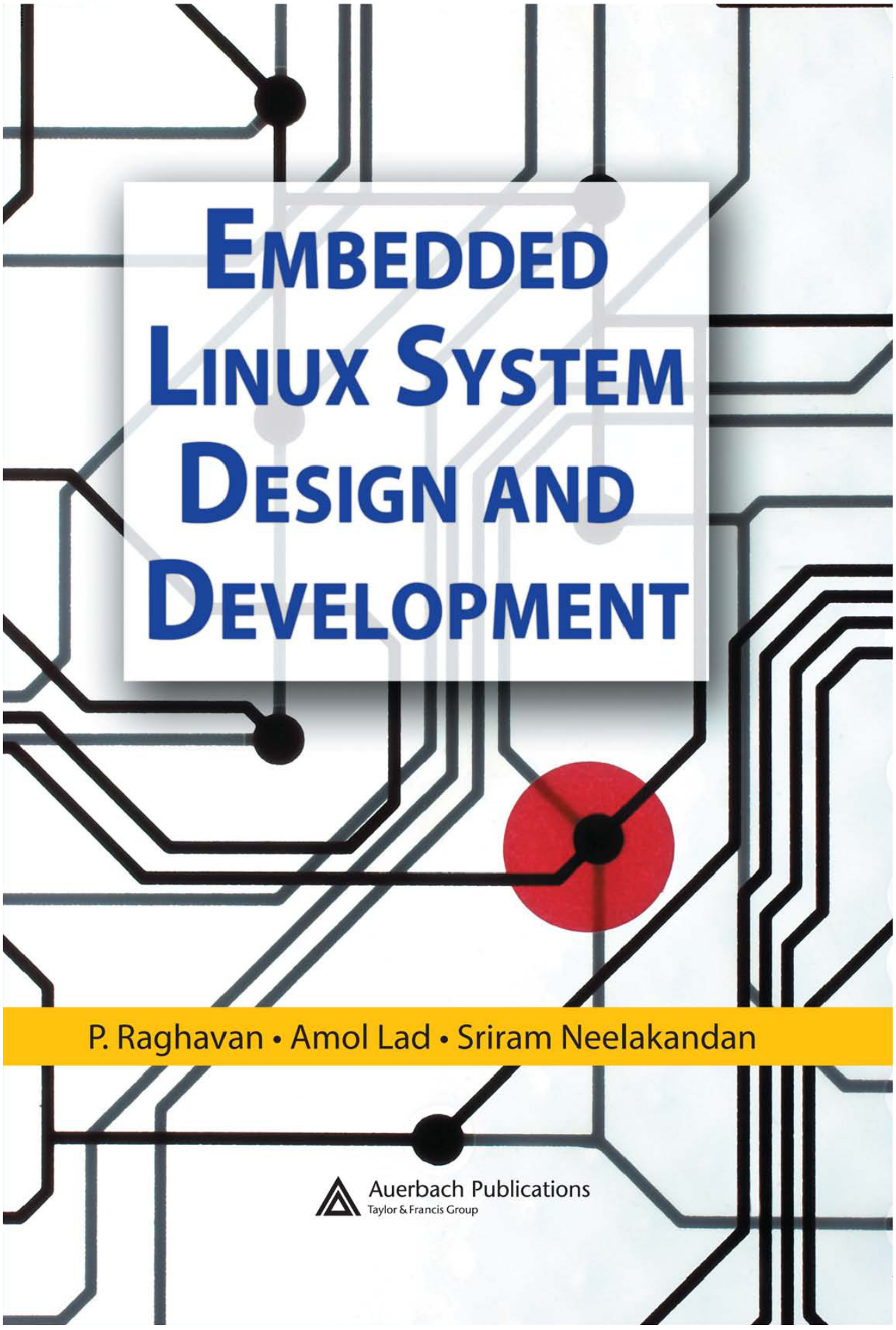
EMBEDDED
LINUX SYSTEM
DESIGN AND
DEVELOPMENT
P. Raghavan • Amol Lad • Sriram Neelakandan
Auerbach Publications
Taylor & Francis Group

Перевод

Разработка и внедрение системы на встраиваемом Linux

V*D*V

Оглавление

Глава 2, Начало работы

Эта глава делится на три части. Первая часть описывает разные архитектуры встраиваемых ОС и сравнивает их с архитектурой Linux. Затем дан краткий обзор ядра и пользовательского пространства Linux. Вторая часть главы объясняет последовательность запуска Linux. Последняя часть рассказывает об инструментах для кросс-разработки.

2.1 Архитектура встраиваемого Linux

ОС Linux является монолитной. Обычно операционные системы поставляются в трёх вариантах: в режиме исполнения реального времени, монолитные и микроядра. В основе такой классификации лежит то, как ОС использует аппаратные средства для защиты.

2.1.1 Выполнение в режиме реального времени

Традиционные ОС исполнения в режиме реального времени предназначены для процессоров без MMU. На этих операционных системах всё адресное пространство плоское или линейное, без защиты памяти между ядром и приложениями, как показано на Рисунке 2.1.

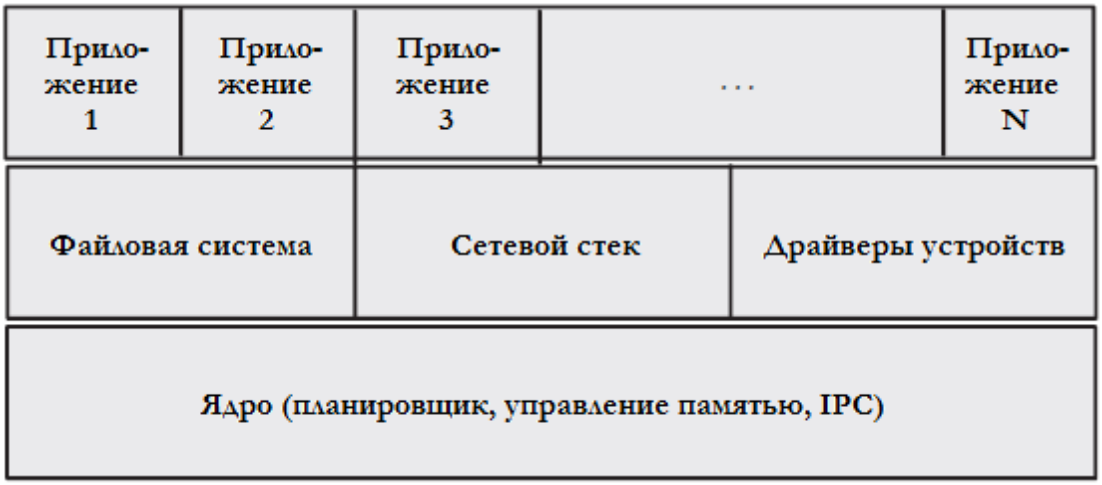

Рисунок 2.1 Архитектура традиционной RTOS.

