd, retval = 0;
//unsigned long tmp, data;
struct rtc_time rtc_tm;

fd = open (Vdev/rtc", 0_RDONLY) ;

/* Чтение времени/даты RTC */
retval = ioctl(fd, RTC_RD_TIME, &rtc_tm) ;

/* вывод времени из переменной rtc tm */
close(fd); return 0;

} /* конец main */
```

Этот код является отрывком более полного примера в /Documentation/ rtc. txt. Две основные строки кода этой программы - это команда open () и вызов io-ctl (); open () сообщает нам, какой драйвер используется (/dev/rtc), а iotcl () указывает конкретный путь сквозь код к физическому интерфейсу RTC и команде RTC_RD_TI'ME. Код драйвера для команды open () описан в исходных кодах драйвера, однако нам для обсуждения необходимо знать только, *какой* драйвер устройства открывается.

7.4.2 Чтение из часов реального времени на PPC

Во время компиляции ядра вставляется соответствующая ветка дерева кода (x86, PPC, MIPS и т. д.). Здесь обсуждается PPC вариант исходного файла для обобщенного драйвера RTC не x86 систем.

```
/drivers/char/genrtc.c
276  static int gen_rtc_ioctl(struct inode *inode, struct file *file,
277    unsigned int cmd, unsigned long arg)
278  {
279   struct rtc time wtime;
280   struct rtc pll info pll;
281
2 82   switch (cmd) {
283
284   case RTC_PLL_GET:

290   case RTC_PLL_SET:
```

```
298   case RTC_UIE_OFF:  /* отключение прерываний от обновлений RTC. */

302   case RTC_UIE_ON:   /* включение прерываний от обновлений RTC. */

305   case RTC RD TIME:  /* Чтение времени/даты из RTC */
306
307     memset (&wtime, 0, sizeof (wtime) ) ;
308     get rtc time(&wtime) ;
309
310   return copy to user ( (void *) arg, &wtime, sizeof (wtime) ) ? -EFAULT: 0 ;
311
312     case RTC SET TIME:     /* Set the RTC */
313       return -EINVAL;
314    }

353    static int gen rtc open(struct inode *inode, struct file *file)
354    {
355   if (gen_rtc_status & RTC_IS_OPEN)
356      return -EBUSY;
357   gen_rtc_status |= RTC_IS_OPEN;
```

Этот код реализует тот же набор команд ioctl. Так как мы выполняем вызов ioctl из проверочной программы пользовательского пространства с флагом RTC_RD_TIME, управление передается на строку 305. Следующий вызов в строке 308 - это get_rtc_time(&wtime) из rtc.h (см. соответствующий код). Перед тем как оставить этот блок кода, обратите внимание на строку 353. Она позволяет получать доступ к драйверу только для одного пользователя зараз с помощью open (), устанавливающей состояние драйвера в RTC_IS_OPEN:

include/asm-ppc/rtc.h

```
45     static inline unsigned int get_rtc_time(struct rtc_time *time)
46     {
047      if (ppc_md.get_rtc_time) {
048       unsigned long nowtime;
049
050       nowtime = (ppc md.get rtc time)();
051
052             to tm(nowtime, time);
053
054       time->tm year -= 1900;
055    time->tm_mon -= 1; /* Make sure userland has a 0-based month */
056      }
```

```
057   return RTC_24H;
058  }
```

Встроенная функция get_rtc_time () вызывает функцию, устанавливающую переменную-указатель в значение ppc_md. get_rtc_time на строке 50. Ранее при инициализации ядра эта переменная устанавливалась в chrp_setup. c:

arch/ppc/platforms/chrp_setup.c
```
447   chrp_init(unsigned long r3,   unsigned long r4,   unsigned long r5,
448   unsigned long r6,   unsigned long r7)
449   {

477   ppc md.time init = chrp time init;
478   ppc md.set  rtc time = chrp set rtc time;
479   ppc_md.get_rtc_time = chrp_get_rtc_time;
480   ppc_md.calibrate_decr = chrp_calibrate_decr;
```

Функция chrp_get_rtc_time () (в строке 479) определена в chrp__time.c в следующем блоке кода. Так как информация о времени в памяти CMOS обновляется на периодической основе, блокировка кода чтения включена в цикл for, который пересчитывает обновляемые в прогрессе блоки:

```
arch/ppc/platforms/chrp time.c
122   unsigned long  _chrp chrp_get__rtc_time(void)
123   {
124    unsigned int year, mon, day, hour, min, sec;
125    int uip, i;

141    for ( i = 0; i<1000000; i++) {
142     uip = chrp cmos clock read(RTC FREQ SELECT);
143     sec = chrp cmos  clock read(RTC SECONDS);
144     min = chrp_cmos_clock__read(RTC_MINUTES);
145     hour = chrp cmos clock read(RTC HOURS);
146     day = chrp cmos clock read(RTC DAY OF MONTH) ;
147     mon = chrp_cmos_clock_read(RTC_MONTH);
148     year = chrp cmos clock read(RTC YEAR);
149     uip |= chrp cmos clock read(RTC FREQ SELECT) ;
150     if ((uip & RTC_UIP)==0) break;
151    }
152    if (!(chrp_cmos_clock_read(RTC_CONTROL)
153    & RTC DM BINARY) || RTC ALWAYS BCD)
154    {
```

```
155      BCD_TO_BIN(sec);
156      BCD_TO_BIN(min);
157      BCD_TO_BIN(hour);
158      BCD_TO_BIN(day);
159      BCD_TO_BIN(mon);
160      BCD TO BIN(year);
161     }

54    int _ chrp chrp_cmos_clock_read(int addr)
55    {   if (nvram_asl != 0)
56     outb(addr»8, nvram asl) ;
57     outb(addr, nvram as0);
58     return (inb(nvram_data));
059   }
```

Наконец, в chrp_get_rtc_time () значения отдельных компонентов структуры времени считываются с устройства RTC с помощью функции chrp_cmos_clock_ read. Эти значения форматируются и возвращаются в структуру rtc_tm, передаваемую в обратном вызове ioctl в пространство проверочной программы.

7.4.3 Чтение из часов реального времени на x86

Метод чтения RTC на платформе x86 довольно похож, но немного более компактен и удобен по сравнению с методом на PPC. Еще раз мы открываем драйвер /dev/rtc, но на этот раз при сборке компилируется файл rtc. с для архитектуры x86. Далее обсуждается вариант исходного кода для x86.

```
drivers/char/rtc.c

352   static int rtc_do_ioctl(unsigned int cmd,
                              unsigned long arg, int kernel)
353   {

switch (cmd) {

482   case RTC RD TIME:      /* Read the time/date from RTC  */
483   {
484    rtc get rtc time (ScWtime) ;
485    break;
486   }

1208   void rtc_get_rtc_tinie (struct  rtc_time  *rtc_tm)
1209   {
```

```
123 8  spin lock irq(&rtc lock) ;
1239    rtc_tm->tm_sec = CMOS_READ(RTC_SECONDS) ;
1240    rtc tm->tm min = CMOS READ(RTC MINUTES);
1241    rtc_tm->tm_hour = CMOS_READ(RTC_HOURS);
1242    rtc tm->tm mday = CMOS READ(RTC DAY OF MONTH) ;
1243    rtc tm->tm mon = CMOS READ(RTC MONTH);
1244    rtc tm->tm  year = CMOS READ(RTC YEAR) ;
1245    Ctrl = CMOS READ(RTC CONTROL);

1249  spin_unlock_irq(&rtc_lock);
1250
12 51  if (!(ctrl & RTC DM BINARY) || RTC ALWAYS BCD)
1252   {
1253    BCD TO BIN(rtc tm->tm sec) ;
1254    BCD_TO_BIN(rtc_tm->tm_min) ;
1255    BCD TO BIN(rtc tm->tm hour);
1256    BCD TO BIN(rtc tm->tm mday);
1257    BCD TO BIN(rtc tm->tm mon) ;
1258    BCD_TO_BIN(rtc_tm->tm_year);
1259   }
```

Программа проверки использует флаг RTC_RD_TIME в вызове ioctl () драйвера в rtc.c;ioctl переключает состояние и затем заполняет структуру из памяти CMOS для RTC. Далее приведена реализация аппаратного чтения из RTC для x86.

```
include/asm-i38б/mcl46818rtc.h

018   #define CMOS READ(addr) ({ \
019    outb p((addr),RTC PORT(0)); \
02 0   inb p (RTC PORT(1)); \
021   })
```

Резюме

В этой главе описаны планировщик Linux, приоритетное прерывание обслуживания, системные часы и таймер Linux.

Точнее говоря, мы обсудили следующие темы:

- мы представили вашему вниманию новый планировщик Linux 2.6 и рассмотрели его особенности;
- мы обсудили, как планировщик выбирает новую задачу из множества задач и как используется алгоритм работы планировщика и т. д.;

- мы обсудили переключение контекста, используемого планировщиком для переключения процессов, и проследили соответствующие функции до уровня аппаратно-специфического кода;
- мы описали, как процессы в Linux могут передавать управление процессором другому процессу с помощью вызова schedule () и как ядро маркирует процессы, требующие «перепланировки»;
- мы углубились в подсчет ядром Linux динамических приоритетов на основе предыдущего поведения отдельного процесса и в то, как процесс удаляется из очереди планировщика;
- далее мы перешли к описанию явных приоритетных прерываний обслуживания пользовательского уровня и уровня ядра и их реализации в ядре Linux 2.6;
- и наконец, мы рассмотрели таймеры, системные часы и реализацию системных часов на архитектурах x86 и PPC.

Упражнения

1. Как Linux сообщает планировщику, чтобы тот периодически запускался?
2. Опишите различие между интерактивными и неинтерактивными процессами.
3. С точки зрения планировщика какими отличиями обладают процессы реального времени?
4. Что происходит, когда у процесса заканчиваются запланированные тики?
5. В чем преимущество 0(1)-планировщика?
6. Какие структуры данных использует планировщик для управления приоритетами запущенных в системе процессов?
7. Что произойдет, если вы вызовите schedule () во время циклической блокировки?
8. Как ядро решает, возможно ли выполнить неявное приоритетное прерывание обработки задачи?

Глава 8

Загрузка ядра

В этой главе:

На данный момент мы рассмотрели подсистемы ядра Linux и используемые в их операциях структуры. Каждая глава подразумевала, что подсистема была настроена и запущена, а мы фокусировались на типичном управлении подсистемами ядра и обработке их операций. Тем не менее каждая из подсистем должна быть инициализирована перед использованием. Эта инициализация происходит во время загрузки ядра, после того как загрузчик завершит загрузку образа ядра в память и передаст ему управление.

Мы выбрали путь следования процессу инициализации ядра в линейном порядке. Начнем мы с обсуждения того, что при этом происходит, начиная с включения и вызова первой архитектурно-зависимой функции, start_kernel (), и заканчивая вызовом процесса инициализации /sbin/init. Рис. 8.1 иллюстрирует порядок сообщений начиная с включения и заканчивая выключением.

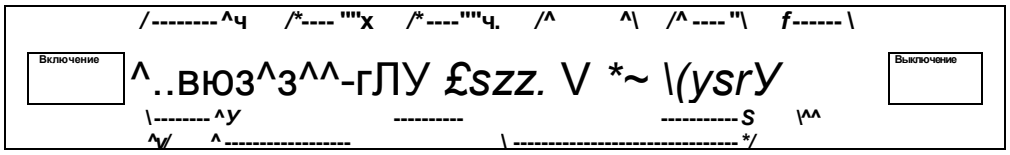


Рис. 8.1. Старт ядра и процесс загрузки

Мы начнем с обсуждения BIOS и Open Firmware, являющихся первым кодом, запускаемым на системах x86 и PPC при включении соответственно. Далее мы обсудим наиболее распространенные загрузчики, используемые в Linux, и то, как они загружают ядро и передают ему управление. После этого мы подробно обсудим шаги инициализации ядра (kernel initialization), во время которой инициализируются все подсистемы. В конце инициализация ядра происходит вызов /sbin/init как процессом 1. Программа init продолжается тем, что называется инициализацией системы (system initialization), с помощью включения процессов, необходимых перед регистрацией пользователя в системе.

Вскоре станет ясно, что часть инициализации ядра состоит из вложенных инициализаций подсистем. Это затрудняет попытки проследить непрерывный процесс инициализации подсистем с начала и до конца. Тем не менее дальнейший линейный порядок загрузки ядра Linux позволяет проследить настройку подсистем ядра и по мере их появления и иллюстрирует сложность процесса загрузки.

Мы коснемся многих структур, представленных в предыдущих главах, по мере их загрузки и инициализации. Мы начнем с рассмотрения первого шага: BIOS и Open Firmware.

8.1 BIOS и Open Firmware

При включении процессор сначала получает доступ к адресам, которые обычно находятся в доступной для чтения области памяти. Эта доступная только для чтения память обычно располагается в Flash ROM (или просто Flash). Там располагается первый код,

который выполняется при каждом запуске системы. Этот код отвечает за включение минимума систем, необходимых для загрузки ядра.

На x86-системах он полностью находится в BIOS (Basic Inpit Output System)[1] - блоке аппаратно-зависимого кода инициализации системы, загружающего систему. На системах x86 загрузчик, и соответственно Linux, зависит от BIOS, приводящего систему в определенное состояние. Интерфейс BIOS составляет унифицированный набор функций, известных как прерывания(interrupts). Во время загрузки Linux использует эти прерывания для запроса доступных ресурсов системы. После того как BIOS закончит инициализацию, он копирует первые 512 байт с устройства загрузки (описываемого в следующем разделе) в адрес 0x7c00 и переходит в него. Несмотря на то что в некоторых случаях BIOS загружает операционную систему через сетевое соединение, мы будем рассматривать процесс загрузки Linux с жесткого диска. После загрузки Linux BIOS все равно находится в памяти и его функции доступны через прерывания.

На PowerPC тип кода инициализации зависит от возраста соответствующей архитектуры PowerPC. Старые системы IBM используют PowerPC Reference Platform (PreP)[2], тогда как более новые системы IBM используют Common Hardware Reference Platform (CHRP)[3]. Системы G4 и позднейшие называются «Новым миром» и используют Open Firmware(OF) в границах реализации конкретной архитектуры. (Более подробную информацию об этих процессорах и системно-зависимом загрузочном firmware, а также совместимости их форматов вам стоит получить с домашней страницы Open Firmware на www. openf irmware. org.)

8.2 Загрузчики

Загрузчики - это программы, находящиеся на загрузочном диске компьютера. Первым устройством загрузки обычно является первый жесткий диск системы. Загрузчик вызывается BIOS (x86) или firmware (PPC) после того, как инициализация системы обеспечит поддержку памяти, прерываний и ввода-вывода, требуемых для загрузки ядра. После загрузки ядро инициализируется и конфигурируется операционной системой.

Для систем x86 BIOS позволяет пользователю установить последовательность устройств загрузки для его системы. Такими устройствами загрузки обычно являются флоппи-дисководы, CD-ROM и жесткие диски. Форматирование диска (например, с помощью f disk) создает на диске Master Boot Record (MBR)[4], располагающуюся в первом секторе (сектор 0, цилиндр 0, головка 0) загрузочного диска. MBR содержит небольшую программу и таблицу разделов из четырех элементов. Конец загрузочного сектора помеча-

[1] **Базовая система ввода-вывода.** ***Примеч. пер.***

[2] **Эталонная платформа PowerPC.** ***Примеч. пер.***

[3] **Простая эталонная аппаратная платформа.** ***Примеч. пер.***

[4] **Главная загрузочная запись.** ***Примеч. пер.***

ется шестнадцатеричных значением 0xAA55 в позиции 510. Табл. 8.1 демонстрирует компоненты MBR.

Таблица 8.1. Компоненты MBR

Отступ	Длина	Назначение
0x00	Oxbd	Код программы MBR
Oxlbe	0x40	Таблица разделов
Oxlfe	0x2	Шестнадцатиричный маркер сигнатуры

Таблица разделов MBR хранит информацию, относящуюся к каждому из главных разделов жестких дисков. Табл. 8.2 демонстрирует, как выглядит каждая 16-битовая запись в разделе MBR.

Таблица 8.2. 16-битовые записи MBR

Отступ	Длина	Назначение
0x00	1	Флаг активного загрузочного раздела
0x01	3	Начальный цилиндр/головка/сектор загрузочного раздела
0x04	1	Тип раздела (Linux использует 0x83, PPC PReP использует 0x41)
0x05	3	Конечный цилиндр/головка/сектор загрузочного раздела
0x08	4	Номер начального сектора раздела
0x0c	4	Длина раздела (в секторах)

В конце самотестирования и идентификации аппаратуры код инициализации системы (Firmware или BIOS) получает доступ к контроллеру жесткого диска для чтения MBR. После того как тип загрузочного устройства определен, код инициализации системы получает через задокументированный интерфейс (например, для диска ШЕ) доступ к головке 0, цилиндру 0 и сектору 0.

После обнаружения устройства загрузки MBR копируется в память по адресу 0x7c00 и выполняется. Маленькая программа в голове MBR распаковывается и ищет в таблице разделов активный загрузочный раздел. Далее MBR копирует код с активного загрузочного раздела в адрес 0x7c00 и начинает его выполнение. Начиная с этой точки DOS обычно загружает систему x86. Тем не менее активный загрузочный раздел может иметь загрузчик, который, в свою очередь, загружает операционную систему. Теперь мы обсудим несколько наиболее распространенных загрузчиков, используемых Linux. Рис. 8.2 демонстрирует то, как выглядит память во время загрузки.

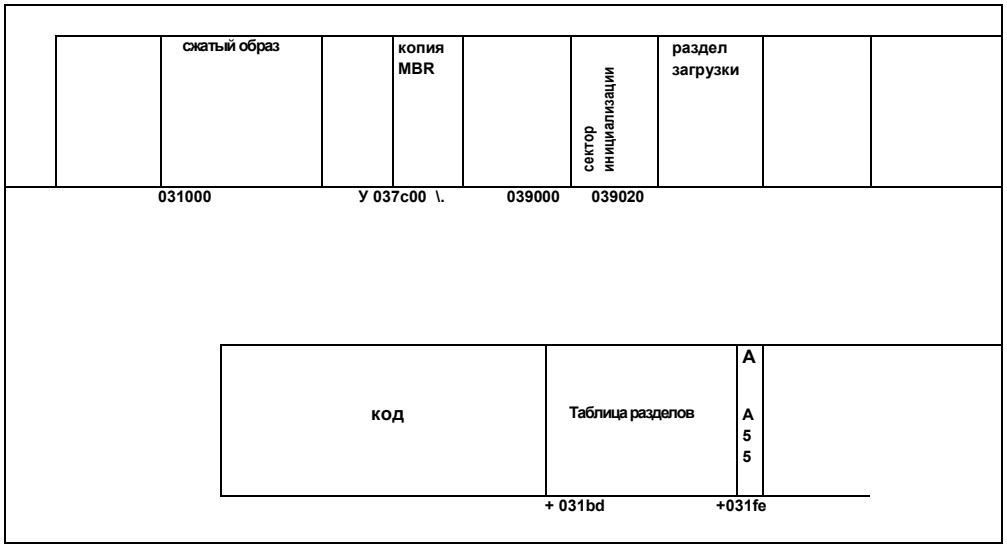


Рис. 8.2. Вид памяти во время загрузки

8.2.1 GRUB

Grand Unified Bootloader (GRUB)[1] - это x86-загрузчик, используемый для загрузки Linux. GRUB 2 на момент написания книги находился в процессе портирования на PPC. Соответствующая документация, включающая его историю и особенности дизайна, находится на www.gnu.org/software/grub. GRUB распознает файловые системы на загрузочных дисках, а ядро может быть загружено из указанного файла, диска и раздела, где оно расположено. GRUB является двухэтапным загрузчиком (two-stage bootloader[2]). Этап 1 инсталлируется в MBR и вызывается из BIOS. Этап 2 вначале загружает этап 1, а по его завершении загружается из файловой системы. Остановки и события, возникающие на каждом из этих этапов, перечислены далее.

Этап 1

1. **Инициализация.**
2. **Определение загрузочного диска.**
3. **Загрузка первого сектора этапа 2.**
4. **Переход на этап 2.**

• Великий унифицированный загрузчик. *Примеч. пер.

[2] Иногда GRUB использует этап 1.5, но мы рассмотрим только два обыкновенных этапа.

Этап 2

1. Загрузка остатка этапа 2.
2. Переход в загруженный код.

Доступ к GRUB можно получить через интерактивную командную строку или интерфейс с набором меню. При использовании интерфейса с меню должен быть создан файл конфигурации. Далее приведена строка из конфигурационного файла GRUB, загружающего ядро Linux.

```
/boot/menu.1st

title     Kernel 2.6.7, test kernel
root       (hd0,0)
kernel          /boot/bzImage-2.6.7-mytestkernel  root=/dev/hdal  ro
```

Пункт title хранит метку для настройки; root устанавливает текущее корневое устройство в hdO, раздел 0; kernel загружает первичный загрузочный образ ядра из указанного файла. Оставшаяся информация о записях ядра передается в качестве параметров во время загрузки ядра.

Некоторые аспекты загрузки, такие, как месторасположение загружаемого и распаковываемого образа ядра, конфигурируется в аппаратно-зависимом разделе кода ядра Linux. Давайте рассмотрим arch/ i3 8 б /boot /setup. S, где хранятся такие настройки для x86.

```
arch/i836/boot/setup.S
61 INITSEG = DEF INITSEG # 0x9000, перемещаем загрузчик сюда с дороги
62 SYSSEG = DEF SYSSEG # 0x1000, система загружается из 0x10000 (65536).
63 SETUPSEG = DEF_SETUPSEG  # 0x9020, это текущий сегмент[1]
```

Эта конфигурация указывает, какой образ Linux загружать в линейный адрес 0x9000, и переходит по адресу 0x9020. В этой точке распакованная часть ядра Linux распаковывает запакованную часть по адресу 0x10000 и начинает инициализацию ядра.

GRUB основан на *Многозагрузочной спецификации(МиШЪоо1 Specification).* На момент написания книги Linux еще не обладал всеми структурами, необходимыми для многозагрузочной компиляции, но обсудить многозагрузочные требования все-таки стоит.

[к] Ядро воспринимает спецификацию во время загрузки через командную строку ядра. Это строка, описывающая список параметров, хранящих информацию об аппаратной специфике, значениях по умолчанию и т. д. Более подробную информацию о загрузочной строке Linux можно найти по адресу www.tldp.org/HOWTO/BootPrompt-HOWTO.html.

8.2.1.1 Многозагрузочная спецификация

Многозагрузочная спецификация описывает интерфейс между любым потенциальным загрузчиком и любой потенциальной операционной системой. Многозагрузочная спецификация не указывает, как должен работать загрузчик, а только указывает интерфейс для работы с загружаемой операционной системой. Текущая цель - это архитектура x86 и свободные 32-битовые операционные системы, для которых предусматривается стандарт передачи конфигурационной информации из загрузчика в операционную систему. Образ ОС может быть любого типа (ELF или специального), но должен содержать **многозагрузочный заголовок (multiboot header)** в первых 8 Кб образа вместе с магическим числом 0xlBADB002. Многозагрузочный загрузчик должен предоставлять и метод для вспомогательных загрузочных модулей или драйверов, используемых некоторыми ОС во время загрузки, так как эти ОС не загружают все необходимое для работы в загрузочный образ ядра. Обычно так поступают модульные загрузочные ядра, для того чтобы размер загрузочного ядра не выходил за разумные пределы.

Многозагрузочная спецификация диктует, что, когда загрузчик вызывает ОС, система должна находиться в специальном 32-битовом реальном режиме, для того чтобы ОС могла выполнять обратные вызовы из BIOS. И наконец, загрузчик должен предоставить ОС структуры данных, заполненные основными машинными данными. Далее мы рассмотрим структуры данных с многозагрузочной информацией.

```
typedef struct multiboot__info
{
ulong flags;       // описывает следующие поля
ulong mem lower;   // если flags[0], количество памяти < 1M
ulong mem_upper;   // если flags[0], количество памяти > 1M
ulong boot_device;  // если flags[l], диск, parti, 2, 3
ulong cmdline;      // если flags[2],адрес командной строки
ulong mods count;   // если flags[3], # загрузочного модуля
ulong mods addr;    // если flags[3], адрес первого загрузочного модуля
union
{
aout_symbol__table_t aout_sym;  // если flags[4], таблица символов
                                // из a.out образа ядра
elf_section_header_table_telf_sec;    //еслиflags [5] , заголовок
                                      // из ELF ядра }
и;
ulong mmap length;  // если flags[б],BIOS длина отображения в память
ulong mmap addr;    // если flags[б], адрес отображения BIOS ulong
drives length;  // если flags[7], информационная структура
                // диска BIOS ulong drives_length;  //
если flags[7], первая информационная
                // структура BIOS.
```

```
ulong config table    // если flags[8],настроечная таблица ROM
ulong boot loader name  // если flags[9],адрес строки
ulong apm table // если flags[10],адрес информационной таблицы APM
ulong vbe_control_info  // если flags[11], настройка видеорежима
ulong vbe_mode_info
ulong vbe mode
ulong vbe interface seg
ulong vbe interface off
ulong vbe_interface__len
};
```

Указатель на эту структуру передается в EBX, когда управление передается ОС. Первое поле, flags указывает, какие из следующих полей верны. Неиспользуемые поля должны иметь значение 0. Вы можете подробнее изучить *Многозагрузочную спецификацию* по адресу www.gnu.org/software/grub/manual/multiboot/multi-boot.html.

8.2.2 LILO

Linux Loader (LILO)[1] используется в качестве x86-загрузчика Linux уже много лет. Это одна из простейших программ загрузки, доступная для настройки и загрузки ядра Linux. LILO похож на GRUB в плане того, что он тоже работает в два этапа. LILO использует файл конфигурации и не имеет интерфейса командной строки.

Мы снова начнем с инициализации BIOS системы и загрузки MBR (этап 1) в память и передачу в него управления. Остановки и события, возникающие на каждом из этапов LILO, описаны ниже.

Этап1

1. Начало выполнения и отображение «L».
2. Распознание геометрии диска и отображения «I».
3. Загрузка кода этапа 2.

Этап 2

1. Начало выполнения и отображение «L».
2. Нахождение данных загрузчика и ОС, отображение «O».
3. Определение, какую ОС загружать, и переход в нее.

Фрагмент из конфигурационного файла LILO выглядит следующим образом:

[1] Загрузчик Linux. *Примеч. пер.*

```
/etc/lilo.conf
image=/boot/bzImage-2.6.7-mytestkernel
label=Kernel 2.6.7, my test kernel
root=/dev/hda6
read-only
```

Параметрами являются image, указывающий путь к ядру; label 1, описывающий строку конфигурации; root, описывающий раздел, где находится корень файловой системы, и read-only, описывающий варианты корневых разделов при загрузке.

Далее приведен список различий между GRUB и LILO.

- LILO хранит конфигурационную информацию в MBR. Если производятся какие-то изменения, для обновления MBR необходимо вызвать /sbin/lilo.
- LILO не может читать различные файловые системы.
- LILO не имеет интерфейса командной строки.

Давайте рассмотрим, что происходит, когда LILO устанавливается в качестве загрузчика. Сначала MBR (содержащий LILO) копируется в 0x7c00 и начинает свое выполнение. LILO начинает с копирования образа ядра, указанного в /etc/lilo.conf, с жесткого диска. Этот образ, созданный с помощью build, с, подготавливается из сектора init (загружаемого в 0x90000), настроечного сектора (загружаемого в 0x90200) и сжатого образа (загружаемого в 0x10000). Далее LILO переходит по метке start_of_setup по адресу 0x90200.

8.2.3 PowerPC и Yaboot

Yaboot - это основанный на OpenFirmware (OF) загрузчик машин PowerPC New World. Аналогично LILO и GRUB, Yaboot использует конфигурационный файл и утилиты наподобие ybin и ybootconf ig для установки загрузочного раздела, содержащего Yaboot. Аналогично BIOS x86, OF позволяет настройку загрузочного диска. Однако в случае OF он различается от системы к системе. Настройки OF обычно можно узнать, введя «Command+Option/Alt+o+f.?».

Yaboot использует следующие шаги загрузки:

1. Yaboot вызывается OF.
2. Ищет загрузочные устройства, загрузочные пути и открытые загрузочные разделы.
3. Открывает /etc/yaboot. conf или командную оболочку.
4. Загружает образ ядра и initrd.
5. Выполняет образ.

Как вы можете видеть, фрагмент загрузочной информации для Yaboot аналогичен LELO и GRUB:

```
yaboot.conf label=Linux
root=/dev/hdall
sysmap=/boot/System.map
read-only
```

Как и в случае LILO, ybin устанавливает Yaboot в загрузочный раздел. Любые обновления-изменения в конфигурацию Yaboot требуют запуска ybin.

Документацию на Yaboot можно найти на www. penguinppc. org.

8.3 Архитектурно-зависимая инициализация памяти

Теперь мы уделим внимание особенностям аппаратного управления на PPC и x86. Обе архитектуры имеют аппаратные особенности управления памятью для поддержки окружения с реальной и виртуальной адресацией. Как и все операционные системы, менеджер памяти Linux зависит от аппаратной архитектуры. Этот раздел описывает аппаратную инициализацию на обеих архитектурах. Так как инициализация менеджера памяти полностью аппаратно-зависима, для понимания последовательности процесса инициализации нужно понимать аппаратную спецификацию. Менеджер памяти является одной из первых инициализируемых подсистем и запускается раньше, чем start_kernel () из-за своей аппаратно-зависимой сущности.

8.3.1 Аппаратное управление памятью на PowerPC

В мире PowerPC также известно как «управление хранением» (storage control). Этот подраздел описывает аппаратно-специфические особенности преобразования адресов на архитектуре PowerPC. Мы продолжим обсуждение того, как Linux использует (или игнорирует в целях портируемости) эти особенности начиная со старта системы и кончая инициализацией ядра.

8.3.1.1 Режим реальной адресации

Начиная со встраиваемых и заканчивая высокопроизводительными все процессоры PowerPC выполняют аппаратную перезагрузку в **реальном режиме (real mode)**[1]**.** Режим реальной адресации PowerPC переводит процессор в режим выключенного преобразования

[1] **Даже 440-я серия процессоров, технологически не обладающая реальным режимом, стартует с «тенью» TLB, отображающей линейные адреса в физические.**

адресов. Преобразования адресов контролируется **instruction relocate (IR)**[1] и **data relocation(DR)**[2] битами в **Machine State Register(MSR)**[3]**.** Для извлекаемых инструкций, если бит *Ж* равен 0, **effective address (EA)**[4] равен реальному адресу. Для загрузки и хранения инструкций бит DR играет ту же роль в MSR.

MSR, иллюстрируемый на рис. 8.3, является 64- или 32-битовым регистром, описывающим текущее состояние процессора. На 32-битовых реализациях IR и DR занимают биты 26 и 27.

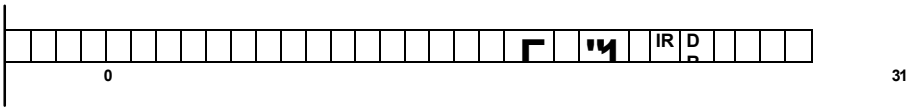


Рис. 8.3. Регистр состояния машины PowerPC (MSR)

Так как преобразование адресов в Linux представляет собой совокупность аппаратных и программных структур, реальный режим является фундаментальным для загрузки процесса, инициализирующего подсистему управления памятью, и управляющих памятью структур в Linux. Необходимость включения преобразования адресов поясняется наследственными ограничениями реального режима. Реальный режим более подходит для адресации определенного диапазона адресов, *для* 64- и (чаще всего) 32-битовых приложений. Существуют следующие два основных ограничения:

- операции загрузки-сохранения не защищены аппаратными средствами;
- любой доступ (инструкции или данные) к адресу или из адреса, не относящегося к физическому устройству, может вызвать машинную проверку (Machine Check) [также известную как проверочная остановка (Checkstop)], из КОТороq в большинстве случаев невозможно восстановиться.

8.3.1.2 Преобразование адресов

Недостатком преобразования адресов является реальная адресация. Преобразование адресов открывает двери для виртуальной адресации, при которой любой возможный адрес не является физически доступным для данного экземпляра, но благодаря хитрому использованию аппаратуры и программного обеспечения каждый возможный адрес может стать виртуально доступным для доступа.

При включенном преобразовании адресов архитектура PowerPC преобразует EA одним из двух способов: с помощью сегментного преобразования адресов (Segment Ad-

[1] **Перераспределитель инструкций.** ***Примеч. пер.***
[2] **Перераспределитель данных.** ***Примеч. пер.***
[3] **- Регистр состояния машины.** ***Примеч. пер.***
[4] **Эффективный адрес.** ***Примеч. пер.***

dress Translation) или **блочного преобразования адресов (Block Address Translation)** (рис. 8.4). Если преобразование EA возможно обеими методами, предпочитается блочное преобразование адресов, которое включено, когда значение MSR_{IR}=1, или MSR_{DR}=1, или оба равны 1. Сегментное преобразование адресов разбивает виртуальную память на сегменты, разделяемые на страницы по 4 Кб, представляющие физическую память. Блочное преобразование адресов разбивает память на области размером от 128 до 256 Мб.

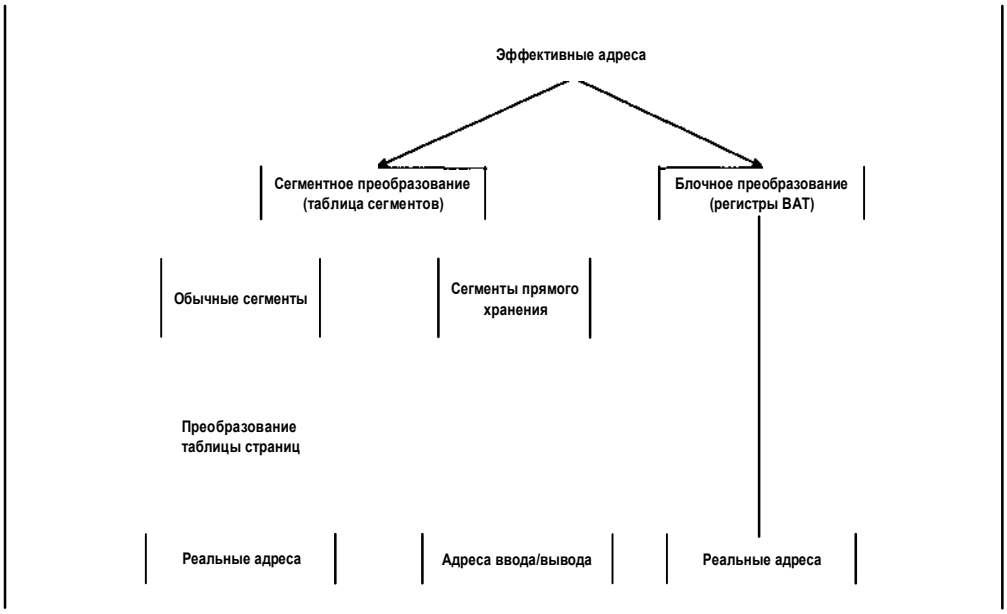


Рис. 8.4. Тридцатидвухбитовое преобразование адресов

Терминология преобразования адресов

Когда мы обращаемся к памяти, у нас есть только два метода обращения: реальная адресация, когда каждый элемент адреса описывает базовую единицу (обычно байт) физической памяти, и виртуальная адресация, когда адрес вычисляется в аппаратуре и/или программе. Вот несколько примеров, использования каждого метода:

- **Реальная адресация.** Физическая память, шины.
- Виртуальная адресация. Эффективные, защищенные и преобразованные адреса.

На PowerPC эффективное адресное пространство считается подмножеством виртуального адресного пространства. Термины *линейный, плоский* или *логический* применимы к обоим методам.

Сегментное преобразование адресов: прямое сохранение сегмента T

Следующий уровень преобразования определяется битом T, который находится в **регистре сегмента.** В PowerPC серии 7xx биты 0:3 EA выбирают один из 16 регистров сегментов (SR). Регистры сегментов продемонстрированы на рис. 8.5.

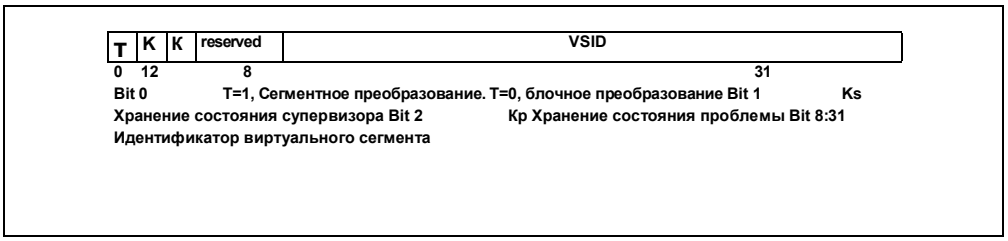


Рис. 8.5. Регистры сегментов

Когда бит T установлен, сегмент считается прямым сегментом хранения для устройства ввода-вывода и не связан с таблицей аппаратных страниц. Адрес ввода-вывода состоит из бита разрешения, BUID, контроллероспецифического поля и битов 4:31 EA. Linux не использует сегментацию прямого хранения.

Когда обычный сегмент T сегментного преобразования адресов не установлен, используется поле **virtual segment ID (VSID)[1].**

В соответствии с рис. 8.6,52-битовый **virtual address (VA)[2]** формируется объединением битов 20:31 EA (отступа внутри данной страницы), битов 4:19 EA и битами 8:31 поля VSID выбранного регистра сегмента. Наиболее важные 40 бит VA образуют **virtual page number (VPN)[3].** Архитектура PowerPC использует таблицу хешированных страниц для отображения CPN в реальные числа (реальные адреса желаемой страницы в памяти). Функция *hash* использует VPN и значение **Storage Description Register 1 (SDR1)[4]** для хранения и получения **Page Table Entry (PTE)[5].** PTE, изображенное на рис. 8.7, представляет собой 8-байтовую структуру, содержащую все необходимые атрибуты страницы в памяти.

Блочное преобразование адресов

Как следует из его имени, **Block Address Translation (BAT)[6],** - это механизм адресации, позволяющий манипулировать последовательными блоками памяти от 125 Кб до 256 Мб. Регистры BAT на архитектуре PowerPC - это привилегированные **special purpose registers (SPRs)[7].** Рис. 8.8 иллюстрирует регистры BAT.

[1] - Идентификатор виртуального сегмента. *Примеч. пер.*

[2] Виртуальный адрес. *Примеч. пер.*

[3] Виртуальный номер страницы. *Примеч. пер.*

[4] Регистр описания хранилища 1. *Примеч. пер.*

[5] Элемент таблицы страниц. *Примеч. пер.*

[6] - Блочное преобразование адресов. *Примеч. пер.*

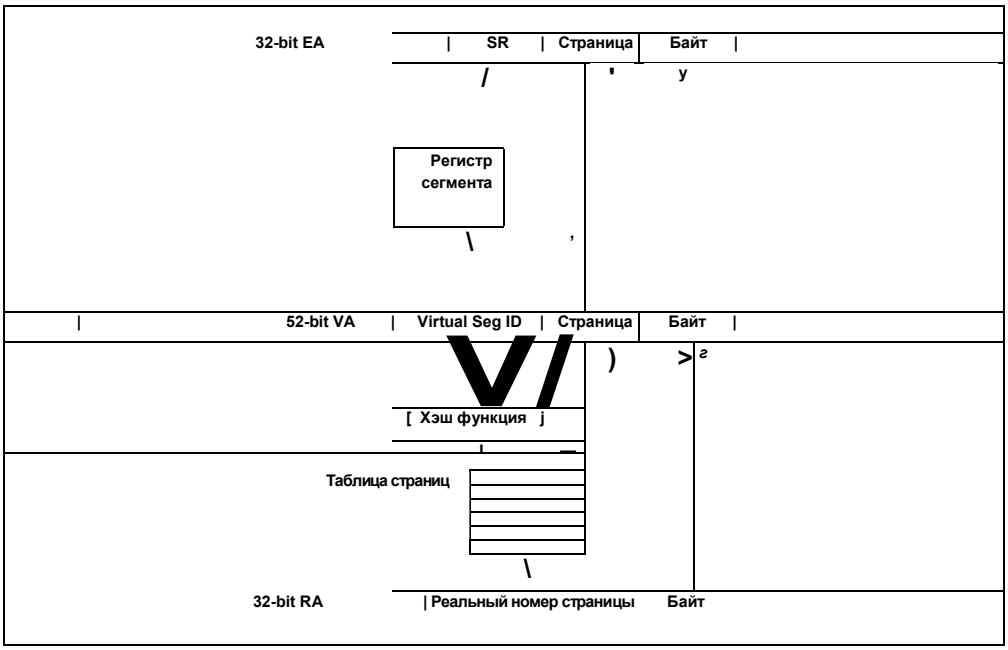
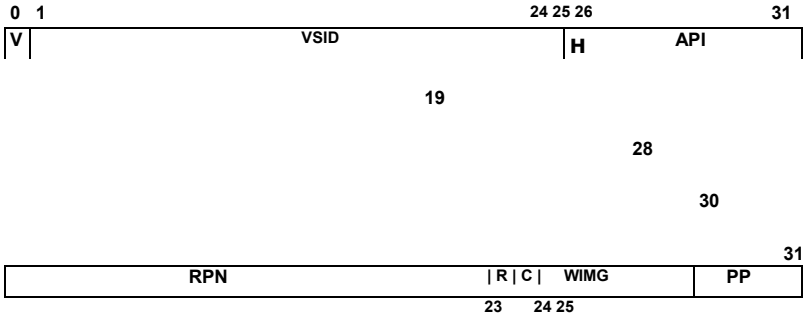


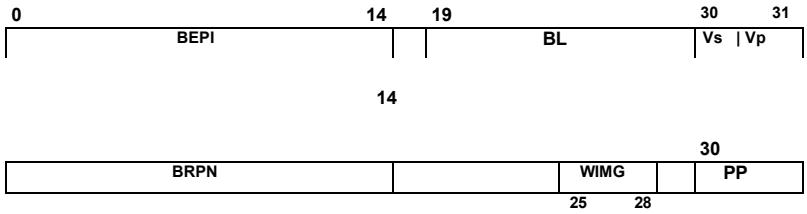
Рис. 8.6. Трансляция сегмента



Слово О	Бит О Биты	V=1, Корректный элемент, V=0. Некорректный элемент
	1:24 Бит 25	Идентификатор виртуального сегмента
	Биты 26:31	Идентификатор хэш функции
	Bits 0:19	Аббревиатурный индекс страницы
Слово 1	Биты 20:22	Реальный номер страницы
	Бит 23 Бит	Зарезервированы
	24 Биты	Бит ссылки
	25:28 Биты	Бит изменения
	30:31	WIMG
		Защита страницы

Рис. 8.7. Элемент таблицы страниц

■ **Регистры специального назначения.** ***Примеч. пер.***



Верхний регистр	Биты 0:14	Эффективный индекс блока страницы
	Биты 15:18, 25	Зарезервированы
	Биты 19:29	Длина блока
	Бит 30 Бит 31	Состояние корректности супервизора
		Состояние корректности проблемы
Нижний регистр	Биты 0:14	
	Биты 15:24	Реальный номер блока страницы
	Биты 25:28	Зарезервированы
	Бит 29 Биты	WIMG (см. сноску)
	30:31	Зарезервирован
		Биты защиты для области BAT

Рис. 8.8. Регистр BAT

Формирование реального адреса из регистра BAT можно увидеть на рис. 8.9. Четыре регистра **Instruction BAT (IBAT)**[1] и четыре регистра **Data BAT (DBAT)**[2] можно читать с помощью PPC-инструкций *mtspr* и *mfspr**.

Буферы просмотра преобразований

Translation Lookaside Buffers (TLBs) можно рассматривать как аппаратный кеш с аппаратной защитой системы страниц. Длина TLB зависит от архитектуры PowerPC, а содержит она индекс наиболее используемых PTE. Программа работы со страницами должна обеспечивать синхронизацию TLB с таблицей страниц. Когда процессор не может найти страницу в хеш-таблице[4], выполняется поиск в таблице страниц Linux. Если страница больше не найдена, генерируется обычная ошибка страницы. Информацию об оптимизации синхронизации таблицы страниц Linux и хеш-таблицы PowerPC можно найти в документе *Низкоуровневая оптимизация в ядре PowerPC/Linux* Пола Макераса[5].

• Инструкции BAT. *Примеч. пер. [2]

BAT-данные. *Примеч. пер.*

[3] **- Блочное преобразование адресов реализовано не на всех процессорах PowerPC. В частности, оно не реализовано на G4 и G5. Оно реализовано во встроенных процессорах 4xx.**

[4] **Хеш-таблица для процессоров PowerPC не реализована. Она отсутствует на встроенных системах 4xx и 8xx, где ошибка TLB генерирует аппаратное и программное исключение, а затем выделяет новую страницу.**

[5] **Low level Optimization in the PowerPC/Linux Kernels by Paul Mackerras. *Примеч. пер.***

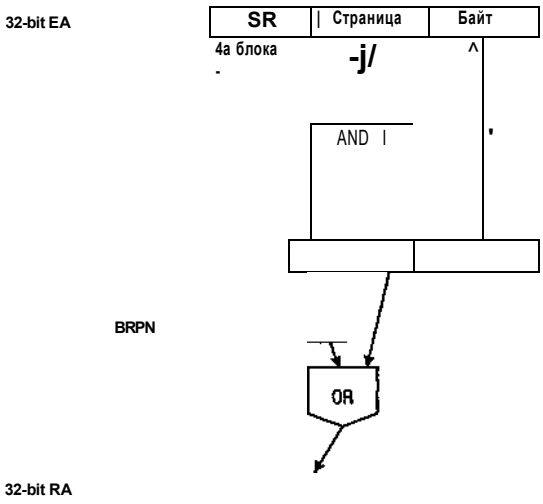


Рис. 8.9. Реальный BAT

Режим управления доступом к хранимому

При включенном преобразовании адресов ($MSRi_R$=l, и/или MSRrjR=l) и используемом сегментном преобразовании адресов или блочном преобразовании адресов режим хранения определяется четырьмя битами: W, I, M и G. Для сегментного преобразования адресов это биты 25:28 второго слова PTE и те же биты для второго SPR DBAT. (Бит G заносится в IBAT.) Для сегментного преобразования адресов доступно еще два бита - ссылка и управление, расположенные в PTE. Биты R и C устанавливаются аппаратным, либо программным путем. (См. следующую вставку, где обсуждаются биты W, I, M, G, RHC.)

8.3.1.3 Как Linux использует преобразование адресов на PPC

Мы знаем, как выглядит код, выполняющий управление памятью на PPC.

Следующий код является первым, получающим управления в ядре. Эта функция выполняет обратный вызов Firmware для выделения временной области с помощью функции claim (). Далее ядро распаковывается в соответствующее место.

arch/ppc/boot/openfirmware/newworldmain.c
```
40      void boot(int al,  int a2,  void  *prom)