На Рисунке 2.1 показана архитектура выполнения реального времени, где основное ядро, подсистемы ядра и приложения совместно используют одно и то же адресное пространство. Эти операционные системы используют немного памяти и имеют небольшой размер, так как ОС и приложения сгруппированы в единый образ. Как следует из названия, они по своей природе работают в режиме реального времени, потому что нет накладных расходов системных вызовов, передачи сообщений или копирования данных. Однако, поскольку ОС не обеспечивает защиты, всё программное обеспечение, работающее в системе, должно быть надёжным. Добавление нового программного обеспечения становится не очень приятным действием, потому что оно должно быть тщательно протестировано, чтобы не обрушить всю систему. Кроме того, очень трудно добавлять приложения или модули ядра динамически, так как система должна быть выключена. Под эту категорию подпадают большинство патентованных и коммерческих RTOS.

В течение последнего десятилетия встраиваемые системы были свидетелями сдвига

парадигмы по отношению к архитектуре.Традиционная модель встраиваемой системы основана на жёстко контролируемом программном обеспечении, работающем на платах устройств; стоимость памяти и размер запоминающего устройства больше ограничивались количеством программ, которые могли быть запущены в данной системе. Надёжность работы в режиме реального времени, использующей плоскую модель памяти, достигалась за счёт процесса тщательного тестирования. Однако, так как цены на оперативную и флеш память упали, а вычислительная мощность стала дешевле, встраиваемые системы начали иметь всё больше и больше программного обеспечения. И многое из этого программного обеспечения было не только системным программным обеспечением (таким как драйверы или сетевой стек), но и приложениями. Так программное обеспечение тоже начало становиться фактором, влияющим на продажи встроенных систем, которые традиционно оценивались главным образом по аппаратным возможностям. Работа в режиме реального времени не подходит для крупномасштабных интеграций программного обеспечения; поэтому были серьёзно рассмотрены альтернативные модели с целью получения большего количества программного обеспечения, работающего в системе. Двумя такими моделями для операционных систем являются монолитные и микроядерные модели. Это подходит для процессоров с MMU. Заметим, что если сам процессор не имеет MMU, то ОС не имеет альтернативы, кроме как обеспечить плоскую модель адресации. (uClinux, происходящий от Linux, работает на процессорах без MMU и обеспечивает плоское адресное пространство.)

2.1.2 Монолитные ядра

Монолитные ядра имеют разграничение между пространством пользователя и пространством ядра. Когда программное обеспечение работает в пространстве пользователя, оно обычно не может получить доступ к системному оборудованию и не может выполнять привилегированные инструкции. Используя специальные точки входа (предоставляемые оборудованием), приложение может войти из пользовательского пространства в режим ядра. Программы пространства пользователя работают с виртуальными адресами, так что они не могут повредить память другого приложения или ядра. Тем не менее, компоненты ядра используют общее адресное пространство, так что плохо написанный драйвер или модуль может привести к падению системы.

На Рисунке 2.2 показана архитектура монолитных ядер, где ядро и его подмодули используют общее адресное пространство, и где приложения имеют свои обособленные адресные пространства.

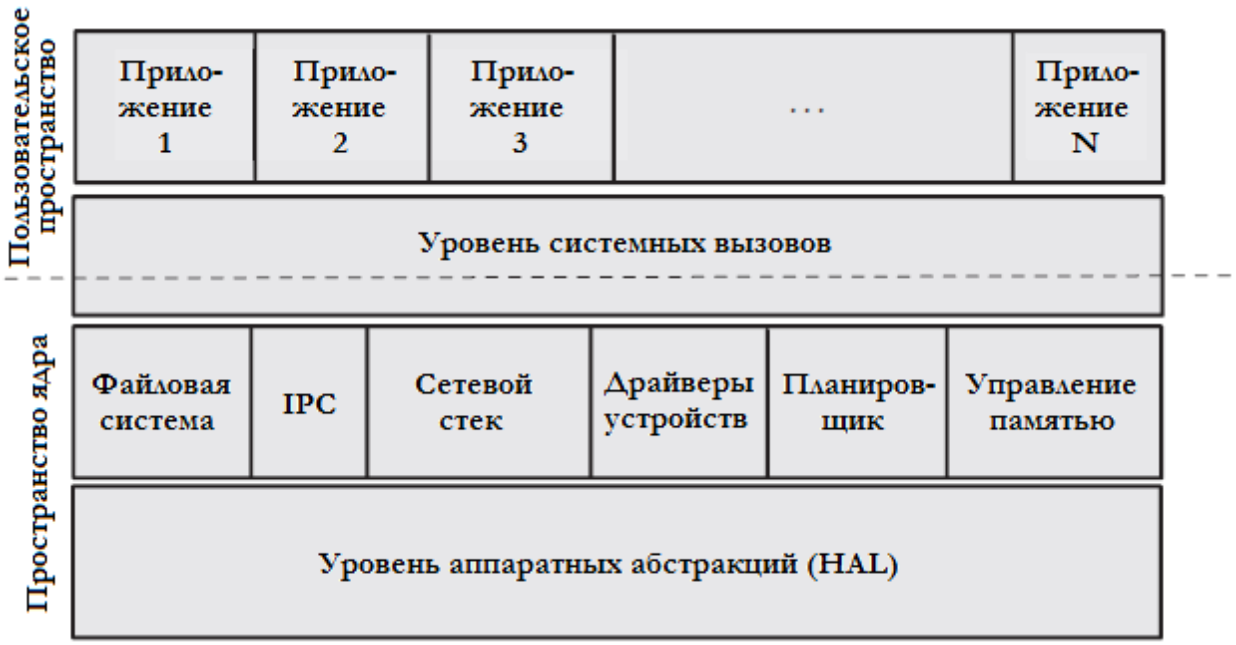

Рисунок 2.2 Архитектура монолитного ядра.

Монолитные ядра могут поддерживать большой набор программного обеспечения. Любая ошибка в приложении вызовет только то, что это приложение будет плохо себя вести, не вызывая краха системы. Кроме того, приложения могут быть добавлены в работающую систему без выключения этой системы. Большинство операционных систем UNIX являются монолитными.

2.1.3 Микроядра

Такие ядра подвергались многочисленным исследованиям, особенно в конце 1980-х, и считались наилучшими по отношению к принципам разработки ОС. Тем не менее, переход от теории к практике привёл к слишком большому числу узких мест; очень немногие из таких ядер были успешными на рынке. Микроядро использует небольшую ОС, которая предоставляет только основной сервис (планирование, обработка прерываний, передача сообщений), а остальная часть ядра (файловая система, драйверы устройств, сетевой стек) работает в качестве приложений. По отношению к использованию MMU, ядра реального времени формируют одну крайность, работая без использования MMU, в то время как микроядра находятся на другом конце, предоставляя подсистемам ядра индивидуальное адресное пространство. Ключевым принципом микроядра является придумать чётко определённые интерфейсы для взаимодействия с ОС, а также надёжные схемы передачи сообщений.

Рисунок 2.3 показывает архитектуру микроядра, где подсистемы ядра, такие как сетевой стек и файловые системы, имеют отдельное адресное пространство, аналогично приложениям. Микроядра требуют надёжных схем передачи сообщений. Работа в режиме реального времени и модульность обеспечиваются только в случае передачи сообщений должным образом. Микроядра активно обсуждались, особенно по отношению к монолитным ядрам. Одна из таких широко известных дискуссий была между создателем Linux, Линусом Торвальдсом и Эндрю Танненбаумом, который был создателем Minix ОС (микроядро). Дискуссия, возможно, не представляет очень большого интереса для читателя, который хочет разобраться во встраиваемом Linux.

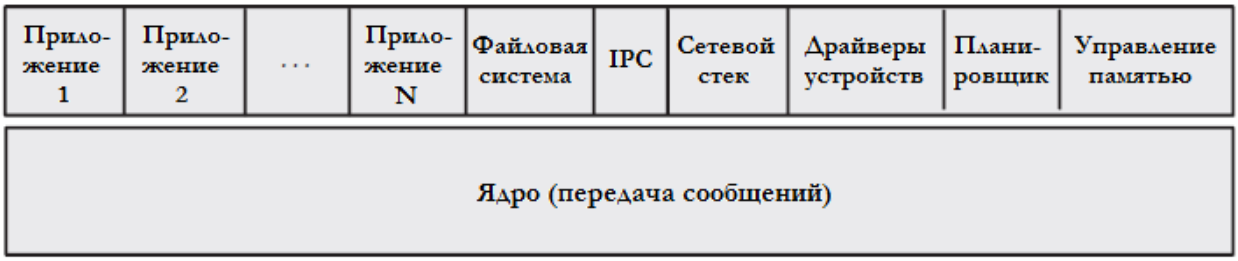

Рисунок 2.3 Архитектура микроядра.

Как видно, эти три типа ОС основаны на совершенно разных философиях. На одном конце спектра находятся ядра реального времени, которые не предоставляют защиты памяти; это сделано для того, чтобы система была более быстрой, но за счёт надежности. На другом конце микроядро, обеспечивающее защиту памяти для отдельных подсистем ядра за счёт сложности. Linux - это средний путь монолитных ядер, где всё ядро целиком работает в едином пространстве памяти. Является ли единое пространство памяти для ядра проблемой? Чтобы убедиться в том, что внедрение нового программное обеспечение ядра не вызывает никаких проблем с надёжностью, любое добавление проходит через большую проверку с точки зрения функциональности, дизайна и производительности прежде, чем оно принимается в основную ветку ядра. Этот процесс экспертизы, который каждый раз может быть очень утомительным, сделал ядро Linux одним из самых стабильных частей программного обеспечения. Это позволило ядру быть использованным в целом ряде разнообразных систем, таких как настольные компьютеры, КПК, и большие серверы.

С введением динамически загружаемых модулей ядра были некоторые сомнения в отношении монолитной архитектуры Linux. Динамически загружаемые модули ядра являются кусками кода ядра, которые не скомпонованы (не включены) непосредственно в ядро. Они компилируются отдельно и их можно вставить и удалить из работающего ядра почти в любое время. Загружаемые модули ядра хранятся отдельно и загружаются в память только при необходимости, что позволяет экономить память. Следует также отметить, что повышение модульности ядра не делает его менее монолитным, поскольку ядро взаимодействует с драйверами используя вместо передачи сообщений прямые вызовы функций.

Следующие два раздела представляют высокоуровневый обзор ядра Linux и пользовательского пространства.

2.2 Архитектура ядра Linux

Хотя версии ядра Linux меняются, базовая архитектура ядра Linux остаётся более или менее неизменной. Ядро Linux можно разделить на следующие подсистемы:

- Уровень аппаратных абстракций
- Диспетчер памяти
- Планировщик
- Файловая система
- Подсистема ввода-вывода
- Сетевая подсистема
- Межпроцессное взаимодействие

Кратко рассмотрим каждую подсистему и детали её использования во встроенных системах.