54      claim(initrd_start, RAM_END - initrd_start, 0);
```

Управляющие биты

Биты W, I, M, G, R и C управляют доступом процессора к кешу и основной памяти:

- W (Write Thought, повсеместная запись). Если данные находятся в кеше и над ними производится операция сохранения, при W=l копия в памяти также обновляется.
- I (Cache Ingibit, запрет кеша). Обновление минует кеш и обращается к основной памяти напрямую.
- M (Memory Coherence, согласование памяти). При W=l выполняется принудительно-аппаратное согласование памяти.
- G (Guarded, безопасность). При G=l конкурентное выполнение запрещено.
- R (Referenced, ссылка). При R=l считаются ссылки на вхождения таблицы страниц.
- C (Changed, изменение). При C=1 вхождения таблицы страниц изменяются.

```
55      printf ("initial  ramdisk moving  Ox%x <-   Ox%p   (%x bytes) \n\r",
56                initrd_start/   (char  * ) ( &  _________ramdisk_begin) ,   initrd_size);
57      memcpy( (char  *) initrd_start,    (char  *) (&  _____________ramdisk_begin),
                                                          initrd_size);

63      /*  выделение  3MB начиная  с  PROG_START  */
64            claim(PROG_START,   PROG_SIZE/   0 ) ;
65            dst   =   (void  *)   PROG_START;
66            if   (im[0]   ==   Oxlf  &&  im[l]   ==   0x8b)    {
67      /*  выделение  памяти для  временного рабочего пространства  */
68      avail_ram =   (char   *)   claim(0,   SCRATCH_SIZE/   0x10);
69      begin_avail   =   avail_high =   avail_ram;
70      end_avail   =   avail_ram +   SCRATCH_SIZE;
71      printf("heap  at  0x%p\n",   avail_ram);
72      printf("gunzipping   (0x%p   <-   0x%p:0x%p)...",   dst,   im,   im+len) ;
73      gunzip(dst,   PROG_SIZE,   im,   &len);
74      printf("done  %u bytes\n^H,   len);
75      printf("%u bytes  of heap consumed,   max in use %u\n"#
76                avail_high   -   begin_avail,   heap__max) ;

86      sa   =   (unsigned  long)PROG_START;
87            printf("start  address   =   0x%x\n",   sa);
88
89             (*(kernel_start_t)sa)(al,   a2,   prom);
```

Строка 40

Входной точкой файла является функция boot (al, a2, *prom).

Строка 54

Функция claim () вызывается для выделения памяти сразу за 1 Мб и копирования диска в памяти (ramdisk) в эту память.

Строка 64

Функция claim () вызывается для выделения 3 Мб начиная с 0x1_0000 для образа.

Строка 68

Функция claim () вызывается для выделения 8 Кб памяти начиная с 0x00 для временной кучи.

Строка 73

Образ распаковывается в адрес 0xl_0000 (PROG_START).

Строка 89

Переход на 0x1_0000 [(*kernel_start_t) sa] с параметрами (al, a2 и prom), где al хранит значение в г3 (равное загрузочному ramdisc start), a2 хранит значение в г4 (равное размеру загрузочного диска или Oxdeadbeef в случае по ramdisk) и prom хранит значение в г5 (код хранится в системной памяти).

Следующий блок кода подготавливает аппаратные особенности управления памятью для различных процессоров PowerPC. Первые 16 Мб памяти отображаются в 0x0000000:

```
arch/ppc/kernel/head.S

131    _ start:

150   bl  early_init  in <arch/ppc/kernel/setup.c> (283)

170    bl  mmu_off

171     RFI: SRR0=>IP, SRRl=>MSR
172    tifndef CONFIG POWER4
173     bl  clear bats
174     bl  flush tlbs
175
176     bl  initial bats
177    #if !defined(CONFIG APUS) && defined(CONFIG BOOTX TEXT)
178     bl  setup disp bat
179    #endif
180    #else /* CONFIG_POWER4 */
181     bl  reloc offset
182     bl  initial mm power4
183    #endif /* CONFIG_POWER4 */
```

```
185     /*
186      * Вызов setup cpu для процессора 0 инициализация бхх Idle
187     */
188    Ы  reloc offset
189    li  r24,0   /* cpu# */
190    Ы  call_setup_cpu  /* Вызов setup_cpu для этого процессора */
195   #ifdef CONFIG POWER4
196    bl  reloc_offset
197    bl  init idle power4
198   #endif /* CONFIG_POWER4 */
199
210    bl  reloc_offset
211    mr  r26,r3
212    addis   r4,r3,KERNELBASE@h  /* current address of _start */
213     cmpwi  0,r4,0   /мы уже работаем в 0?
214     bne   relocate_kernel 215

224     turn  on mmu:
225     mfmsr             rO
226     ori       rO,rO,MSR DR|MSR IR
227     mtspr           SRRl,rO
228     lis         rO,start here@h
229     ori        rO,rO,start here@l
230     mtspr           SRR0,r0
231     SYNC
232     RFI         /*  включение MMU */
```

Строка 131

Это точка входа в код. Получает установленное mnu окружение. (Обратите внимание, что APUS расшифровывается как Amiga Power Up System.)

Строка 150

Адрес, куда распаковывается ядро, и с чем оно связывается могут отличаться. Функция early_init возвращает физический адрес текущего кода.

Строка 170

Отключение модуля управления памятью на PPC. Если включены IR и DR, он остается включенным; в противном случае переразмещение отключено.

Строки 173-176

Если процессор не является процессором power4 или G5, то регистры BAT очищаются, сбрасываются TLB, а BAT настраивается для отображения первых 16 Мб памяти в OxcOOOOOOO.

Обратите внимание, как внутри ядра используются различные метки для памяти ядра.

arch/ppc/defconfig
CONFIG_KERNEL_START=0xc0 0 0 0 0 0 0

include/asm-ppc/page.h
#define PAGE_OFFSET CONFIG_KERNEL_START
#define KERNELBASE PAGE_OFFSET

Строки 181-182

При использовании сегментации настраивается память ядра для power4 и G5.

Строки 188-198

setup_cpu () инициализирует особенности ядра и использования, такие, как настройка кеша и наличие FPU и MMU. (Примите во внимание, что на момент написания книги init__idle_power4 ничего не выполняет.)

Строка 210

Перевыделение ядра в KERNELBASE или 0x00 в зависимости от платформы.

Строки 224-232

Включение MMU (если он еще не включен) с помощью включения IR и DR в MSR. Далее выполняется инструкция RFI, производящая переход к метке start_here:. (Внимание: инструкция RFI загружает в MSR содержимое SRR1 и переходит к SRR0.)

Далее стартует следующий код. Он настраивает всю память системы на основе строки команд:

```
arch/ppc/kernel/head.S
1337     start_here:

13 64  bl    machine_init
13 65  bl    MMU_init

1385  lis     r4,2f@h
1386  ori     r4,r4,2f@l 13
87  tophys(r4,r4)
13 88  li     r3,MSR_KERNEL & -(MSR_IR|MSR_DR)
```

```
1389  FIX_SRR1(r3 , r5)
1390  mtspr     SRR0,r4
1391  mtspr     SRR1,r3
1392  SYNC
1393  RFI
1394  /* Загрузка контекста ядра */
1395  2:     bl     load_up_mmu

1411  /* Now turn on the MMU for real! */
1412  li     r4,MSR_KERNEL
1413  FIX SRR1(r4,r5)
1414  lis    r3,start_kernel@h
1415  ori    r3,r3,start_kernel@l
1416  mtspr     SRR0,r3
1417  mtspr     SRR1,r4
1418  SYNC
1419  RFI
```

Строка 1337

Это строка является точкой входа для этого кода.

Строка 1364

machine_init () (см. файл arch/ppc/kernel. setup. c, строка 532) настраивает машинно-зависимую информацию, такую, как NVRAM, L2, количество линий кеша CPU, отладка и т. д.

Строка 1365

MMU_init() (см. файл arch/ppc /mm/ init .c, строка 234) определяет общий размер для highmem и lowmem. Далее она инициализирует аппаратный MMU (MMU_init_hw(), строка 267), настраивает таблицу хеша страниц (arch/ppc/ mm/hashtable.s), отображает всю память в KERNELBASE (mapin_ram(), строка 272), отображает весь ввод-вывод (setup_io_mapping(), строка 285) и инициализирует управление контекстом (mmu_context_init (), строка 288).

Строка 1385

Отключение IR и DR для включения SDR1. Хранит реальный адрес таблицы страниц и то, сколько битов из хеша используется в индексе таблицы страниц.

Строка 1395

Очистка TLB, загрузка SDR1 (основа хеша таблиц и размер), устанавливает сегментацию и в зависимости от конкретной платформы PPC инициализирует регистры BAT.

Строки 1412-1419

Включение IR, DR и RFI для start_kernel в /init/main. c. Обратите внимание, что в момент прерывания на архитектуре PowerPC содержимое Instruction Address Register (ISR)[1] хранит адреса, которые должен возвращать процессор после обслуживания прерывания. Это значение сохраняется в Save Restore Register 0 (SRR0)[2]. Регистр состояния машины, в свою очередь, сохраняется в Save Restore Register 1 (SRR1). Коротко говоря, во время прерывания:

- **IAR->SRR0**
- **MSR->SRR1**

Инструкция RFI, которая обычно выполняется в конце функции прерывания, является обратной процедурой, когда SRR0 восстанавливается в IAR и SRR1 восстанавливается в MSR. Коротко говоря:

- **SRR0->IAR**
- **SRR1->MSR**

Код в строках 1385-1419 использует эту методику для включения управления памятью и выключения за три шага:

1. Установка желаемого бита для MSR (в соответствии с рис. 8.1) в SRR1.
2. Запись желаемого адреса, куда мы хотим перейти в SRR0.
3. Выполнение инструкции RFI.

8.3.2 Управление памятью на x86-аппаратуре

При включении все процессоры Intel работают в реальном режиме адресации. Реальная адресация - это режим, совместимый с ранними Intel процессорами. По мере усложнения процессоров наследуемый код всегда использовался в новых процессорах для совместимости. В реальном режиме адресации процессор может выполнять программы, написанные для 8086 и 8088 с использованием тех же самых инструкций и, что гораздо важнее, с помощью того же метода адресации или преобразования адресов (address **translation).** В результате преобразования адресов процессор получает доступ к системной памяти. Ранние процессоры Intel имели 20-битовую адресную шину, через которую они могли адресовать до 64 Кб памяти. Это ограничение накладывалось и на код ранних систем. В реальном режиме адресации *линейные адреса* равны *физическим адресам.* По мере нашего продвижения по коду инициализации менеджера памяти мы увидим и другие особенности последних процессоров, используемые в аппаратных и более комплексных структурах вдобавок к программным.

■ Регистр адресов инструкций.** ***Примеч. пер.

[2] Регистр сохранения-восстановления. ***Примеч. пер.***

Код в setup. S выполняет несколько важных функций, связанных с инициализацией памяти.

```
arch/i386/boot/setup.S
3 07                  #define SMAP  0x534d4150
308
3 09                            meme820:
310    xorl    %ebx, %ebx   # последовательный счетчик
311    movw  $E820MAP, %di   # указатель в списке записи
312              # так что мы можем иметь прямую запись в bios.
314
315    jmpe820:
316     movl  $0x0000e820, %eax # e820, верхнее слово обнулено
317     movl  $SMAP, %edx   # ascii 'SMAP'
318     movl  $20, %ecx   # размер e820rec
319     pushw %ds  # запись данных.
320     popw %es
321     int  $0x15   # вызов
322     jc bail820  # сброс e801, если произошла неудача 323
324     cmpl  $SMAP, %eax # проверка, возврата из 'SMAP'
325     jne bail820   # сброс в e801 если произошла неудача 326

333    good820:
334     movb  (E820NR), %al    # до 32 элементов
335     cmpb  $E820MAX, %al
336     jnl  bail820
337
338                          incb  (E820NR)
33 9                         movw %di, %ax
340     addw $20, %ax
341     movw %ax, %di
342    again820:
343     cmpl  $0, %ebx       # проверка установлен ли #
344     jne  jmpe820        %ebx в EOF
345    bail820:
```

Строки 307-345

Если посмотреть на этот блок кода, мы сначала увидим (в строке 321) вызов функции BIOS intl5h с ax= 0xe82 0. Она возвращает адрес и длину различных типов памяти, распознанных BIOS. Эта простая карта памяти представляет простой пул, из которого выбираются все страницы памяти в Linux. Как видно из рассмотренного

ранее кода, карту памяти можно получить тремя способами: 0xe820, 0x801 и 0x88. Все эти три метода совместимы с существующими BIOS и соответствующими платформами.

```
arch/i386/boot/setup.S
595  # Теперь мы переносим систему в ее рабочее место, но проверяем
# наличие большого ядра. В этом случае мы не должны его перемещать..
   597  testb     $LOADED HIGH, %cs:loadflags
598 jz  do_move0   # .. когда у нас есть нижний
599         # загруженный zImage
600         # .. или в противном случае верхний загруженный
602      bzlmage jmp    end_move  # ... и продолжаем перемещение
603
604 do move0: movw
605 $0x100,
606                  %ax
607  movw                  # начало сегмента назначения %cs, %bp
608  subw    # как SETUPSEG $DELTA INITSEG, %bp   # как INITSEG
609  movw    %cs:start_sys_seg, %bx  # начало сегмента источника
610  eld
611  do move:
612  movw    %ax, %es   # сегмент назначения
613  incb    %ah        # вместо add ax,#0x100
614  movw    %bx, %ds   # сегмент-источник
615  addw    $0x100, %bx %di, %di %si, %si
616  subw    $0x800, %cx
617  subw
618  movw
619  rep     %bp, %bx # представим start sys seg > 0x200,
620  movsw               # поэтому мы, возможно, читаем на одну строку
621  empw                больше
622                      # чем нужно, но никогда не перезаписываем
625              INITSEG, так как
626              # назначение - это минимум одна страница ниже источника
627              j b do_move
628 end_move:
```

Строки 595-628

Этот код представляет собой образ ядра, созданный build, с и загруженный LILO. Он выполняется для сектора init (по адресу 0x9000), сектора setup (по адресу 0x9200) и сжатого образа. Образ изначально загружается по адресу 0x10000. Если он LARGE (>0x7FF), он остается на месте; в противном случае перемещается в 0x1000.

```
arch/  i386/boot/setup.S
723     # Попытка включения A20 через контроллер клавиатуры
724    #endif /* CONFIG X86 VOYAGER */ a20 kbc:
725     call    empty_8042
726
727    #ifndef CONFIG_X86_VOYAGER call  a2 0_test   #
728     Только в случае работы BIOS jnz  a20 done    #
729     но с отложенной реакцией.
730    #endif
731
               # команда записи
       movb    $0xDl,  %al  %al,
       outb    $0x64 empty_8042
       call
               $0xDF, %al %al, $0x60 # A20 включена
       movb    empty_8042
       outb
       call
732
733
734
735
736
737
738
739
```

Строки 723-739

Этот код представляет собой возвращение к старым процессорам Intel. В настройке менеджера памяти он выглядит немного неуместным.

```
arch/i386/boot/setup.S 790    #
    настройка gdt и  idt
791 lidt    idt_48     #  загрузка  idt  с  0,0
792 xorl    %eax,  %eax                 #  вычисление gdt_base
793 movw    %ds,  %ax                  #  (Преобразование  %ds:gdt  в линейный ptr)
794 shll    $4,  %eax
795 addl    $gdt, %eax
796 movl    %eax, (gdt 48+2)
797 lgdt gdt_48     # загрузка в gdt соответствующего значения

981     gdt:
982       .fill GDT__ENTRY_BOOT_CS ,8,0
983
984       .word  OxFFFF      # 4Gb - (0x100000*0x1000      4Gb)
985       .word 0            # базовый адрес = 0
986       .word 0x9A00      # чтение-выполнение кода
987       .word  OxOOCF      # гранулярность = 4096, 386
                                 #  (+5-й полубайт предела)
        .word  OxFFFF       # 4Gb       (0x100000*0x1000 = 4Gb)
988
989
990
```

Формирование 20-битового физического адреса в режиме реальной адресации Intel

Процессор Intel 8088 в оригинальном IBM PC имел только 20 линий адресации [0...19]. Это позволяло системе адресовать до 1 Мб плюс приблизительно 64 Кб внутренней памяти (от 0 до 0xl0_FFEF), но *физически* (через шину) последние 64 Кб адресуемой памяти были на самом деле *первыми* 64 Кб реальной памяти!

Внутри процессора 20-битовый адрес формировался из 16-битового сегмента селектора и 16-битового сегмента отступа. Селектор сдвигался на 4 байта и добавлялся к отступу, расширяя его на 4 бита. Сумма этих регистров равнялась физическому адресу, видимому через шину.

Например, для получения высших адресов мы загружаем в сегмент селектора (CS, DS, ES и т. д.) значение **OxFFFF,** а значение **OxFFFF** в регистр индекса (SI, DI и т. д.). Внутри процессора сегмент селектора сдвигается на 4 бита и добавляется к отступу.

OxFFFF сдвигается на 4 6nTa=0x0F_FFF0
Добавляется oTCTyn+0x00_FFFF Внутренняя
cyMMa=0xl0_FFEF Внешний физический
aflpec=0x00_FFEF

В результате физический адрес равен сегментному селектору со значением **0x0000** и отступу со значением **OxFEFF (0000:FFEF).**

Доступ к наивысшему адресу и выше спустит нас в OxFFEF. Некоторые написанные для этого процессора программы зависят от этого 20-битового циклического поведения. Представленные процессоры Intel 286 и более поздние с более широкими шинами адресов использовали реальную адресацию для сохранения совместимости с 8088 и 8086. Режим реальной адресации не учитывал потребностей старых, зависящих от циклического 20-битового эффекта программ. Была добавлена сигнальная дорожка A20M# для имитации этой «особенности» ранних процессоров. Получение этого сигнала маскировало сигнал A20 для доступа к нижней памяти.

Для включения и выключения сигнала A20 использовался логический вентиль. Оригинальный дизайн получения этого сигнала предусматривал использование дополнительного ввода-вывода от контроллера клавиатуры, управляемого портами ввода-вывода 0x60 и 0x64. Позднее был разработан «быстрый вентиль A20», использовавший порт 0x92, встроенный в материнскую плату. Так как все процессоры x86 сбрасываются в реальном режиме адресации, при загрузке имеет смысл включить режим A20 одним из двух доступных способов.

```
991    .word 0           # базовый адрес = 0
992    .word  0x9200     # чтение-запись данных
993    .word  OxOOCF    # гранулярность = 4096, 386
994         #  (+5-й полубайт предела)
995  gdt end:
996    .align 4
997
998    .word  0      # байт выравнивания
999  idt_48:
```

```
1000  .word  0          # ограничение idt = 0
1001  .word  0, 0
1002                    # основа idt = 0L
1003  .word  0
1004 gdt_48:            # байт выравнивания
1005  .word  gdt_end
1006  .word  0, 0       - gdt - 1    # ограничение gdt
                          # основа gdt (заполняется позже)
```

***Строки** 790-797*

Структуры и данные для временных GDT и ЮТ компилируются в конце setup. S. Эти таблицы реализованы в своей простейшей форме.

Строки 981-1006

Эти строки являются откомпилированными значениями временного GDT, который имеет код и описатель данных, каждый из которых соответствуют 4 Гб памяти, начиная с 0x00. ГОТ инициализируется в 0x00 и заполняется позднее.

На этом этапе разворачивания менеджера памяти для платформы Intel одной из важнейших фаз является переход в защищенный режим. В этой точке аппаратура начинает построение пространства виртуальных адресов для операционной системы.

Защищенный режим

Метод управления памятью Intel называется защищенным режимом. Защита относится к множеству независимых сегментированных адресных пространств, защищенных друг от друга. Другой половиной менеджера памяти Intel являются страницы и их преобразование. Системные программисты могут использовать различные комбинации сегментации с разделением на страницы, однако Linux использует плоскую модель, где сегментация не учитывается. В плоской модели каждый процесс может адресовать полное 32-битовое пространство (4 Гб).

```
arch/i386/boot/setup.S
                        бит защищенного режима (PE)
                        Вот оно!
830 movw  $1, %ax    # би
831 lmsw  %ax        # Во
832 jmp   flush_instr
833                         Флаг для обозначения загрузки
834  flush_instr:           Указатель на код реального режима
835 xorw  %bx, %bx       #
836 xorl  %esi, %esi     #   %si
837 movw  %cs, %si
838 subw  $DELTA_INITSEG, %si
839 shll  $4, %esi
```

Строки 830-831

Установка бита PE в слове состояния машины для входа в защищенный режим. Инструкция jmp начинает выполнение в защищенном режиме.

Строки 834-839

Сохранение 32-битового указателя в защищенном режиме для распаковки и загрузки ядра позже в startup_32 ().

Вспомните, что в реальном режиме адресации код выполняется с помощью 16-битовых инструкций. Текущий файл компилируется с помощью ассемблерной директивы . codel6, включающей данный режим, также известный как 16-битовый модуль в Intel Programmer's Reference. Для перехода от 16-битового модуля к 32-битовому модулю архитектура Intel (и магия ассемблера) позволяет строить 32-битовые инструкции в 16-битовом модуле.

Построение и выполнение 32-битового перехода:

```
arch/i3 8б/boot/setup.S
841   # переход к startup 32 в arch/i38б/kernel/head.S
842   #
843   # ПРИМЕЧАНИЕ. Для большой загрузки ядра в верхнюю память нужно
844   #  jmpi    0x100000, __BOOT_CS
846   #  но мы еще не загрузили регистр CS, так что размер по умолчанию
847   #  отступа задачи до сих пор 16-битовый.
848   # Однако при использовании префиксного операнда (Oxбб) процессор
849   #  корректно получит наш 48-битовый указатель. (INTeL 80386
850   #  Programmer's Reference Manual, Mixing 16-bit and 32-bit code,
851   #  страница 16-6)
852    .byte 0x66, Oxea   # prefix + jmpi-opcode
853   code32:  .long  0x1000     # устанавливается в 0x100000
854          # для большого ядра
855    .word      _ BOOT_CS
```

Строка 852

Эта строка строит 32-битовую инструкцию перехода.

После выполнения перехода система использует временный GDT, а код выполняется в 32-битовом защищенном режиме начиная с метки startup_32 в arch/i386 /kernel/ head. s, строка 57.

8.3.2.1 Защищенный режим

До этого момента обсуждение подбиралось к моменту, когда система Intel становится готовой к настройке страниц. По мере нашего продвижения по коду head. S мы увидели,

как происходит инициализация и как Linux использует x86-защищенный режим системы страниц. Это последний код, выполняемый перед стартом main. с ядра. За полной информацией о множестве доступных режимов и установок, связанных с инициализацией памяти и процессоров Intel, см. *Intel Architecture Software Developer Manual,* (Т. 3).

```
arch/i386/kernel/head.S
057                         ENTRY(s tartup_3 2)
058
059   /*
60     * Установка сегмента в известное значение.
61     */
62      eld
63      lgdt boot_gdt_descr -  _ PAGE_OFFSET
64      movl $(   BOOT DS),%eax
65      movl %eax,%ds
66      movl %eax,%es
67      movl %eax,%fs
68      movl %eax,%gs
068
081    /*
82      * Инициализация таблиц страниц. Создает PDE и настраивает таблицы
83      * страниц, расположенные сразу за  end. Переменная
84      * init pg tables end устанавливается указывающей на первую
85      * "безопасную" позицию. Отображения создаются в виртуальном
86      * адресе 0 (тождественное отображение)и PAGE OFFSET до
87      * _end+sizeof(page tables)+INIT_MAP_BEYOND_END.
88      * ВНИМАНИЕ! Не используйте %esi или стек в этом коде. Однако %esp
89       * может быть использован как GPR, если вам это необходимо ...
90      */
091      page pde offset = (   PAGE OFFSET » 20);
092
93       movl $(pg0 -    PAGE OFFSET), %edi
94       movl $(swapper pg dir -    PAGE OFFSET), %edx
95       movl $0x007, %eax/* 0x007 = PRESENT+RW+USER */
096    10:
97       leal 0x007(%edi),%ecx  /* Создание элемента PDE */
98       movl %ecx,(%edx)/* Сохранение единичного элемента PDE */
99       movl %ecx,page_pde_of fset (%edx) /* Сохранение элемента PDE ядра*/
100      addl $4,%edx
101      movl $1024, %ecx
102    11:
103      stosl
104      addl $0x1000,%eax
105      loop lib
```

```
106      /* Конечное состояние: мы должны выполнить отображение до /*
         INIT MAP BEYOND END включительно */
107      /* и до конца наших таблиц страниц; +0x007 - бит атрибутов */
108     leal (INIT MAP BEYOND END+0x007)(%edi),%ebp
109      cmpl %ebp,%eax
110      jb 10b
111     movl %edi,(init pg tables end -   PAGE OFFSET)
112
113    #ifdef CONFIG_SMP

156     3:
157     #endif /* CONFIG SMP */
158
159     /*
160      * Включение подкачки
161      */
162      movl $swapper_pg_dir- _ PAGE_OFFSET,%eax
163      movl %eax,%cr3     /* настройка указателя таблицы страниц.. */
164      movl %cr0,%eax
165      orl $0x80000000,%eax
166      movl %eax,%cr0           /* ..установка бита страницы (PG) */
167      ljmp $   BOOT CS,$lf   /* Очистка выборки и нормализация %eip */
168     1:
169      /* Настройка указателя стека */
170      lss stack_start,%esp

177      pushl $0
178      popfl
179
180     #ifdef CONFIG SMP
181      andl %ebx,%ebx
182      jz  If                    /* Начальная очистка BSS */
183      jmp checkCPUtype
184     1:
185     #endif /* CONFIG SMP */
186
187     /*
188      * Настройка 32-битового запуска системы. Нам нужно переделать
189      * кое-что, что было выполнено в "реальном" 16-битовом режиме.
190      */
191        call setup idt
192
193     /*
194      *  Копирование загрузочных параметров.
195      * ПРИМЕЧАНИЕ.  %esi до сих пор указывает на данные реального режима.
196      */
```

```
197      movl $boot params,%edi
198      movl $(PARAM SIZE/4),%ecx
199      eld
200      rep
2 01     movsl
2 02     movl boot params+NEW CL POINTER,%esi
203      andl %esi,%esi
2 04     jnz 2f                  # Протокол новой командной строки
205      empw $ (OLD CL  MAGIC) , OLD CL MAGIC ADDR
206      jne If
207      movzwl OLD  CL OFFSET, %esi
208      addl $(OLD CL BASE ADDR),%esi
209     2:
210      movl $saved command JLine,%edi
211      movl $(COMMAND LINE SIZE/4),%ecx
212      rep
213      movsl
214     1:
215      checkCPUtype:

279      lgdt cpu_gdt_descr
280      lidt idt_descr

303    call start_kernel
```

Строка 57

Эта строка представляет собой точку входа в 32-битовом режиме в код ядра. Сейчас этот код использует временные GDT.

Строка 63

Этот код инициализирует GDT базовыми адресами загрузочного GDT. Загрузка GDT представляет собой то же самое, что и GDT, используемая в setup.S (4 Гб кода и данных начиная с адреса 0x00000000), и применяется только для загрузки кода.

Строки 64-68

Инициализация оставшихся сегментов регистров с __ BOOT_DS начиная с 24 (см. /include/asm-i386/segment.h). Это значение указывает на селектор 24h (начиная с 0) в финальном GDT, устанавливаемом далее в этом коде.

Строки 91-111

Создание записи директории страниц (PDE) в swapper_pg_dir, которая относится к таблице страниц (рдО) с записями, начинающимися с виртуального адреса 0 (тождественные), и с записями-копиями PAGE_OFFSET (память ядра).

Строки 113-157

Этот блок кода инициализирует дополнительные (незагрузочные) процессоры в таблице страниц. Для нашего обсуждения мы сфокусируемся на загрузочном процессоре.

Строки 162-164

Регистр сг3 является точкой вхождения для аппаратных страниц x86. Этот регистр инициализируется в точке на основе **Директории страниц (Page Directory),** находящейся в нашем случае в swapper_pg_dir.

Строки 165-168

Установка бита PG (страниц) в сгО загрузочного процессора. Бит PG включает механизм страниц x86. Инструкция перехода (в строке 167) необходима при смене бита PG для понимания, что все инструкции *в* процессоре сериализируются в момент вхождения или выхода из режима страниц.

Строка 170

Инициализация стека в начале сегмента данных (см. также строки 401-403).

Строки 177-178

Регистр ef lags является системным регистром чтения-записи, содержащим состояния прерываний, моделей и разрешений. Регистры очищаются нулем в стеке и прямо извлекаются в регистры с помощью инструкции popf 1.

Строки 180-185

Регистр общего назначения ebx используется в качестве флага для обозначения того, загрузочный ли процессор выполняет этот код. Так как мы прослеживаем этот код, выполняющийся как на загрузочном процессоре, ebx очищается (0) и мы переходим к вызову setup_idt.

Строка 191

Функция setup_idt инициализирует таблицу описателей прерываний (IDT), в которой каждой точке вхождения соответствует обработчик-лтустыш/ся. ГОТ обсуждается в гл. 7, «Планировщик и синхронизация ядра», и представляет собой таблицу функций (или *обработчиков),* вызываемых, когда процессору нужно выполнить критический по времени код.

Строки 197-214

Во время загрузки пользователь может передать Linux несколько параметров. Они сохраняются здесь и могут быть использованы позже.

Строки 215-303

Код, приведенный в этих строках, выполняет основную необходимую работу по проверке версии x86 процессора и некоторую дополнительную инициализацию.

С помощью инструкции cupid (или ее отсутствия) некоторые биты устанавливаются в регистрах ef lags и сгО. Одна из важных настроек, тип расширения (ET), содержится в бите 4 сгО. Этот бит означает поддержку инструкций математического сопроцессора на старых, x86-процессорах. Наиболее важными строками в этом блоке являются строки 279-280. Именно здесь IDT GDT загружаются (с помощью инструкций lidt и lgdt) в регистры idtr и gdtr. И наконец, в строке 303 мы переходим в функцию start_kernel ().

С кодом из head. S система может теперь отображать **логические** адреса в **линейные** адреса, а затем и в **физические** адреса (рис. 8.10). Начиная с **логических адресов селектор** (в регистрах CS, DS, ES и т. д.) ссылается на **описатели** в **GDT. Отступ** является искомым нами плоским адресом. Информация из **описателя** и **отступа** комбинируется для формирования **логических** адресов.

В этой прогулке по коду мы увидели, как создаются директория страниц (swapper_pg_dir) и таблица страниц (рдО) и как сгЗ инициализируется в точке директории страниц. Как обсуждалось ранее, процессор становится внимательным к просмотру компонентов страниц в соответствии с настройками сгЗ и настройками сг0 (бит PG) для того, чтобы процессор был проинформирован об их использовании. В **логических** адресах биты 22:31 означают **элемент директории страниц (PDE),** биты 12:21 означают **элемент таблицы страниц (PTE)** и биты 0:11 означают **отступ** (в данном примере 4 Кб) в физической странице.

Теперь у системы есть 8 Мб отображенной памяти для использования временной системы страниц. Следующим шагом является вызов функции start_kernel () в init/ main. c.

8.3.3 Похожесть кода PowerPC и x86

Обратите внимание, что код для PowerPC и x86 объединяется вместе в функции start_kernel () в init /main. c. Эта функция, расположенная в архитектурно-независимой части код, вызывает архитектурно-зависимые функции для завершения инициализации памяти.

Первой вызываемой инструкцией в этом файле является setup_arch(), которая определена для платформы x86 в arch/i3 8 б /kernel/setup. c и которая далее вызывает pagetable_init () в этом же файле. Остаток системной памяти выделяется для составления последней таблицы страниц.

В мире PowerPC многое уже сделано. Функция setup_arch() в файле arch/ ppc/kernel/setup. c вызывает paging__init () в arch/ppc/mm/init. c. Одна из важных функций для PPC paging_init () устанавливает все страницы в зоне DMA.

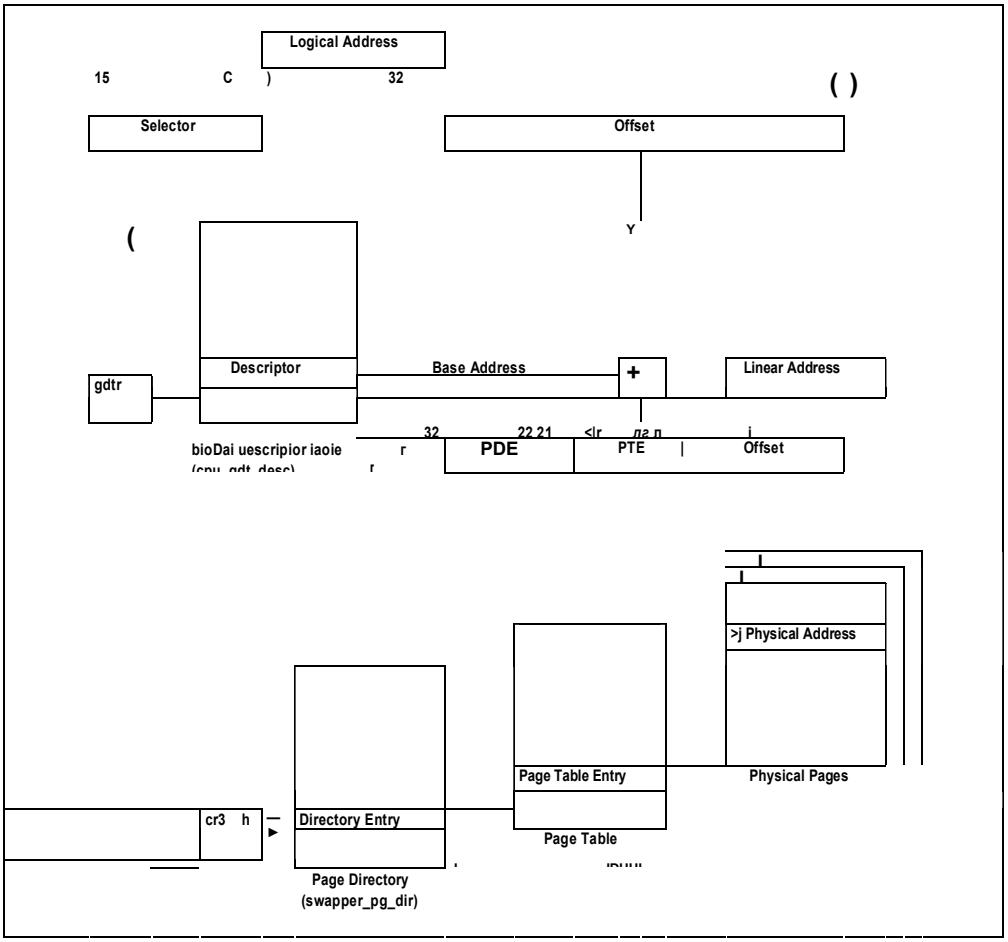


Рис. 8.10. Загрузочные страницы

8.4 Диск начальной загрузки

LILO, GRUB и Yaboot поддерживают диск начальной загрузки (initrd), работающий как корневая файловая система до того, как настоящая файловая система загружается и инициализируется. Завершением загрузки настоящей файловой системы мы считаем ее монтирование к корню.

Начальные шаги позволяют Linux загрузить несколько скомпилированных модулей и динамически загрузить другие модули и драйверы из initrd. Главное отличие от за-

грузчика заключается в том, что он загружается как минимальное ядро и диск в оперативной памяти во время шага 2. Ядро инициализирует использование диска в памяти, монтирует финальную корневую файловую систему и затем удаляет initrd. initrd позволяет нам:

- настроить ядро во время загрузки;
- сохранить ядро минимально функциональным;
- обладать ядром для нескольких аппаратных конфигураций.

Предыдущие строки являются основными при загрузке с помощью Yaboot, GRUB и LILO. Каждый загрузчик имеет богатый набор команд для своих настроечных файлов. Для измененного или использующего специальные функции загрузочного процесса, быстрого поиска в веб в GRUB и использования настроечных файлов LILO требуется дополнительная информация.

Теперь, когда мы увидели, как ядро загружается и как начинается инициализация памяти, давайте рассмотрим процессы инициализации ядра.

8.5 Начало: start_kernel()

Это обсуждение началось с перехода в функцию start_kernel () (init/main. c), которая является первой вызываемой архитектурно-зависимой частью кода.

Во время перехода в start_kernel () мы выполняем процесс 0, также известный как root thread (корневой процесс). Процесс 0 порождает процесс 1, известный как процесс инициализации. Процесс 0 становится ожидающим потоком процессора. Когда вызывается /sbin/init, у нас есть только два запущенных процесса:

```
init/main.c
396  asmlinkage void __init start_kernel(void)
397  {
3 98 char * command_line;
3 99 extern char saved_command_line[] ;
400  extern struct kernel_param ______ start___param[], __stop__param[];

405   lock_kernel();
406   page_address_init();
407   printk(linux_banner);
408   setup_arch(&command_line);
409   setup_per_cpu_areas();

415  smp_prepare_boot_cpu () ;

422    sched_init();
```

```
423
424    build_all_zonelists();
425    page_alloc_init() ;
426    printk("Kernel command line: %s\n", saved_command_line) ;
427    parse_args("Booting kernel", command_line,  __start___param,
428     ___stop___param -  __start___param,
429       &unknown_bootoption) ;
43 0      sort_main_extable();
431    trap_init();
432    rcu_init();
433    init_IRQ();
434    pidhash_init();
435    init_timers () ; 43
6      softirq_init();
437    time_init() ;

444    console_init () ;
445    if (panic_later)
446    panic (panic_later, panic_param)
447    profile_init();
448    local_irq_enable();
449   #ifdef CONFIG_BLK_DEV_INITRD
450    if (initrd_start && ! initrd_below_start_ok &&
451      initrd_start < min_low_pfn « PAGE_SHIFT) {
452     printk(KERN_CRIT "initrd overwritten (0x%081x < 0x%081x) - "
453       "disabling it.\n",initrd_start,min_low_pfn « PAGE_SHIFT);
454     initrd_start = 0;
455           }
456    #endif
457      mem_init();
458      kmem_cache_init () ;
459      if (late_time_init)
460       late_time_init();
461      calibrate_delay();
462      pidmap_init () ;
463      pgtable_cache_init();
464      prio_tree_init();
465      anon_vma_init() ;
466     #ifdef CONFIG_X86
467      if (efi_enabled)
468       efi_enter_virtual_mode();
469     #endif
470      fork_init(num_physpages);
471      proc_caches_init();
472      buffer_init();
473      unnamed_dev_init();
```

```
474      security scaffolding startup();
475    vf s_caches_init (num__physpages) ;
47 6   radix tree init();
477      signals_init();
478    /* содержимое rootfs нужно перезаписать обратно */
47 9   page_writeback_init();
480     #ifdef CONFIG PROC FS
481      proc_root_init();
482     #endif
483      check_bugs();

490      init_idle(current,  smp_processor_id());

493      rest_init();
494     }
```

8.5.1 Вызов lock_kernel()

Строка 405

В ядре Linux 2.6 конфигурация по умолчанию представляет собой вытесняющее ядро. Вытесняющее ядро означает, что само ядро может быть прервано высокоприоритетной задачей, такой, как аппаратное прерывание, и управление будет передано высокоприоритетной задаче. Ядро должно сохранять достаточно состояний, чтобы продолжать выполнение после обработки высокоприоритетной задачи.

Ранние версии реализации Linux включали приоритетное прерывание обработки ядра и блокировку SMP с помощью **Big Kernel Lock (BKL)**[1]. Позднейшие версии Linux абстрагировали приоритетную обработку прерываний в различные вызовы, такие, как preenipt_disable(). Во время инициализации BKL используется до сих пор. Это рекурсивная циклическая блокировка, отбирающая некоторое время у данного процессора. Подобным эффектом использования BKL является отключение приоритетного прерывания обработки, что во время инициализации крайне важно.

Блокировка ядра предотвращает его от того, что оно будет прервано или вытеснено какой-либо другой задачей. Linux использует для этой цели BKL. Когда ядро блокируется, другие процессы не выполняются. Этот антитезис приоритетного прерывания обработки ядра может быть прерван в любой момент. В ядре Linux 2.6 мы используем BKL для блокировки ядра во время загрузки и инициализации различных объектов ядра без опасения, что они будут прерваны. Ядро разблокирует строку 493 в функции rest_init (). Поэтому start_kernel () возникает при заблокированном ядре. Давайте рассмотрим, что происходит в lock_kernel ().

[1] Большая блокировка ядра. *Примеч. пер.*

```
include/linux/smp lock.h
42 static inline void lock kernel(void)
43 {
44   int depth = current->lock depth+1;
45   if (likely(!depth))
46     get kernel lock();
47   current->lock depth = depth;
48 }
```

Строки 44-48

Задача init имеет специальную lock_depth -1. Это позволяет мультипроцессорным системам не пытаться заблокировать выполнение ядра сразу на нескольких процессорах. Так как задачу init выполняет только один процессор, только он может блокировать выполнение ядра, так как только init имеет depth 0 (в противном случае depth больше 0). Похожий трюк используется в unlock_kernel (), где мы тестируем (--current->lock_depth < 0). Давайте рассмотрим, что происходит в get_kernel_lock ().

```
include/linux/smp lock.h
10 extern spinlock_t kernel_flag;
11
12 #define kernel locked()    (current->lock depth >= 0)
13
14 #define get kernel lock()   spin lock(&kernel flag)
15 tdefine put_kernel_lock ()   spin_unlock(&kernel_flag)

59 tdefine lock kernel()       do { } while(0)
60 #define unlock_kernel()       do { } while(O)
61 #define release kernel JLock(task)     do { } while(0)
62 #define reacquire_kernel_lock(task)    do { } while(0)
63 #define kernel_locked()    1
```

Строки 10-15

Этот макрос описывает большую блокировку ядра, использующую стандартную функцию циклической блокировки. На многопроцессорных системах возможна ситуация, когда два процессора могут попытаться получить доступ к одной и той же структуре данных. Циклический блок, описанный в гл. 7, предотвращает этот тип соглашения.

Строки 59-63

В этом случае ядро не вытесняется и не работает на многопроцессорных системах, мы просто ничего не делам для lock_kernel (), так как прерывания не поступают.

Теперь ядро исчерпало BKL и избавляется от него до конца start_kernel (), в результате все следующие команды не могут быть вытеснены.

8.5.2 Вызов page_addressjnit()

Строка 406

Вызов page_address_init () является первой функцией, связанной с инициализацией подсистемы памяти в архитектурно-зависимой части кода. Определение page_address_init О варьируется в зависимости от определения трех параметров компиляции. Первые два отключают page_address_init (), замещая тело функции на do {} while (0), как показано в следующем коде. Третью операцию мы рассмотрим подробнее. Давайте рассмотрим их определение и способ включения.

```
include/linux/mm.h
376 #if defined(WANT PAGE VIRTUAL)
382 #define page address  init ()  do { } while (0)
385 #if defined(HASHED PAGE VIRTUAL)
388 void page address init(void);
3 91 #if ! defined (HASHED PAGE VIRTUAL) && ! defined (WANT PAGE VIRTUAL)
3 94 #define page_address__init ()  do { } while (0)
```

#define для WANT__PAGE_VIRTUAL устанавливается, когда система использует прямое отображение памяти, и при этом просто рассчитывает виртуальный адрес в памяти для доступа к адресу в памяти. В случае, когда вся память не отображается в пространство адресов ядра (как часто бывает при использовании himem), нам нужен более удобный способ получения адресов. Поэтому инициализация странниц адресов определяется только в случае установленного HASHED_PAGE_VIRTUAL.

Теперь рассмотрим случай, когда ядру указывается использование HASHED_PAGE_VIRTUAL и когда нам нужно инициализировать применяемую ядром виртуальную память. Имейте в виду, что это происходит, только если настроена himem; при этом количество памяти, к которому может обратиться ядро, может быть больше, чем отображено в адресное пространство ядра (обычно 4 Гб).

В процессе последующих определений функций создаются и пересматриваются различные объекты ядра. Табл. 8.3 иллюстрирует объекты ядра, представленные в процессе рассмотрения page_address_init ().

Таблица 8.3. Представленные при вызове page_addr-ess_init() объекты

Объект	Описание
page_address_map	Структура
page_address_slot	Структура
page_address_pool	Глобальная переменная
page_address_maps	Глобальная переменная
page_address__htable	Глобальная переменная

```
mm/highmem.c
510 static struct page_address_slot {
511   struct list head lh;
512   spinlock t lock;
513 }  _ cacheline_aligned_in_smp page_address_htable[l«PA_HASH_ORDER] ;

591 static struct page_address_map page_address_maps[LAST_PKMAP];
592
593 void  _ init page_address_init(void)
594 {
595    int i;
596
597    INIT_LIST_HEAD(&page_address_pool);
598    for (i = 0; i < ARRAY_SIZE(page address maps) ; i++)
599      list add(&page address maps [i] .list, &page address pool) ;
600    for (i = 0; i < ARRAY_SIZE(page_address_htable); i++) {
601      INIT LIST HEAD(&page address htable[i].lh);
602      spin_lock_init(&page_address_htable[i].lock) ;
603    }
604    spin_lock_init(&pool_lock);
605 }
```

Строка 597

Главное назначение этой строки заключается в инициализации глобальной переменной page_address_global (), являющейся структурой типа list_head и указывающей на список свободных выделенных из page_address_maps (строка 591) страниц. Рис. 8.11 иллюстрирует page_address_pool.

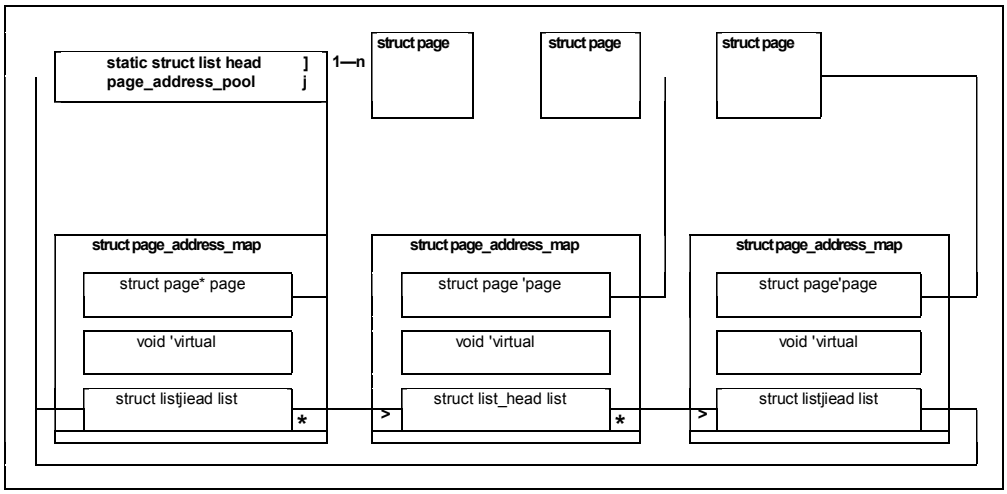


Рис. 8.11. Структуры данных, связанные с пулом карты адресов страниц

Строки 598-599

Мы добавляем каждый список страниц из page___address_maps в двусвязный список в page_address__pool. Далее мы подробно опишем структуру page_address_maps.

Строки 600-603

Мы инициализируем каждый list_head хеш-таблицы адресов страниц и циклические блокировки. Переменная page__address_htable хранит список элементов хеша для тех же участков памяти. Рис. 8.12 иллюстрирует хеш-таблицу адресов страниц.

Строка 604

Мы инициализируем циклическую блокировку page_address_pool.

Давайте рассмотрим структуру page_address_map, чтобы лучше понять, что за список мы только что инициализировали. Главное назначение этой структуры - поддержание связи между страницами и их виртуальными адресами. Если страницы отображаются в виртуальные адреса линейно, это совершенно не нужно. Необходимость в этой структуре появляется только при хешировании адресов:

```
mm/highmem.c
490 struct page address  map {
491   struct page *page;
492   void *virtual;
```

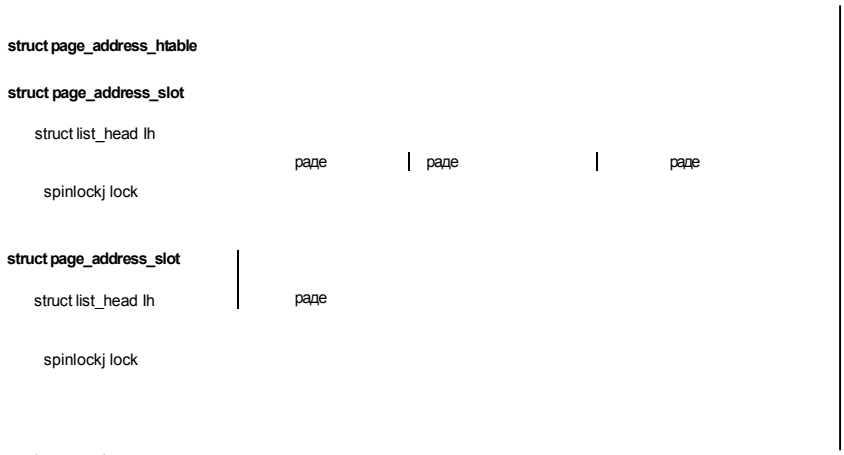


Рис. 8.12. Хеш-таблица адресов страниц

```
493    struct list_head list;
494  };
```

Как вы можете видеть, объект хранит указатель на структуру страницы, связанную с данной страницей, указатель на виртуальный адрес и структуру list_head для обозначения позиции в двусвязном списке адресов страниц.

8.5.3 Вызов printk(linux_banner)

Строка 407

Вызов отвечает за первый вывод в консоль, выполняемый ядром Linux. Он представлен глобальной переменной linux_banner:

```
init/version.c
31   const char *linux banner =
32    "Linux version " UTS_RELEASE " (" LINUX_COMPILE_BY "@"
          LINUX__COMPILE_HOST ") (" LINUX_COMPILER ") " UTS_VERSION "\n";
```

Файл version, с определяет linux_banner, как показано выше. Эта строка информирует пользователя о версии ядра Linux, версии дсс, с помощью которой оно откомпилировано, и название релиза.

8.5.4 Вызов setup_arch

Строка 408

Функция setup_arch() в arch/i386/kernel/setup.c преобразуется в тип _ init (обратитесь к описанию ____init в гл. 2), запускаемый только один раз во время инициализации системы. Функция setup_arch () получает указатель на заданные во время загрузки в командной строке Linux данные и инициализирует множество архитектурно-зависимых подсистем, таких, как память, ввод-вывод, процессоры и консоли.

```
arch/i3 86/kernel/setup.c
1083  void   init setup arch(char **cmdline p)
1084  {
1085   unsigned long max low pfn;
1086
1087   memcpy(&boot cpu data, &new cpu data, sizeof(new cpu data));
1088   pre setup arch hook();
1089   early_cpu_init();
1090
1091   /*
1092    * FIXME: сейчас это неофициальный loader type
1093    * хотя он и работает с elilo.
1094    * Если мы настраиваем ядро EFI, нужно проверить,
1095    * что загрузка из elilo прошла успешно и что системная
1096    * таблица верна. Если нет, инициализация проводится как обычно.
1097    */
1098   #ifdef CONFIG EFI
1099    if ((LOADER TYPE == 0x50) && EFI SYSTAB)
1100     efi_enabled = 1;
1101   #endif
1102
1103    ROOT_DEV = old_decode_dev(ORIG_ROOT_DEV);
1104    drive info = DRIVE INFO;
1105    screen info = SCREEN INFO;
1106    edid info = EDID INFO;
1107    apm info.bios = APM BIOS INFO;
1108    ist info = IST INFO;
1109    saved_videomode = VIDEO_MODE;
1110    if( SYS_DESC_TABLE.length != 0 ) {
1111     MCA_bus = SYS_DESC_TABLE. table [3] &0x2 ;
1112     machine id = SYS DESC TABLE.table[0];
1113     machine submodel id = SYS DESC TABLE.table[1];
1114     BIOS revision = SYS DESC TABLE.table[2];
1115    }
```

```
1116       aux_device_present = AUX_DEVICE__INFO;
1117
1118    #ifdef CONFIG BLK DEV  RAM
1119     rd image start = RAMDISK FLAGS & RAMDISK IMAGE  START  MASK;
1120     rd  prompt = ( (RAMDISK FLAGS & RAMDISK PROMPT FLAG) != 0) ;
1121     rd doload = ( (RAMDISK FLAGS & RAMDISK LOAD FLAG) != 0) ;
1122    #endif
1123     ARCH SETUP
1124     if (efi_enabled)
1125      efi init();
1126     else
1127      setup memory region();
1128
1129                        copy_edd();
1130
1131     if (!MOUNT ROOT RDONLY)
1132      root  mountflags &= ~MS  RDONLY;
1133     init mm.start code = (unsigned long)   text;
1134     init mm.end code = (unsigned long)  etext;
1135     init mm.end data = (unsigned long)  edata;
113 6    init mm.brk = init pg tables end + PAGE OFFSET;
1137
1138     code resource.start = virt to phys( text);
1139     code  resource.end = virt  to phys ( etext)-1;
1140     data resource.start = virt to phys( etext) ;
1141     data resource.end = virt to phys( edata)-1;
1142
1143             parse cmdline early(cmdline p);
1144
1145              max .low pfn = setup  memory () ;
1146
1147     /*
1148     * ПРИМЕЧАНИЕ. Перед этой точкой  nobody  позволено выделить
1149     * всю память, используя выделитель памяти bootmem.
1150     */
1151
1152    #ifdef CONFIG_SMP
1153     smp_alloc_jnemory(); /* Стеки реального режима процессора AP[1]
                               в нижней памяти*/
1154    #endif
1155     paging init();
1156
```

[1] **AP (application processor) - любой процессор SMP-системы, который не является BSP (bootstrap processor)-процессором, т. е. не является процессором, которому передается управление в момент загрузки.** ***Примеч.*** *науч. ред.*

```
1157   #ifdef CONFIG EARLY PRINTK
1158    {
1159     char *s = strstr(*cmdline p/ "earlyprintk^');
1160     if (s) {
1161      extern void setup early printk(char *) ;
1162
1163      setup early printk(s);
1164      printk("early console enabled\n");
1165     }
1166    }
1167   #endif

1170     dmi scan machine();
1171
1172   #ifdef CONFIG_X86_GENERICARCH
1173    generic apic probe(*cmdline p);
1174   #endif
1175    if (efi enabled)
1176     e f i_map_memmap();
1177
1178     /*
1179     * Парсинг таблицы ACPI для возможной конфигурации SMP
1180     * времени загрузки. */
1181     acpi boot init();
1182
1183   #ifdef CONFIG X86  LOCAL APIC
1184    if (smp found config)
1185     get smp__conf ig() ;
1186   #endif
1187
1188     register memory(max low pfn);
1188
1190   #ifdef CONFIG VT
1191   #if defined(CONFIG_VGA_CONSOLE)
1192    if (!efi enabled || (efi mem type(OxaOOOO) !=
                                        EFI CONVENTIONAL MEMORY))
1193     conswitchp = &vga con;
1194   #elif defined(CONFIG DUMMY CONSOLE)
1195    conswitchp = &dummy_con;
1196   #endif
1197   #endif
1198  }
```

Строка 1087

Получение boot_cpu_date, являющегося указателем на структуру cpuinfo_ x8б, заполняемую во время загрузки. Это похоже на то, что происходит на PPC.

Строка 1088

Активация любых машинно-специфических функций идентификации. Можно найти в arch/xxx/machine-ddefault/setup.c.

Строка 1089

Идентификация специфического процессора.

Строки 1103-1116

Получение параметров загрузки системы.

Строки 1118-1122

Получение диска в памяти при установке в arch/<arch>/def conf ig.

Строки 1124-1127

Инициализация расширенного интерфейса Firmware (если установлено в /def config) или просто вывод карты памяти BIOS.

Строка 1129

Сохранение параметров с загрузочного расширенного привода диска.

Строки 1133-1141

Инициализация структуры менеджера памяти из предоставленной BIOS карты памяти.

Строка 1143

Запуск парсинга командной строки Linux. (См. arch/<arch>/kernel/setup, с.)

Строка 1145

Инициализация/резервирование загрузочной памяти. (См. arch/i3 8 б /kernel/ setup.c.)

Строки 1153-1155

Получение страницы для инициализации SMP или инициализации страниц за пределами 8 Мб памяти, уже инициализированной в head.S. (См. arch/i3 8 б/mm/ init .c.)

Строки 1157-1167

Получение printk (), запущенной, даже если консоль еще не полностью инициализирована.

Строка 1170

Это строка Desktop Management Interface (DMI)[1], собирающего информацию о специфических аппаратно-зависимых конфигурациях из BIOS. (См. arch/i386/ kernel /dmi_s can. c.)

Строки 1172-1174

Если конфигурация его вызывает, просматривается APIC для заданной командной строки. (См. arch/ i3 8 б /kernel /probe, c.)

Строки 1175-1176

Если используется интерфейс расширяемого Firmware, то перезаполнить карту памяти EFI. (См. arch/i386/kernel/efi.c.)

Строка 1181

Просмотр APIC локального и ввода-вывода. (См. arch/i386/kernel/acpi/ boot. c.) Поиск и проверка контрольной суммы таблицы описания системы. (См. drivers/acpi/tables . c.) Для лучшего понимания ACPI обратитесь к проекту **ACPI4LINUX** в сети.

Строки 1183-1186

Сканирование конфигурации SMP. (См. arch/i386/kernel/mpparse.c.) Этот раздел также использует настроечную информацию ACPI.

Строка 1188

Запрос ввода-вывода и пространства памяти для стандартных ресурсов. (См. регистрацию ресурсов в arch/i3 86/kernel/std_resources.c.)

Строки 1190-1197

Настройка структуры переключения VBGA консоли (См. drivers/video/console/vgacon. c.)

Похожую, но более короткую версию setup_arch () для PowerPC можно найти и в arch/ppc /kernel /setup. c. Эта функция инициализирует большую часть структуры ppc_md. Вызов pmac_f eature_init () в arch/ppc /plat f orm/pmac_f eature. c вызывает начальную проверку и инициализацию оборудования ртас.

8.5.5 Вызов setup_per_cpu_areas()

Строка 409

Функция setup_per_cpu_areas () существует для настройки многопроцессорного окружения. Если ядро Linux скомпилировано без поддержки SMP, setup_per_cpu_areas () замещается ничегонеделанием следующим образом:

[1] **Интерфейс управления рабочим столом.** ***Примеч. пер.***

init/main.c
317 static inline void setup_per_cpu_areas(void) { }

Если ядро Linux компилируется с поддержкой SMP, setup__per_cpu_areas () определяется следующим образом:

```
init/main.c
327 static void   init setup per cpu areas(void)
328 {
329   unsigned long size, i;
33 0   char *ptr;
331   /* Создание с помощью магии компоновщика */
332   extern char   per cpu start[],    per cpu end[ ] ;
333 /* Копирование выбранного для каждого процессора (оригинал
334  * затирается) */
335   size = ALIGN(   per cpu end -    per cpu start, SMP CACHE BYTES) ;
33 6 #ifdef CONFIG_MODULES
337   if (size< PERCPU ENOUGH ROOM)
338    size= PERCPU ENOUGH ROOM;
33 9 #endif
340
341   ptr = alloc bootmem(size * NR CPUS);
342
343   for (i = 0; i < NR CPUS; i++, ptr += size) {
344        per  cpu of fset [i] = ptr -    per cpu start;
345      memcpy(ptr,    per cpu start,    per cpu end -    per cpu start) ;
346   }
347 }
```

Строки 329-332

Инициализируются переменные для управления последовательными блоками памяти. Переменные «магического связывания» определяются во время связывания в соответствующих каталогах архитектуры ядра (например, arch/i386/kernel/ vbmlinux. Ids . S).

Строки 334-341

Мы определяем размер памяти, требуемой одному процессору, и выделяем эту память для каждого процессора системы в виде единого последовательного блока памяти.

Строки 343-346

Мы проходим по всей новой выделенной памяти и инициализируем блок памяти для каждого процессора. Концептуально мы берем блок данных, предназначенный для отдельного процессора (__per_spu_start в _____per_cpu_end), и копируем его для каждого процессора системы. Таким способом каждый процессор получает данные, с которыми он может работать.

8.5.6 Вызов smp_prepare_boot_cpu()

Строка 415

Аналогично smp_per_cpu_areas (), smp_per_boot_cpu () заменяется в случае, если ядро Linux не поддерживает SMP:

```
include/1inux/smp.h
106 #define smp_prepare_boot_cpu()                  do {} while (0)
```

Тем не менее, если ядро Linux откомпилировано без поддержки SMP, нам нужно позволить загрузочному процессору получить доступ к драйверу консоли и хранилищу для процессора, которое мы только что инициализировали. Это выполняется с помощью битовой маски для процессора.

Битовая маска процессора определяется следующим образом:

```
include/asm-generic/cpumask.h
10 #if NR CPUS > BITS PER LONG && NR CPUS != 1
11 tdefine CPU ARRAY SIZE  BITS TO LONGS(NR CPUS)
12
13 struct cpumask
14 {
15   unsigned long mask[CPU ARRAY SIZE];
16 };
```

Это означает, что у нас есть платформонезависимая битовая маска, содержащая такое же количество битов, как и количество процессоров в системе.

smp_prepare_boot_cpu() реализована как архитектурно-зависимый блок ядра Linux, однако, как мы вскоре увидим, для x86 и PPC она совпадает:

```
arch/i38 б/kernel/smpboot.c
66 /* карта для включенных процессоров */
67 cpumask_t cpu_online_map;
```

```
70 cpumask_t cpu_callout_map;

1341 void   devinit smp prepare boot cpu(void)
1342 {
1343   cpu set(smp processor  id(), cpu online map);
1344   cpu set (smp  processor id() , cpu callout map);
1345 }
```

```
arch/ppc/kernel/smp.c
49 cpumask_t cpu_online_map;
50 cpumask_t cpu_possible_map;

331 void __devinit smp__prepare_boot_cpu(void)
332 {
333   cpu set(smp processor  id(), cpu online map);
334   cpu_set(smp_processor_id(), cpu_possible_map);
335 }
```

Для обеих функций cpu_set () просто устанавливает бит smp_proces sor_id в битовой карте cpumask_t. Установка бита предполагает, что значение установленного бита равно 1.

8.5.7 Вызов schedjnit()

Строка 422

Вызов sched_init () помечает инициализацию всех объектов, с которыми работает планировщик, для назначения процессорного времени системным процессам. Имейте в виду, что в этой точке существует только один процесс, а именно процесс init, выполняемый shed_init ().

```
kernel/sched.c
3896 void __init sched_init(void)
3897 {
3898    runqueue t * rq;
3 899   int i, j, k;
3900

3919       for (i = 0; i < NR CPUS; i++) {
3 92 0                 prio array t *array;
3921
3 922                        rq = cpu rq(i);
3923            spin_lock_init(&rq->lock);
```

```
3 924                    rq->active = rq->arrays;
3 925                  rq->expired = rq->arrays + 1;
3 92 6               rq->best_expired_prio = MAX_PRIO;

3938                   for (j = 0; j < 2; j++) {
3 93 9                    array = rq->arrays + j;
3940            for (k = 0; к < MAX_PRIO; k++) {
3941              INIT LIST HEAD(array->queue + k);
3942              ___ clear_bit(k, array->bitmap);
3943            }
3944            // разделитель для битового поиска
3945              setjDit(MAX PRIO, array->bitmap) ;
3946         }
3947      }
3948      /*
3 949* Мы применяем небольшую магию для получения первого
3 950               * потока в SMP-режиме.
3951      */
3952      rq = this rq();
3 953                rq->curr = current;
3 954                rq->idle = current;
3955   set task cpu(current, smp processor id());
3 956        wake up f orked process (current) ;
3957
3958                           /*
3 959* Загрузка ожидающего потока,выполняющего ленивое переключение
MMU:
3960      */
3961      atomic inc(&init mm.iran count);
3962      enter lazy tlb(&init mm, current);
3963   }
```

Строки 3919-3926

Инициализируется очередь выполнения для каждого из процессоров: активная очередь, очередь истекших и циклическая блокировка инициализируются именно в этом блоке. Вспомните из гл. 7, что spin_lock_init () устанавливают циклическую блокировку в 1, что означает, что данные объекта разблокированы.

Рис. 8.13 иллюстрирует инициализированную очередь выполнения.

Строки 3938-3947

Для каждого возможного приоритета мы инициализируем связанный с ним список и очищаем все биты для обозначения того, что в очереди выполнения нет процессов. (Если вы во всем этом запутались, обратитесь к рис. 8.14. Также вернитесь к гл. 7,

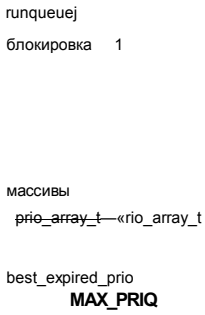


Рис. 8.13. Инициализированная очередь выполнения rq

где приведен обзор того, как планировщик управляет очередями выполнения.) Этот блок кода просто проверяет, что все готово для первого процесса. В строке 3947 планировщик знает, что процессы не существуют; он игнорирует текущий и ожидающий процессы.

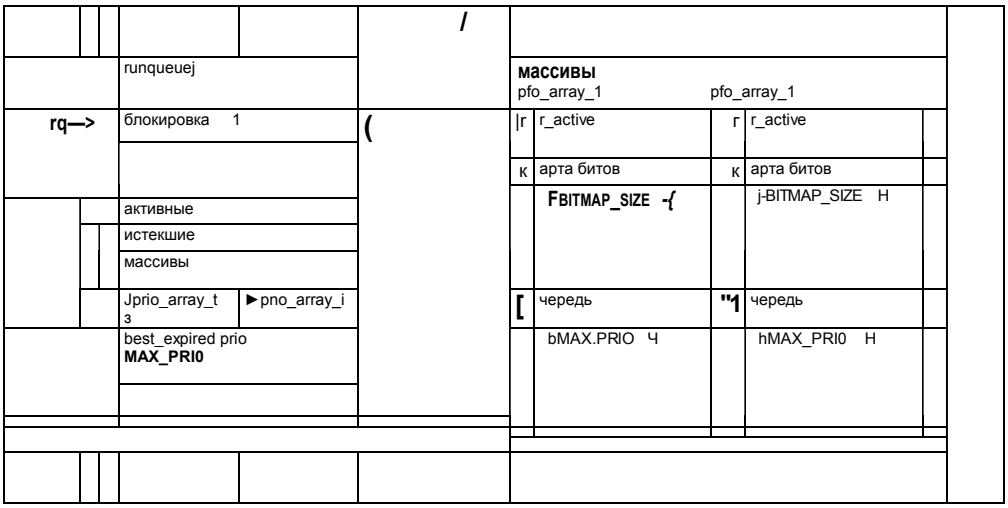


Рис. 8.14. Инициализированная очередь выполнения rq

Строки 3952-3956

Мы добавляем текущий процесс в очередь выполнения текущего процессора и вызываем wake_up_f orked_process () для самого текущего процесса с целью его

инициализации в планировщике. Теперь планировщик знает, что у нас есть как минимум один процесс - init.

Строки 3961-3962

Когда переключатель ленивого MMU включен, многопроцессорная система Linux может выполнять переключение контекстов быстрее. TLB является буфером предпросмотра транзакций, содержащим последние преобразования адресов страниц. Обработка TLB занимает много времени, так что, если это возможно, мы ее заменяем; enter_lazy__tlb() проверяет, что структура inin_struct_init_mm не используется для нескольких процессоров и может быть лениво переключена. На однопроцессорной системе эта функция приравнивается NULL.

8.5.8 Вызов build_all_zonelists()

Строка 424

Функция build_all_zonelists () разделяет память в зависимости от типа зоны ZONE_DMA, ZONE_NORMAL и ZONE_HIGHMEM. Как упоминалось в гл. 6, «Файловые системы», зоны - это линейное разделение физической памяти, используемое в основном для адресации аппаратных ограничений. Стоит заметить, что все зоны строятся именно в этой функции. После того как зоны построены, страницы сохраняются во фреймах страниц, расположенных в зонах.

Вызов build_all_zonelists () представляет numnodes и NODE_DATA. Глобальная переменная numnodes хранит количество узлов (или разделов) физической памяти.

Разделы определяются в зависимости от скорости доступа для процессора. Обратите внимание, что в этой точке таблица страниц полностью настроена.

mm/page_alloc.c

```
1345    void  ___ init build_all_zonelists(void)
1346    {
1347      int  i;
1348
1349     for(i = 0 ; i < numnodes ; i++)
13 50      build_zonelists(NODE_DATA(i));
1351     printk("Built %i zonelists\n", numnodes);
1352    }
```

build_all_zonelists () вызывает build_zonelists() для каждого узла и заканчивает распечаткой количества созданных зон. Эта книга не рассматривает подробностей, связанных с узлами. Стоит заметить, что для примера с одним процессором

numnodes равно 1 и каждый узел содержит зоны всех трех типов. Макрос NODE_DATA возвращает описатель узла из списка описателей узлов.

8.5.9 Вызов page_allocJnit

Строка 425

Функция page_alloc_init () просто регистрирует функцию в цепи уведомлений[1]. Зарегистрированная функция page_alloc_cpu_notify () является функцией утечки страниц[2], связанной с динамической настройкой процессора.

Динамическая настройка процессора связана с добавлением и удалением процессоров во время работы системы Linux при наступлении события «горячее подключение» процессора; несмотря на то что технически процессоры не вставляются и не извлекаются физически во время работы машины, в некоторых системах они могут включаться и выключаться, как в IBM p-Series 690. Давайте рассмотрим эту функцию.

mm/page_alloc.c

```
1787  tifdef CONFIG HOTPLUG CPU
1788  static int page_alloc_cpu_notify(struct notifier_block *self,
1789      unsigned long action, void *hcpu)
1790  {
1791   int cpu = (unsigned long)hcpu;
17 92  long *count;
1793
if (action == CPU_DEAD) {

1796    count = &per_cpu(nr__pagecache_JLocal, cpu) ;
1797    atomic__add(*count, &nr_pagecache);
1798    *count = 0;
1799    local_irq_disable();
1800   __ drain__pages (cpu) ;
1801    local_irq_enable();
1802    }
1803    return NOTIFY_OK;
1804  }
1805  #endif /* CONFIG HOTPLUG CPU */
1806
1807 void   init page alloc init(void)
1808  {
1809   hotcpu  notifier(page alloc cpu notify, 0);
1810  }
```

[L] Цепь уведомлений обсуждается в гл. 2.

2. Утечка страниц связана с удалением больше не используемых процессором страниц.

Строка 1809

В этой строке регистрируется функция page_alloc_cpu_notify () в цепи уведомлений hotcpu_notifier. Функция page_alloc_cpu_notify () создает notif ier_block, указывающую на функцию page_alloc_cpu_notify (), и затем регистрирует объект в цепи уведомления с приоритетом 0 (kernel/cpu. c).

Строка 1788

page_alloc__cpu_notify () имеет параметры, связанные с вызовом уведомителя, как описано в гл. 2. Системно-зависимый указатель указывает на целое, означающее номер процессора.

Строки 1794-1802

Если процессор мертв, его страницы освобождаются. При выключении процессора переменная действия устанавливается в CPU_DEAD. (См. drain_pages () в том же файле.)

8.5.10 Вызов parse_args()

Строка 427

Функция parse_args () выполняет парсинг аргументов, передаваемых в ядро Linux.

Например, nfsroot- это параметр ядра, устанавливающий корень файловой системы NFS для систем без диска. (Вы можете найти полный список параметров ядра в Documentation/kernel-parameters . txt.)