2.2.1 Уровень аппаратных абстракций (HAL)

Уровень аппаратных абстракций (hardware abstraction layer, HAL) виртуализирует оборудование платформы, так что различные драйверы могут быть легко перенесены на любое оборудование. HAL эквивалентен BSP (board support package, пакет поддержки аппаратуры), предоставляемому на большинстве RTOS, за исключением того, что BSP на коммерческих RTOS обычно имеет стандартные API, которые позволяют легко его переносить. Почему HAL Linux не имеют стандартных интерфейсов для подключения к остальной части ядра? Это наследство; поскольку Linux изначально предназначался для настольных систем x86 и поддержка других платформ был добавлена позже, первые разработчики не думали о стандартизации HAL. Однако, в последних версиях ядра идея придумать стандартные интерфейсы для подключения зависящего от платы программного обеспечения перенимается. Две известные архитектуры, ARM и PowerPC, имеют хорошо описанные структуры данных и API, которые упрощают перенос на новую плату.

Ниже приведены некоторые встраиваемые процессоры (кроме x86), поддерживаемые ядром Linux 2.6.

- MIPS
- PowerPC
- ARM
- M68K
- CRIS
- V850
- SuperH

HAL имеет поддержку следующих аппаратных компонентов:

- Процессор, кэш и MMU
- Создание карты памяти
- Поддержку обработки исключений и прерываний
- DMA
- Таймеры
- Системная консоль
- Управление шиной
- Управление питанием

Функции, которые инициализируют платформы, более подробно объясняются в Разделе 2.4 13. Глава 3 29 подробно объясняет шаги по переносу Linux на платформы на базе MIPS.

2.2.2 Диспетчер памяти

Диспетчер памяти Linux отвечает за управление доступом к аппаратным ресурсам памяти. Диспетчер памяти несёт ответственность за предоставление динамической памяти подсистемам ядра, таким как драйверы, файловые системы и сетевой стек. Он также реализует программное обеспечение, необходимое, чтобы обеспечить виртуальной памятью пользовательские приложения. Каждый процесс в подсистеме Linux работает в своём отдельном адресном пространстве, называемом виртуальным адресом. Работая с виртуальным адресом, процесс не может повредить ни другой процесс, ни память операционной системы. Любой повреждённый указатель внутри процесса локализован в

процессе, не обрушивая систему; это очень важно для надёжности системы.

Ядро Linux делит общий доступный объём памяти на страницы. Типичный размер страницы равен 4 Кб. Хотя ядру доступны все страницы, только некоторые из них используются ядром; остальные используются приложениями. Обратите внимание, что страницы, используемые ядром, не являются частью процесса подкачки; по требованию в основную память загружаются только страницы приложений. Это упрощает разработку ядра. Когда приложение должно быть выполнено, нет необходимости быть загруженным в память всему приложению; между памятью и хранилищем переключаются только используемые страницы.

Наличие отдельной памяти пользователя и ядра является самым радикальным изменением, которое разработчик может ожидать при переходе от собственной RTOS. Во первых, все приложения являются частью одного и того же образа, содержащего и эту ОС. Таким образом, когда этот образ загружается, приложения также копируются в память. В Linux, однако, операционная система и приложения скомпилированы и собраны раздельно; каждое приложение имеет свой собственный экземпляр в хранилище, часто называемый программой.

2.2.3 Планировщик

Планировщик Linux обеспечивает многозадачность и развивается от версии к версии ядра с целью обеспечения детерминированной политики планирования. Прежде чем углубляться в историю улучшений планировщика, давайте разберёмся в примерах выполнения, которые понятны планировщику.

- ***Потоки ядра***: это процессы, которые не имеют пользовательского контекста. Они выполняются в пространстве ядра до тех пор, пока работают.
- ***Пользовательский процесс***: благодаря виртуальной памяти, каждый пользовательский процесс имеет своё собственное адресное пространство. Они входят в режим ядра, когда выполняется прерывание, исключение или системный вызов. Обратите внимание, что когда процесс входит в режим ядра, он использует совершенно другой стек. Он называется стеком ядра и каждый процесс имеет свой собственный стек ядра.
- ***Пользовательские потоки***: потоки представляют собой различные исполняющиеся объекты, которые сопоставляются с одним пользовательским процессом. Потоки пользовательского пространства имеют общее пространство текста, данных и "кучи". Они имеют отдельные адреса стека. Другие ресурсы, такие как открытые файлы и обработчики сигналов, также общие для всех потоков.

Поскольку Linux стал популярен, спрос на поддержку приложений реального времени увеличился. В результате планировщик Linux постоянно улучшался, так что его политика планирования стала детерминированной. Ниже приведены некоторые из важных вех в эволюции ядра Linux в отношении возможностей реального времени.

- Начнём с ядра версии 1.3.55, где была поддержка кругового планирования и планирования на основе FIFO наряду с классическим планировщиком Linux с разделением времени. Также оно имело возможность отключения подкачки для отдельных областей памяти приложения; это называется блокировкой памяти (поскольку запрос подкачки делает систему недетерминированной).
- Ядро версии 2.0 предоставило новую функцию **nanosleep()**, которая позволила процессу спать или выполнять задержку в течение очень короткого времени. До этого

минимальное время составляло около 10 мс; с **nanosleep()** процесс может спать от нескольких микросекунд до нескольких миллисекунд.
- Ядро версии 2.2 имело поддержку сигналов реального времени POSIX.
- Ядра серии 2.4 получили много улучшений в области планирования в режиме реального времени. Самым главным был патч MontaVista для вытеснения ядра и патч низкой латентностью Эндрю Мортона. Это в конечном счёте привело к ядру версии 2.6.
- Ядро версии 2.6 имеет совершенно новый планировщик, называемый планировщиком O (1), который приносит детерминизм в политику планирования. Также стало больше функций реального времени, таких как таймеры POSIX, добавленные в ядро версии 2.6.

Политики режима реального времени Linux более подробно обсуждаются в Главе 7 176.