```
kernel/params.c
116 int parse args(const char *name,
117      char *args,
118      struct kernel param *params,
119      unsigned num,
120      int (*unknown)(char *param, char *val))
121 {
122       char *param, *val;
123
124    DEBUGP("Parsing ARGS: %s\n", args);
125
126        while (*args) {
127               int ret;
128
129   args = next arg(args, &param, &val);
130ret = parse one(param, val, params, num, unknown);
131       switch (ret) {
132       case -ENOENT:
```

```
133         printk(KERN_ERR "%s: Unknown parameter N %s•\n",

134           name, param);
135         return ret;
136       case -ENOSPC:
137         printk(KERN_ERR
138           "%s: 8%s' too large for parameter 4%s'\n"/
139           name, val ?: "", param);
140         return ret;
141       case 0:
142         break;
143       default:
144         printk(KERN_ERR
145           "%s: x%s' invalid for parameter 4%s'\n",
146           name, val ?: "", param);
147         return ret;
148       }
149     }
150
151     / Все прошло успешно. */
152    return 0;
153 }
```

Строки 116-125

В parse_args () передаются следующие параметры:

- name. Строка символов, отображаемая при возникновении ошибки, когда ядро пытается выполнить парсинг аргументов параметров ядра. В стандартной операции это означает, что отображается сообщение «Booting kernel: Unknown parameter X».
- args. Список параметров ядра в *виде/оо=Ьаг, bar2 baz-fuz wix*.
- params. Указывает на структуру параметров ядра, хранящую все корректные параметры для данного ядра. В зависимости от откомпилированного ядра некоторые параметры могут присутствовать или отсутствовать.
- num. Количество параметров ядра для данного ядра, не равное количеству аргументов в args.
- unknown. Указывает на функцию, вызываемую, когда параметр ядра не распознан.

Строки 126-153

Мы циклически проходим по строке args, устанавливаем указатель рагаш на первый параметр и присваиваем val первое значение (иначе, val может быть равно нулю). Это выполняется с помощью next_args () (например, первый вызов next_args () с args, равным *foo=bar, bar2 baz=fuz wix.)* Мы присваиваем param

foo и val - *bar, bar2*. Пространство после bar2 перезаписывается \ 0 и args устанавливает указатель на первый символ *baz*.

Мы передаем наши указатели раташ и val в parse_one(), выполняющую работу по настройке настоящих параметров структур данных:

```
kernel/params.c
46 static int parse one(char *param,
47       char *val,
48       struct kernel param *params,
49       unsigned num params,
50       int (*handle_unknown)(char *param, char *val))
51 {
52   unsigned int i;
53
54   /* Поиск параметра */
55   for (i = 0; i < num__params; i++) {
56     if (parameq(param, params[i].name)) {
57       DEBUGP("They are equal!  Calling %p\n",
58        params[i].set);
59       return params[i].set(val, &params[i]);
60     }
61   }
62
63   if (handle unknown) {
64     DEBUGP("Unknown argument: calling %p\n", handle_unknown);
65     return handle_unknown(param, val);
66   }
67
68   DEBUGP("Unknown argument "%s'\n", param);
69   return -ENOENT;
70 }
```

Строки 46-54

Эти параметры те же, что и описанные в parse_args () с раташ и val, указывающими на подраздел args.

Строки 55-61

Мы проходим по всем определенным параметрам для проверки любых совпадений раташ. Если совпадение найдено, мы используем val для вызова связанного набора функций. Далее набор функций обрабатывает несколько или 0 аргументов.

Строки 62-66

Если параметр ядра не найден, мы вызываем функцию handle_unknown(), передаваемую через parse_args ().

После вызова parse_args () для каждой комбинации параметра-значения, указанного в args, мы устанавливаем набор параметров ядра и готовимся продолжить запуск ядра Linux.

8.5.11 Вызов trapjnit()

Строка 431

В гл. 3 мы представляли вашему вниманию исключения и прерывания. Функция trap_init () специфична для обработки прерываний на архитектуре x86. Коротко говоря, эта функция инициализирует таблицу связи x86 с аппаратурой. Каждый элемент в таблице имеет функцию для обработки ядром связанных с пользователем вещей, таких, как неправильные инструкции или связь с не находящимися в памяти страницами. Несмотря на то что на PowerPC существуют аналогичные вещи, его архитектура обрабатывает их по-другому. (Это уже обсуждалось в гл. 3.)

8.5.12 Вызов rcu_init()

Строка 432

Функция rcu_init() инициализирует Read-Copy-Update (RCU)[1] ядра Linux 2.6. RCU контролирует доступ к критическим разделам кода и включает обоюдное исключение в системе за счет получения блокировки в зависимости от скорости чипа. Реализация RCU в Linux выходит за пределы рассмотрения этой книги. Мы уже мимоходом упоминали вызовы к подсистеме RCU при анализе нашего кода, но в подробности мы вдаваться не будем. (Более подробную информацию о подсистеме RCU Linux см. в странице Linux Scalability Effort в http://Ise. source-forge, net/locking/rcupdate. html.)

```
kernel/rcupate.c
2 97 void   init rcu init(void)
298 {
299   rcu cpu notify(&rcu nb, CPU UP PREPARE,
300        (void *)(long)smp_processor_id());
301   /* регистрация уведомителя для незагрузочного процессора */
3 02   register_cpu_notifier(&rcu_nb);
303 }
```

[1] Чтение-копирование-обновление. *Примеч. пер.*

8.5.13 Вызов initJRQ()

Строка 433

Функция init_IRQ() в arch/i386/kernel/i8259. с инициализирует контроллер аппаратных прерываний, таблицу векторов прерываний и системный таймер для x86. В гл. 3 приводится подробное обсуждение прерываний как для x86, так и для PPC, с использованием в качестве примера прерывания часов реального времени.

```
arch/i386/kernel/i8259.c
410 void   init init IRQ(void)
411 {
412  int i;

422  for (i =0; i < (NR VECTORS - FIRST EXTERNAL VECTOR); i++) {
423   int vector = FIRST_EXTERNAL_VECTOR + i;
424   if (i >= NR IRQS)
425    break;

43 0   if (vector != SYSCALL_VECTOR)
431    set_intr_gate(vector, interrupt[i]);

432  }

437  intr_init_hook();

443  setup_timer();

449  if (boot_cpu_data.hard_math && !cpu_has_fpu)
450   setup__irq(FPU_IRQ, &fpu_irq) ;
451 }
```

Строки 422-432

Инициализация вектора прерываний. Связывание IRQ (аппаратных) x86 с соответствующими кодами обработки.

Строка 437

Установка машинно-специфических IRQ, таких, как усовершенствованный программируемый контроллер прерываний (APIC).

Строка 443

Инициализация таймера времени.

Строки 449-450

Установка FPU при необходимости.

Следующий код представляет собой реализацию init_IRQ () для PPC.

```
arch/ppc/kernel/irq.c
700  void   init init IRQ(void)
701  {
702   int i;
703
704   for (i = 0; i < NR_IRQS; ++i)
705    irq affinity[i] = DEFAULT CPU AFFINITY;
706
707    ppc md.init  IRQ();
708  }
```

Строка 704

На многопроцессорных системах прерывания могут быть родственными для отдельных процессоров.

Строка 707

Для платформы PowerMac эта функция находится в arch/ppc /platforms/ pinac_pic.c. Она устанавливает часть программируемого контроллера прерываний (PIC) ввода-вывода.

8.5.14 Вызов softirq_init()

Строка 436

Функция sof tircx_init () подготавливает загрузочный процессор к приему уведомлений оттасклетов. Рассмотрим содержимое sof tirq__init ().

```
kernel/softirq.c
317 void __ init softirq_init(void)
318 {
319   open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
320   open softirq(HI SOFTIRQ, tasklet hi action, NULL);
321 }

327 void _ init softirq_init(void)
328 {
329   open softirq(TASKLET SOFTIRQ, tasklet action, NULL);
330   open softirq(HI SOFTIRQ, tasklet hi action, NULL);
331  tasklet_cpu_notify(&tasklet_nb, (unsigned long)CPU_UP_PREPARE,
332           (void *)(long)smp processor id());
333  register cpu notifier(&tasklet jib);
334 }
```

Строки 319-320

Мы инициализируем действия, выполняемые, когда мы получаем прерывания TASKLET__SOFTIRQ или HI_SOFTIRQ. Если мы передаем NULL, мы сообщаем ядру Linux вызвать tasklet_action(NULL) и tasklet_hi_action(NULL (в строках 319 и 320 соответственно). Следующая реализация open_sof tirq() демонстрирует, как ядро Linux сохраняет информацию об инициализации тасклета.

```
kemel/sof tirq. c
177 void open_softirq(int nr, void (*action) (struct softirq_action*),
                            void * data)
178 {
179   softirq vec[nr].data = data;
180   softirq_vec[nr].action = action;
181 }
```

8.5.15 Вызов timejnit()

Строка 437

Функция time_init () выбирает и инициализирует системный таймер. Эта функция, как и trap_init (), сильно зависит от архитектуры; гл. 3 описывает ее при рассмотрении прерывания таймера. Системный таймер дает Linux временной обзор мира, позволяя планировщику определять, когда и на какое время запускать задачи. **ffigh Performance Event Timer (HPET)**[1] от Intel является приемником аппаратных 8254 PIT и RTC; он использует отображенный в память ввод-вывод, что означает возможность доступа к управляющим регистрам HPET так, как будто они находятся в памяти. Для доступа к области ввода-вывода необходимо правильно сконфигурировать память. Если это установлено в arch/i3 86/defconf ig.h, time_init () необходимо отложить до завершения mem_init (), когда она настроит области памяти. (См. следующий код.)

```
arch/i386/kernel/time.c
376 void   init time init(void)
377 {

378 #ifdef CONFIG HPET TIMER
379  if (is hpet capable()) {
3 80    late_tinie_init = hpet_time_init; 3 81
return; 382   }
```

[1] Высокопроизводительный таймер событий. *Примеч. пер.*

```
3 87 #endif
3 88    xtime.tv_sec = get_cmos_time();
3 89    wall_to_monotonic.tv_sec = -xtime.tv_sec;
390     xtime. tv_nsec = (INITIAL_JIFFIES % HZ) * (NSEC_PER_SEC / HZ);
391     wall_to_monotonic. tv_nsec = -xtime.tv_nsec; 392
3 93    cur_timer = select_timer();
394   printk(KERN_INFO "Using %s for high-res timesource\n",
                  cur_timer->name); 395
3 96    time_init_hook(); 397 }
```

Строки 379-387

Если HPET настроен, после инициализации памяти запускается time_init () . Код late_time_init () (строки 358-373) аналогичен коду time_init ().

Строки 388-391

Инициализация структуры времени xtime, используемой для хранения времени суток.

Строка 393

Выбор первого инициализируемого таймера. Это действие может быть переназначено. (См. arch/i386/kernel/timers/timer.c.)

8.5.16 Вызов consoleJnit()

Строка 444

Компьютерная консоль - это устройство, куда ядро (и другие компоненты системы) выводит сообщения. Также она употребляется для регистрации пользователя. В зависимости от системы консоль может быть монитором или просто последовательным портом. Функция console_init () вызывается для предварительной инициализации устройства консоли, что позволяет сообщать о статусе загрузки системы.

```
drivers/char/tty_io.c
2347 void __init console__init (void)
2348 {
2349  initcall_t *call;

2352  (void) tty_register_ldisc(N_TTY, &tty_ldisc_N_TTY);
```

```
2358 #ifdef CONFIG_EARLY_PRINTK
2359          disable early printk();
2360 #endif

23 66  call = &   con initeall start ;
23 67  while (call < &   con initcall end) {
2368     (*call)();
2369     call++;
2370    }
2371  }
```

Строка 2352

Установка порядка обслуживания строки.

Строка 2359

Сохранение ранней поддержки printk при необходимости. Ранняя поддержка printk позволяет системе сообщать о своем состоянии во время процесса загрузки, перед завершением полной инициализации консоли. В частности, инициализируется последовательный порт (например, ttysO) или системный VGA с минимумом функциональности. Поддержка раннего printk запускается в setup_arch(). (Более подробно эта информация обсуждается в строке 408 в этом разделе и файлах /kernel/printk.c и/arch/i386/kernel/early_printk.c.)

Строка 2366

Инициализация консоли.

8.5.17 Вызов profileJnit()

Строка 447

prof ile_init () выделяет память для хранения ядром данных профилирования. Профиль в терминах компьютерных наук используется для описания набора данных во время работы программы. Данные профилирования применяются для анализа производительности и изучения работы программы (в нашем случае самого ядра Linux).

```
kernel/profile.c
3 0 void   init profile init(void)
31  {
32   unsigned int size;
33
34   if (!prof on) 3
5      return; 36
```

```
37  /* профилируется только текст */
38 prof 1en =  etext -  stext;
39 prof 1 en »= prof shift;
40
41  size = prof len * sizeof(unsigned int) + PAGE SIZE - 1;
42  prof buffer = (unsigned int *) alloc bootmem(size) ;
43 }
```

Строки 34-35

Ничего не делаем, если профилирование ядра выключено.

Строки 38-39

_extaxt и _jstext определены в kernel /head. c. Мы определяем длину профилирования на основе _extaxt и _stext и затем смещаем значение на prof_shif t, определенное в параметрах ядра.

Строки 41-42

Мы выделяем последовательный блок памяти для хранения данных профилирования, определенного параметрами ядра размера.

8.5.18 Вызов localJrq_enable()

Строка 448

Функция local_irq__enable () позволяет прерывания текущего процессора. Обычно является парной для функции local_irq_disable(). В предыдущих версиях ядра для этих целей использовались пары `sti()` исН(). Несмотря на то, что эти макросы до сих пор ссылаются на sti () и cli (), ключевым словом здесь является **local.** Они действуют только на текущий работающий процессор:

include\asm-i386\system.h
```
446 #define local_irq_disable()  _asm___________volatile ___ ("cli":   :   : "memory")
447 #define local_irq__enable ()  _____ asm_____volatile ___ ("sti":   :   : "memory")
```

Строки 446-447

В соответствии с тем, что описано в разд. 2.4, элементы в кавычках являются ассемблерными инструкциями и окружают список памяти.

8.5Л 9 Настройка initrd

Строки 449-456

Эта инструкция # define выполняет проверку initrd - начального загрузочного диска.

Система, использующая initrd, загружает ядро и монтирует начальный диск в памяти как корневую файловую систему. Программы могут запускаться с диска в памяти и, когда приходит время для новой корневой файловой системы, как и в случае с жестким диском, могут быть смонтированы, а диск в памяти размонтирован.

Эта операция просто проверяет корректность указанного диска в памяти. Если это не так, мы устанавливаем initrd_start в 0, т. е. указываем ядру не использовать диск в памяти.

8.5.20 Вызов memjnit()

Строка 457

Как на x86, так и на PPC вызов mem_init () ищет свободные страницы и посылает информацию о них на консоль. Вспомните из гл. 4, что ядро Linux разбивает доступную память на *зоны*. На данный момент у Linux есть три зоны:

- Zone_DMA. Память меньше 16 Мб.
- Zone_Normal. Память начиная с 16 Мб и меньше 896 Мб. (Ядро использует последние 128 Мб.)
- Zone_HIGHMEM. Память больше 1 Гб.

Функция mem_init () ищет общее количество свободных фреймов страниц во всех зонах памяти. Эта функция выводит на печать информационные сообщения ядра в зависимости от начального состояния памяти. Эта функция архитектурно-зависима, так как она управляет ранними размещенными в памяти данными. Каждая архитектура поддерживает свои собственные функции, несмотря на то что все они выполняют одни и те же действия. Сначала мы рассмотрим, как это делает x86, а затем перейдем к PPC.

```
arch/i386/mm/init
445  void   init mem init(void)
446  {
447   extern int ppro with ram bug(void);
448   int codesize, reservedpages, datasize, initsize;
449   int trap;
450   int bad_ppro;

459  #ifdef CONFIG_HIGHMEM
460   if (PKMAP BASE+LAST PKMAP*PAGE SIZE >= FIXADDR START) {
461   printk(KERN_ERR "fixmap and kmap areas overlap - this will crash\n");
462   printk(KERN_ERR "pkstart: %lxh pkend:%lxh fixstart %lxh\n",
463    PKMAP BASE, PKMAP BASE+LAST PKMAP*PAGE SIZE, FIXADDR START);
464   BUG();
465  }
```

```
466                                  #endif
467
468                              s et_max_mapnr_ini t();

476   /* помещение всей нижней памяти в свободный список */
477    totalram_pages  += ______ free_all_bootmem() ;
478
479
480    reservedpages = 0;
481    for (tmp = 0; tmp < max_low_pfn; tmp++)

485    if (page_is_ram(tmp) && PageReserved(pfn_to__page(tmp) ) )
486     reservedpages++;
487
488                set highmem pages init (bad  ppro) ;
490    codesize = (unsigned long) &  etext - (unsigned long) & text;
491    datasize = (unsigned long) & edata - (unsigned long) & etext;
492    initsize = (unsigned long) &   init end - (unsigned long)
                                    &   ini t begin ;
493
494    kclist add(&kcore mem,    va(0), max low pfn « PAGE SHIFT);
495    kclist_add(&kcore_vmalloc, (void *)VMALLOC_START,
496     VMALLOC END-VMALLOC START) ;
497
498    printk(KERN INFO "Memory: %luk/%luk available (%dk kernel code,
              %dk reserved, %dk data, %dk init, %ldk highmem)\n",
499     (unsigned long) nr free pages() « (PAGE  SHIFT-10),
500    num_physpages « (PAGE_SHIFT-10),
501    codesize » 10,
502    reservedpages « (PAGE SHIFT-10),
503    datasize » 10,
504    initsize » 10,
505    (unsigned long) (totalhigh pages « (PAGE SHIFT-10))
506     );

521   #ifndef CONFIG SMP
522     z ap_low_mappings();
523   #endif
524   }
```

Строка 459

Эта строка последовательно проверяет ошибки так, чтобы восстановленная память и память ядра не перекрывались.

Строка 469

Функция set_max_mapnr_init () (arch/i3 86/mm/init .c) просто устанавливает значение num_physpages, являющейся глобальной переменной (определенной в mm/memory. c), хранящей количество доступных фреймов страниц.

Строка 477

Вызов ___f ree_all_bootniem () помечает как освобождаемые все страницы в нижней памяти. Во время загрузки все страницы резервируются. В данной точке поздней загрузки доступные страницы в нижней памяти освобождаются. Поток вызовов функций показан на рис. 8.15.

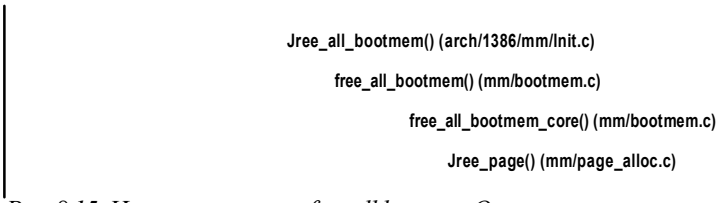


Рис. 8.15. Иерархия вызова free all bootmemQ

Давайте рассмотрим основную часть f ree_all_bootmem_core (), для того чтобы понять, что происходит.

mm/bootmem.c

```
257   static unsigned long ______ init free_all_bootmem_core(pg_data_t *pgdat)
258   {
259    struct page *page;
260    bootmem  data t *bdata = pgdat->bdata;
261    unsigned long i, count, total = 0;

295    page = virt_to_page(bdata->node_bootmem_map);
296    count = 0;
297     for  (i = 0;  i <  ( (bdata->node__low__pfn- (bdata->node_boot_start »
                PAGE_SHIFT))/8  +  PAGE_SIZE-1)/PAGE_SIZE;  i++/page++)   {
298      count++;
299      ClearPageReserved(page);
300      set page count(page, 1); 3 01
         free_page(page);
302                          }
3 03                  total += count;
3 04           bdata->node bootmem map = NULL;
305
3 06                   return total;
307                          }
```

Для всех доступных страниц в нижней памяти мы очищаем флаг PG_reserved[1] в поле flags структуры страницы. Далее мы устанавливаем поле count структуры страницы в 1 для обозначения того, что она используется, и вызываем___ f ree_page (), передавая в нее выделитель близнеца. Как вы помните из описания системы близнецов гл. 4, мы говорили, что эта функция освобождает страницы и добавляет их в список свободных.

Функция___free_all_Jbootmem() возвращает количество доступных в нижней памяти страниц, добавляемое к количеству запущенных totalram_pages (unsigned long, определенное в mm/page_alloc. c).

Строки 480-486

Эти строки обновляют количество зарезервированных страниц.

Строка 488

Вызов set_highmem_pages_init () помечает как инициализированные страницы в верхней памяти. Рис. 8.16 иллюстрирует иерархический вызов set_highmem_ pages_init ().

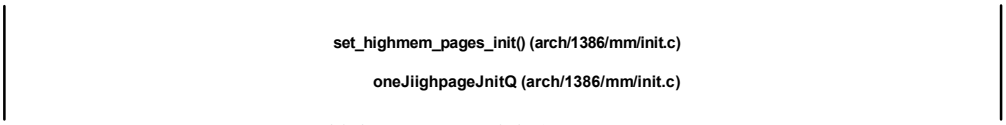


Рис. 8.16. Иерархия вызова set_highmem_pages_init ()

Давайте рассмотрим фрагмент кода, выполняемый в one_highpage_init ().

```
arch/i38 б/mm/ini t.c
253  void _ init one_highpage_init(struct page *page, int pfn,
                                   int bad ppro)
254  {
255       if (page is ram(pfn) && ! (bad ppro && page kills ppro(pfn) ) ) {
256                    ClearPageReserved(page);
257                    set bit(PG highmem, &page->f lags) ;
258                    set page count(page, 1) ;
259                     _ free_page(page);
260                    totalhigh_pages++;
261         } else
262                    SetPageReserved(page);
263  }
```

[1] Из гл. 6 вы помните, что этот флаг устанавливается в страницах, отображаемых в памяти, и для страниц нижней памяти во время ранней загрузки.

Почти аналогично __ free_all__bootmem() все страницы в верхней памяти имеют в своих структурах поле flags с очищенным флагом PG_re served, установлен ным PG__highmem, и полем count, равным 1; ___ f ree_page () также вызывается для добавления этих страниц в свободный список, при этом счетчик totalhigh_pages увеличивается.

Строки 490-506

Этот блок кода собирает и распечатывает информацию о размере областей и количестве доступных страниц.

Строки 521-523

Функция zap_low__mapping сбрасывает начальные TLB PGD в нижней памяти.

Функция mem_init () помечает конец файлы загрузки выделения памяти и начало выделения памяти, которая будет использоваться на протяжении жизни системы. Код mem_ini t () для PPC ищет и инициализирует все страницы для зон.

```
arch/ppc/mm/init.c
3 93  void  __init mem_init (void)
394   {
3 95  unsigned long addr;
3 96   int codepages = 0;
3 97   int datapages = 0;
398  int initpages = 0;
3 99   #ifdef CONFIG HIGHMEM
400  unsigned long highmem mapnr;
402   highmem  mapnr = total lowmem » PAGE SHIFT;
403   highmem start page = mem map + highmem mapnr;
404  #endif /* CONFIG_HIGHMEM */
405   max mapnr = total memory » PAGE SHIFT;
407   high memory = (void *)      va(PPC MEMSTART + total lowmem) ;
408   num  physpages = max mapnr;     /* RAM is assumed contiguous */ 410
totalram pages += free all bootmem();
412   #ifdef CONFIG_BLK_DEV_INITRD
413  /* Если мы загружаемся из BootX с начальным диском в памяти,
414  проверяем, чтобы страницы диска в памяти не были зарезервированы. */
415   if (initrd start) {
416  for (addr = initrd start; addr < initrd end; addr += PAGE SIZE)
417    ClearPageReserved(virt to page(addr));
418  }
419   #endif /* CONFIG BLK  DEV INITRD */
421   #ifdef CONFIG PPC OF
422   /* пометка RTAS-страниц как зарезервированных */
423   if ( rtas_data )
```

```
424     for (addr = (ulong) ___va(rtas_data);
425      addr < PAGE ALIGN( (ulong)    va (rtas data) +rtas size) ;
426      addr += PAGE_SIZE)
427      SetPageReserved(virt to page(addr));
428     #endif
42 9                    #ifdef CONFIG PPC PMAC
43 0                     if (agp special page)
431       SetPageReserved(virt_to_page(agp_special_page) ) ;
432     #endif
433      if ( sysmap )
434       for (addr = (unsigned long)sysmap;
435        addr < PAGE ALIGN( (unsigned long) sysmap+sysmap size) ;
43 6       addr += PAGE SIZE)
437              SetPageReserved(virt_to_page(addr));
43 9   for (addr = PAGE OFFSET; addr < (unsigned long)high memory;
440       addr += PAGE SIZE) {
441       if (!PageReserved(virt to page(addr)))
442        continue;
443       if (addr < (ulong) etext)
444        codepages++;
445       else if (addr >= (unsigned long)&    init begin
446        && addr < (unsigned long)&    init end)
447        initpages++;
448       else if (addr < (ulong) klimit)
449        datapages++;
450      }
452    #ifdef CONFIG HIGHMEM
453     {
454      unsigned long pfn;
456     for (pfn = highmem mapnr; pfn < max mapnr; ++pfn) {
457       struct page *page = mem map + pfn;
459       ClearPageReserved(page);
460       set_bit(PG_highmem, &page->flags);
461       set page count(page, 1);
462      ___free_page(page);
463       totalhigh  pages++;
464      }
465      totalram pages += totalhigh pages;
466     }
467    #endif /* CONFIG HIGHMEM */
469   printk( "Memory: %luk available (%dk kernel code, %dk data,
              %dk init, %ldk highmem)\n",
470      (unsigned long)nr free pages()« (PAGE SHIFT-10),
471      codepages« (PAGE SHIFT-10), datapages« (PAGE SHIFT-10) ,
472      initpages« (PAGE_SHIFT-10),
473      (unsigned long) (totalhigh_pages « (PAGE_SHIFT-10)));
```

```
474   if (sysmap)
475    printk("System.map loaded at 0x%08x for debugger,
                size: %ld bytes\n",
476      (unsigned int)sysmap, sysmap_size);
477   #ifdef CONFIG PPC PMAC
478    if (agp special page)
479     printk(KERN INFO "AGP special page: 0x%081x\n",
                agp  special page) ;
480   #endif
482    /* Будьте уверены, что все наши страницы таблиц страниц имеют
483     корректно установленные page->mapping и page->index. */
484    for (addr = KERNELBASE; addr != 0; addr += PGDIR SIZE) {
485     struct page *pg;
486     pmd t *pmd = pmd offset(pgd offset k(addr), addr);
487     if (pmd__present (*pmd) ) {
488      pg = pmd page(*pmd);
489      pg->mapping = (void *) &init_mm;
490      pg->index = addr;
491     }
492    }
493    mem init done = 1;
494   }
```

Строки 399-410

Эти строки определяют количество доступной памяти. Если используется HIGHMEM, эти страницы также считаются. Глобальная переменная totalram__pages перезаписывается полученными данными.

Строки 412-419

Если используется, очищает любые страницы, применяемые RAM-диском (initrd).

Строки 421-432

В зависимости от окружения загрузки резервируются страницы для служб абстрагирования реального времени и AGP (видео) при необходимости.

Строки 433-450

Если это необходимо, резервируется несколько страниц для карты системы.

Строки 452-467

Если используется HIGHMEM, очищаются любые зарезервированные страницы и изменяется глобальная переменная totalram__pages.

Строки 469-480

Распечатка информации о памяти в консоль.

Строки 482-492

Циклический перебор директорий страниц и инициализация mm__struct и индекса каждой.

8.5.21 Вызов late_timejnit()

Строки 459-460

Функция late_time_J.nit () использует HPET (можете обратиться к обсуждению в подразделе «Вызов timeinit»). Эта функция используется только на архитектуре Intel с HPET. Функция имеет аналогичный time_in.it () код; она просто вызывается после инициализации памяти для выполнения отображения HPET в физическую память.

8.5.22 Вызов calibrate.delayQ

Строка 461

Функция calibrate_delay () в ini t /main. c рассчитывает и выводит значения откалиброванных «BogoMips», означающих количество итераций delay (), которые может выполнить ваш процессор за один тик таймера; calibrate_delay () позволяет примерно уравнять задержки на процессорах с разной скоростью. Результирующее значение - это в основном индикатор скорости работы процессора, который хранится в loop_pre_j if fy, а функции udelay () и mdelay () используются для установки количества выполняемых итераций delay ():

```
Init/main.c
void _ init calibrate_delay(void)
{
  unsigned long ticks, loopbit;
  int lps precision = LPS PREC;
186   1 oops per jiffy = (1«12) ;
  printk("Calibrating delay loop... ");
189  while (loops per jiffy «= 1) {
   /* ожидание "запускающего" тика часов */
   ticks = jiffies;
   while (ticks == jiffies)
    /* ничего*/; /*
   Запуск.. */ ticks
   = jiffies;
     delay(loops per jiffy) ;
ticks = jiffies - ticks; if
(ticks) break; 200   }
```

```
/* Выполнение бинарной аппроксимации для установки loops_per_jiffy,
   равной одному (до lps_precision бит) */
2 04  loops per jiffy »= 1;
  loopbit = loops_per_jiffy; 206  while (
lps precision-- && (loopbit »= 1) ) {
   loops_per_jiffy |= loopbit;
   ticks = jiffies;
   while (ticks == jiffies);
   ticks = jiffies;
   __delay(loops_per_jiffy);
if (jiffies != ticks)     /* longer than 1 tick */
loops per jiffy &= -loopbit; 214 }
/* Округление и вывод значения */ 217
printk("%lu.%021u BogoMIPSXn",
   loops per jiffy/(500000/HZ), 219
(loops_per_jiffy/(5000/HZ)) % 100); }
```

Строка 186

Начало в 0x800.

Строки 189-200

Сохранение удвоенного loops_per_jif fy до тех пор, пока занимаемое функцией delay (loops_per_j iffy) не достигнет одного мгновения.

Строка 204

Деление loops_per_j if fy на 2.

Строки 206-214

Добавление убывающей степени 2 к loop_per_j if f y до тех пор, пока тик не станет равен моменту.

Строки 217-219

Распечатка значений в вещественном формате.

8.5.23 Вызов pgtable_cache_init()

Строка 463

Ключевой функцией в блоке кода x86 является системная функция kmem_ceche_ create (). Эта функция создает именованный кеш. Первый параметр - это строка, используемая для ее идентификации в /proc/slabinf o.

arch/i38 б/mm/ini t.c

```
52 9 kmem_cache_t *pgd_cache;
53 0 kmem_cache_t *pmd_cache; 531
532 void    init pgtable cache init(void)
533 {
53 4   if (PTRS PER PMD > 1) {
53 5   pmd cache = kmem cache create("pmd",
53 6           PTRS PER PMD*sizeof(pmd t),
537           0,   53 8    SLAB HWCACHE ALIGN | SLAB MUST H WCACHE ALIGN,
53 9          pmd_ctor,
540           NULL);
541    if (!pmd_cache)
542      panic("pgtable cache init(): cannot create pmd cache");
543  }
544  pgd_cache = kmem_cache_create("pgd",
545         PTRS PER PGD*sizeof(pgd t),
546         0,
547         SLAB HWCACHE ALIGN | SLAB MUST HWCACHE ALIGN,
548         pgd ctor,
549         PTRS PER PMD == 1 ? pgd dtor : NULL);
550  if (!pgd cache)
551    panic("pgtable cache init(): Cannot create pgd cache");
552 }
```

```
arch/ppc64/ram/init.c
976 void pgtable_cache_init(void)
977 {
978   zero cache = kmem cache create("zero",
979         PAGE.SIZE,
980         0,
981         SLAB HWCACHE ALIGN | SLAB MUST HWCACHE ALIGN,
982         zero_ctor,
983         NULL);
984   if (!zero cache)
985     panic("pgtable__cache_init(): could not create zero_cache !\n");
986 }
```

Строки 532-542

Создание кеша pmd.

Строки 544-551

Создание кеша pgd.

На PPC с аппаратно-назначенным хешированием pgtable_cache__in.it () не является операцией.

```
include\asmppc\pgtable.h
685  #define pgtable_cache_init()  do { } while (0)
```

8.5.24 Вызов bufferjnit()

Строка 472

Функция buf f er_init () в f s/buf fer .c хранит данные устройства файловой системы.

```
fs/buffer.c
3031 void __ init buffer_init(void)
{
  int i ;
  int nrpages; 3 036   bh cachep =
kmem cache create("buffer  head",
    sizeof(struct buffer head), 0,
    0, init buffer head, NULL); 3039    for (i = 0; i <
ARRAY SIZE(bh wait queue heads) ; i++)
    init waitqueue head(&bh wait  queue heads[i].wqh);
3044   nrpages = (nr free buffer pages() * 10) / 100;
    max_buffer_heads = nrpages * (PAGE_SIZE /
    sizeof(struct buffer head));
    hotcpu  notifier(buffer cpu notify, 0);
3048   }
```

Строка 3036

Выделение буфера для хеш-таблицы кеша.

Строка 3039

Создание таблицы буфера очереди ожидания хеша.

Строка 3044

Ограничение занятости нижней памяти до 10 %.

8.5.25 Вызов security_scafolding_startup()

Строка 474

Ядро Linux 2.6 содержит код для загрузки модулей ядра с использованием различных мер безопасности; security_scafolding_startup() просто

проверяет наличие объекта операции безопасности, и если он присутствует, то вызывает функции инициализации модулей безопасности.

Какие модули могут быть созданы и с чем может столкнуться их автор - выходит за пределы рассмотрения книги. Более подробную информацию можно найти в модулях безопасности Linux (http: //Ism. immunix. org/) и списках рассылки модулей безопасности Linux (http: //mail .wirex.com/mailman/listinfo/linux-securi-ty-module).

8.5.26 Вызов vfs_cachesjnit()

Строка 475

Подсистема VFS зависит от кешей памяти, называемых SLAB-кешами, хранящими структуры, которыми эта подсистема управляет. Гл. 4 подробно описывает SLAB-кеши. Функция vfs__caches_init () инициализирует используемые системой кеши SLAB. На рис. 8.17 показан обзор иерархии вызовов из vf s_cache_init (). Мы подробно рассмотрим каждую функцию из этой иерархии вызовов. Вы можете сверяться с этой иерархией по мере того, как мы будем рассматривать входящие в нее функции.

В табл. 8.4 подводится итог представляемых vf s_caches_init () или вызываемыми из нее функциями новых функций.

Таблица 8.4. Представляемые vfs_caches_init объекты

Объект	Описание
names_cachep	Глобальная переменная
f lip_cachep	Глобальная переменная
inode_cache	Глобальная переменная
dentry_cache	Глобальная переменная
mnt_cache	Глобальная переменная
namespace	Структура
mount_hashtable	Глобальная переменная
root_f s_type	Глобальная переменная
f ile_system_type	Структура (описывается в гл. 6)
bdev_cachep	Глобальная переменная

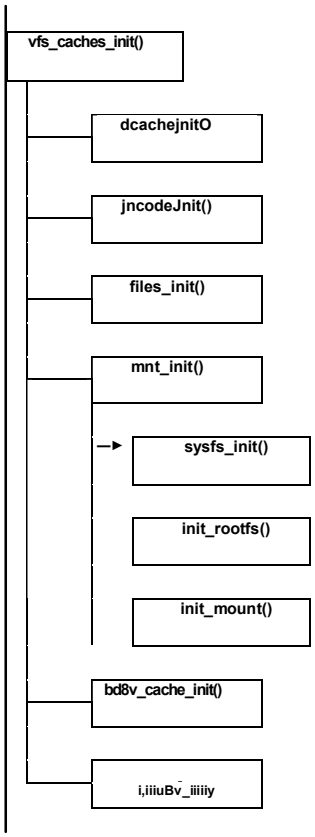


Рис. 8.17. Иерархия вызовов vfs_caches_init()

```
fs/dcache.c
1623  void   init vfs caches init(unsigned long mempages)
1624  {
1625   names cachep = kmem cache create("names cache",
1626     PATH MAX, 0,
1627     SLAB HWCACHE ALIGN, NULL, NULL);
1628   if (!names cachep)
1629    panic("Cannot create names SLAB cache");
1630
1631   filp cachep = kmem cache  create ("f ilp",
1632     sizeof(struct file), 0,
1633     SLAB_HWCACHE_ALIGN, filp_ctor, filp_dtor);
```

```
1634    if(!filp cachep)
1635     panic("Cannot create filp SLAB cache");
1636
1637    dcache init(mempages);
1638    inode init(mempages);
1639    files init(mempages);
1640    mnt init(mempages);
1641    bdev  cache init () ;
1642    chrdev init();
1643   }
```

Строка 1623

Функция получает глобальную переменную num_physpages [значение которой вычисляется во время mem_init O] в качестве параметра, хранящего число доступных физических страниц, в системной памяти. Это число влияет на создание кеша SLAB, как будет описано позже.

Строки 1625-1629

Следующим шагом является создание области памяти nanies_cachep. В гл. 4 подробно описана функция kmem_cache_create (). Эта область памяти хранит объекты размера PATH_MAX, т. е. максимально доступной длины пути в символах. (Значение устанавливается в linux/limits. h в значение 4096.) В этой точке кеш создан как пустой объект или область памяти размера PATH_MAX. Настоящая область памяти выделяется во время первой потенциальной последовательности вызовов к getname ().

Как сказано в гл. 6, функция get паше () вызывается в начале того же связанного с файлом системного вызова [в нашем примере sys_open ()] для чтения имени файла из адресного пространства процесса. Объекты освобождаются из кеша с помощью функции putname ().

Если names_cache не может быть создан, ядро переходит к функции паники и забирает управление у потока функции.

Строки 1631-1635

Далее создается f ilp_cache с объектами размером со структуру файла. Объект, хранящий структуру файла, выделяется с помощью функции get_empty_f ilp () (fs/file_table.c), вызываемой, например, при создании канала или открытии файла. Объект описателя файла освобождается с помощью вызова file_free() (fs/file_table.c).

Строка 1637

Функция dcache_init () (f s/dcache.c) создает кеш SLAB, хранящий пустой описатель[1]. Сам кеш называется dentry_cache. Сам описатель dentry создается

для каждого иерархического компонента в пути файла, запрошенном процессом при доступе к файлу или директории. Структура связывает компонент директории с представляющей ее inode для дальнейшего ускорения доступа к компоненту через соответствующий inode.

Строка 1638

Функция inode_init () (fs/inode. c) инициализирует хеш-таблицу inode и голову массива очереди ожидания, используемую для хранения хешированных inode, которые хочет заблокировать ядро. Голова очереди ожидания (wait_queue_head_t) для хешируемых inode хранится в массиве, называемом i_wait_queue_head. Этот массив инициализируется в данной точке процесса запуска системы.

В данной точке создается inode_hashtable. Эта таблица ускоряет поиск inode. Последним происходит кеширование SLAB, используемое для хранения созданных объектов inode. Она называется inode__cache. Область памяти для этого кеша выделяется при вызове alloc_inode (fs/inode.c) и освобождается при вызове destroy_inode() (fs/inode.c).

Строка 1639

Функция f iles_init () вызывается для определения максимального количества памяти, разрешенной для файлов на один процесс. Устанавливается поле max_f iles структуры f iles_stat. Далее она используется при создании файла для определения того, достаточно ли памяти для открытия файла. Давайте рассмотрим эту функцию.