2.2.4 Файловая система

В Linux различные файловые системы управляются уровнем, называемым VFS или виртуальной файловой системой. Виртуальная файловая система обеспечивает согласованное представление данных, хранящихся в системе на разных устройствах. Она делает это путём отделения пользовательского представления файловых систем с использованием стандартных системных вызовов, но позволяя разработчикам ядра реализовывать логические файловые системы на любом физическом устройстве. Таким образом, это абстрагирует детали физического устройства и логической файловой системы, и позволяет пользователям получать доступ к файлам одинаковым способом.

Любому устройству Linux, будь то встроенная система или сервер, необходима по крайней мере одна файловая система. Это отличается от исполняющих систем реального времени, которые не требуют вообще какой-либо файловой системы. В Linux необходимость иметь файловые системы проистекает из двух фактов:

- Приложения имеют отдельные программные образы и, следовательно, они должны иметь место для хранения в файловой системе.
- Доступ ко всем низкоуровневым устройствам реализуется так же, как к файлам.

Каждой системе Linux необходимо иметь основную файловую систему, корневую файловую систему. Она монтируется при старте системы. Позже с помощью этой файловой системы могут быть установлены многие другие файловые системы. Если система не может смонтировать корневую файловую систему на указанном устройстве, оно будет паниковать, а не выполнять запуск системы.

Наряду с дисковыми файловыми системами, Linux поддерживает специализированные файловые системы, которые для встроенных систем реализованы на основе флеш-памяти и ПЗУ. Также в Linux есть поддержка NFS, которая позволяет на встроенной системе смонтировать файловую систему, находящуюся на сетевом устройстве. Linux поддерживает файловые системы на основе памяти, что также полезно на встраиваемых системах. Также есть поддержка логических или псевдо-файловых систем; они могут быть использованы для получения информации о системе, а также использоваться в качестве средств отладки. Ниже приведены некоторые из наиболее часто используемых встраиваемых файловых систем.

- EXT2: классическая файловая система Linux, которая широко используется пользователями
- CRAMFS: сжатая файловая система только для чтения
- ROMFS: файловая система только для чтения

- RAMFS: файловая система для чтения и записи на основе памяти
- JFFS2: журналируемая файловая система, специально созданная для хранения на флеш
- PROCFS: псевдо-файловая система, используемая для получения системной информации
- DEVFS: псевдо-файловая система для хранения файлов устройств

Более подробно эти файловые системы рассматриваются в Главе 4 59.

2.2.5 Подсистема ввода/вывода

Подсистема ввода/вывода Linux обеспечивает простой и универсальный интерфейс для устройств, находящихся на плате. Подсистемой ввода/вывода поддерживаются три вида устройств:

- Символьные устройства для поддержки последовательных устройств.
- Блочные устройства для поддержки устройств с произвольным доступом. Блочные устройства имеют важное значение для реализации файловых систем.
- Сетевые устройства, которые поддерживают широкий спектр устройств на канальном уровне.

Более подробно архитектура драйверов устройств Linux вместе с конкретными примерами описана в Главе 5 100.

2.2.6 Сетевые подсистемы

Одним из основных преимуществ Linux является его надёжная поддержка разных сетевых протоколов. В Таблице 2.1 перечислены основные наборы функций вместе с версиями ядра, в которых они поддерживаются.

Таблица 2.1 Возможности сетевого стека в ядрах версии 2.2, 2.4 и 2.6

	Наличие в ядре		
Возможность	***2.2***	***2.4***	***2.6***
Уровень 2			
Поддержка сетевого моста	Есть	Есть	Есть
X.25	Есть	Есть	Есть
LAPB	Пробная	Есть	Есть
PPP	Есть	Есть	Есть
SLIP	Есть	Есть	Есть
Ethernet	Есть	Есть	Есть
ATM	Нет	Есть	Есть
Bluetooth	Нет	Есть	Есть
Уровень 3			
IPV4	Есть	Есть	Есть

IPV6	Нет	Есть	Есть
Перенаправление IP	Есть	Есть	Есть
Групповая передача по IP	Есть	Есть	Есть
Защита IP	Есть	Есть	Есть
Туннелирование IP	Есть	Есть	Есть
ICMP	Есть	Есть	Есть
ARP	Есть	Есть	Есть
NAT	Есть	Есть	Есть
IPSEC	Нет	Нет	Есть
Уровень 4 (и выше)			
UDP и TCP	Есть	Есть	Есть
BOOTP/RARP/DHCP	Есть	Есть	Есть

2.2.7 Межпроцессное взаимодействие

Взаимодействие между процессами в Linux включает в себя сигналы (для асинхронного взаимодействия), каналы и сокеты, а также IPC механизмы System V, такие как разделяемая память, очереди сообщений и семафоры. Ядро версии 2.6 имеет дополнительную поддержку для очередей сообщений POSIX.

2.3 Пользовательское пространство

Пользовательское пространство на Linux основывается на следующих принципах.

- ***Программа***: это образ приложения. Он находится в файловой системе. Когда приложение необходимо запустить, образ загружается в память и начинает работать. Обратите внимание, что из-за виртуальной памяти весь образ процесса в память не загружается, а будут загружены только необходимые страницы памяти.
- ***Виртуальная память***: она позволяет каждому процессу иметь своё собственное адресное пространство. Виртуальная память позволяет иметь дополнительные возможности, такие как разделяемые библиотеки. Каждый процесс имеет свою собственную карту памяти в виртуальном адресном пространстве; она является уникальной для каждого процесса и совершенно не зависит от карты памяти ядра.
- ***Системные вызовы***: это точки входа в ядро, так что ядро может выполнять работу от имени приложения.