```
fs/file_table.c
292   void  ___init  files_init(unsigned  long mempages)
293   {
294     int  n;

299   n = (mempages * (PAGE_SIZE / 1024)) / 10;
3 00  files stat.max files = n;
3 01  if  (files_j3tat.max_files < NR_FILE)
3 02      files_stat.max_files  = NR_FILE;
303   }
```

Строка 299

Размер страницы делится на количество пространства (вместе со связанными inode и кешем), которое займет файл (в данном случае 1 Кб). Это значение умножается на

[1] Вспомните, что dentry - это сокращение от «элемент директории».

количество страниц для получения общего количества «блоков», которые можно использовать для файлов. Деление на 10 дает значение предела использования памяти для файлов, при котором они будут занимать не более 10 % доступной памяти.

Строки 301-302

NR_FILE (include/linux/f s.h) устанавливается в 8192.

Строка 1640

Следующая функция, mnt_init (), создает кеш, хранящий объекты vf smount, используемые VFS для монтирования файловых систем. Кеш называется mnt_cache. Также функция создает массив mount_hachtable, хранящий ссылки на объекты в mnt_cache для быстрого доступа. Далее выполняются вызовы для инициализации файловой системы sysf s и монтирования файловой системы root. Давайте рассмотрим подробнее создание таблицы хеширования.

```
fs/namespace.c
1137 void __ init mnt_init(unsigned long mempages)
{
113 9                 struct list head *d;
1140   unsigned long order;
1141   unsigned int nr hash;
1142   int i;

1149   order = 0;
1150   mount hashtable = (struct list head *)
1151      get free pages(GFP ATOMIC, order);
1152
1153   if (!mount hashtable)
1154    panic("Failed to allocate mount hash table\n");

1161  nr  hash = (1UL « order) * PAGE  SIZE / sizeof (struct list head) ;
1162   hash_bits = 0;
1163   do {
1164     hash bits++;
1165     } while ((nr_hash » hash_bits) != 0);
1166   hash_bits—;

1172   nr_hash = 1UL « hash_bits;
1173   hash mask = nr hash-l;
1174
1175printk("Mount-cache hash table entries: %d (order: %ld,
              %ld bytes)\n", nr__hash, order, (PAGE_SIZE « order));

1179   d = mount_hashtable;
```

```
1180    i = nr hash;
1181    do {
1182     INIT_LIST_HEAD(d) ;
1183     d++;
1184     i—;
1185    } while (i);

1189                }
```

Строки 1139-1144

Массив хеш-таблицы состоит из полной страницы памяти. Гл. 4 подробно описыва ет работу этой функции ___ get_f ree_pages (). Самое главное в том, что эта функция возвращает указатель на область памяти в размере страниц второго порядка. В этом случае мы выделяем одну страницу для хранения таблицы хеша.

Строки 1161-1173

Следующим шагом является определение количества вхождений в таблице; nr_hash настраивается для хранения номера порядка (степень двойки) головы списка, который помещается в таблицу, hash__bits вычисляется как количество необходимых для представления наибольшей степени двойки hr_hash битов. Строка 1172 переопределяется nr__hash, чтобы она состояла из одного левого бита. Далее битовая маска пересчитывается на основе нового значения hr_hash.

Строки 1179-1185

Наконец, мы инициализируем хеш-таблицу с помощью вызова макроса INIT_LIST__HEAD, получающего указатель на область памяти, в которой инициализируется новая голова списка. Мы проделываем это nr_hash раз (по количеству элементов, которые может хранить таблица).

Давайте рассмотрим пример. Мы предполагаем, что PAGE_SIZE равна 4 Кб и list_head равна 8 байтам. Из-за того что порядок равен 0, значение nr_hash становится равным 500; поэтому в таблицу 4 Кб может входить до 500 элементов; (1UL«order) становится количество выделенных страниц. Например, если порядок равен 1 (что значит, что мы запрашиваем 21 страницу в хеш-таблице), сдвинутый влево бит 00000001 становится равным 00000010 (или 2 в двоичной системе счисления). Далее мы рассчитываем количество битов, необходимых хеш-коду. Проследив каждую итерацию цикла, мы получим следующее.

Начальные значения hash_bits = 0 и nr_hash = 500.

- Итерация 1: hash_bits = 1, и (500 » 1)! = 0
  (000111110100» 1) = 000011111010
- Итерация 2: hash__bits = 2, и (500 » 2)! = 0

(000111111010 » 2) = 000001111110

- Итерация 3: hash_bits = 3, и (500 » 3)! = 0
  (000111111010 » 3) = 000000111111
- Итерация 4: hash_bits = 4, и (500 » 4)! = 0
  (000111110100» 4) = 000000011111
- Итерация 5: hash_bits = 5, и (500 » 5)! = 0
  (000111111010» 5) = 000000001111
- Итерация 6: hash__bits = 6, и (500 » 6)! = 0
  (000111111010» 6) = 000000000111
- Итерация 7: hash_bits = 7, и (500 » 7)! = 0
  (000111110100 » 7) = 000000000011
- Итерация 8: hash_bits = 8, и (500 » 8)! = 0
  (000111111010 » 8) = 000000000001
- Итерация 9: hashjaits = 9, и (500 » 9)! = 0
  (000111111010 » 3) = 000000000000

После остановки цикла while функция hash_bits декрементируется до 8, nr_hash устанавливается в 000100000000 и hash_mask устанавливается в 000011111111.

После того как функция mnt_init () инициализирует mount_hashtable и создает mnt_cache, она выполняет три вызова:

```
fs/namespace.c

1189  sysfs init();
1190  init_rootfs();
1191  init mount tree();
1192 }
```

sysf s__in.it () отвечает за создании файловой системы sysf s; init__f ootds () и init_mount__tree () вместе отвечают за монтирование корневой файловой системы. Мы подробно рассмотрим каждый шаг этих функций.

```
fs/ramfs/inode.c
218  static struct file system type rootfs fs type = {
219    .name = "rootfs",
22 0   .get sb = rootf s  get  sb,
221    .kill_sb = kill_litter_super,
```

```
222  };

237  int   init init rootf s (void)
238  {
239   return register filesystem(&rootfs fs type);
240  }
```

Файловая система rootf s - это первая монтируемая ядром файловая система. Это простая пустая директория, которая *перемонтируется* реальной файловой системой на более поздних этапах загрузки ядра.

Строки 218-222

Этот блок кода объявляет структуру rootf s_fs_type f ile_system_type. Для получения и удаления связанных суперблоков определяются только два метода.

Строки 237-240

Функция init__rootfs () просто регистрирует rootf s в ядре. Таким образом, вся связанная с типом файловой системы информация становится доступной (информации хранится в структуре f ile_system_type) внутри ядра.

```
init_mount_tree()
fs/namespace.c
1107  static void   init init mount tree(void)
1108  {
1109   struct vfsmount *mnt;
1110   struct namespace *namespace;
1111   struct task  struct *g, *p;
1112
1113   mnt = do kern mount("rootfs", 0, "rootfs", NULL);
1114   if (IS ERR(mnt))
1115    panic("Can't create rootfs");
1116   namespace = kmalloc(sizeof(*namespace), GFP KERNEL);
1117   if (Inamespace)
1118    panic("Can*t allocate initial namespace");
1119   atomic_set(&namespace->count, 1);
1120   INIT LIST HEAD(6cnamespace->list) ;
1121   init_rwsem(&namespace->sem) ;
1122   list add(&mnt->mnt list, &namespace->list);
1123   namespace->root = mnt;
1124
1125   init  t ask. namespace = namespace;
1126   read lock(&tasklist lock);
1127   do.each.threadfg, p) {
1128    get_namespace(namespace);
```

```
112 9       p->namespace  = namespace;
ИЗО      }  while_each_thread(g,  p) ; 1131
read_unlock(&tasklist_lock); 1132
1133   set_fs_pwd(current->fs, namespace->root,
                  namespace->root->mnt root);
1134   set_fs_root(current->fs, namespace->root,
                   namespace->root->mnt root);
1135   }
```

Строки 1116-1123

Инициализация пространства имен процесса. Эта структура хранит указатели на смонтированные древовидные структуры и соответствующие dentry. Выделяется объект namespace, счетчик устанавливается в 1, инициализируется список полей типа list_head, инициализируются семафоры, блокирующие пространство имен (и дерево монтирования), а поле root, соответствующее структуре vf smount, устанавливается таким образом, чтобы указывать на новые выделенные vf smount.

Строка 1125

Поле namespace описателя процесса текущей задачи (задача init) устанавливается так, чтобы указывать на только что выделенный и инициализированный объект пространства имен. (Текущим процессом является процесс 0.)

Строки 1134-1135

Следующие две функции устанавливают значения четырех полей в f s_struct, связанной с нашим процессом; f s_struct, хранящая поле для корня, и элементы текущей рабочей директории устанавливаются следующими двумя функциями.

Мы только что закончили рассмотрение того, что происходит в функции mnt_init. Давайте продолжим рассмотрение vf s_mnt_init.

```
1641 bdev_cache_init()
fs/block dev.c
290 void   init bdev cache init(void)
291 {
292  int err;
293  bdev_cachep = kmem_cache_create(Hbdev_cache",
294    sizeof(struct bdev_inode),
295    0,
2 96  SLAB HWCACHE ALIGN | SLAB RECLAIM ACCOUNT,
2 97   init  once,
2 98   NULL);
299   if (!bdev_cachep)
```

```
3 00    panic("Cannot  create bdev_cache  SLAB cache");
3 01    err  =  register_filesystem(&bd_type);
302     if  (err)
303     panic("Cannot register bdev pseudo-fs");
3 04    bd mnt = kern mount(&bd type);
3 05    err = PTR_ERR(bd_mnt);
306     if (IS  ERR(bd mnt) )
307       panic("Cannot  create bdev pseudo-fs");
3 08    blockdev_superblock  = bd_mnt->mnt_sb;      /* Для обратной записи */
309     }
```

Строки 293-298

Создаем кеш SLAB bdev_cache, хранящий bdev__inodes.

Строка 301

Регистрация специальной файловой системы bdev. Определяется следующим образом:

```
fs/block dev.c
2 94   static struct file system type bd  type = {
295      .name  = "bdev",
296      .get sb  = bd get sb,
297      .kill_sb  = kill_anon_super,
298    };
```

Как вы можете видеть, структура f ile_system специальной файловой системы bdev имеет только две определенные функции: одну для выборки суперблоков файловой системы и другую для удаления-освобождения суперблока. В этой точке может удивить, почему блочное устройство регистрируется как файловая система. В гл. 6 мы видели, что системы, не являющиеся технически файловыми системами, могут использовать структуры файловой системы ядра; поэтому они не имеют точек монтирования, но могут использовать структуры VFS ядра для поддержки файловых систем. Блочные устройства являются одним из видов псевдофайловых систем, которые используют структуры файловой системы VFS ядра. В случае bdev эти структуры представляют собой ограниченный набор полей, так как не все из них имеют смысл для отдельных приложений.

Строки 304-308

Вызов kern_mount () устанавливает все связанные с монтированием VFS-структуры и возвращает структуру vf smount. (См. подробное описание глобальной переменной bd_mnt, указывающей на структуру vf smount, и blockdev_superblock, указывающей на суперблок vf smount, в гл. 6.)

Эта функция инициализирует объекты символьного устройства, которые описывают модель драйвера:

```
fs/char_dev.c
void _ init chrdev_init(void)
{
433   subsystem init(&cdev subsys);
434   cdev_map = kobj_map_init(base_probe, &cdev_subsys);
435   }
```

8.5.27 Вызов radix_treejnit()

Строка 476

Ядро Linux 2.6 использует корневое дерево для управления страницами в кеше страниц. Здесь мы просто инициализируем последовательную секцию пространства ядра для хранения корневого дерева страницы кеша:

lib/radix-tree.c

```
798  void  ___init  radix__tree_init (void)
799  {
800    radix tree node cachep = kmem cache create(Hradix tree node",
801       sizeof (struct radix  tree node) , 0,
802       SLAB_PANIC, radix_tree_node_ctor, NULL);
803    radix tree init maxindex();
804    hotcpu notifier(radix tree callback, 0) ;
```

```
lib/radix-tree.c
768  static   init void radix tree init maxindex(void)
769  {
770    unsigned int i;
771
772    for (i = 0; i < ARRAY_SIZE(height_to_maxindex); i++)
773      height_to_maxindex[i] = _ maxindex(i);
774  }
```

Обратите внимание, как radix_tree_init () выделяет пространства кеша страниц, а radix_tree_init_maxindex () настраивает хранилище данных корневого дерева, height_to_maxindex [ ].

hotcpu_notif ier () (в строке 804) связана с возможностью горячего подключения процессоров в Linux 2.6. Когда процессор подключается вгорячую, ядро вызывает

radix_tree_callback (), которая пытается очистить все части кеша страниц, связанные с горячим подключением процессоров.

8.5.28 Вызов signalsjnit()

Строка 477

Функция signals_init () в kernel /signal. с инициализирует очередь сигналов ядра:

```
fs/buffer.c
2565  void  _ init signals_init(void)
2566  {
2567  sigqueue cachep =
2 568      kmem__cache_create ("sigqueue",
2569        sizeof(struct sigqueue),
2570        __ alignof__ (struct sigqueue),
2571        0, NULL, NULL);
2572  if (!sigqueue cachep)
2573   panic("signals_init(): cannot create sigqueue SLAB cache");
2574  }
```

Строки 2567-2571

Выделение памяти SLAB для sigqueue.

8.5.29 Вызов page_writebackjnit()

Строка 479

Функция page_writeback_init () инициализирует значение, контролируемое, когда грязные страницы записываются обратно на диск. Грязные страницы записываются на диск не сразу; они записываются по прошествии некоторого времени или после того, как некоторая часть в процентах памяти будет помечена как грязная. Эта функция init пытается определить оптимальное количество страниц, которые должны быть помечены как грязные, перед запуском фоновой и специальной записей. Фоновая запись грязных страниц занимает намного меньше процессорного времени, чем специальная запись грязных страниц:

```
mni/page-writeback. c
488 /*
489  * Если в машине много верхней памяти, рейтинг нижней памяти
490  * снижается до значения порога грязной памяти по умолчанию,
491  * предлагая больше места грязной верхней памяти, чем количество
492  * buffer_heads. */
```

```
493 void __init page_writeback_init(void)
494 {
495   long buffer_pages = nr_free_buffer_pages();
496   long correction;
497
498   total_pages = nr_free_pagecache_pages();
499
500correction = (100 * 4 * buffer_pages) / total_pages;
501
502   if (correction < 100) {
503     dirty_background_ratio *= correction;
504     dirty_background_ratio /= 100;
505     vm_dirty_ratio *= correction;
506     vm_dirty_ratio /= 100;
507   }
508   mod_timer(&wb_timer, jiffies +
                              (dirty_writeback_centisecs * HZ) / 100) ;
509   set_ratelimit();
510   register_cpu_notifier(&ratelimit_nb);
511 }
```

***Строки** 495-507*

Если мы работаем на машине с большим кешем страниц по сравнению с количеством буферов страниц, мы уменьшаем порог записи грязных страниц. Если мы выбираем не снижать порог, который увеличивает частоту записи при каждой записи, мы будем использовать чрезмерное количество buf fer_heads. [Это означает и комментарий перед page_writevack ().]

Фоновая запись по умолчанию dirty_background_ratio начинается, когда грязными становятся 10 % страниц. Специальная запись, vm_dirty__ratio начинается при 40 % грязных страниц.

Строка 508

Мы модифицируем время записи, wb_timer, для периодического запуска (каждые 5 с по умолчанию).

Строка 509

Вызывается отлично документированная set_ratelimit (). Предоставим слово встроенным комментариям.

mm/page-writeback.c

```
450 /*
451  * Если ratelimit_pages слишком высок, то у нас может случиться
452  * переизбыток грязных данных, если несколько процессов выполняет
```

```
453  * запись в одно и то же время. Если он слишком мал, машина SMP
454  * часто вызывает (дорогостоящую) get_writeback_state.
455  *
456  * Здесь мы устанавливаем ratelimit pages до уровня, на котором все
457  * процессоры имеют одинаковый уровень грязной памяти, и мы не можем
458  * превысить 3% (1/32) от порога грязных страниц до прекращения
459  * записи.
460  * Этот предел нельзя задавать слишком большим. Из за-того, что он
461  * также контролирует размер памяти с помощью вызова
462  * balance_dirty_pages () для записи. Если оно слишком велико,
463  * вызывающей код каждый раз должен блокировать очередь ввода-вывода.
464  * Предел равен 4 Мб, а вызывающий balance dirty_pages ()
465  * код записывает до максимальных б Мб. */
466
467 static void set_ratelimit(void)
468 {
469   ratelimit pages = total pages / (num online cpus () * 32);
470   if (ratelimit pages < 16)
471     ratelimit__pages = 16;
472   if (ratelimit pages * PAGE CACHE SIZE > 4096 * 1024)
473     ratelimit pages = (4096 * 1024) / PAGE CACHE SIZE;
474 }
```

Строка 510

Последняя команда, page_writeback_init (), регистрирует блок уведомителя рейтинга, ratelimit_nb, с уведомлением процессора. Блок уведомителя рейтинга вызывает ratelimit_handler () при уведомлении и, в свою очередь, вызывает set_ratelimit (). Это сделано для пересчета retelimit_pages, когда изменяется количество включенных процессоров.

```
mm/page-writeback.c
483 static struct notifier block ratelimit nb = {
484     .notifier call  = ratelimit handler,
485     .next           = NULL,
486 };
```

И наконец, нам нужно понять, что происходит, когда wb_timer (из строки 508) истекает и вызывается wb_time_f n ().

```
mm/page-writeback.c
414  static void wb_timer_fn(unsigned long unused)
415  {
```

```
416   if (pdflush operation(wb kupdate, 0) < 0)
417    mod timer(&wb_timer, jiffies + HZ); /* delay 1 second */
418 }
```

Строки 416-417

По истечении таймера ядро активизирует pdf lush_operation(), которая будит один из потоков pdf lush для выполнения настоящей записи грязных страниц на диск. Если pdf lush_operation() не может разбудить ни один из потоков pdf lush, она сообщает таймеру записи сработать снова через 1 с для повторной попытки пробуждения потока pdf lush. (См. более подробную информацию о pdf lush в гл. 9, «Сборка ядра Linux».)

8.5.30 Вызов proc_rootJnit()

Строки 480-482

Как описано в гл. 2, CONFIG_* #def ine связан с переменной времени компиляции. Если во время компиляции выбрана файловая система proc, то следующим шагом инициализации будет вызов proc_root_init ():

```
fs/proc/root.c
40  void   init proc root init(void)
41  {
42   int err = proc_init_inodecache();
43   if (err)
44    return;
45   err = register f ilesystem(&proc fs  type) ;
46   if (err)
47    return;
48   proc_mnt = kern_mount(&proc_fs_type);
49   err = PTR_ERR(proc_mnt);
50   if (IS_ERR(proc_rant)) {
51    unregister filesystem(&proc  fs type);
52    return;
53   }
54   proc misc init();
55   proc_net = proc_mkdir("net", 0);
56  #ifdef CONFIG SYSVIPC
57   proc_mkdir("sysvipc", 0);
58  #endif
59  #ifdef CONFIG_SYSCTL
60   proc_sys__root = proc_mkdir ("sys", 0) ;
61  #endif
62  #if defined(CONFIG_BINFMT_MISC)  ||  defined(CONFIG_BINFMT_MISC_MODULE)
```

```
63   proc mkdir("sys/fs", 0);
64   proc mkdir( "sys/f s/binfm^misc", O);
65  #endif
66   proc root fs = proc  mkdir ("f s", 0) ;
67   proc root driver = proc mkdir("driver", 0);
68   proc_mkdir("fs/nfsd", 0); /* somewhere for the nfsd filesystem
                                  to be mounted */
69  #if defined(CONFIG SUN OPENPROMFS) ||
                                 defined(CONFIG SUN OPENPROMFS MODULE)
70   /* просто получаем точку монтирования */
71   proc mkdir("openprom", 0);
72  #endif
73   proc tty init();
74  #ifdef CONFIG_PROC_DEVICETREE
75   proc device tree init();
76  #endif
77   proc_bus = proc_mkdir("bus",  0);
78  }
```

Строка 42

Эта строка инициализирует inodeKeuia, хранящий inode для данной файловой системы.

Строка 45

Структура f ile__system_type proc__f s_type регистрируется в ядре. Давайте подробнее рассмотрим эту структуру.

```
fs/proc/root.c
33  static struct file system type proc fs type = {
34     .name           = "proc",
3 5    .get sb     = proc get sb,
36     .kill_sb          = kil^anon^super,
37  };
```

Структура f ile_system_type, определяющая имя файловой системы как proc, имеет функцию для получения и освобождения структуры суперблока.

Строка 48

Мы монтируем файловую систему proc. (См. подробности в сноске для kern_mount, чтобы понять, что там происходит.)

Строки 54-78

Далее вызов proc_misc__init () создает большинство элементов, которые можно увидеть в файловой системе /proc. Она создает элементы с помощью вызова create_j?roc_read_entry(), create_proc_entry() и create_proc_ sec£_entry (). Оставшийся код блока содержит вызовы к proc_mkdir для создания директорий в /proc, вызов функции proc_tty_init () для создания дерева внутри /proc/tty, и если установлена переменная настройки CONFIG_ PROC_DEVICETREE, то вызывается функция proc__device_tree_init () для создания поддерева /proc/device-tree.

8.5.31 Вызов initjdle()

Строка 490

init_idle () вызывается почти в конце start__kernel () с параметрами current и smp_j?rocessor_id() для подготовки start_Jkernel () для перепланировки.

```
kernel/sched.c
2643 void   init init idle(task  t *idle, int cpu)
2644 {
2 645  runqueue t *idle rq = cpu rq(cpu), *rq = cpu rq(task cpu(idle) ) ;
2 646  unsigned long flags;
2647
2648   local irq save(flags);
2649   double_rq_lock(idle_rq, rq) ;
2650
2651   idle rq->curr = idle rq->idle = idle;
2652   deactivate task(idle, rq);
2653   idle->array = NULL;
2 654    idle->prio = MAX_PRIO;
2655    idle->state = TASKJRUNNING;
2656    set^ask^cpuddle,  cpu) ;
2 657   double_rq__unlock(idle_rq, rq) ;
2658   set  tsk  need resched(idle) ;
2659   local irq restore(flags);
2660
2661   /* Установка зарезервированного числа соединительных блокировок
        * _outside__ ! */ 2
662 #ifdef CONFIG  PREEMPT
2 663   idle->thread info->preempt count = (idle->lock depth >= 0);
2664 #else 2 665   idle->thread_info->preempt__count = 0;
```

```
2666  #endif
2667  }
```

Строка 2645

Мы сохраняем очередь запроса процессора для текущего процессора и очередь запросов процессора, на котором выполняется задача idle. В нашем случае при current и smp_processor_id () эти очереди запросов будут одинаковые.

Строки 2648-2649

Мы сохраняем флаги IRQ и выполняем блокировку обеих очередей запросов.

Строка 2651

Мы устанавливаем текущую задачу очереди запроса процессора для процессора, на котором выполняется задача idle.

Строки 2652-2656

Эти строки убирают задачу idle из очереди запросов и перемещает ее в очередь запросов процессора ери.

Строки 2657-2659

Мы освобождаем блоки очереди запроса для заблокированной до этого очереди выполнения. Далее мы отмечаем задачу idle для перепланировки и восстанавливаем IRQ, которые были сохранены ранее. Наконец, мы устанавливаем счетчик приоритетного прерывания обслуживания, если настроено приоритетное прерывание обслуживания ядра.

8.5.32 Вызов rest_init()

Строка 493

Функция rest_init () кажется прямолинейной. Она просто создает поток, называемый init, убирает блокировку инициализации ядра и вызывает поток idle:

```
init/main.c
388  static void noinline rest_init(void)
389  {
390   kernel_thread(init, NULL, CLONE_FS | CLONE_SIGHAND);
3 91   unlock kernel();
3 92   cpu idle();
393  }
```

Строка 388

Вы могли обратить внимание, что первая вызываемая start__kernel () функция не имеет приставки __ init. Из гл. 2 вы можете вспомнить, как мы говорили, что, если функция имеет приставку ____ init, это значит, что все используемые ей переменные и память будут очищены/освобождены после завершения инициализации. Это выполняется с помощью функции f ree_initmem (), которую мы видели, когда рассматривали, что происходит в init(). Причина, по которой rest_init () не объявляется_____init, заключается в том, что она вызывает поток init до его завершения (имеется в виду вызов cpu_idle). Так как поток init выполняет вызов free_initmem(), возможно возникновение соревновательной ситуации, если free_initmeni() вызывается перед завершением rest_init() (или корневым потоком).

Строка 390

Эта строка создает поток init, связанный также с процессом init, или процессом 1. Коротко говоря, все, что мы здесь сказали, распространяется на все структуры данных для вызывающих процессов. Поток ядра вызывает функцию init O, которую мы рассмотрим в следующем разделе.

Строка 391

Функция unlock_kernel () ничего не делает на однопроцессорной системе. В противном случае она освобождает BKL.

Строка 392

Вызов cpu_idle () превращает корневой поток в поток ожидания. Эта функция передает процессор планировщику и возвращается, когда у планировщика не остается необработанных процессов для запуска.

В этой точке мы закончили основную работу по инициализация ядра Linux. Теперь мы вкратце рассмотрим, что происходит в вызове init ().

8.6 Поток init (или процесс 1)

Теперь мы рассмотрим поток init. Обратите внимание, что мы пропускаем все связанные с SMP функции.

```
init/main.c
601   static frit init (void * unused)
602   {
603    lock kernel();

612    child_reaper = current;
```

```
627   populate_rootfs();
629   do__basic_setup () ;

63 5  if (sys_access((const char     user *) "/init", 0) == 0)
63 6   execute_command = "/init";
637   else
63 8   prepare_namespace();

645    free_initmem();
646    unlock kernel();
647    system state = SYSTEM RUNNING;
649    if (sys open( (const char    user *) "/dev/console", 0 RDWR, 0) < 0)
650     printk("Warning: unable to open an initial console.\n");
651
652    (void) sys_dup(0);
653    (void) sys_dup(0);

662    if (execute command)
663     run_init__process (execute_command) ;
664
665    run init process("/sbin/init");
666    run init process("/etc/init");
667    run init process("/bin/init");
668    run init process("/bin/sh");
669
670    panic("No init found.  Try passing init= option to kernel.");
671   }
```

Строка 612

Поток init настроен на уничтожение любых потоков, чьи родители мертвы. Переменная child_reaper является глобальным указателем на task_struct и определена в init /main. c. Эта переменная входит в игру в «повторяющейся функции» и используется для ссылки на поток, который должен стать новым родителем. Мы ссылаемся на функции reparent_to_init () (kernel/exit .c), choose_new_parent () (kernel/exit .c) и f orget__original_parent () (kernel/exit.c), так как они используют child_reaper для сброса вызова родителя потока.

Строка 629

Функция do_basic_setup() инициализирует модель драйвера, интерфейс sy-sctl, интерфейс сетевых сокетов и поддержку рабочей очереди.

```
init/main.c
5 51  static void  _ init do_basic_setup(void)
```

```
552  {
553   driver_init () ;
554
555  #ifdef CONFIG SYSCTL
556    sysctl_init();
557  #endif

560   sock_init () ;
561
562   init_workqueues();
563   do_initcalls () ;
564  }
```

Строка 553

Функция driver_init () (drivers/base/init .c) инициализирует все подсистемы, связанные с поддержкой драйверов. Это первая часть инициализации драйверов. Вторая появляется в строке 563 с вызовом do__initcalls ().

Строки 555-557

Интерфейс sysctl предоставляет поддержку динамического изменения параметров ядра. Это означает, что параметры ядра, поддерживающие sysctl, могут быть изменены во время работы без необходимости перекомпиляции и перезагрузки ядра; sysctl_init () (kernel/sysctl .c) инициализирует интерфейс sysctl. Более подробную информацию о sysctl читайте в соответствующих мэн-страницах (man sysctl).

Строка 560

Функция sock_init() представляет собой пустышку с простым printk, если ядро сконфигурировано без поддержки сети. В этом случае sock_init () определяется в net /nonet. c. В случае поддержки сети, когда sock_init () определена в net /socket, c, она инициализирует кеш памяти для использования поддержки сети и регистрации файловой системы, поддерживающей сеть.

Строка 562

Вызов init_workqueue настраивается для работы очереди цепи уведомлений. Гл. 10, «Добавление вашего кода в ядро», обсуждает рабочую очередь.

Строка 563

Функция do_initcalls () (init/main.c) составляет вторую часть инициализации драйвера устройства. Эта функция последовательно вызывает элементы массива указателей на функции, соответствующие функциям инициализации встроенных устройств[1].

Строки 635-638

Если раннее пользовательское пространство функции init существует, ядро не подготавливает пространство имен; оно позволяет выполнение этой функции. В противном случае выполняется вызов prepare_namespace (). Пространство имен связано с точкой монтирования иерархии файловой системы.

```
init/do_mounts.c
383  void __init prepare namespace(void)
384  {
385   int is floppy;
386
387   mount_devfs();

391   if (saved .root name[0] ) {
392   root_device_name = saved_root_name;
393   ROOT DEV = name to dev t(root device name);
394   if (strncmp (root .device .name, " /dev/", 5) ==0)
395     root device name += 5;
396   }
397
398   is floppy = MAJOR(ROOT DEV) == FLOPPY MAJOR;
399
400    if (initrd load())
401     goto out;
402
403    if (is floppy && rd doload && rd_load_disk(0))
404     ROOT_DEV = Root_RAM0 ;
405
406    mount root();
407   out:
408    umount devfs("/dev");
409    sys mount(".", "/", NULL, MS MOVE, NULL);
410    sys_chroot(".");
411    security sb post mountroot();
412    mount devfs fs ();
413   }
```

Строка 387

Функция mount_devf s () создает структуры, связанные с монтированием /dev. Нам нужно смонтировать /dev, так как мы используем ее для ссылок на имя корневого устройства.

'Обратитесь по адресу http://geek.vtnet.ca/doc/initcall/ за подробным объяснение работы initcall от Тревора Военера (Trevor Woerner).

Строки 391-396

Этот блок кода устанавливает глобальную переменную ROOT_DEV для инициализации корневого устройства, передаваемого через параметры загрузки ядра.

Строка 398

Простое сравнение старшего числа, означающего, что корневым устройством является флоппи-диск.

Строки 400-401

Вызов initrd_load() монтирует диск в памяти, если диск в памяти нужно обозначить как корень файловой системы ядра. В этом случае она возвращает 1 и выполняет переход к нашей метке, которая отменяет все выполненные приготовления для устройства корневой файловой системы.

Строка 406

Вызов mount_root выполняет основную часть монтирования корневой файловой системы. Давайте рассмотрим эту функцию поближе.

```
init/do mounts.c
3 53  void __init mount_root(void)
354   {
355   #ifdef CONFIG ROOT NFS
3 56  if (MAJOR (ROOT DEV) == UNNAMED„MAJOR) {
3 57            if (mount nfs root())
3 58                  return;
359
360printk(KERN_ERR "VFS: Unable to mount root fs via NFS,
               trying floppy.\n");
3 61           ROOT DEV = Root FD0;
362                     }
3 63                  #endif
3 64          #ifdef CONFIG BLK DEV FD
365    if (MAJOR(ROOT DEV) == FLOPPY MAJOR) {

3 67            if (rd_doload==2) {
368           if (rd load disk(l)) {
3 69           ROOT_DEV = Root_RAM1;
370       root device name = NULL;
371       }
372     } else
373      change_floppy("root floppy");
374    }
375   #endif
376    create_dev( "/dev/root",  ROOT_J3EV,  root_device_name) ;
```