Чтобы понять, как в Linux работает приложение, рассмотрим небольшой пример. Допустим, что следующему фрагменту кода необходимо запускать приложение на устройстве на базе MIPS.

```
#include <stdio.h>
char str[] = “hello world”;
void myfunc()
{
```

```
  printf(str);
}
main()
{
  myfunc();
  sleep(10);
}
```

Чтобы сделать это, необходимы следующие шаги:

1. ***Компиляция и сборка исполняемой программы***: программы для встроенной системы не собираются на целевой платформе, а требуют систему со средствами кросс-платформенной разработки. Подробнее это будет обсуждаться в Разделе 2.5 20; на данный момент предположим, что у нас есть базовый компьютер и инструменты для создания приложения, которое называется **hello_world**.
2. ***Загрузка исполняемой программы в файловую систему целевой платы***: процесс создания корневой файловой системы и загрузку приложений на целевую платформу рассматривает Глава 8 240. Сейчас предположим, что этот шаг является доступным для вас; некоторым образом вы сможете загрузить **hello_world** в каталог **/bin** корневой файловой системы.
3. ***Выполнение программы через запуск в оболочке***: оболочка является интерпретатором командного языка; она может быть использована для выполнения файлов. Не вдаваясь в подробности работы оболочки, предположим, что при вводе команды **/bin/hello_world**, ваша программа работает, и вы видите строку в консоли (которая, как правило, последовательный порт).

Для платформы MIPS для создания исполняемого файла используется следующая команда:

```
#mips_fp_le-gcc hello_world.c -o hello_world
#ls -l hello_world
-rwxrwxr-x 1 raghav raghav 11782 Jul 20 13:02 hello_world
```

Это происходит за четыре шага: сначала идёт обработка препроцессором, потом генерируется выходная сборка языка, за которой следует генерация объектного вывода, а затем последний этап - компоновка. Выходной файл **hello_world** является исполняемым файлом MIPS в формате, называемом ELF (Executable Linkage Format, формат исполняемой сборки). Все исполняемые файлы имеют два формата: двоичный формат и файлы сценариев (скриптов). Исполняемыми двоичными форматами, которые наиболее популярны на встраиваемых системах, являются COFF, ELF и простой (flat) формат. Формат FLAT используется в системах без MMU на uClinux и обсуждается в Главе 10 322. COFF первоначально был форматом по умолчанию и был заменён на более мощный и гибкий формат ELF. Формат ELF состоит из заголовка, за которым следуют многие разделы, включая текст и данные. Чтобы найти список символов в исполняемом файле, можно использовать команду **nm**, как показано в Распечатке 2.1 12.

Как вы можете видеть, функциям **main** и **myfunc**, а также глобальным данным, **str**, были присвоены адреса, но функция **printf** не определена (указывается как "U") и определяется как **printf@@GLIBC**. Это означает, что **printf** - не часть образа **hello_world**. Тогда где определена эта функция и как определяются эти адреса? Эта функция является частью библиотеки ***libc*** (библиотеки языка Си). Libc содержит список наиболее часто используемых функций. Например, функция **printf** используется почти во всех приложениях. Таким

образом, вместо того, чтобы находиться в каждом образе приложения, её заменителем становится библиотека. Если библиотека используется как общая библиотека, это не только оптимизирует место для хранения, но также оптимизирует расход памяти, делая так, что в памяти находится только одна копия текста. Приложение может иметь несколько библиотек, общих или статических; это может быть указано во время компоновки. Список зависимостей может быть найден с помощью следующей команды (зависимостями от общих библиотек являются динамический компоновщик **ld.so** и библиотека языка Си).

```
#mips_fp_le-ldd hello_world
libc.so.6
ld-linux.so.2
```

Таким образом, на момент создания исполняемого файла всех перемещений и разрешений символов не произошло. Всем функциям и данным глобальных переменных, которые не являются частью общих библиотек, были присвоены адреса и их адреса разрешены так, что вызывающий знает их адреса выполнения. Тем не менее, адреса выполнения разделяемых библиотек ещё не известны, и, следовательно, их разрешение (например, из функции **myfunc** вызывается **printf**) не закончено. Всё это происходит во время выполнения, когда программа будет реально запускаться из оболочки.

Отметим, что существует альтернатива использованию разделяемых библиотек, так что все ссылки компонуются статически. Например, приведённый выше код может бы быть скомпонован со статической библиотекой языка Си **libc.a** (которая представляет собой архивный набор объектных файлов), как показано ниже:

```
#mips_fp_le-gcc -static hello-world.c -o hello_world
```

Если вывести список символов файла, как показано выше, функции **printf** адрес задан. Использование статических библиотек имеет тот недостаток, что тратится место и память за счёт более быстрой скорости запуска приложения. Теперь давайте запустим программу на плате и изучить её карту памяти.

```
#/bin/hello_world &
[1] 4479
#cat /proc/4479/maps
00400000-00401000 r-xp 00000000 00:07 4088393 /bin/hello_world
00401000-00402000 rw-p 00001000 00:07 4088393 /bin/hello_world
2aaa8000-2aac2000 r-xp 00000000 00:07 1505291 /lib/ld-2.2.5.so
2aac2000-2aac4000 rw-p 00000000 00:00 0
2ab01000-2ab02000 rw-p 00019000 00:07 1505291 /lib/ld-2.2.5.so
2ab02000-2ac5f000 r-xp 00000000 00:07 1505859 /lib/
                                               libc-2.2.5.so
2ac5f000-2ac9e000 ---p 0015d000 00:07 1505859 /lib/
                                               libc-2.2.5.so
2ac9e000-2aca6000 rw-p 0015c000 00:07 1505859 /lib/
                                               libc-2.2.5.so
2aca6000-2acaa000 rw-p 00000000 00:00 0
7ffef000-7fff8000 rwxp ffff8000 00:00 0
```

Видно, что вместе с основной программой **hello_world**, выделен диапазон адресов для **libc** и динамического компоновщика **ld.so**. Во время выполнения создаётся карта памяти приложения, а затем выполняется разрешение символов (в нашем случае это **printf**). Это делается за несколько этапов. Загрузчик ELF, который построен как часть ядра, сканирует

исполняемый файл и узнаёт, что процесс имеет зависимость от общей библиотеки; поэтому он вызывает динамический компоновщик **ld.so**. **Ld.so**, который также реализован в виде общей библиотеки, является самозагружаемой библиотекой; он загружает себя и остальные разделяемые библиотеки (**libc.so**) в память, фиксируя таким образом карту памяти приложения и выполняя оставшуюся часть разрешения символов.