```
377    mount_block_root("/dev/root", root_mountflags);
378   }
```

Строки 355-358

Если ядро сконфигурировано для монтирования файловой системы NFS, мы выполняем mount_nf s_root (). Если монтирование NFS не удалось, ядро выводит соответствующее сообщение и далее пытается смонтировать в качестве корневой файловой системы флоппи-диск.

Строки 364-375

В этом блоке кода ядро пытается смонтировать корневой флоппи-диск[1].

Строка 377

Эта функция выполняет основную часть монтирования корневого устройства. Теперь мы возвращаемся в init ().

Строка 645

Вызов f ree_initmem() освобождает все сегменты памяти, используемые функциями с приставкой___ init. Она отмечает наш выход из чистого пространства ядра и начало настройки пользовательских данных.

Строки 649-650

Открытие консоли инициализации.

Строки 662-668

Переменная execute_command, настраиваемая в init_setup (), хранит значение параметров загрузки, содержащих имя программы init, вызываемой, если мы не хотим вызывать /sbin/init по умолчанию. Если имя программы init принято, она получает приоритет обычной /sbin/init. Обратите внимание, что вызов run_init_process () (init/main. c) не возвращается, так как он заканчивается вызовом execve (). Поэтому первый вызов функции init используется только для первого удачного запуска. В случае, если программа init не найдена, мы можем использовать оболочку bash для ее запуска.

Строка 670

Это выражение паники достигается, только если все наши попытки выполнения различных программ init провалились.

Она завершает инициализацию ядра. Отсюда процесс init связывается с инициализацией системы и запускает все необходимые для регистрации и поддержки работы с пользователями процессы и демоны.

• Глобальная переменная rd_doload хранит значение 0, если нет загруженного RAM-диска, значение 1, если этот диск загружен, и значение 2 для «настройки удвоенного initrd/ramload».

Резюме

Эта глава описывает, что происходит между включением и загрузкой ядра. Мы обсудили, что такое BIOS и Open Firmware и как они взаимодействуют с загрузчиком ядра. Мы обсудили LILO, GRUB и Yaboot как наиболее распространенные загрузчики. Мы рассмотрели, как они работают и как вызывают начальные функции инициализации ядра.

Также мы отследили функции, выполняющие инициализацию ядра. Мы проследили код на протяжении всего процесса инициализации, касаясь представленных в этой главе концепций. Точнее говоря, мы отследили инициализацию ядра Linux на следующих этапах:

- запуск и блокировка ядра;
- инициализация кеша страниц и адресов страниц для управления памятью в Linux;
- подготовка множества процессоров;
- демонстрация приветствия Linux;
- инициализация планировщика Linux;
- парсинг аргументов, переданных ядру;
- инициализация прерываний, таймеров и обработчиков сигналов;
- монтирование начальных файловых систем;
- завершение инициализации системы и передача управления из init обратно в систему.

Как только мы покинули инициализацию ядра, мы должны понять, что в этой точке ядро подготовлено и могут быть запущены приложения более высокого уровня Linux, такие, как XII, sendmail и др. Все эти программы опираются на базовые настройки, в том числе и на только что описанные.

Упражнения

1. В чем разница между большой блокировкой ядра (BLK) и обычной циклической блокировкой?
2. Какой сценарий init позволяет добавлять дополнительные функции безопасности в ядро Linux?
3. Что инициализирует структуры данных для управления страницами ядра?
4. Какой процент страниц может быть грязным для активизации фоновой записи грязных страниц на диск? Какой процент страниц запускает специальную запись?
5. Почему функция rest__init () не имеет приставки _____ init?

Глава 9

Сборка ядра Linux

В этой главе:

До сих пор мы видели только подсистемы ядра и рассматривали функции инициализации этих подсистем. Важно понимать, как собирается образ ядра. Эта глава описывает процесс компиляции и компоновки образа ядра. Также мы рассмотрим процесс построения ядра изнутри.

9.1 Цепочка инструментов

Цепочка инструментов (toolchain) - это набор программ, необходимых для создания образа ядра Linux. Концепция цепочки заключается в том, что выход одного инструмента становится входом другого. Наша цепочка инструментов включает в себя компилятор, ассемблер и компоновщик. Технически в нее также следует включить текстовый редактор, но данный раздел касается только трех первых инструментов. Цепочка инструментов - это то, что необходимо нам для разработки программного обеспечения. Необходимые инструменты также зачастую называют Software Development Kit (SDK)[1].

Компилятор (compiler) - это преобразующая программа, которая берет исходники на языке высокого уровня и переводит их в низкоуровневый **объектный язык (object language).** Объектный код представляет собой последовательность машинно-зависимых команд, выполняемых на целевой системе. **Ассемблер (assembler)** - это программа трансляции, берущая программу на языке ассемблера и преобразующая ее в тот же самый объектный код, который имеет и компилятор. Различие здесь в том, что каждая строка языка ассемблера соответствует машинной инструкции, тогда как код высокого уровня может быть преобразован во множество машинных инструкций. Как вы уже видели, некоторые файлы в архитектурно-зависимой части исходных кодов Linux написаны на ассемблере. Они компилируются (в объектный код) с помощью вызова ассемблера.

Редактор связей (link editor), или **компоновщик (linker),** группирует исполняемые модули в исполняемую единицу.

Рис. 9.1 демонстрирует «цепочку в действии». Компоновщик связывает объектный код нашей программы со всеми используемыми библиотеками. У компиляторов есть флаги, позволяющие задавать глубину компиляции. Например, на рис. 9.1 мы видим, что компилятор может создавать машинный код напрямую, а может также компилировать сначала ассемблерный исходный код, который в дальнейшем превращается в машинный, который, в свою очередь, может быть исполнен на компьютере.

9.1.1 Компиляторы

Обычно компиляторы имеют «механизм цепочки» внутри, т. е. они выполняются в виде серии фаз или шагов, выход каждой из которых является входом следующей. На рис. 9.2 приведена диаграмма этих фаз. Первым шагом компиляции является фаза сканирования,

[1] Набор для разработки программного обеспечения. *Примеч. пер.*

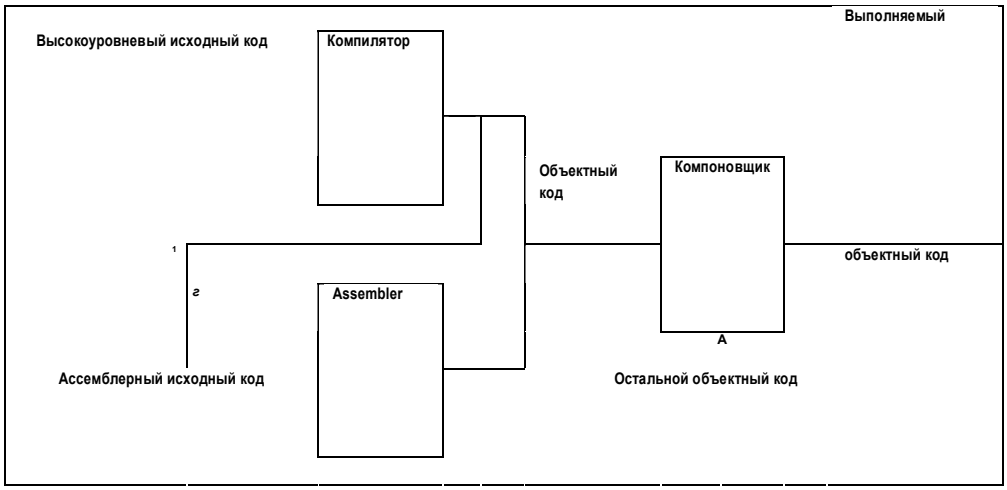


Рис. 9.1. Цепочка инструментов

группирующая токены в зависимости с правилами *синтаксиса* и фазой контекстного анализа, группирующей их по атрибутам семантики. Далее оптимизатор пытается увеличить эффективность отпарсеных токенов, а фаза генерации кода создает объектный код. Вывод компилятора - это таблица символов и настраиваемый объектный код. Поэтому начальный адрес каждого компилируемого модуля равен 0 и должен быть перемещен в нужное место при компоновке.

Рис. 9.2. Операции компилятора

9.1.2 Кросскомпиляторы

Наборы инструментов обычно запускают на месте, что означает, что генерируемый ими объектный код обычно выполняется на той же системе, на которой и компилируется. Если вы разрабатываете ядро на системе x86 и загружаете его на другой (или такой же)

системе x86, вы можете использовать компилятор, идущий в комплекте с системой. Power Macs и мириад x86 ящиков компилируются в код, запускаемый на соответствующей системе. Но что если мы хотим написать код на одной системе и запустить на другой?

Это совсем не так странно, как может показаться. Представьте себе работу со встроенными устройствами. Встроенные системы обычно разрабатываются таким образом, чтобы иметь минимально необходимое количество памяти и способности к вводу-выводу для выполнения своей работы. Когда они контролируют автомобиль, роутер или мобильный телефон, у них уже не остается места для окружения разработки на самой встроенной системе (с помощью отдельного монитора и клавиатуры). Решением этой проблемы является использование для разработки относительно недорогой рабочей станции в качестве хост системы (host system) для разработки кода, которую можно загрузить и протестировать на целевой системе (target system). Отсюда и появился термин кросскомпилятор!

Например, мы можете разрабатывать программу для встроенной PowerPC-системы с процессором 405. Большинство настольных систем разработки работают на x86. С помощью gcc, например, вы можете разрабатывать программу на настольной системе и компилировать с параметром -mcpu=405[k]. Таким образом, создается объектный код, использующий специальные 405-инструкции и адресацию. Затем вы можете загрузить код на встроенную систему, запустить его и отлаживать. Наверное, это будет звучать довольно нудно, но на ограниченных в ресурсах встроенных системах это сохраняет много памяти.

Для конкретного окружения многие инструменты на рынке помогают разрабатывать и отлаживать кросскомпилированный встроенный код.

9.1.3 Компоновщик

При компиляции С-программ (например, «hello world!») в . с-файле обычно не три или четыре строки, а намного больше. Работа компоновщика заключается в том, чтобы найти все ссылки на внешние модули и связать их. Внешние модули или библиотеки предоставляются разработчиком, операционной системой или С-библиотекой времени исполнения [где располагается printf ()]. Компоновщик извлекает эти библиотеки, исправляет указатели [перераспределение (reloaction)] и ссылки **[разрешение символов (symbol resolution)]** по модулям для создания выполнимого модуля. Символы могут быть глобальные и локальные. Глобальные символы могут быть определены в модуле или ссылаться на модуль извне. Работа компоновщика заключается в нахождении определения для каждого символа, связанного с модулем. (Обратите внимание, что библиотеки пользовательского пространства недоступны программисту ядра.) Для обычных функций у ядра есть специальные версии. Статические библиотеки **(static libraries)** находятся и копируются во время компоновки, а **динамические библиотеки (dynamic libraries)**

[k] Другие параметры gcc, связанные с IBM RS/6000 (POWER) и PowerPC, см. на http://gcc.gnu.org/onlinedocs/gcc/RS_002f6000-and-PowerPC-Options.html#RS_002f6000-and-PowerPC-Options.

или разделяемые библиотеки (shared libraries) могут загружаться во время выполнения и разделяться между процессами. Microsoft и OS/2 вызывают разделяемые библиотеки библиотеками динамической компоновки (dll). Linux предоставляет системный вызов dlopen (), dlsym () и die lose () для загрузки-открытия разделяемых библиотек, поиска символов в библиотеке и затем закрытия разделяемых библиотек.

9.1.4 Объектные ELF-файлы

Формат объектных файлов варьируется от производителя к производителю. На сегодня большинство систем UNIX использует формат выполнения и компоновки [Executable and Linking Format (ELF)]. Существуют многие типы ELF-файлов, каждый из которых выполняет различные функции. Основными типами файлов ELF являются выполнимые файлы, файлы перераспределяемых объектов и файлы ядра или разделяемые библиотеки. Формат ELF позволяет компилировать объектные файлы для других платформ и архитектур. Рис. 9.3 иллюстрирует выполнимые и невыполнимые объектные ELF-файлы.

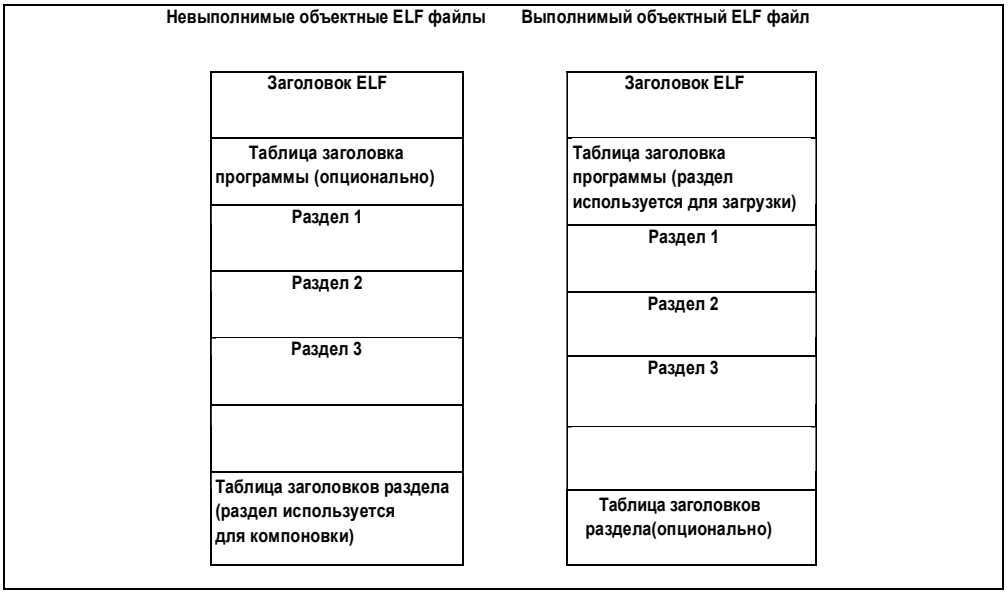


Рис. 9.3. Выполнимые и невыполнимые объектные ELF-файлы

Заголовок ELF всегда находится по нулевому отступу в ELF-файле. Все в файле можно найти через заголовок ELF. Так как заголовок ELF представляет собой фиксированную структуру в объектном файле, он должен указывать на подструктуры внутри этого файла и определять их размер. Невыполнимый объектный файл содержит секции и таблицу за-

головков секций, тогда как выполнимые объектные файлы должны содержать сегменты и таблицу заголовков программ.

9.1.4.1 Заголовок ELF

Заголовок ELF следит за структурой elf 32_hdr (для 32-битовых систем; и структура elf 64_hdr для 64-битовых). Давайте рассмотрим эту структуру.

```
include/linux/elf.h
234  #define EI_NIDENT   16
235
236 typedef struct elf32_hdr{
237  unsigned char  e_ident[EI_NIDENT];
238  Elf32_Half   e_type;
239  Elf32_Half   e_machine;
240  Elf32_Word    e_version;
241  Elf32_Addr   e_entry;  /* Entry point */
242  Elf32_Off    e_phoff;
243  Elf32 Off    e shoff;
244  Elf32_Word    e_flags;
245  Elf32_Half    e_ehsize;
246  Elf32_Half    e phentsize;
247  Elf32_Half    e_phnum;
248  Elf32_Half    e_shentsize;
249  Elf32_Half    e_shnum;
250  Elf32_Half    e_shstrndx;
251  } Elf32 Ehdr,
```

Строка 237

Поле e_ident хранит 16-битовое магическое число, идентифицирующее файл как ELF-файл.

Строка 238

Поле e_type определяет тип объектного файла, т. е. выполнимый, перераспределяемый или разделяемый объект.

Строка 239

Поле e_machine идентифицирует архитектуру системы, для которой компилируется файл.

Строка 240

Поле e_version определяет версию объектного файла.

Строка 241
Поле e_entry хранит начальный адрес программы.

Строка 242
Поле e_phof f хранит отступ в байтах таблицы заголовка программы.

Строка 243
Поле e_shof f хранит отступ для отступа таблицы заголовков модуля в байтах.

Строка 244
Поле e_f lags хранит процессорно-специфические флаги.

Строка 245
Поле e_ehsize хранит размер заголовка ELF.

Строка 246
Поле e_phentsize хранит размер каждого элемента в таблице заголовка программы.

Строка 247
Поле e_phnum содержит количество элементов в заголовке программы.

Строка 248
Поле e_shentsize хранит размер каждого из элементов таблицы заголовков разделов.

Строка 249
Поле e_shnum хранит количество элементов в заголовке раздела, обозначающее количество разделов в файле.

Строка 250
Поле e_shstrndx хранит индекс строки раздела в заголовке раздела.

9.1.4.2 Таблица заголовков разделов

Таблица заголовков разделов - это массив типа Elf 32_Shdr. Его отступ в ELF-файле задается полем e_shof f в ELF-заголовке. Для каждого раздела в файле существует своя таблица заголовка раздела.

```
include/linux/elf.h
332   typedef struct {
333    Elf32 Word   sh name;
334    Elf32 Word   sh type;
335   Elf32_Word    sh_flags;
33 6   Elf32 Addr   sh addr;
337   Elf32 Off     sh offset;
33 8   Elf32_Word   sh_size;
```

```
339    Elf32_Word   sh_link;
340    Elf32_Word   sh_info;
341    Elf32_Word   sh_addralign;
342    Elf32_Word   sh_entsize;
343   } Elf32_Shdr;
```

Строка 333
Поле sh__name содержит имя раздела.

Строка 334
Поле sh_type хранит содержимое раздела.

Строка 335
Поле sh_f lags хранит информацию, связанную с различными атрибутами.

Строка 336
Поле sh_addr хранит адрес раздела в образе в памяти.

Строка 337
Поле sh_of f set хранит отступ к первому байту раздела в ELF-файле.

Строка 338
Поле sh_size содержит размер раздела.

Строка 339
Поле shJLink содержит индекс в таблице ссылок, зависящий от sh_type.

Строка 340
Поле sh_inf o содержит дополнительную информацию, зависящую от значения sh_type.

Строка 341
Поле sh_addralingh содержит константы выравнивания адреса.

Строка 342
Поле sh_entsize содержит размер элемента раздела, когда он хранится в таблице фиксированного раздела.

9.1.4.3 Разделы невыполнимого ELF-файла

ELF-файл делится на несколько разделов, каждый из которых содержит информацию специального типа. Табл. 9.1 описывает типы разделов. Некоторые из этих разделов присутствуют, только если установлены специальные флаги компилятора при компиляции. Вспомните, что ELF32_Ehdr->e_shnum хранит количество разделов в ELF-файле.

Таблица 9.1. Разделы ELF-файла

Раздел	Описание
.bss	Неинициализированные данные
.comment	Использует это поле для отображения версии компилятора
.data	Инициализированные данные
.debug	Символическая отладочная информация в форме таблицы символов
.dynamic	Информация о динамическом связывании
.dynstr	Строки динамического связывания
. f ini	Код завершения процесса, код выхода GCC
.got	Таблица глобального отступа
.hash	Таблица хеширования символов
. init	Код инициализации
.interp	Имя, где находится интерпретатор программы
.line	Количество строк для отладки
.note	Используется компилятором для обозначения версий.
.pit	Таблица связи процедур
.relname	Информация о перераспределении
.rodata	Данные только для чтения
.shstrtab	Имена разделов
. symtab	Таблица символов
.text	Выполняемые инструкции

9.1.4.4 Таблица заголовка программы

Таблица заголовка для выполняемого или разделяемого объектного файла - это массив структур, каждая из которых описывает сегменты или другую информацию о выполнении.

```
include/linux/elf.h
276  typedef struct elf32 phdr{
277   Elf32  Word   p type;
278   Elf32 Off   p offset;
279   Elf32 Addr  p vaddr;
280   Elf32_Addr   p_paddr;
```

```
281       Elf32_Word p_filesz;
2 82      Elf32_Word p_memsz;
2 83      Elf32_Word p_flags;
2 84      Elf32_Word p_align;
2 85    } Elf32_Phdr;
```

Строка 277
Поле p_type описывает тип данного сегмента.

Строка 278
Поле p_of f set хранит отступ от начала файла до начала сегмента.

Строка 279
Поле p_vaddr хранит виртуальный адрес сегмента, если он используется.

Строка 280
Поле p_paddr хранит физический адрес сегмента, если он используется.

Строка 281
Поле p_f ilesz хранит количество байтов в образе файла сегмента.

Строка 282
Поле p_memsz хранит количество байтов в образа сегмента в памяти.

Строка 283
Поле p_f lags хранит флаги, зависящие от p_type.

Строка 284
Поле p_align описывает размещение сегментов в памяти. Значение представляет собой степень двух.

С помощью этой информации функция exec () вместе с компоновщиком создает образ процесса исполняемой программы в памяти. Сюда входит:

- перемещение сегментов в память;
- загрузка разделяемых библиотек, которые нужно загрузить;
- выполнение перераспределения по необходимости;
- передача управления программы.

Получив понимание формата объектного файла и доступных инструментов, вы сможете успешнее решать проблемы компиляции (такие, как неразрешенные связи) и проблемы, возникающие в момент работы программ, так как вы будете знать, *куда* загружается и *где* переразмещается код.

9.2 Сборка исходников ядра

Теперь мы рассмотрим, как ядро компилируется в бинарный образ и загружается в память перед выполнением. Как разработчик ядра, вы будете тесно связаны с его исходным кодом. Необходимо понимать, как ориентироваться в этом коде и как его редактировать для добавления своих изменений.

Этот раздел проведет вас по всему пути от загрузки исходного кода до компиляции и загрузки образа ядра. Мы рассмотрим, как создается образ ядра. Это не подробное пошаговое руководство. Более подробную онлайновую документацию для текущей версии можно найти в HOWTO (www. tldp. org/HOWTO/Kernel-HOWTO/). Здесь же мы приведем только информацию, необходимую для внесения изменений в систему сборки.

Системы сборки, так называемые make-файлы (Makefile), не пользуются большим интересом среди разработчиков, однако нам нужно понимать, как работает система сборки ядра и как оно обновляется при изменении исходного кода. В ядре версии 2.6 у вас есть больше инструментов, которые помогут вам понять все особенности системы сборки ядра. Кроме этого, система сборки была значительно подчищена и перепроектирована, а также еще лучше задокументирована.

Этот раздел описывает, где располагается код, как собирается ядро и как работает Makefile.

9.2.1 Разъяснение исходников

Сайт, на котором размещается официальный код Linux, находится по адресу www. kernel . org. Источник доступен для скачивания в виде . tar. gz-файла с компрессией gzip или . tar .Ьг2-файла с компрессией bzip2. Эти пакеты содержат исходный код для всех доступных архитектур.

Когда разработчик ядра вносит изменения в исходный код ядра, он отправляет его хранителю ядра. Хранитель решает, какие из изменений следует внести в следующую стабильную ветвь. Урезанная PPC-разработка находится в отдельном дереве по адресу www.penguinppc.org. Изменения, вносимые в дерево PPC, передаются далее в главное дерево, где они компонуются. На данный момент сообщество PPC Linux перемещается в главное дерево разработки.

Расположение исходного кода обычно зависит от вашего дистрибутива. Например, в системе Rad Hat исходный код располагается (по умолчанию он инсталлируется только через RMP) в /usr/src/linux-<version>/. Если вы занимаетесь кросскомпиляци-ей, исходный код может находиться в другом месте в /opt/distribution name> или альтернативно в образе корневой системы, задаваемой пользователем с помощью enroot. Например, Montavista - распространенный на рынке встроенных систем дистрибутив - хранит по умолчанию исходный код (и кросскомпилятор) в /opt/mvista.

В этом разделе корень файловой системы исходного кода связан для простоты с root. В дистрибутиве Rad Hat корень исходного кода находится в /usr/src/linux-<version>. Рис. 9.4 показывает иерархическую структуру исходного кода.

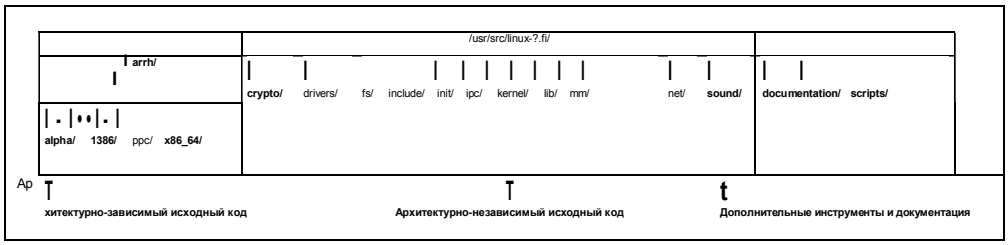


Рис. 9.4. Размещение исходного кода

Исходный код делится на *архитектурно-зависимую* и *архитектурно-независимую* части. Директория /arch в корне хранит весь архитектурно-зависимый код. В этой директории исходный код, скачанный с kernel. org, содержит список всех поддерживаемых архитектур. Для каждой поддерживаемой архитектуры в директории /arch имеется поддиректория, содержащая дальнейшую иерархию архитектурно-зависимого кода для данной архитектуры. Рис. 9.5 демонстрирует поддерживаемые архитектуры, перечисленные в списке из директории /arch.

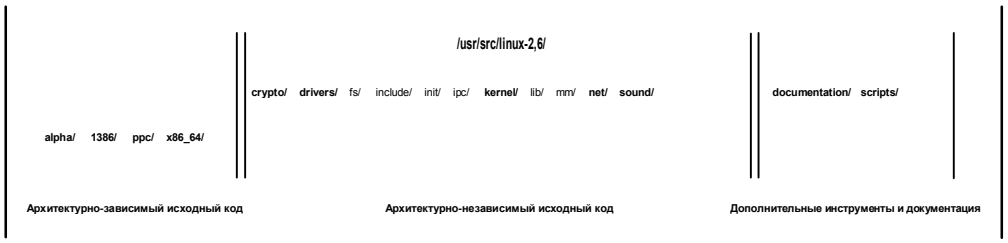


Рис. 9.5. Is /usr/src/linux/arch

Мы начнем с рассмотрения структуры архитектурно-независимой части исходного кода для понимания его разделения. Далее мы представим вашему вниманию обзор архитектурно-независимой части исходного кода, за которым последует краткий обзор дополнительных файлов, не попадающих ни в одну из категорий.

9.2.1.1 Архитектурно-независимый код

Архитектурно-независимая часть исходного кода делится на 11 поддиректорий, категоризирующихся по функциональности. В табл. 9.2 приведены эти поддиректории.

Таблица 9.2. Архитектурно-независимые поддиректории

Поддиректория	Описание
crypto	Хранит код для криптографического API и различных алгоритмов шифрования-дешифрования
drivers	Код для драйверов устройств
fs	Код для VFS и всех поддерживаемых в Linux подсистем
include	Заголовочные файлы. Эта директория имеет несколько поддиректорий с префиксом asm. Эти директории хранят архитектурно-зависимые заголовочные файлы. Оставшиеся директории хранят архитектурно-независимые заголовочные файлы
init	Архитектурно-независимая часть кода загрузки и инициализации
ipc	Код для поддержки межпроцессорной коммуникации (IPC)
kernel	Код для специфического кода ядра
lib	Код для вспомогательных функций
mm	Код для менеджера памяти
net	Код для поддержки различных сетевых протоколов
sound	Код для поддержки звуковой системы

На протяжении предыдущих глав мы рассматривали исходный код, находящийся в одной из этих поддиректорий. Для правильного контекста обсуждения следующий подраздел представляет собой обобщенный обзор этих поддиректорий. Мы оставим за пределами обсуждения только те из них, которые не будем рассматривать.

fs/

Директория f s/ делится далее на исходные файлы С-поддержки VFS и поддиректории для каждой из поддерживаемых файловых систем. Как подробно описано в гл. 7 «Планировщик и синхронизация ядра»[1], VFS - это слой абстрагирования для различных типов файловых систем. Код, находящийся в каждой из поддиректорий, состоит из кода, связывающего устройство хранения и слой абстрагирования VFS.

[к] Очевидно, имеется в виду гл. 6, «Файловые системы». *Примеч. науч. ред.*

in it/

Директория init / содержит весь необходимый код для инициализации системы. Во время выполнения этого кода инициализируются все подсистемы ядра и создаются процессы инициализации.

kernel/

Основная часть кода ядра находится в директории kernel/. Большинство подсистем ядра расположены здесь, но некоторые, такие, как файловая система и память, имеют собственные директории того же уровня, что и kernel/. Имена файлов обычно соответствуют содержащемуся в них коду.

mm/

Директория mm/ содержит код управления памятью. Мы рассматривали примеры этого кода в гл. 4, «Управление памятью».

9.2.1.2 Архитектурно-зависимый код

Архитектурно-зависимый код - это часть кода ядра, связанная с настоящим аппаратным обеспечением. При рассмотрении этой части кода необходимо помнить, что изначально Linux разрабатывался для x86. Для минимизации сложности портирования некоторые x86-специфиические термины отражаются в глобальных структурах ядра, а также в именах переменных. Если вы рассмотрите код PPC и увидите имена, связанные с режимом преобразования адресов, которые не существуют на PPC, не паникуйте.

Сравнив код arch/i3 8 б и arch/ppc, вы увидите три очень похожих файла- def-conf ig, Kconf ig и Makefile. Эти файлы связаны с инфраструктурой системы сборки ядра. Назначение этих трех файлов объясняется в подразд. 9.2.2, «Сборка образа ядра».

В таблице 9.3 приведен обзор файлов и директорий, показанных в списке arch/ppc. Как только вы доберетесь до структур Makefiles и Kconf ig, стоит рассмотреть каждый из файлов в соответствующих директориях, где находится их код.

Таблица 9.3. Список исходных кодов arch/ppc

Поддиректория	Описание
4xx__io	Исходный код для MPC4xx-специфической части ввода-вывода, т. е. для последовательного порта IBM STB3xxx SICC
82 60_io	Исходный код для настроек связи с MPC8260
8xx_io	Исходный код для настроек связи с MPC8xx
amiga	Исходный код для компьютеров Amiga на основе PowerPC
boot	Исходный код связан с загрузкой PPC. Эта поддиректория также содержит поддиректорию с именем images, в которую сохраняются откомпилированные загрузочные образы

Таблица 9.3. Список исходных кодов arch/ppc (Окончание)

conf ig	Файлы настройки для сборки специфических PPC-платформ и архитектур
kernel	Исходный код для аппаратных зависимостей подсистем ядра
lib	Исходный код для специфических файлов PPC
math-emu	Исходный код для эмуляции математики PPC
mm	Исходный код для специфической PPC-части менеджера памяти. (Гл. 6[a] описывает эту тему подробнее)
platforms	Специфический код для платформы, на которой смонтированы чипы PPC
syslib	Часть ядра исходного кода для обобщенных аппаратно-специфических подсистем
хлюп	Исходный код для PPC-специфического отладчика

[a] Очевидно, имеется в виду гл. 4, «Управление памятью». *Примеч. науч. ред.*

Директории в arch/x86 хранят структуру, подобную той, которую можно увидеть в архитектурно-зависимой директории PPC. В табл. 9.4 перечислены все возможные поддиректории.

Таблица 9.4. Список исходных кодов в arch/x86

Поддиректория	**Описание**
boot	Исходный код, связанный с загрузкой x86 и процессом инсталляции
kernel	Исходный код для аппаратных зависимостей подсистем ядра
lib	Исходный код для x86-специфических файлов библиотек
mach-x	Исходный код для разновидностей архитектуры x86
math-emu	Исходный код для функций математической эмуляции x86
mm	Исходный код для x86-специфической части менеджера памяти. (В гл. 6 он описывается подробно)
oprof ile	Исходный код для инструмента профилирования ядра oprof ile
pci	Драйверы PCI x86
power	Исходный код для управления питанием x86

Вы можете удивиться, почему два архитектурно-специфических списка не слишком похожи. Причина в том, что разбиение этой функциональности для данной архитектуры

может не подходить для других архитектур. Например, на PPC драйверы PCI варьируются на различных платформах и подплатформах, усложняя структуру поддиректории PCI по сравнению с ее аналогом из x86.

9.2.1.3 Дополнительные файлы и директории

В корне исходных файлов присутствует несколько файлов, не относящихся к архитектурно-зависимому и архитектурно-независимому коду. Они перечислены в табл. 9.5.

Таблица 9.5. Дополнительные файлы

Файл/директория	Описание
COPYING	Лицензия GPL, под которой распространяется Linux
CREDITS	Список разработчиков проекта Linux
MAINTAINERS	Список поддержки и инструкций по отправке изменений в ядро
README	Сведения о версии
REPORTING-BUGS	Описание процедуры сообщения об ошибках
documentation/	Директория с документацией о различных аспектах ядра Linux и исходного кода. Полезный, хотя и быстро стареющий источник информации
scripts/	Хранилище инструментов и сценариев, использующихся во время процесса сборки ядра

9.2.2 Сборка образа ядра

Система сборки ядра, или kbuid, - это механизм, с помощью которого можно выбирать варианты настройки сборки ядра. Для ядра из ветки 2.6 они обновились. Новая версия, kbuid, быстрее предшественника и соответственно лучше документирована. Система kbuid жестко зависит от иерархической структуры исходного кода.

9.2.2.1 Инструмент конфигурирования ядра

Инструмент конфигурирования ядра автоматически генерирует файлы настройки ядра с именем . c on fig. Это первый шаг сборки ядра. Файл . conf ig находится в корне исходного кода; он содержит описание всех настроек ядра, которые можно задать с помощью инструмента настройки. Каждая настройка сборки ядра имеет имя и связанное с ним значение. Имя имеет форму GONFIG__<NAME>, где <NAME> - метка, связанная с настройкой. Это значение может хранить одно из трех значений: y, m или n; y означает «yes» и то, что настройка должна быть включена в ядро или при компиляции; m означает «module» и, что настройка должна компилироваться отдельно от исходных кодов ядра. Если

настройка не выбрана (или ее значение установлено в п для «по»), то в файле . с on fig соответствующая GONFIG_<NAME> закомментирована, т. е. не установлена. Настроечный файл . conf ig организован в порядке использования настроек инструментом настройки ядра и прокомментирован для обозначения каждой из настроек. Давайте рассмотрим пример файла . conf ig.

```
.config
1    #
2    # Automatically generated make config: don't edit
3    #
4    CONFIG X86=y
5    CONFIG_MMU=y
6    CONFIG UID16=y
7    CONFIG_GENERIC_ISA_DMA=y 8
9    #
10    # Code maturity level options
11    #
12    CONFIG_EXPERIMENTAL=y
13    CONFIG CLEAN COMPILE=
14    CONFIG STANDALONE=y
15    CONFIG_BROKEN_ON_SMP=y 16
17    #
18    # General setup
19    #
20    CONFIG SWAP=y
21    CONFIG SYSVIPC=y
22    #CONFIG POSIX MQUEUE is not set
23    CONFIG_BSD_PROCESS_ACCT=y
```

Этот . conf ig-файл означает, что опции в строках с 4 до 7 находятся на верхнем уровне, настройки из строк с 12 до 15 находятся в меню завершающих настроек кода, а настройки в строках с 20 по 23 находятся в меню общих настроек.

Рассмотрев меню с помощью инструментов конфигурирования, вы увидите, что первые несколько настроек относятся к корню вместе с настройками кода общего уровня и общими настройками. Два последних пункта подразделяются на другие подменю. Это можно увидеть в qconf, вызываемой на выполнение при вызове xconf ig. Меню инструментов настройки по умолчанию показывает настройки для x86. Для просмотра связанных с PPC настроек, показанных на рис. 9.6, вызов xconf ig нужно дополнить параметром ARCH=ppc.

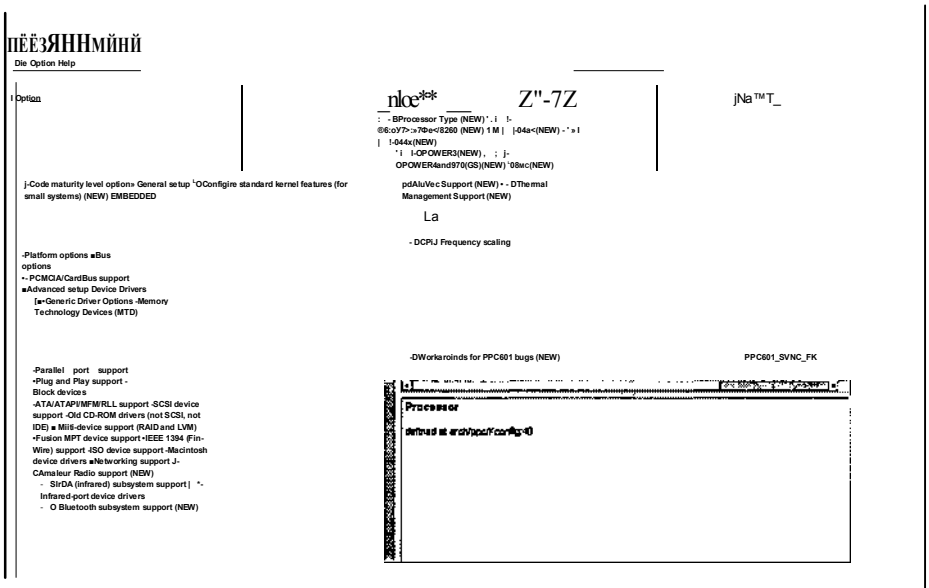


Рис. 9.6. Снимок qconf

Файл . config, генерируемый инструментами настройки, читается Makefile при сборке ядра с помощью вызова make bzlmage. Корневой Makefile тоже получает информацию, собираемую аппаратно-специфическим Makefile, который находится в arch/<arch>. Это делается с помощью директивы include:

```
Makefile
434   include .config
. ./
450   include $(srctree)/arch/$(ARCH)/Makefile
```

В этой точке Makefile всегда определяет, для какой архитектуры выполняется компиляция. Корневой Makefile определяет архитектуру компиляции одним из трех способов:

1. С помощью параметра командной строки ARCH.
2. С помощью переменной окружения ARCH.
3. Автоматически, получая информацию от вызова uname на хосте, на котором он выполняется.

Если ядро компилируется не для той архитектуры, на которой производится сборка, передается параметр CROSS_COMPILE, означающий префикс используемого кросскомпилятора. Альтернативно может быть изменен сам Makefile, и переменная получит свое значение. Например, если выполняется компиляция для процессора PPC на платформе x86, будут выполняться следующие команды:

```
lkp:-#make xconfig ARCH=ppc
lkp:-#make ARCH=ppc  CROSS_COMPILE=ppc-linux-
```

Файл .config также генерирует include/linux/autoconf .h, в котором #de-fine определяет выбираемое значение CONFIG_<NAME> и #undef сигнализирует о том, что это значение сбрасывается.

9.2.2.2 Sub-Makefiles

Система сборки опирается на sub-Makefiles, находящиеся в каждой из поддиректорий. Каждый Makefile поддиректории (называемый sub-Makefile или kbuild Makefile) определяет правила сборки объектных файлов из исходных, находящихся в поддиректориях, и применяется только для этой директории. Вызов каждого Makefile делается рекурсивно с проходом по всем поддиректориям: init/ , drivers/, sound/, net, / .libHusr/.

Перед выполнением рекурсивных вызовов kbuild нужно убедиться в присутствии нескольких вещей, включая обновленный при необходимости include/linux/version, h и настройки символических ссылок inxlude/asm, указывающих на архитектурно-специфические файлы для компилируемой архитектуры. Например, если компиляция выполняется для PPC, include /asm указывает на include/asm-ppc. Кроме этого, kbuild собирает include/linux/autoconf .h и include/linux/config. После этого kbuild начинает рекурсивный спуск вниз по дереву.

Если вы разработчик ядра, вы можете добавить отдельные подсистемы, разместив файлы в специальных поддиректориях и обновив Makefile для внесения ваших изменений. Если ваш код внедряется в уже существующий файл, вы можете завернуть в блок #ifdef (CONFIG_<NAME>). Если это значение выбрано в файле . config, оно определяется с помощью #def ine в include/linux/autoconf .h и ваши изменения будут учтены при компиляции.

Строки sub-Makefile имеют специальный формат, которого следует придерживаться при задании настроек сборки объектного файла. Эти Makefile располагаются последовательно из-за того, что информация с именами компилятора и библиотек определяется в корневом Makefile и архитектурно-специфическом корневом Makefile, а правила определяются в scripts/Makefile; *s. sub-Makefile составляет три возможных списка:

- $(obj-y). Перечисляет объектные файлы, связываемые внутри bui 11- in. o и далее внутри vmlinux.
- $(obj-m). Перечисляет объектные файлы, собираемые как модули.
- $(lib-y). Перечисляет объектные файлы, собираемые в lib. a.

Другими словами, когда мы делаем вызов make bzlmage, kbuild, мы собираем объектные файлы в ob j -y и связываем их. Базовая строка в sub-Makefile имеет вид

```
obj-$(CONFIG_FOO)   +=   foo.o
```

Если CONFIG_FOO установлен в y в файле . conf ig для чтения корня Makefile, эта строка становится эквивалентной obj-y += foo.o kbuild, собирающей этот объектный файл из соответствующих f o o. c и файлов f o o. S из директорий, определенных правилами в scripts/makefile .build. (Скоро мы рассмотрим их подробнее.) Если f oo. c и f oo. S не существуют, выводится замечание

```
make[l] :   *** No rule to make target   x<subdir>/foo.c',   needed by 4<subdir>/build-in.o'.
Stop.
```

Способ, которым kbuild углубляется в директории, определяется ob j -y или ob j -m. Вы можете добавить директорию для настройки ob j -y, означающей, что нужно углубиться в указанную директорию:

```
obj-$(CONFIG_FOO)   +=   /foo
```

Если /foo не существует, выдается следующее замечание:

```
Make[2]:   ***  No rule  to make  target   *<dir>/foo/Makefile' .   Stop.
```

CML2

Откуда программа настройки, которая помогает нам при выборе настроек ядра, получает информацию? Система kbuild зависит от CML2, являющегося предметно-ориентированным языком, разработанным для настройки ядра. CML2 задает базовые правила, в соответствии с которыми интерпретируются читаемые данные и генерируются config-файлы. Этот файл описывает синтаксис и семантику языка. Базовые правила CML2 читаются программами настройки и сохраняются в файлы defconfig и Kconfig. Файлы defconfig находятся в корне архитектурно-зависимых директорий, arch/*/. Файлы Kconfig находятся в других поддиректориях. Файлы Kconfig хранят информацию, связанную с настройками сборки, такими, как меню перечисления вариантов, сопроводительная информация, значение имени config и информация о том, встраивается файл или компилируется в отдельный модуль. Более подробную информацию о CML2 и файле Kconfig см. в Documentation/kbuild/kconfig-language.txt

Давайте еще раз рассмотрим, что мы узнали о процессе kbuild. Первым шагом является вызов инструмента настройки с помощью make xconf ig и make xconf ig ARCH=ppc в зависимости от архитектуры, для которой производится сборка. Сделанный в программе выбор сохраняется в файле .config. Самый верхний Makefile читает . config и вызывает make bz Image для сборки образа ядра. Далее верхний Makefile выполняет следующие процедуры, постепенно спускаясь вниз по поддиректориям:

1. Обновляет include/linux/version.h.
2. Устанавливает символическую связь include /asm указывать на архитектурно-специфические файлы архитектуры, для которой производится компиляция.
3. Собирает include/linux/autoconf .h.
4. Собирает include/linux/conf ig.h.

Далее kbuild спускается по директориям, вызывая sub-Makefile и создавая объектные файлы внутри каждой.

Мы рассмотрели структуру sub-Makefile. Теперь мы обратимся к Makefile высокого уровня и посмотрим, как он используется для управления системой сборки.

9.2.2.3 Makefile ядра Linux

Makefile Linux достаточно сложен. Этот подраздел описывает внутреннюю связь между всеми Makefile в дереве исходников и объясняет подробности реализации make. Тем не менее, если вы хотите расширить свои знания о make, необходимые для понимания всех особенностей kbuild Makefile, это будет хорошим началом. Более подробную информацию о make можно найти на www.gnu.org/software/make/ make.html.

В дереве исходников виртуально каждая директория имеет свой Makefile. Как было сказано в предыдущем разделе, Makefiles в поддеревьях разделяются на отдельные категории исходного кода (или подсистемы ядра) достаточно предсказуемо и соответственно определяют целевые исходные файлы, добавляемые в список при их поиске во время сборки. Расположенные рядом другие 5 Makefile определяют правила и выполняют их. Сюда входят корневой Makefile, arch/$ (ARCH) /Makefile, scripts/ Makefile, build, scripts/Makefile, clean и scripts/Makefile. Рис. 9.7 демонстрирует связи между различными Makefile. Мы определяем связи с помощью «включения» и «выполнения». Когда мы указываем связь «включения», мы имеем в виду, что Makefile вбирает в себя информацию о правилах из файла, указанного include <f ilename>. Когда мы говорим о связи «выполнения», мы имеем в виду то, что оригинальный Makefile выполняет вызов make -f для второго Makefile.

Когда выполняем вызов make для корня дерева исходников, мы вызываем корневой Makefile. Корневой Makefile определяет переменные, экспортируемые в остальные

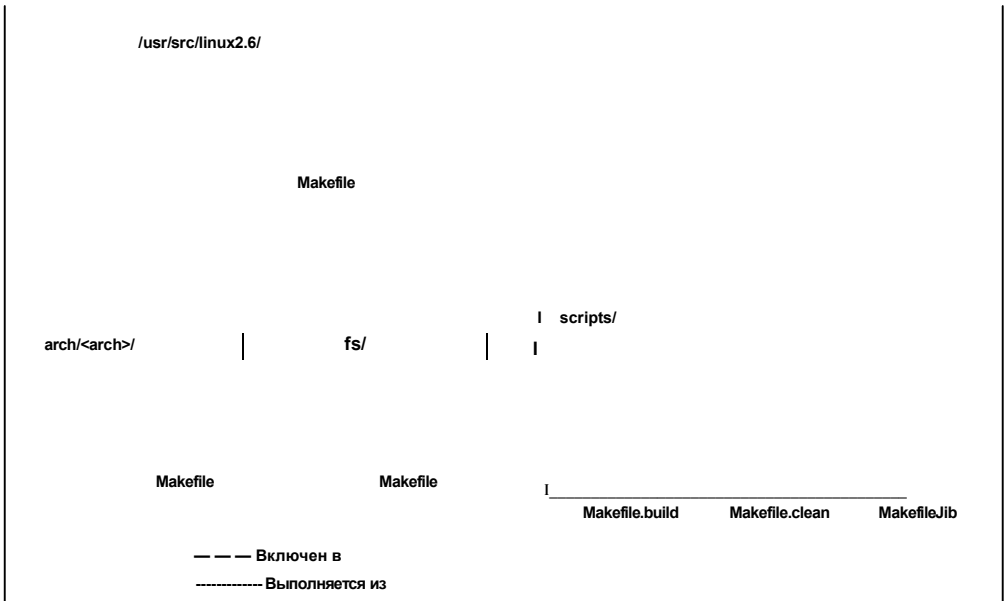


Рис. 9.7. Связи Makefile

Makefile, и выполняет дальнейшие вызовы make для каждой поддиректории корня, передавая в них управление.

Вызовы компилятора и компоновщика определены в scripts/Makefile .build. Это значит, что, когда мы попадаем в поддиректорию и собираем объект с помощью вызова make, мы тем самым выполняем правила, определенные в Make file, build. Это делается с помощью сокращенного вызова $ (Q) $ (MAKE) $ (build) =<dir>. Это правило используется для make внутри каждой поддиректории. Переменная build-это сокращение для

```
Makefile
1157      build := -f $(if $(KBUILD_SRC),$(srctree)/)scripts/
Makefile.build obj
```

Вызов $ (Q) $ (MAKE) $ (build) =<dir> превращается в

```
w@ make -f /path/to/source/scripts/Makefile.build obj=fs".
```

Далее scripts/Makefile.build читает Makefile директории, которая была передана в качестве параметра (fs в нашем примере). Эта sub-Makefile определяет

один из нескольких списков obj-y, obj-m, lib-y и др. Файл scripts/Makefile, build вместе с определениями из включаемого scripts /Makefile, lib компилирует исходные файлы в поддиректории и спускается в другие перечисленные поддиректории. Вызов происходит аналогичным образом.

Давайте посмотрим, как это работает на примере. Если в инструменте настройки мы спустимся в меню File Systems и выберем поддержку журналируемой файловой системы Ext3, в . conf ig-файле будет установлена настройка CONFIG_EXT3_FS. Отрывок из соответствующего f s Makefile приведен далее.