Остался последний вопрос: как на самом деле работает **printf**? Как уже говорилось выше, любое обслуживание, которое должно быть сделано ядром, требует, чтобы приложение сделало системный вызов. После выполнения своих внутренних дел **printf** тоже делает системный вызов. Поскольку фактическая реализация системных вызовов сильно зависит от аппаратного обеспечения, библиотека языка Си скрывает всё это, обеспечивая обёртки, которые на самом деле делают системный вызов.Список всех системных вызовов, которые сделаны приложением, можно узнать используя приложение под названием **strace**; например, запуск **strace** для нашего приложения даёт следующий вывод, часть которого приведена ниже:

```
#strace hello_world
...
write(1, "hello world", 11) = 11
...
```

Теперь когда стала понятна основная идея пространства ядра и пространства пользователя, перейдём к процедуре запуска системы Linux.

Распечатка 2.1 Распечатка символов с помощью nm

Распечатка 2.1

```
#mips_fp_le-nm hello_world
0040157c A __bss_start
004002d0 t call_gmon_start
0040157c b completed.1
00401550 d __CTOR_END__
0040154c d __CTOR_LIST__
0040146c D __data_start
0040146c W data_start
00400414 t __do_global_ctors_aux
004002f4 t __do_global_dtors_aux
00401470 D __dso_handle
00401558 d __DTOR_END__
00401554 d __DTOR_LIST__
00401484 D _DYNAMIC
0040157c A _edata
00400468 r __EH_FRAME_BEGIN__
00401580 A _end
00400438 T _fini
0040146c A __fini_array_end
0040146c A __fini_array_start
00400454 R _fp_hw
00400330 t frame_dummy
00400468 r __FRAME_END__
00401560 D _GLOBAL_OFFSET_Table_
         w __gmon_start__
```

```
00400254 T _init
0040146c A __init_array_end
0040146c A __init_array_start
00400458 R _IO_stdin_used
0040155c d __JCR_END__
0040155c d __JCR_LIST__
         w _Jv_RegisterClasses
004003e0 T __libc_csu_fini
004003b0 T __libc_csu_init
         U __libc_start_main@@GLIBC_2.0
00400374 T main
0040035c T myfunc
00401474 d p.0
         U printf@@GLIBC_2.0
         U sleep@@GLIBC_2.0
004002ac T _start
00401478 D str
```

2.4 Последовательность запуска Linux

Теперь, когда есть понимание высокоуровневой архитектуры Linux, понимание последовательности запуска даст информацию о том, как стартуют различные подсистемы ядра, и как Linux передаёт управление в пользовательское пространство. Последовательность запуска Linux описывает последовательность шагов, которые начинаются с момента начального старта Linux до того, как пользователю в консоли предоставляется приглашение на вход. Почему на данном этапе вы должны понимать последовательность запуска? Понимание последовательности запуска необходимо, чтобы отметить этапы в цикле разработки. Также, после понимания процесса запуска станут поняты основные части, необходимые для создания системы Linux, такие как загрузчик и корневая файловая система. На встроенных системах время запуска часто должно быть как можно меньше; понимание деталей поможет пользователю настроить систему для быстрого запуска. Для более подробной информации об ускорении загрузки обратитесь к Приложению A 364.

Последовательность запуска Linux может быть разбита на три этапа:

- ***Фаза начальной загрузки***: обычно этот этап выполняет инициализацию оборудования и тестирование, загружает ядро, и передаёт управление ядру Linux.
- ***Фаза инициализации ядра***: этот этап выполняет зависящую от платформы инициализацию, запускает подсистемы ядра, включает многозадачность, монтирует корневую файловую систему и переходит в пространство пользователя.
- ***Фаза инициализации пользовательского пространства***: обычно эта фаза запускает службы, выполняет инициализацию сети, а затем выдаёт строку приглашения.

2.4.1 Фаза начальной загрузки

Подробно загрузчики рассматриваются в Главе 3 29. В этом разделе описывается последовательность шагов, выполняемых загрузчиком.

Инициализация оборудования

Она обычно включает в себя:

1. Настройку частоты процессора
2. Инициализацию памяти, такую как настройка регистров, очистка памяти и определение размера встроенной памяти
3. Включение кэшей
4. Настройку последовательного порта для консоли загрузки
5. Выполнение диагностики оборудования или POST (Power On Self-Test diagnostics, самотестирование после включения)

После успешного завершения этих действий, следующим шагом является загрузка ядра Linux.

Загрузка образа ядра и начального RAM диска

Загрузчик должен найти образ ядра, который может быть на системной флеш-памяти, или может быть доступен по сети. В любом случае, этот образ должен быть загружен в память. В случае, если образ сжат (что бывает часто), образ должен быть распакован. Также, если присутствует начальный RAM (электронный) диск, загрузчик должен загрузить в память образ начального диска. Обратите внимание, что адрес памяти, куда загружается ядро, определяется загрузчиком чтением заголовка ELF файла образа ядра. В случае, если образ ядра представляет собой простые двоичные данные, загрузчику должна быть передана дополнительная информация относительно размещения частей ядра и адреса запуска.

Аргументы настройки

Передача аргументов является очень мощным инструментом, поддерживаемым ядром Linux. Linux обеспечивает универсальный способ передачи аргументов ядру на всех платформах. Глава 3 29 объясняет это в деталях. Обычно загрузчик должен настроить область памяти для передачи аргументов, проинициализировать её необходимыми структурами данных (которые могут быть идентифицированы ядром Linux), а затем заполнить их требуемыми значениями.

Переход в точку входа ядра

Точка входа ядра определяется скриптом компоновщика при сборке ядра (который обычно присутствует в скрипте компоновщика в зависимом от архитектуры каталоге). Как только загрузчик переходит к точке входа ядра, его работа сделана и он больше не нужен. (Есть исключения из этого; некоторые платформы предлагают службу загрузки PROM, которая может использоваться операционной системой для ведения платформо-зависимых операций.) Если это так, и если загрузчик выполняется из памяти, эта память может быть передана ядру. Это должно быть учтено при построении карты памяти системы.