```
Makefile
49    obj-$(CONFIG_EXT3_FS)       +=   ext3/
```

Когда в соответствии с этим правилом запускается make, он определяет ob j -y + = ext3, делая ext3/ одним из элементов obj-y.make и распознав, что это поддиректория вызывает $ (Q) $ (MAKE) $ (build) =ext3.

$(Q)

Переменная $(Q) представляет собой префикс вызова $(MAKE). В дереве ядра 2.6 и в его улучшенной инфраструктуре вы можете подавить многословный режим вывода make, который выводит команды перед тем, как их выполнять. Когда строка начинается с @, вывод (echo) подавляется:

```
Makefile
254  ifeq($(KBUILD_VERBOSE),l)
255  quiet =
256  Q =
257  else
258  quiet=quiet_
259   Q=@
260  endif
```

Как вы можете видеть из этих строк, Q устанавливается в @, если KBUILD_VERBOSE установлена в 0, что означает, что мы не хотим компилироваться в многословном режиме.

После завершения процесса сборки мы заканчиваем работу с образом ядра. Этот загрузочный образ ядра называется zImage или vmlinuz, из-за того что ядро упаковывается с помощью алгоритма zlib. Обычные соглашения Linux также определяют местонахождение загрузочного образа в файловой системе; образ должен находиться в /boot или /. Теперь образ ядра готов для загрузки в память и загрузки системы.

Резюме

В этой главе рассмотрен процесс компиляции и компоновки структуры объектного файла, дающий возможность понять, что происходит с кодом и как он может быть исполнен. Также мы рассмотрели инфраструктуру, касающуюся системы сборки ядра, и то, как с этой системой сборки связана сама структура исходного кода. Мы бросили беглый взгляд на разделение исходного кода по функциональности в соответствии с подсистемами, которые мы рассматривали в предыдущих главах.

Упражнения

1. Опишите различные типы ELF-файлов и для чего они используются.
2. Какое место занимают разделы в объектном файле?
3. Рассмотрите arch/ppc /Kconfig и arch/i3 86 /Kconfig и определите, какие процессоры поддерживаются в каждой архитектуре.
4. Рассмотрите arch/ppc и arch/i38б и определите, какие у них общие файлы и директории. Изучите список их поддержки. Насколько они похожи?
5. Если вы кросскомпилируете ядро, какой параметр вам нужно задать в виде префикса кросскомпилятора?
6. В каком случае нужно задавать архитектуру через параметр ARCH?
7. Что такое sub-makefile? Как она работает?
8. Ознакомьтесь с scripts /Makefile, build, scripts/Makefile, clean и scripts/Makefile. lib. Перечислите, что они делают.

Глава Ю

Добавление вашего кода в ядро

В этой главе:

В этой главе можно выделить две главные части: «Обход исходников» и «Написание кода».

«Обход исходников» посвящен обзору драйвера устройства /dev/random, который является общим для всех систем Linux, и демонстрирует, как с ним связано ядро. Во время обзора мы вспомним некоторые особенности внутренней работы ядра, рассмотренные ранее, и осветим их с практической точки зрения.

«Написание кода» - это руководство по написанию драйвера и затрагивает распространенные ситуации, с которыми сталкивается разработчик драйверов.

После этих разделов мы перейдем к описанию того, как вы можете отлаживать драйвер устройства с помощью системы /proc. Может быть, это и есть третья | сторона монеты?

10.1 Обход исходников

Этот раздел включает представление концепции системных вызовов и драйверов (также известных как модули) в Linux. Системные вызовы используются пользовательскими программами для общения с операционной системой для запроса служб. Добавление системного вызова - это один из способов создания новой службы ядра. Гл. 3, «Процессы: принципиальная модель выполнения», описывает внутреннюю реализацию системного вызова. Эта глава описывает практические аспекты встраивания вашего системного вызова в ядро Linux.

Драйвер устройства представляет собой интерфейс, используемый ядром Linux для того, чтобы разрешать программисту управлять системным вводом-выводом устройств. Эта глава подробно разъясняет все тонкости. В этом разделе мы проследим работу драйвера с его представления в файловой системе и вплоть до контролирующего его кода ядра. В следующем разделе мы покажем, как использовать то, что мы изучили в первой части разработки функционального символьного устройства. Заключительная часть гл. 10 описывает, как писать системные вызовы и собирать ядро. Мы начнем с рассмотрения файловой системы и покажем, как эти файлы связаны с ядром.

10.1.1 Познакомимся с файловой системой

Устройства в Linux доступны через /dev. Например, -1 /dev/random выдает следующее:

```
crw-rw-rw- 1 root root 1, 8 Oct 2 08:08 /dev/random
```

Первая «с» означает, что устройство является символьным; «Ь» означает блочное устройство. После владельца и колонок группы идут два числа, разделенные запятыми (в данном случае 1, 8). Первое число - это старший номер драйвера, а второе число - младший номер. Когда драйвер устройства регистрируется в ядре, он регистрирует старший номер. Когда данное устройство открывается, ядро использует старший номер устройства

для нахождения драйвера, зарегистрированного для этого старшего номера[1]. Младший номер передается через ядро в сам драйвер устройства, так как один драйвер может управлять несколькими устройствами. Например, /dev/urandom имеет старший номер 1 и младший номер 9. Это значит, что драйвер устройства зарегистрирован со старшим номером 1, обрабатывающим как /dev/random, так и /dev/urandom.

Для генерации случайного числа мы просто выполняем чтение из /dev/random. Следующим способом можно считать 4 байта случайных данных[2]:

```
lkp@lkp:~$ head -c4 /dev/urandom | od -x
0000000 823a 3be5
0000004
```

Если вы повторите эту команду, вы увидите что 4 байта [823a 3be5] продолжают изменяться. Для демонстрации того, как ядро Linux использует драйверы устройств, мы проследим по шагам, что делает ядро, когда пользователь получает доступ к /dev/random.

Мы знаем, что файл устройства /dev/random имеет старший номер 1; мы можем определить, какой драйвер контролирует этот узел через /proc/devices:

```
lkp@lkp:~$ less /proc/devices
Character devices: 1 mem
```

Давайте рассмотрим драйвер устройства mem и поищем вхождение «random»;

```
653 static int memory_open(struct inode * inode, struct file * filp)
654 {
655   switch (iminor(inode)) {
656     case 1:

676     case 8:
677       filp->f_op    &random_fops;
678       break;
679     case 9:
680       filp->f_op    &urandom_fops;
681       break;
```

[1] mknod создает файлы блочных и символьных устройств.

[2] head -c4 берет первые 4 байта, а od -x форматирует их в шестнадцатеричный вид.

Строки 655-681

Конструкция switch инициализирует поддержку драйверов на основе младшего номера устройства, с которым мы работаем. Точнее говоря, устанавливаются f ilp и fops.

Возникает вопрос, что такое f ilp и что такое fop?

10.1.2 FilpsnFops

f ilp - это просто указатель на файловую структуру, а fop - это указатель на структуру f i l e_operations. Ядро использует структуру f ile_operations для определения того, какую функцию при работе с данным файлом вызывать. Здесь выбирается раздел структур, используемых в драйвере устройства random.

```
556 struct file {
557   struct list head   f list;
558   struct dentry  *f dentry;
559   struct vfsmount  *f_vfsmnt;
560   struct file operations  *f  op;
561   atomic_t   f_count;
562   unsigned int  f_flags;

581   struct address_space  *f_mapping;
582 };


863 struct file operations {
864   struct module *owner;
865   loff_t (*llseek) (struct file *, loff_t, int);
866   ssize t (*read) (struct file *, char   user *, size t, loff t *);
867   ssize t (*aio read) (struct kiocb *, char   user *, size t, loff t) ;
868   ssize_t (*write) (struct file *, const char _user *, size_t, loff_t *) ;
869   ssize_t (*aio write)(struct kiocb *, const char   user *,
                size t, loff t);
870   int (*readdir) (struct file *, void *, filldir  t);
871   unsigned int (*poll) (struct file *, struct poll table  struct *);
872   int (*ioctl) (struct inode *, struct file *, unsigned int,
                  unsigned long);

888  };
```

Драйвер устройства random определяет, какую операцию производить следующим образом: функции, реализованные в драйвере, должны соответствовать прототипам, перечисленным в структуре f ile_operations:

```
1824 struct file operations random fops = {
1825   .read   = random_read,
1826   .write    = random write,
1827   .poll    = random_poll,
1828   .ioctl    = random_ioctl,
1829 };
1830
1831 struct file operations urandom fops = {
1832    .read   = urandom read,
1833    .write   = random_write,
1834    .ioctl    = random ioctl,
1835  };
```

Строки 1824-1829

Устройство random реализует операции read, write, poll и ioctl.

Строки 1831-1835

Устройство urandom реализует операции read, write и ioctl.

Операция poll позволяет программисту выполнять проверку перед выполнением операции для проверки блокировки. Существует соглашение, что /dev/random блокируется, если затребовано больше байтов, чем находится в пуле энтропии[1]; /dev/urandom не блокируется, но может вернуть не полностью случайные данные, если пул энтропии слишком мал. (Более подробную информацию см. в man pages, особенно в man 4 random.)

По мере углубления в код обратите внимание, что, когда с /dev/random выполняются операции чтения, ядро передает управление в функцию random_read () (см. строку 1825); random_rand() определен следующим образом:

```
drivers/char/random.c
1588 static ssize t
1589 random_read(struct file * file, char  _user * buf, size_t
nbytes, loff_t *ppos)
```

У этой функции следующие параметры:

- file. Указывает на структуру устройства.
- buf. Указывает на область пользовательской памяти, где сохраняется результат.

[L] **В драйвере устройства random энтропия означает системные данные, которые невозможно предсказать. Обычно, она собирается из клавиатурного времени, перемещений мыши и другого нерегулярного ввода.**

- nbytes. Размер требуемых данных.
- ppos. Указывает на позицию в файле, к которой получает доступ пользователь.

Получается интересный результат: если драйвер выполняется в пространстве ядра, но буфер находится в пользовательском пространстве, как мы можем безопасно получить доступ к данным в buf? Следующий подраздел объясняет процесс перемещения данных между пользовательской памятью и памятью ядра.

10.1.3 Пользовательская память и память ядра

Если мы просто употребляем memcpy () для копирования буфера из пространства ядра в пользовательское пространство, операция копирования может не сработать из-за того, что адрес пользовательского пространства может быть замещен при возникновении mempcy (). В Linux есть функции copy_to_user () и copy_f rom_user (), что позволяет драйверам перемещать данные между пространством ядра и пользовательским пространством. В read_random () это делается в функции extract_entropy (), но есть и дополнительные особенности:

```
drivers/char/random.c
 1:  static ssize t extract^entropy(struct entropy store *r, void *
buf,
 2:           size_t nbytes, int flags)
 3:  { 1349 static ssize t extract entropy(struct entropy store *r, void *
buf, 13 50        size_t nbytes, int flags) 1351 {

1452      /* Копирование данных в буфер назначения */
1453      i = min(nbytes, HASH BUFFER SIZE*sizeof (   u32)/2);
1454      if (flags & EXTRACT_ENTROPY_USER) {
1455        i -= copy to user(buf, (   u8 const *)tmp, i);
1456        if (!i) {
1457          ret = -EFAULT;
1458          break;
1459        }
1460      } else
1461        memcpy(buf, (__u8 const *)tmp, i);
```

extract_entropy () имеет следующие параметры:

- г. Указатель на внутреннее хранилище информации, игнорируемое в целях нашей дискуссии.
- buf. Указатель на область памяти, заполняемую данными.

- **nbytes.** Размер записываемых в **buf** данных.
- flags. Информируем функцию, где находится **buf** в пользовательской памяти или же в памяти ядра.

extract_entropy () возвращает ssize_t, хранящую размер в байтах случайно сгенерированных данных.

Строки 1454-1455

Если flags говорит нам, что buf указывает на место в нашей памяти, мы используем copy_to_user () для копирования указываемой trap памяти ядра в пользовательскую память, указываемую buf.

Строки 1460-1461

Если buf указывает на место в памяти ядра, мы просто используем memcpy () для копирования данных.

Получение случайных данных необходимо как пользовательским программам, так и программам ядра; пространство программ ядра может избегать накладок copy_to_user (), не устанавливая соответствующий флаг. Например, ядро может реализовывать зашифрованную файловую систему и может избегать накладок на копирование в пользовательское пространство.

10.1.4 Очереди ожидания

Мы немного отвлечемся на объяснение того, как данные перемещаются между пользовательской памятью и памятью ядра. Давайте вернемся в read_random () и посмотрим, как она применяет очередь ожидания.

Иногда драйверу может потребоваться подождать некоторого события, например доступа к системному ресурсу. В этом случае мы не хотим, чтобы ядро ожидало возможности доступа. Заставлять ядро ждать проблематично, так как все остальные системы в этом случае замрут[1]. С помощью объяснения очереди ожидания вы можете отложить время обработки до тех пор, пока не произойдет ожидаемое вами событие.

Для обеспечения этого ожидания используются две структуры: очередь ожидания и голова очереди ожидания. Модуль должен создавать голову очереди ожидания и иметь части, использующие макросы sleep_on и wake_up для управления. Именно это происходит в random_read():

[к] **На самом деле выполняющий задачу процессор будет ожидать. На многопроцессорных системах другие процессоры продолжают выполнение.**

```
1588 static ssize_t
1589 random__read(struct file * file, char   user * buf, size t nbytes,
                 loff t *ppos)
1590 {
1591   DECLARE WAITQUEUE(wait, current);

1597              while (nbytes > 0) {

1608    n = extract entropy(sec random state, buf, n,
1609          EXTRACT ENTROPY USER |
1610          EXTRACT ENTROPY LIMIT |
1611          EXTRACT ENTROPY SECONDARY);

1618    if (n == 0) {
1619       if (file->f_flags & 0_NONBLOCK) {
1620         retval = -EAGAIN;
1621         break;
1622       }
1623       if (signal__pending( current) ) {
1624         retval = -ERESTARTSYS;
1625         break;
1626       }

1632     set_current_state(TASK_INTERRUPTIBLE);
1633      add wait queue(&random read wai t, &wai t);
1634
1635      if (sec random_state->entropy_count / 8 == 0)
1636        schedule0;
1637
1638      set_current_state (TASK_RUNNING) ;
1639      remove_wait_queue(&random_read_wait, &wait);

1645      continue;
1646 }
```

Строка 1591

Очередь ожидания wait инициализируется текущей задачей. Макрос current ссылается на указатель task_struct текущей задачи.

Строки 1608-1611

Мы извлекаем порцию случайных данных из устройства.

Строки 1618-1626

Если мы не извлекли необходимое количество случайных данных из пула энтропии и задача не заблокирована или поступил требуемый ею сигнал, мы возвращаем ошибку в код вызова.

Строки 1631-1633

Настройка очереди ожидания; random__read () использует собственную очередь ожидания, а random_read_wait, наоборот, использует системную очередь ожидания.

Строки 1635-1636

В данном месте мы выполняем блокирующее чтение, и, если у нас не хватает 1 байта энтропии, мы освобождаем управление процессором с помощью вызова schedule (). (Переменная entropy_count хранит биты и не биты, поэтому необходимо производить деление на 8 для определения наличия полного байта энтропии.)

Строки 1638-1639

Когда мы в конце концов выполняем перезапуск, мы очищаем нашу очередь ожидания.

ПРИМЕЧАНИЕ. Устройство random в Linux требует заполнения очереди энтропии перед возвратом. Устройство urandom не предъявляет таких требований и возвращается независимо от количества данных, доступных в пуле энтропии.

Давайте подробнее рассмотрим, что происходит при вызове schedule () задачи.

```
2184 asmlinkage void  _sched schedule(void)

2185 {

2209                        prev = current;

2233   switch_count = &prev->nivcsw;
2234   if (prev->state && !(preempt_count() & PREEMPT_ACTIVE)) {
2235     switch count = &prev->nvcsw;
2236     if (unlikely((prev->state & TASK_INTERRUPTIBLE) &&
2237       unlikely(signal_pending(prev))))
2238       prev->state = TASK RUNNING;
2239     else
2240       deactivate task(prev, rq);
2241     }
2242 ...
```

Строка 2209

Указатель на структуру текущей задачи сохраняется в переменную prev. В случае, когда сама задача вызывает schedule () , current указывает на эту задачу.

Строка 2233

Мы сохраняем счетчик переключения задач nivcsw в switch_count. Далее после удачного переключения он будет увеличен[1].

Строка 2234

Мы попадаем сюда только в том случае, когда состояние задачи prev->state не равно нулю и нет приоритетного прерывания обслуживания ядра. Другими словами, мы попадаем в этот блок кода, только если состояние задачи не равно TASK_RUNNING и ядро не выполняет приоритетного прерывания обслуживания задачи.

Строки 2235-2241

Если задача прерывается, мы можем быть уверены, что она хочет освободить управление. Если подан сигнал для задачи, которая хочет освободить управление, мы устанавливаем состояние задачи в TASK_RUNNING, чтобы иметь возможность выбрать с помощью планировщика, какой задаче передать управление. Если сигнал не поступил, что обычно и происходит, мы деактивируем задачу и устанавливаем switch_count в nivcsw. Планировщик увеличит switch__count позже. При этом увеличивается nvcsw или nivcsw.

Далее функция schedule () выбирает следующую задачу в очереди выполнения планировщика и переключает управление на эту задачу[2].

С помощью вызова schedule () мы можем позволить задаче передать управление процессором другой задаче ядра, когда текущая задача решит, что она будет по какой-либо причине ожидать. Другие задачи в ядре могут использовать это время, и можно надеяться, что, когда вызов schedule () вернет управление этой задаче, причина ожидания исчезнет.

Возвращаясь из нашего отступления по поводу планировщика к функции random., read (), ядро в конечном счете передает управление обратно в random_read () и мы очищаем нашу очередь ожидания и продолжаем. Это происходит в цикле, и, если система сгенерировала достаточно энтропии, мы можем вернуться с требуемым количеством случайных байтов.

random_read () устанавливает свое состояние в TASK_INTERRUPTIBLE перед вызовом schedule () для того, чтобы позволить прерывать себя поступающими сигналами во время нахождения в очереди ожидания. Собственный код драйвера генерирует

[1] **См. гл. 4 и 7, где описано использование счетчиков переключения контекста.**

[2] **Подробную информацию см. в подразд. «swith_toQ» гл. 7.**

эти сигналы, когда накапливается избыточное количество энтропии, с помощью вызова wake_up_interruptible () в batch_entropy_j?rocess () и random_ioctl (). TASK_UNINTERRUPTIBLE обычно используется когда процесс ожидает ответа от аппаратного обеспечения в отличие от программного обеспечения (когда обычно используется TASK_INTERRUPTIBLE).

Код, используемый random_read () для передачи управления другой задаче (см. строки 1632-1639, drivers /char /random, c) - является вариацией interruptible_sleep_on () из кода планировщика.

```
2489 #define SLEEP ON VAR       \
2490    unsigned long flags;      \
2491    wait_queue_t wait;        \
2492    init waitqueue entry(&wait, current); 2493
2494 #define SLEEP  ON HEAD             \
2495    spin lock irqsave(&q->lock,flags);      \
2496    __ add_wait_queue(q, &wait) ;           \
2497    spin unlock(&q->lock);
2498
2499 #define SLEEP ON  TAIL              \
2500    spin_lock_irq(&q->lock);           \
2501     _remove_wait_queue(q,  &wait);          \
2502    spin__unlock_irqrestore(&q->lock,  flags);
2503
2504 void  fastcall _____sched interruptible_sleep_on(wait_queue_head_t  *q)
2505 {
2506  SLEEP ON VAR
2507
2508  current->state = TASK INTERRUPTIBLE;
2509
2510  SLEEP_ON_HEAD
2511  scheduleO;
2512  SLEEP_ON_JTAIL
2513 }
```

q-это структура wait_queue_head, координирующая сон и ожидание модуля.

Строки 2494-2497

Атомарное добавление нашей задачи в очередь ожидания q.

Строки 2499-2502

Атомарное удаление задачи из очереди ожидания q.

Строки 2504-2513

Добавление очереди ожидания. Передает управление процессором другой задаче. Когда мы передаем управление, сама задача удаляется из очереди ожидания.

random_read () использует собственную очередь ожидания вместо стандартного макроса, но, по существу, выполняет interruptible_sleep_on() за исключением того, что, если у нас больше чем нужно байтов энтропии, мы не передаем управление, а повторяем цикл и получаем всю требуемую энтропию. Если энтропии недостаточно, random_read () ожидает до тех пор, пока не будет пробуждена с помощью interruptible_sleep_on () из собирающего энтропию процесса драйвера.

10.1.5 Очереди выполнения прерывания

Драйверам устройств в Linux приходится работать с прерываниями, генерируемыми устройствами, которым они предоставляют интерфейс. Прерывания запускают обработчики прерываний в драйвере устройства и заставляют весь выполняемый в данный момент код - как пользовательский, так и код ядра - прервать свое выполнение. Поэтому желательно, чтобы обработчик прерывания драйвера устройства выполнялся как можно быстрее для предотвращения длительных остановок процессов ядра.

Тем не менее это приводит к стандартной дилемме обработки прерываний: как нам обрабатывать прерывания, которые требуют выполнить достаточно много работы? Стандартный ответ заключается в использовании функций верхней половины и нижней половины. Функции верхней половины - это быстрые обработчики, прерывающие и планирующие функции нижней половины, в которых выполняется основная работа при возникновении такой возможности. Обычно функции верхней половины запускаются при отключенных прерываниях, для того чтобы обработчик прерывания не был прерван тем же прерыванием. Поэтому драйвер устройства не обрабатывает рекурсивные прерывания. Функции нижней половины обычно запускаются при включенных прерываниях, поэтому другие прерывания могут быть обработаны во время выполнения основной работы.

В предыдущих ядрах Linux это разделение на функции верхней половины и нижней половины также называлось быстрыми и медленными прерываниями, которые обрабатывались очередями задач. Теперь в ядре Linux 2.6 появилась концепция очередей выполнения, которые в данный момент являются стандартным способом работы с прерываниями нижней половины.

Когда ядро получает прерывание, процессор останавливает выполнение текущей задачи и немедленно обрабатывает прерывание. Когда процессор входит в этот режим, он обычно переводится в контекст прерывания. Ядро в контексте прерывания определяет, какому обработчику прерываний передать управление. Когда драйвер устройства хочет обработать прерывания, он использует request_irq () для запроса номера прерывания и регистрирует функцию-обработчик, которая будет вызвана при обнаружении этого прерывания. Регистрация обычно выполняется во время инициализации модуля. Функции

обработчиков верхней половины регистрируются с помощью request_irq(), выполняют минимум работы и помещают выполняемую далее работу в очередь выполнения.

Как и request_irq() в верхней половине, очереди выполнения обычно регистрируются во время инициализации модуля. Они могут инициализироваться статически с помощью макроса DECLARE_WORK () или структуры выполнения, которую можно выделить и инициализировать динамически с помощью вызова INIT_WORK (). Вот определение этих макросов:

```
3 0  #define DECLARE_WORK(n, f, d)                    \
31     struct work_struct n = ________ WORK_INITIALIZER(n, f, d)

45  #define INIT_WORK(_work, _func, _data)                    \
46     do {                                  \
47        INIT_LIST_HEAD(&(_work)->entry);                  \
48        (_work)->pending = 0 ;                  \
49        PREPARE_WORK((_work), (_func), (_data));              \
50        init_timer(&(_work)->timer);                  \
51     } while (0)
```

Оба макроса получают следующие аргументы:

- **n** или **work.** Имя создаваемой и инициализируемой структуры выполнения.
- **f** или **func.** Функция, запускаемая, когда структура выполнения удаляется из очереди выполнения.
- **d** или **data.** Хранит данные, передаваемые в функцию f или func, когда они запускаются.

Функция-обработчик прерывания, зарегистрированная в register_irq(), далее принимает прерывание и посылает соответствующие данные из обработчика прерывания верхней половины в нижнюю половину с помощью настройки раздела данных work_struct и вызова schedule_work () для очереди выполнения.

Присутствующий в очереди выполнения код функции работает в контексте процесса и поэтому может выполнять работу, которую невозможно выполнить в контексте прерывания, такую, как копирование из пользовательского пространства или в него или сон.

Тасклеты похожи на очереди выполнения, но работают внутри контекста прерывания. Они полезны, когда вам нужно выполнить немного работы в нижней половине и вы хотите сохранить служебные обработчики прерываний верхней половины и нижней половины. Тасклеты инициализируются с помощью макроса DECLARE_TASKLET ():

```
include/linux/interrupt.h
136 #define DECLARE_TASKLET(name, func, data) \
```

```
137 struct tasklet_struct name = { NULL, 0, ATOMIC_INIT (0) , func, data }
```

- name. Имя создаваемой структуры тасклета.
- func. Вызываемая планировщиком функция тасклета.
- data. Хранит данные, передаваемые в func при выполнении тасклета.

Для планирования тасклета используется tasklet_schedule ():

```
include/linux/interrupt.h
171 extern void FASTCALL(   tasklet schedule(struct tasklet struct *t) ) ;
172
173 static inline void tasklet schedule(struct tasklet struct *t)
174 {
175   if ( !test and setjDit(TASKLET STATE  SCHED, &t->state) )
17 6      tasklet schedule(t) ;
177 }
```

- tasklet_struct. Имя тасклета, создаваемого с помощью DECLARE_TASKLET ().

В обработчике прерывания верхней половины вы можете вызвать tasklet_schedule () и гарантировать, что когда-нибудь в будущем определенная в тасклете функция будет выполнена. Тасклеты отличаются от очередей выполнения тем, что различные тасклеты могут одновременно выполняться на разных процессорах. Если тасклет уже запланирован и перепланирован снова перед его выполнением, то он будет выполнен только один раз. Так как тасклеты выполняются в контексте прерывания, они не могут спать или копировать данные в пользовательское пространство. Из-за работы в контексте прерывания, если различным тасклетам требуется общаться друг с другом, они могут делать это только с помощью циклических блокировок.

10.1.6 Системные вызовы

Существуют и другие способы добавления кода в ядро помимо драйверов устройств. Системные вызовы ядра Linux (syscalls) представляют собой метод, с помощью которого программы пользовательского пространства могут получать доступ к службам ядра и аппаратному обеспечению системы. Программам пользовательского режима доступны многие библиотечные С-функции и один или несколько системных вызовов для выполнения отдельных функций. На самом деле получить доступ к системным вызовам можно и из кода ядра.

По своей натуре реализация системных вызовов является аппаратно-специфической. На архитектуре Intel все системные вызовы используют программное прерывание 0x80. Параметры системных вызовов передаются через регистры общего назначения. Реализа-

ция системных вызовов на архитектуре x86 ограничивает количество параметров пятью. Если требуется больше 5 параметров, можно передать указатель на блок параметров. Во время выполнения ассемблерной инструкции **int 0x80** вызывается специальная функция ядра с помощью механизма обработки исключений процессора.

10.1.7 Другие типы драйверов

До сих пор все драйверы устройств, с которыми мы имели дело, были символьными устройствами. Их намного проще понять, но вы можете захотеть написать и другие драйверы, которые общаются с ядром по-другому.

Блочные устройства похожи на символьные устройства в способе доступа к ним через файловую систему; /dev/hda - это файл устройства в файловой системе для первичного ШЕ-диска. Блочные устройства регистрируются и удаляются из системы так же, как и символьные устройства, с помощью функций register_blkdev() и unregister_blkdev().

Основная разница между блочными устройствами и символьными устройствами заключается в том, что блочные устройства не предоставляют собственной функциональности для чтения и записи; вместо этого они используют метод запросов.

Ядро 2.6 претерпело несколько серьезных изменений в подсистеме блочных устройств. Старые функции, такие, как block__read() и block_write(), а также структуры ядра, такие, как blk_size и blksize_size, убраны. Этот раздел фокусируется исключительно на реализации блочных устройств в ядре 2.6.

Если вам нужно, чтобы ядро Linux работало с дисковым устройством (или подобным диску устройством), вам нужно написать драйвер блочного устройства. Драйвер должен информировать ядро о том, с каким диском оно имеет дело. Это делается с помощью структуры gendisk:

```
include/linux/genhd.h
82 struct gendisk {
83 int major;      /*старший номер драйвера */
84 int first minor;
85 int minors;
86 char disk name[32];     /* имя старшего драйвера */
87 struct hd_struct **part;   /* [индексация по младшему] */
88 struct block device operations *fops;
89 struct request queue *queue;
90 void *private_data;
91 sector_t capacity;
```

Строка 83

ma j or - это старший номер блочного устройства. Он может устанавливаться статически или может динамически генерироваться с помощью register_ blkdev (), как и в случае с символьным устройством.

Строки 84-85

first__minor и minors используются для определения количества разделов в блочном устройстве; minors содержит максимальное количество младших номеров имеющихся устройств; f irst_minor содержит первый младший номер устройства для блочного устройства.

Строка 86

disk_name - это 32-символьное имя для блочного устройства. Оно появляется в файловой системе /dev, sysf s и /proc/partitions.

Строка 87

hd_struct устанавливает разделы, связанные с блочным устройством.

Строка 88

fops - это указатель на структуру block_operations, содержащую операции open, release, ioctl, media_changed и revalidate__disk. (См. include/linux/f s.h.) В ядре 2.6 каждое устройство имеет собственный набор операций.

Строка 89

reques t_cueue - это указатель на очередь, помогающую управлять связанными с устройством операциями.

Строка 90

private_data указывает на информацию, недоступную из блочной подсистемы ядра. Обычно используется для сохранения данных, используемых в низкоуровневых устройствозависимых операциях.

Строка 91

capacity - это размер блочного устройства в секторе из 512 байт. Если устройство является съемным, таким, как флоппи-диск или CD, capacity 0 означает, что диск отсутствует. Если ваше устройство не использует секторы 512 байт, вам нужно установить эту переменную в соответствующий эквивалент. Например, если выше устройство имеет тысячу 256-байтовых секторов, это эквивалентно пятистам 512-байтовых секторов.

Дополнительно для того, чтобы иметь структуру gendisk, блочному устройству нужна структура циклической блокировки для использования с очередью запроса.

Как циклическая блокировка, так и поля структуры gendisk должны быть инициализированы драйвером устройства. (Демонстрацию инициализации диска в памяти драйвером блочного устройства можно найти по адресу http://en.wikipedia.org/wiki/Ram_disk.) После инициализации устройства и готовности к обработке запросов для добавления блочного устройства в систему должна быть вызвана функция add_disk().

Наконец, если блочное устройство будет использоваться в качестве источника энтропии для системы, модуль инициализации может также вызывать и add_disk_randomness (). (Более подробную информацию см. в drivers/char/ random, с.)

Теперь, когда мы раскрыли основы инициализации блочного устройства, мы можем рассмотреть его добавление, выход и очистку драйвера блочного устройства. В ядре Linux 2.6 это делается очень просто.

del_gendisk (struct gendisk) удаляет gendisk из файловой системы и очищает информацию о разделах. За этим вызовом должен следовать put disk (struct gendisk), освобождающий ссылку в ядре на gendisk. Блочное устройство удаляется с помощью вызоваunregister_blkdev(int major, char [16] device_name), что позволяет далее освободить структуру gendisk.

Также нам нужно освободить связанную с драйвером блочного устройства очередь выполнения. Это делается с помощью blk_cleanup_queue (struct * request_ queue).

> ПРИМЕЧАНИЕ. Если вы можете только ссылаться на очередь выполнения через структуру gendisk, необходимо вызывать blk_cleanup_queue перед освобождением gendisk.

В обзоре инициализации и выключения блочного устройства мы можем легко избежать разговора о специфике очереди выполнения. Но теперь, когда драйвер настроен, нужно что-то сделать, а очередь выполнения - это именно то, что предоставляет основные функции для чтения и записи блочного устройства.

include/linux/blkdev.h
576 extern request queue t *blk init queue(request fn proc *,
spinlock_t *);

Строка 576

Для создания очереди выполнения мы используем blk_init_queue и передаем указатель на циклическую блокировку для управления доступом к очереди и указатель на функцию запроса, вызываемого при доступе к устройству. Функция доступа должна иметь следующий прототип:

```
static void my_request__function( request_queue *q) ;
```

Внутренняя реализация функции запроса обычно использует несколько вспомогательных функций. Для определения следующего обрабатываемого процесса вызывается функция elv_next_request () и возвращается указатель на структуру запроса или возвращается нуль, если следующего запроса нет.

В ядре 2.6, драйвер блочного устройства работает через структуру ВЮ в структуре запроса. ВЮ расшифровывается как Block I/O (блочный ввод-вывод) и полностью определяется в include/linux/bio.h.

Структура ВЮ содержит указатель на список структур biovec, определенных следующим образом:

```
include/linux/bio.h
47 struct bio vec {
48 struct page    *bv_page;
49 unsigned int   bv len;
50 unsigned int   bv offset;
51 };
```

Каждый biovec использует свою структуру страниц для хранения буферов данных, в которые в конечном счете пишутся и считываются данные с диска. Ядро 2.6 имеет несколько вспомогательных bio для перемещения по данным, хранящимся в структуре bio.

Для определения размера операции ВЮ, вы можете также обратиться к структуре bio_size в структуре ВЮ для получения результата в байтах или использовать макрос bio_sectors () для получения размера в секторах. Тип блочной операции, READ или WRITE, можно определить с помощью bio_data_dir ().

Для перемещения по списку biovec в структуре ВЮ используется макрос bio_f or_each_segment (). Внутри цикла по мере углубления в biovec можно использовать и другие макросы - bio_page (), bio_of f set (), bio_curr_sectors () и bio_data(). Более подробную информацию можно найти в include/ linux. bio. h и Documentation/block/biodoc . txt.

Некоторые комбинации информации, содержащейся в biovec и структурах страниц, позволяют вам определить, какие данные нужно читать и какие записывать на блочное устройство. Низкоуровневые детали того, как читать и писать на устройство, связаны с аппаратным обеспечением этого драйвера блочного устройства.

Теперь мы знаем, как работать со структурой ВЮ, и нам остается только понять, как работать со структурой списка запросов структур ВЮ. Это делается с помощью другого макроса - rq_f or_each__bio.

include/linux/blkdev.h
```
495 #define rq_for_each_bio(„bio, rq)                    \
496   if ((rq->bio))                                     \
497     for (_bio = (rq)->bio; _bio; _bio = bio->bi_next)
```

Строка 495

bio - это текущая структура BIO, а rq - итератор для перемещения по ней.

После обработки каждой структуры ВЮ драйвер должен обновить ядро в соответствии со своим прогрессом. Это делается с помощью end_that_request_f irst ().

include/linux/blkdev.h
```
557 extern int end_that_request_first(struct request *, int, int);
```

Строка 557

Первый аргумент int должен быть ненулевым во избежание возникновения ошибки, а второй аргумент int представляет количество секторов, обработанных устройством.

Когда end_that_request__f irst () возвращает 0, все процессы обработаны и необходимо произвести очистку. Это делается с помощью вызова blkdev_dequeue_ request () и end_that_request_last () в том же порядке - обе функции получают запрос как единственный аргумент.

После этого функция запроса выполняет свою работу, а блочная подсистема использует функцию очереди запроса драйвера блочного устройства для выполнения дисковой операции. Устройству нужно обработать несколько функций ioctl, так как наш диск в памяти обрабатывает разделы, хотя это тоже зависит от типа блочного устройства.

Этот раздел касается только основ блочного устройства. Существуют перехватчики Linux для операций DMA, кластеризация, подготовка команд очереди запросов и другие особенности более сложных блочных устройств. Теперь вы можете переходить к чтению документации из Documentation/block.

10.1.8 Модель устройства и sysfs

В ядре Linux 2.6 появилась новая модель устройств, тесно связанная с sysfs. Модель устройства хранит набор внутренних данных, связанных с устройствами и драйверами в системе. Система следит за существованием этих устройств и разбивает их на классы: блоки, ввод, шины и т. д. Кроме этого, система следит за тем, какие драйверы существуют и как они связаны с управляемыми ими устройствами. Модель устройства присутствует в ядре, а sysfs - это окно в эту модель. Из-за того что некоторые устройства

и драйверы не показывают себя через sysf s, о sysf s стоит думать как об общедоступной демонстрации модели устройств ядра.

Некоторые устройства имеют несколько вхождений в sysf s.

В модели устройства хранится только копия данных, хотя получить доступ к этим данным можно множеством способов, как показано в символической ссылке в дереве sysf s.

Иерархия sysf s связана со структурами ядра kobj ee t и kset. Модель достаточно сложна, но большинство написанных драйверов не опускаются до реализации ее мелких деталей[1]. С помощью концепции атрибутов sysf s вы работаете с kobject в абстрактном режиме. Атрибуты - это части устройства или модели драйвера, к которым можно получить доступ через файловую систему sysf s. Они могут быть внутренней переменной модуля, контролирующей, как модуль управляет задачами, или могут быть напрямую связаны с различными аппаратными настройками. Например, RF-передатчик может иметь базовую частоту, на которой он работает, в то время как индивидуальная настройка реализована в виде отступа от базовой частоты. Изменение базовой частоты может быть выполнено с помощью изменения атрибута модуля в драйвере RF в sysf s.

При доступе к атрибуту sysf s вызывает функцию для обработки этого доступа, show () для чтения и store () для записи. Существует одностраничное ограничение на размер данных, которые можно передать в функции show () или store ().

Вооружившись пониманием того, как работает sysf s, мы можем перейти к тонкостям того, как драйвер регистрируется в sys f s, отображает атрибуты и регистрирует специальные функции show () и store () для работы с атрибутами, к которым мы получаем доступ.

Первой задачей является определение, к какому классу относится ваше новое устройство и под какой класс попадает его драйвер (например, usb_device/ net_devi.ee, pci_device, sys_device и т. д.). Все эти структуры имеют поле char *name; sysf s употребляет это поле с именем для демонстрации нового устройства в иерархии sysf s.

После выделения и именования структуры устройства вам нужно создать инициализацию структуры devicer_driver.

```
include/linux/device.h
102 struct device_driver {
103     char     * name;
104     struct bus_type      *  bus; 105
10 6    struct  semaphore     unload_sem;
107     struct kobject              kobj;
108     struct  list_head     devices; 109
```

[1] Обратитесь к файлу Documentation/f ilesystems/sysf s . txt в исходниках ядра.

```
110   int   (*probe)    (struct device * dev);
111   int   (*remove)    (struct device * dev);
112   void  (*shutdown)   (struct device * dev);
113   int   (*suspend)    (struct device * dev, u32 state, u32 level);
114   int   (*resume)    (struct device * dev, u32 level);
115 };
```

Строка 103

name связана с отображаемым именем драйвера в иерархии sysf s.

Строка 104

bus обычно заполняется автоматически; автору драйвера не нужно из-за волноваться.

Строки 105-115

Программисту не нужно заполнять оставшиеся поля. Они должны инициализировать автоматически на уровне шины.

Мы должны зарегистрировать наш драйвер во время инициализации с помощью вызова driver_register (), который передает большую часть работы bus_add_ driver (). Аналогично при выходе из драйвера нужно добавить вызов driver_unregister ().

drivers/base/driver.c

```
8 6  int  driver__register(struct  device_driver  *  drv)
87  {
88   INIT LIST HEAD(Scdrv->devices) ;
89   init MUTEX LOCKED(&drv->unload sem);
90   return bus add driver(drv);
91  }
```

После регистрации драйвера атрибуты драйвера могут быть созданы с помощью структур driver_attribute и вспомогательных макросов DRIVER_ATTR:

include/linux/device.h

```
133 #define DRIVER ATTR( name, mode, show, store)    \
13 4 struct driver attribute driver attr ## name = {        \
135    .attr = {.name =  _stringify(_name), .mode = _mode, .owner =
THIS_MODULE },    \
13 6    .show    =  show,      \
137    .store   =  store,       \
138 };
```

Строка 135

name - это имя атрибута драйвера; mode - это битовая карта, описывающая уровень защищенности атрибута; include/linux/stat .h содержит большинство из этих режимов, примерами которых могут служить S_IRUGO (только чтение) и S_IWUSR (доступ на запись только для root).

Строка 136

show - имя функции драйвера, используемой при чтении атрибута через sysf s. Если чтение не разрешено, следует использовать NULL.

Строка 137

store - это имя функции драйвера, используемой при записи атрибутов через sysf s. Если запись не разрешена, следует использовать NULL.

Функции драйвера, реализующие show() и store () для специального драйвера, должны иметь приведенные ниже прототипы:

```
include/linux/sysfs.h
34 struct sysfs ops {
35 ssize t (*show)(struct kobject *, struct attribute *,char *);
36 ssize_t (*store)(struct kobject *,struct attribute *,const char *,
                      size t);
37 };
```

Вспомните, что размер читаемых и записываемых через sysf s данных атрибутов ограничен PAGE_SIZE байтами. Функции атрибутов драйвера show () и store () должны учитывать это ограничение.

Эта информация должна позволить вам добавлять базовую функциональность в драйвер устройства ядра. Подробности реализации sysf s и kobj ect можно прочитать в директории Documentation/device-model.

Другим типом драйверов устройств являются драйверы сетевых устройств. Сетевые устройства передают и получают пакеты данных и могут быть не обязательно аппаратными устройствами - устройство обратной связи (loopback) - это программное сетевое устройство.

10.2 Написание кода

1 0.2 Л Основы устройств

Когда вы создаете драйвер устройства, он связывается с операционной системой через некоторый элемент (файл) в файловой системе. Этот элемент имеет старший номер, который показывает для ядра, какой драйвер использовать, когда на файл ссылаются. Этот

файл имеет также младший номер, который может использовать сам драйвер для уточнения информации. Когда драйвер устройства загружен, он регистрирует свой старший номер. Эту регистрацию можно увидеть через /proc/devices.

```
lkp# less /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  6 lp
  7 vcs
 10 misc
 29 fb
 12 ptm
 13 pts
Block devices:
  1 ramdisk
  2 fd
  3 ideO
  7 loop
 22 idel
```

Это число вводится в /proc/devices, когда драйвер регистрирует себя в ядре; для символьных устройств оно вызывается функцией register_chrdev ().

include/linux/fs.h 1: int register_chrdev(unsigned int major, const char *name, 2: struct file_operations *fops)

- major. Старший номер регистрируемого устройства. Если ma j or равен 0, ядро динамически назначает старшее число, не конфликтующее с другими загруженными модулями.
- name. Строка представления устройства в дереве /dev файловой системы.
- fops. Указатель на структуру файловых операций, определяющих, какие операции можно выполнить на зарегистрированном устройстве.

Использование 0 в качестве старшего номера связано с методом создания номеров устройств для устройств, которые не устанавливают старшие номера (приводы IDE всегда используют 3, SCSI - 8, флоппи-дисководы - 2). С помощью динамического назначения

старших номеров мы можем избежать проблемы выбора старшего номера, который уже был выбран другим устройством[1]. Последовательность создания узла файловой системы немного усложняется из-за того, что после загрузки модуля мы должны проверить, какой старший номер назначается устройству. Например, во время тестирования устройства вам нужно сделать следующее:

```
lkp@lkp# insmod my_module.o
lkp@lkp# less /proc/devices
1 mem

233 my module
lkp@lkp# mknod c /dev/my moduleO 233 0
lkp@lkp# mknod c /dev/my_modulel 233 1
```

Этот код показывает, как мы можем вставлять наш модуль с помощью команды insmod, которая инсталлирует загружаемый модуль в запущенное ядро. Код нашего модуля содержит следующие строки:

```
static   int my_module_major=0;

module_param(my_module_major/  int,  0);

result = register_chrdev(ray_module_major/  "my_module",
&my_module_fops) ;
```

Первые две строки демонстрируют, как мы создаем старшее число по умолчанию со значением 0 для динамического назначения и даем возможность пользователю его переопределять с помощью переменной my_rnodule_major в качестве параметра:

```
include/linux/moduleparam.h
 1: /* Это фундаментальная функция для регистрации параметров загрузки/
модуля; perm устанавливает видимость в driverfs: 000 значит, что его
нет; read-биты значат, что он читаемый; write значит, что он
перезаписываемый. */

/* Вспомогательные функции: byte, short, ushort, int, uint, long, ulong,
   charp, bool или invbool, или XXX, если вы определите param get XXX,
   param_set_XXX и param_check_XXX. */
```

[1] Функция register_chrdev() возвращает назначенный старший номер. Он может быть полезным для получения информации при динамическом назначении старших номеров.

```
2:  #define module_param(name,  type,  perm)
```

В предыдущих версиях Linux макросом module_param был MODULE_PARM; в версии 2.6 он упразднен и вместо него нужно использовать module_param.

- name. Строка, используемая для доступа к переменной параметра.
- type, тип значения, сохраняемого в параметре name.
- perm. Имя параметра, видимого в sysf s. Если вы не знаете, как его отображать в sysf s, используйте значение 0, что означает, что параметр не будет виден через sysf s.

Вспомните, что мы передаем в register_chdev() указатель на структуру fops. Это говорит ядру, что функции обрабатываются ядром. Мы определяем только те функции, которые обрабатывает модуль. Мы объявляем, что read, write, ioctl и open-корректные операции для зарегистрированного нами устройства, и добавляем следующий код:

```
struct  file_operations  my_mod_fops  =  {
    .read    = my_mod_read,
    .write  = my_mod_write,
    .ioctl   = my_mod_ioctl,
    .open    = my_niod_open/};
```

1 0.2.2 Символьное экспортирование

Во время написания сложного драйвера устройства может возникнуть необходимость в экспорте некоторых объявленных в драйвере символов для использования другим модулем ядра. Обычно это используется в низкоуровневых драйверах, над которыми надстраиваются драйверы более высоких уровней.

При загрузке драйвера устройства любые символы экспорта помещаются в таблицу символов ядра. Загружаемые последовательно драйверы могут использовать символы, экспортированные предыдущими драйверами. Когда модули зависят друг от друга, становится важным порядок их загрузки; вызов insmod провалится, если символы, используемые модулем высокого уровня, отсутствуют.

В ядре Linux 2.6 программисту драйверов доступны два макроса для экспорта символов:

```
include/linux/module.h
187  #define  EXPORT_SYMBOL(sym)            \
```

```
188     EXPORT_SYMBOL(sym, "")
189
190 #define EXPORT_SYMBOL_GPL(sym)      \
191   __EXPORT_SYMBOL(sym, "_gpl")
```

Макрос EXPORT_SYMBOL позволяет данному символу стать видимым для других частей ядра с помощью помещения его в таблицу символов ядра. EXPORT_SYMBOL_GPL позволяет только модули, объявленные совместимыми с лицензией GPL в своем атрибуте MODULE_LICENSE. (См. полный список лицензий в include/linux/module.h.)

10.2.3 IOCTL

До сих пор мы в основном имели дело с драйверами устройств, выполняющими действия по чтению и записи только со своего собственного устройства. Что произойдет, когда у вас будет устройство, способное делать больше чем чтение и запись данных? Или у вас есть устройство, которое может выполнять разные операции чтения и записи? Или ваше устройство требует интерфейса для аппаратного контроля? В Linux драйверы устройств обычно используют для решения этих проблем метод ioctl.

ioctl - это системный вызов, позволяющий драйверам устройств обрабатывать специальные команды, которые можно использовать для управления каналами ввода-вывода. Вызов ioctl драйвера устройства должен следовать за определением в структуре file_operations:

```
include/linux/fs.h
863 struct file_operations {

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

Из пользовательского пространства вызов функции ioctl определен следующим образом:

```
int ioctl (int d, int request, ...);
```

Третий аргумент в определении пользовательского пространства является нетипизированным указателем на память. Через него данные передаются из пользовательского пространства в реализацию ioctl драйвера устройства. Это может сложно звучать, но на самом деле использовать ioctl с драйвером достаточно просто.

Во-первых, мы хотим определить, какие номера IOCTL верны для нашего устройства. Мы должны проконсультироваться с файлом Documentation/ioctl-num-ber. txt и выбрать код, не используемый машиной. Проконсультировавшись с текущим

файлом 2.6, мы увидим, что код ioctl для 'д' сейчас не используется. В нашем драйвере мы требуем это в следующем коде:

```
#define MYDRIVER_IOC_MAGIC 'д'
```

Для каждого получаемого драйвером управляющего сообщения нам нужно определить уникальный номер ioctl. Он основан на только что определенном магическом числе:

```
#define MYDRIVER_IOC_OPl      _I0(MYDRIVER_IOC_MAGIC,    0)
#define MYDRIVER_IOC_OP2    _IOW(MYDRIVER_IOC_MAGIC,   1)
#define MYDRIVER_IOC_OP3    _IOW(MYDRIVER_IOC_MAGIC,   2)
#define MYDRIVER_IOC_OP4  _I0RW(MYDRIVER_I0C_MAGIC,    3)
```

Четырем перечисленным операциям (opl, op2, op3 и op4) назначены уникальные номера ioctl, с использованием определенных в include/asm/ioctl.h макросов и MYDRIVER_IOC_MAGIC, являвшимся нашим магическим номером ioctl. Документация красноречиво сообщает о том, что это значит:

```
6  Если вы добавляете новые ioctl's в ядро,  вам нужно использовать макрос
             7  _10  определенный  в  <linux/ioctl.h>:
9     io     8
10   _I0W    an  ioctl with no parameters
11   _I0R    an  ioctl  with write parameters   (copy_from_user) an  ioctl with read
12   _I0WR   parameters      (copy_to_user) an ioctl with both write and read
13           parameters.
14   •Write'
             и  'read'  с  пользовательской  точки  зрения выглядят  как
15  системные  вызовы  'write'  и  'read'.   Для  примера,  SET_FOO  ioctl  будет
16  _IOW,  так  как  ядро  будет  читать  данные  из  пользовательского
17  пространства,  а GET_F00  ioctl  будет _IOR,  так  как ядро  будет
18  записывать  данные  в  пользовательское  пространство.
```

Из пользовательского пространства мы можем вызвать команду ioctl следующим образом:

```
ioctl(fd, MYDRIVER I0C 0P1, NULL);
ioctl(f d, MYDRIVER I0C 0P2, &mydata);
ioctl(fd, MYDRIVER I0C 0P3, mydata);
ioctl(fd, MYDRIVER_I0C_0P4, &mystruct);
```

Программе пользовательского пространства необходимо знать, какую ioctl команду (в нашем случае MYDRIVER_I0C_0P1 . . . MYDRIVER_I0C_0P4) и тип аргументов команды ожидать. Мы должны вернуть значение с помощью кода возврата системного вызова ioctl или мы можем интерпретировать параметр как указатель на набор для чтения. В последнем случае помните, что указатель связан с разделом в пользовательском пространстве памяти, которая должна быть скопирована из ядра или в ядро.

Простейшим способом перемещения памяти между пользовательским пространством и пространством ядра в функции ioctl является применение функций put_user () и get_user (), определенных следующим образом:

```
include/asm-i386/uaccess.h
* get_user: - Получение простых переменных из пользовательского
* пространства.
* @x:   переменная для сохранения результата.
* @ptr: Исходный адрес в пользовательском пространстве.

* put_user: - Запись простого значения в пользовательское пространство.
* @x:   Значение для копирования в пользовательское пространство.
* @ptr: Адрес назначения в пользовательском пространстве.
```

put_user () и get_user () проверяют, чтобы память пользовательского пространства была доступна для чтения или записи во время вызова.

Существует еще одно дополнительное ограничивающее условие, которое вы можете захотеть добавить к функциям вашего драйвера устройства: аутентификация.

Одним из способов проверки того, что процессу, который вызывает вашу функцию ioctl, разрешено это делать, является использование совместимостей. Обычно для авторизации используется CAP_SYS_ADMIN:

```
     include/linux/capability.h
20  / Позволяет настройку ключа сообщения безопасности */
20  / Позволяет администрировать устройство random */
20  / Позволяет увидеть и сконфигурировать квоты диска */
20  / Позволяет настроить syslog ядра (поведение printk */
20  / Позволяет установить domainname */
20  / Позволяет установить hostname */
20  / Позволяет вызвать bdflush0 */
20  / Позволяет установить mount () и umount0 для нового smb-соединения */
21  / Позволяет некоторые autofs root ioctls */
21  / Позволяет nfsservctl */
21  /        Позволяет VM86 REQUEST IRQ */ 213 /* Позволяет читать-
записывать pci config на alpha процессорах*/
```

```
214 /* Позволяет irix prctl Hamips (setstacksize) */
      215    /* Позволяет обрабатывать весь ktin на m68k
21 /         (sys cacheflush) */ Позволяет удаление семафоров */
21 /  Используется вместо CAP_CHOWN для "chown" очередей сообщений
21    IPC, семафоров и разделяемой памяти */ Позволяет блокирование-
21 /  разблокирование сегментов разделяемой
0  *
22 /   памяти */ Позволяет включать-выключать
0  *
22 /  свопинг */
22 /  Позволяет обмануть pids для переданного мандата сокета */
22 /  Позволяет установку буферов readahead и flushing для блочных
2  *
22 /   устройств */ Позволяет установку геометрии для драйвера флопи-
2  *
22 /  диска */ Позволяет настроить включение-выключение DMA on/off в
2  *
22 /  драйвере
xd *
22 /  Позволяет администрирование устройств md (обычно выше, но иногда
22    и дополнительные ioctls) */
22 /  Позволяет настраивать драйверы ide */
22 /  Позволяет получить доступ к устройству nvram */
22 /  Позволяет администрирование apm bios, последовательного
0  *
23 /  и bttv (TV) устройства*/
0  *
23 /  Позволяет использовать команды производителя в драйвере
1  *
23 /   поддержки isdn CAPI  */ Позволяет считывать
1  *
23 /   нестандартизированные порции пространства
2  *
23 /   настройки pci  */ Позволяет отладку DDI debug ioctl для
2  *
233 /  драйвера sbpcd */
234 /* Позволяет настраивать последовательные порты */
235 /* Позволяет посылать сырые команды qic-117 */
235 /* Позволяет включить/выключить теговые запросы для контроллера
       SCSI и посылки 23 6    специальных SCSI-команд*/ 237 /*
Позволяет настроить ключ раскодирования для циклической файловой
       системы */ 238 239
#define CAP_SYS_ADMIN 21
```

Многие другие совместимости из include/linux/compatibility.h могут лучше подходить для вашего драйвера, но CAP_SYS_AMIN вмещает их все.

Для проверки совместимости вызываемого процесса с вашим драйвером вы можете добавить нечто наподобие следующего кода:

```
if (! capable(CAP_SYS_ADMIN)) {
 return -EPERM;
```

10.2.4 Организация пула и прерывания

Когда драйвер устройства посылает команды в управляемое устройство, существует два способа определения успешности выполнения команды: можно организовать пул устройства или использовать прерывания устройства.

При организации пула устройства драйвер устройства периодически проверяет устройства для проверки того, что команда успешно доставлена. Из-за того что драйвер устройства является частью ядра, при прямой организации пула присутствует риск заставить ядро ожидать завершения операции пула. Пул драйвера устройства может быть организован через таймер. Когда драйвер устройства хочет заполнить пул устройства, он помещает в планировщик вызов функции из драйвера устройства. Эта функция выполняет проверку устройства без остановки ядра.

Перед тем как углубиться в подробности работы прерываний ядра, мы должны объяснить основные методы блокировки доступа к критическим секциями в коде ядра - циклическим блокировкам. Циклические блокировки работают с помощью установки специального флага для некоторых значений перед вхождением в код критической секции и сбрасывания этого значения после оставления кода критической секции. Циклическая блокировка должна применяться, когда контекст задачи не блокируется, что и происходит в коде ядра. Давайте рассмотрим код циклической блокировки для архитектур x86 и PPC.

```
include/asm-i386/spinlock.h
32 #define SPIN_LOCK_UNLOCKED (spinlock_t) { 1 SPINLOCK_MAGIC_INIT }
33
34 #define spin_lock_init(x) do { *(x) = SPIN_LOCK_UNLOCKED; } while(O)

43 #define spin_is_locked(x) (*(volatile signed char *)(&(x)->lock) <= 0)
44 #define spin_unlock_wait(x) do { barrier(); }
                                    while(spin_is_locked(x))
```

```
include/asm-ppc/spinlock.h
25 idefine SPIN_LOCK_UNLOCKED (spinlock_t) { 0 SPINLOCK_DEBUG_INIT }
26
27 #define spin lock init (x) do { * (x) = SPIN LOCK UNLOCKED; } while(O)
28 #define spin_is_locked(x) ((x)->lock != 0)
                             while(spin is locked(x))
29 tdefine spin_unlock_wait(x) do { barrier(); }
                                 while(spin_is_locked(x))
```

На архитектуре x86 флаг циклической блокировки устанавливается в 1 если она снята, а на PPC в 0. Отсюда видно, что при написании драйверов вам нужно использовать

предоставляемый макрос вместо сырых значений для кроссплатформенной совместимости.

Задачи, которые хотят получить блокировку, в пустом цикле последовательно проверяют значение специального флага до тех пор, пока оно не станет меньше 0, т. е. выполняется циклическое ожидание в задаче. См. sipn__unlock_wait () в двух блоках кода.

Циклические блокировки для драйверов обычно используются при обработке прерываний, когда коду ядра необходимо выполнить критическую секцию без прерывания другими прерываниями. В предыдущих версиях ядра Linux использовались функции cli () и sti () для отключения и включения прерываний. Как и в 2.5.28, cli () и sti() поэтапно заменены циклическими блокировками. Новый способ выполнения разделов кода ядра, которые не могут быть прерваны - следующий:

```
  Documentation/cli-sti-removal.txt
1 spinlock_t driver_lock = SPIN_LOCK_UNLOCKED;
2 struct driver_data;
3
4 irq handler (...) {
5 unsigned long flags;
6
7 spin_lock_irqsave(&driver_lock, flags);
8
9 driver data.finish = 1;
10 driver_data.new_work = 0;
11
12 spin__unlock_irqrestore (&driver__lock, flags)
13
14
15
16
17
18 ioctl_func  (...) {
19
20 spin_lock_irq(&driver_lock);
21
22 driver_data.finish = 0;
23 driver_data.new_work = 2;
24
25 spin__unlock_irq(&driver_lock) ;
26
27
28
29
```

Строка 8

Перед началом кода критической секции прерывание сохраняется во флаг и выполняется блокировка driver_lock.

Строки 9-12

Этот код критической секции может выполняться только одной задачей одновременно.

Строка 27

Эта строка заканчивает код критической секции. Восстанавливается состояние прерываний и разблокируется driver_lock.

С помощью spin_lock_irq_save () [и spin_lock_irq_reatore ()] мы проверяем, что прерывание отключено перед запуском обработчика прерывания и осталось отключенным после его завершения.

Когда ioctl__f unc () блокирует driver_lock, могут запускаться другие вызовы ircj__handler (). Поэтому нам нужно убедиться, что критическая секция в ioctl_ f unc () завершается как можно быстрее, для того чтобы гарантировать минимальное время ожидания irq_handler (), обработка прерывания верхнего уровня.

Давайте рассмотрим последовательность создания обработчика прерывания и его высокоуровневого обработчика (см. нижнюю половину, использующую очередь выполнения в подразд. 10.2.5)

```
#define mod num tries 3
static int irq = 0;

int count = 0;
unsigned int irqs = 0;
while ((count < mod num tries) && (irq <= 0) ) {
 irqs = probe irq on(); /* Заставляет устройство запустить
прерывание. Некоторая задержка, требуемая на подтверждение получения
прерывания */
  irq = probe_irq_off(irqs);
  /* If irq < 0 получение множественных прерываний.
     If irq == 0 нет полученных прерываний. */
  count++; } if ((count == mod_num__tries) &&
(irq <=0) ) {
printk(*Couldn't determine interrupt for %s\n",
MODULE_NAME); }
```

Этот код будет частью раздела инициализации драйвера устройства и завершится неудачно, если прерывания не будут обнаружены. Теперь, когда у нас есть прерывание, мы можем зарегистрировать это прерывание и наш высокоуровневый обработчик прерывания в ядре:

```
retval = request_irq(irq, irq_handler, SA__INTERRUPT,
     DEVICE NAME, NULL);
if (retval < 0) {
  printk(*Request of IRQ %n failed for %s\n",
   irq, MODULE NAME);
return retval; }
```

request_irq () имеет следующий прототип:

```
arch/ i38б/kernel/irq.c
590 /**
591  *  request  irq - выделение строки прерывания
592  *  @irq: строка прерывания для выделения
593  *  ©handler: Функция, вызываемая при наступлении IRQ
594  *  @irqflags: тип флага прерывания
595  *  ©devname: ascii имя для обрабатываемого устройства
596  *  @dev_id: cookie передаваемое в функцию-обработчик

622 int request  irq(unsigned int irq,
623     irqreturn t (*handler) (int, void *, struct pt regs *) ,
624     unsigned long irqflags,
625     const char * devname,
626     void *dev_id)
```

Параметр irqflags может иметь значение ord следующего макроса:

- SAJSHIRQ для разделяемого прерывания;
- SA__INTERRUPT для отключения локальных прерываний во время выполнения handler;
- SA_SAMPLE_RANDOM если прерывание является источником энтропии.

dev_id должно равняться NULL, если прерывание неразделяемое, и, если оно все-таки разделяемое, обычно адресует структуру данных устройства, так как это значение получает handler.

В этом месте полезно вспомнить, что любое запрашиваемое прерывание нужно освободить при выходе из модуля с помощью f ree_irq ():

```
arch/ i386/kernel/irq.c
669 /**
67 0  *   free irq - освобождение прерывания
671  *   @irq: линия прерывания для освобождения
672  *   @dev id: идентификатор устройства для освобождения

682  */
683
684 void  free_irq(unsigned  int  irq,   void  *dev_id)
```

Если dev_id разделяет irq, модуль должен быть уверен, что прерывания выключены перед вызовом этой функции. Кроме того, f ree_irq () никогда нельзя вызывать из контекста прерывания. Вызов f ree_irq () в модуле - стандартный способ очистки. [См. spin_lock_irq () и spin__unlock_irq ().]

В данной точке мы зарегистрировали наш обработчик прерывания и связали его с irq. Теперь нам нужно написать настоящий обработчик верхнего уровня, определяемый как irq_handler ().

```
void  irq_handler(int  irq,   void  *dev_id,   struct pt_regs   *regs) {
   /*  См.   выше   код циклической  блокировки  */
   /*  Копирование  данных прерывания  в  очередь  выполнения для
        обработчика  нижней половины  */
   schedule_work(   WORK_QUEUE   );
   /*  Освобождение  spin_lock  */ }
```

Если вам нужен быстрый обработчик прерывания, вы можете использовать тасклет вместо очереди выполнения:

```
}
```

10.2.5 Очереди выполнения и тасклеты

Основную работу в обработчике прерываний обычно выполняет очередь выполнения. В предыдущем подразделе мы рассмотрели, как обработчики прерываний верхнего уровня копируют нужные им данные из прерывания в структуры данных и затем вызывают schedule_work ().

Для запуска задачи из очереди выполнения она должна быть помещена в work__struct. Для объявления структуры выполнения во время компиляции используется макрос DECLARE_WORK (). Например, следующий код помещает в наш модуль инициализацию структуры выполнения и связывает ее с функцией и данными.

```
struct bh data struct {
  int data one; int
  *data_array; char
  *data_text;
}

static bh__data_struct bh_data;

static DECLARE_WORK(my_mod_work, my_mod_bh, &bh_data);

static void my_mod_bh(void *data)
{
  struct bh_data_struct *bh_data = data;
  /* весь замечательный код нижней половины */
}
```

Обработчик верхней половины настраивает все требуемые данные через my_mod_bh в bh_data и затем вызывает schedule_work(my_niod_work).

schedule_work () - это функция, доступная для любого модуля; тем не менее это значит, что работающий планировщик помещает «события» в общую очередь выполнения. Некоторые модули могут желать обладать собственными очередями выполнения, но требуемые для этого функции экспортируются только для GPL-совместимых модулей. Поэтому, если вы хотите оставить ваш модуль проприетарным, вы должны использовать общую очередь выполнения.

Очередь выполнения создается с помощью макроса create__workqueue (), ко торый вызывает __create_workqueue () со вторым параметром, равным 0.

```
kernel/workqueue.c
3 04 struct workqueue_struct * _ create_workqueue(const char *name,
305
```

name может иметь длину до 10 символов.

Если single thread равно 0, ядро создает поток workqueue для процессора; если single thread равен 1, ядро создает единственный поток single thread для всей системы.

Структуры выполнения создаются тем же способом, который был описан ранее, но они помещаются в отдельную очередь выполнения с помощью queue_work () вместо schedule_work ().

```
kernel/workqueue.c
97 int fastcall queue work(struct workqueue struct *wq, struct
work_struct *work)
98 {
```

wq - это специальная очередь выполнения, созданная с помощью create_workqueue ().

work - это структура выполнения, помещаемая в wq.

Другие функции очереди выполнения можно найти в kernel /workqueue. c, включая следующее:

- queue_work_delayed(). Проверяет, что функция структуры выполнения вызывается по истечении указанного количества мгновений.
- flush_workqueue(). Заставляет вызывающего подождать до тех пор, пока завершится работа планировщика с очередью. Обычно используется при выходе из драйвера устройства.
- destroy_workqueue(). Обрабатывает и затем освобождает очередь выполнения.

Похожие функции schedule_work_delayed() и flush_sheduled_work() существуют и для общей очереди выполнения.

10.2.6 Дополнение кода для системного вызова

Мы можем изменить Makefile в /kernel для включения файла в нашу функцию, но проще будет включить код функции в уже существующий файл в дереве исходников. Файл /kernel/sys . c содержит функции ядра для системных вызовов, а файл arch/ i386/kernel/sys_i3 86 .c содержит системные вызовы x86 с нестандартной последовательностью вызова. Мы добавляем туда исходный код для нашего системного вызова, написанный на C. Этот код запускается в режиме ядра и выполняет всю работу. Все остальное в этой процедуре помогает нам получить эту функцию. Она обрабатывается через обработчик исключения x86:

```
kernel/sys.c 1:
  . .
2 /* где-то после последней функции*/
3
4 /* простая функция для демонстрации системного вызова.    */
5 /* получение в номер , вывод, возвращение number+1 */
б
7 asmlinkage long sys ourcall(long num) {
8 printk("Inside our syscall num =%d \n", num);
9   return(num+1); }
10
11
```

Когда обработчик исключения выполняет int 0x80, он индексируется внутри таблицы системного вызова. Файл /arch/i3 8 б /kernel /entry. S содержит функции обработчики прерывания нижнего уровня и таблицу системного вызова sys_ call_table. Таблица - это реализация в ассемблерном коде массива С с элементами длиной 4 байта. Каждый элемент или вхождение в этой таблице инициализируется для адресации функции. По соглашению мы должны приготовить имя нашей функции в sys__. Из-за того, что позиция в таблице определяется номером системного вызова, мы должны добавить имя нашей функции в конец списка. См. следующий код для изменения таблицы:

```
arch/i38б/kernel/entry.S
   : .data 608:
ENTRY(sys call table)
    .long sys_restart_syscall    /*  0  - old  "setupO"  системный  вызов
используется для перезапуска  */

    .long  sys_tgkill                /*  270  */
    .long sys_utimes .1ong
    sys_fadvi s e б 4_6 4
    .long sys_ni_syscall      /*  sys_vserver  */
    .long  sys_ourcall               /*  our  syscall will  be  274  */
884:  nr_syscalls=(.-sys_call_table)/4
```

Файл include/asiri/unistd.h связывает системные вызовы с их номерами позиции в sys_call_table. Также в этом файле находятся макросы для помощи пользовательским программам (написанным на С) загружать параметры в регистры. Здесь мы изменяем unistd.h и вставляем наш системный вызов:

```
  include/asm/unistd.h
 1 /*
 2 *  Этот  файл  содержит  номерасистемных  вызовов.
 3 */
 4
 5 tdefine __________ NR_restart_syscall          0
 6 #define __________ NR_exit                    1
 7 tdefine __________ NR_fork                    2
 8
 9 #define __________ NR_utimes                 271
10 #define __________ NR_fadvise64_64           272
11 #define __________ NR_vserver                273
12 #define __________ NR_ourcall                274
13 /*    #define NR_syscalls  274    это  старое  значение  перед нашим
14 системным вызовом    */
15:  #define NR_syscalls                    275
```

Наконец, мы хотим создать пользовательскую программу для проверки нового системного вызова. Как говорилось ранее в этом разделе, существует набор макросов, помогающих программисту ядра загружать параметры из кода С в регистры x86. В /usr/include/asm/unistd.h существует 7 макросов: _sy sea llx (type, name, . .),гдех - номер параметра. Каждый макрос предназначен для загрузки определенного количества параметров от 0 до 5, а syscall б (. . .) позволяет загрузить указатель на большее число параметров. Следующая демонстрационная программа получает один параметр. Для этого примера (в строке 5) мы используем макрос _syscall (type, name, typel, namel) из /unistd.h, преобразующийся в вызов int 0x80 с правильными параметрами:

```
  mytest.c
1 #include <stdio.h>
2 tinclude <stdlib.h>
3 #include  Vusr/include/asm/unistd.h"
4
5 _syscall(long,ourcall,long, num);
б
7 main() {
8   printf(*our syscall --> num in=5, num out = %d\n", ourcall(5)); }
9
10
```

10.3 Сборка и отладка

Добавление вашего кода в ядро обычно требует нескольких циклов программирования и отладки. В этом разделе мы опишем, как можно отладить написанный вами код ядра и как собрать связанные с отладкой инструменты.

10.3.1 Отладка драйвера устройства

В предыдущих разделах мы использовали файловую систему /proc для получения информации об ядре. Мы можем сделать доступной информацию о нашем драйвере устройства для пользователя через /proc, что является отличным способом отладки части вашего драйвера устройства. Каждый узел в файловой системе /proc связан с функцией ядра при чтении и записи. В ядре 2.6 большинство записей в части ядра, включая устройства, выполняется через sysf s вместо /proc. Операции модифицируют специальные атрибуты объектов ядра во время выполнения ядра; /proc остается полезным инструментом для операций только чтения, требующих большего количества данных, чем пара атрибут-значение, а этот раздел работает только с чтением из вхождений /proc.

Первый шаг позволяет получить доступ к чтению с вашего устройства с помощью создания вхождения в файловой системе /proc с помощью create_proc_read_entry ():

```
include/linux/proc fs.h
146 static inline struct proc_dir_entry *create_proc_read_entry(const
char *name,
147   mode t mode, struct proc dir entry *base,
148   read_proc_t *read_proc, void * data)
```

*name - это вхождение узла, появляющегося в /proc, mode 0 позволяет файлу быть читаемым отовсюду. Если вы создаете несколько различных файлов proc для одного драйвера устройства, сначала стоит создать директорию proc с помощью proc_mkdir () и затем помещать каждый файл в нее; *base - это путь директории внутри /proc для размещения файлов; значение NULL помещает файл прямо в /proc. Функция *read_proc вызывается при чтении файла, а указатель *data передается обратное *read_proc:

```
include/linux/proc fs.h
44 typedef int (read_proc_t)(char *page, char **start, off_t off,
45    int count, int *eof, void *data);
```

Это прототип для функций, которые будут осуществлять чтение через файловую систему /proc; *page - это указатель на буфер, куда функции записывают свои данные во

время чтения файла /proc. Функция должна начать с записи по байту of f в *page и записи остальных count байт. Так как обычно чтение возвращает только небольшое количество информации, многие реализации игнорируют как off, так и count. Дополнительно **start обычно игнорируется и редко используется в ядре. Если вы реализуете функцию чтения, возвращающую определенное количество данных, то **start, off и count можно использовать для управления чтением небольших порций за определенное время. Когда чтение закончено, функция должна записывать 1 в * eof. Наконец, *data является параметром, передаваемым в функцию чтения, определенную в create^ proc__read_entry ().

Резюме

Эта глава описывает драйвер устройства, модули и системные вызовы. Мы описали несколько способов, которыми Linux использует драйверы устройств. Точнее говоря, мы рассмотрели следующие темы:

- в Мы описали дерево /dev в файловой системе Linux и объяснили, как определить, какое устройство контролируется и с помощью какого драйвера устройства.
- Мы объяснили, как драйверы устройств используют структуры файлов и структуры файловых операций для обработки ввода-вывода файловой системы.
- Мы описали разницу между памятью пользовательского уровня и памятью пространства ядра и то, как драйвер устройства может копировать данные из одной области в другую.
- Мы рассмотрели конструкцию очереди ожидания в ядре и показали, как она используется, когда драйверу устройства нужно подождать освобождения определенного ресурса.
- Мы исследовали теорию, стоящую за очередью ожидания и прерываниями, являющимися методами, используемыми ядром Linux для очистки прерываний и обработки драйверов устройств, когда процессор нужно передать другому процессу.
- Мы представили вашему вниманию системные вызовы Linux и описали их базовые функции.
- Мы описали разницу между блочными и символьными устройствами и новую модель устройств, представленную в Linux 2.6. Также сюда включен короткий обзор sysfs.

В первой части гл. 10 об этих темах говорилось на абстрактном уровне, и мы проследили их практическое применение на примере специального драйвера устройства /dev/random. Вторая часть гл. 10 предоставляет более конкретные примеры и код для непосредственного создания драйвера устройства.

Точнее говоря, мы рассмотрели детали следующих концепций:

- Мы показали, как мы конструируем узлы в /dev, которые нужно связать с драйверами устройств, и как создаются динамические модули.
- Мы описали новые методы в Linux 2.6 для экспорта символов из модуля драйвера устройства.
- Мы продемонстрировали, как драйвер устройства предоставляет функцию IOCTL, позволяющую устройству работать с Linux через файловую систему.
- Мы объяснили, как возникают прерывания и помещение в пул, а также разницу между циклическими блокировками на архитектурах x86 и PPC.
- Мы объяснили, как добавить простой системный вызов в ядро Linux.

Гл. 10 представляет мощную основу для разработки драйверов устройств в 2.6 и объединяет в единое целое идеи и концепции, представленные ранее в этой книге.

Упражнения

1. См. создание ядра и пользовательского кода в гл. 3, «Процессы: принципиальная модель выполнения». Перекомпилируйте ядро и скомпилируйте my test. c. Запустите my test. c и рассмотрите вывод.
2. Добавьте другой параметр в ourcall.
3. Создайте системный вызов из ourcall.
4. Объясните сходства и различия между системными вызовами и драйверами устройств.
5. Почему мы не можем использовать memcpy для копирования данных между пользовательским пространством и пространством ядра?
6. В чем разница между функциями верхней половины и нижней половины?
7. В чем разница между тасклетами и work_queue?
8. Когда устройство обрабатывает больше чем просто чтение и запись, как с ним общается Linux?
9. Чему равно численное значение снятой циклической блокировки на архитектуре x86?HaPPC?
10. Одним предложением объясните разницу между блочными и символьными устройствами.