2.4.2 Запуск ядра

Запуск ядра можно разделить на следующие этапы.

Инициализация, зависящая от CPU/платформы

Если вы переносите Linux на вашу платформу, этот раздел очень важен, поскольку он

знаменует важный этап в переносе BSP. Платформо-зависимая инициализация состоит из следующих шагов.

1. ***Настройка окружения для первой процедуры на Си***: точка входа ядра является процедурой на языке ассемблера; название этой точки входа варьируется (**stext** для ARM, **kernel_entry** для MIPS и так далее). Чтобы узнать точку входа для вашей платформы, посмотрите скрипт компоновщика. Эта функция обычно находится в файле **arch/<name>/kernel/head.S**. Эта функция делает следующее:
   a. На машинах, которые не имеют включённого MMU, она включает MMU. Большинство загрузчиков не работают с MMU, так что виртуальный адрес эквивалентен физическому адресу. Тем не менее, ядро собрано с виртуальным адресом. Функция должна включить MMU, чтобы ядро могло начать использовать виртуальный адрес в обычном режиме. Это не является обязательным на таких платформах, как MIPS, где MMU включён сразу при включении питания.
   b. Выполняет инициализацию кэша. Это снова зависит от платформы.
   c. Настраивает BSS путём его обнуления (как правило, нельзя полагаться, что это сделает загрузчик).
   d. Настраивает стек, чтобы могла быть вызвана первая процедура на языке Си. Такой первой процедурой на языке Си является функция **start_kernel()** в **init/main.c**. Это большая функция, которая делает много всего, пока не заканчивается фоновой задачей (первой задачей в системе, имеющей идентификатор процесса 0). Эта функция вызывает функции остальной инициализации платформы, которые будут рассмотрены ниже.
2. ***Функция* setup_arch()**: эта функция выполняет зависимую от платформы и процессора инициализацию, так что оставшаяся инициализация может быть вызвана безопасно. Приведём только общую функциональность, так как она сильно зависит от платформы:
   a. Распознавание процессора. Поскольку архитектура процессора может иметь различные особенности, эта функция распознаёт процессор (например, если выбран процессор ARM, она распознаёт особенности ARM) используя оборудование или информацию, которая может быть передана во время сборки. Опять же в этом коде могут быть сделаны какие-либо зависящие от процессора исправления.
   b. Распознавание платы. Снова, поскольку ядро поддерживает множество плат, эта опция распознаёт плату и выполняет зависимые от платы исправления.
   c. Анализ параметров командной строки, переданной ядру.
   d. Определение электронного диска, если он был создан загрузчиком, таким образом, чтобы ядро позже могло смонтировать его в качестве корневой файловой системы. Обычно загрузчик передаёт начало области электронного диска в памяти и размер.
   e. Вызов функций ***bootmem***. Bootmem является неправильным наименованием; оно относится к начальной памяти, которую ядро может зарезервировать для различных целей, прежде чем код подкачки захватывает всю память. Например, вызовом распределителя **bootmem** вы можете зарезервировать большой кусок непрерывной памяти, который может быть использован вашим устройством для DMA.
   f. Вызов функции инициализации подкачки, которая забирает оставшуюся память для создания страниц для системы.
3. ***Инициализация исключений - функция* trap_init()**: эта функция настраивает зависимые от ядра обработчики исключений. До этого момента, если происходит исключительная ситуация, результат зависит от платформы. (Например, на некоторых платформах вызываются обработчики исключений, зависимые от загрузчика.)
4. ***Инициализация процедуры обработки прерываний - функция* init_IRQ()**: эта

функция инициализирует контроллер прерываний и дескрипторы прерываний (это структуры данных, которые используются BSP для маршрутизации прерываний; об этом в следующей главе). Обратите внимание, что на данный момент прерывания не разрешены, за их включение отвечают отдельные драйверы, владеющие линиями прерываний, что происходит во время их инициализации, которая вызывается позже. (Например, инициализация таймера убедилась бы, что линия прерывания таймера включена.)

5. ***Инициализация таймеров - функция*** **time_init()**: эта функция инициализирует таймерное оборудование, так что система начинает генерировать периодические временные сигналы (тики), которые являются сердцебиением системы.
6. ***Инициализация консоли - функция*** **console_init()**: эта функция выполняет инициализацию последовательного устройства в качестве консоли. Как только консоль включена, на экране появляются все сообщения о ходе запуска. Для печати сообщения из ядра должна быть использована функция **printk()**. (**printk()** является очень мощной функцией, она может быть вызвана из любого места, даже из обработчиков прерываний.)
7. ***Расчёт циклов задержки для данной платформы - функция*** **calibrate_delay()**: эта функция используется для реализации в ядре микрозадержек с помощью функции **udelay()**. Функция **udelay()** прокручивает несколько циклов за число микросекунд, указанных в качестве аргумента. Чтобы **udelay** работала как надо, ядру необходимо знать число тактов в микросекунду. Это как раз и делается с помощью этой функции; она калибрует количество циклов задержки. Это гарантирует, что циклы задержки работают одинаково на всех платформах. Обратите внимание, что выполнение этого зависит от прерывания таймера.

Инициализация подсистем

Она включает:

- Инициализацию планировщика
- Инициализацию контроллера памяти
- Инициализацию VFS

Обратите внимание, что большая часть инициализации подсистем выполняется в функции **start_kernel()**. В конце этой функции ядро создаёт новый процесс, процесс инициализации, **init**, чтобы выполнить остальную инициализацию (инициализацию драйверов, initcalls, монтирование корневой файловой системы и переход в пространство пользователя) и текущий процесс становится фоновым процессом с идентификатором процесса, равным 0.

Инициализация драйверов

Инициализация драйверов выполняется после того, как запущен управляющий процесс и управление памятью. Это может быть сделано в контексте процесса **init**.

Монтирование корневой файловой системы

Напомним, что корневая файловая система является основной используемой файловой системой с помощью которой могут быть смонтированы другие файловые системы.

Монтирование её знаменует собой важный процесс в стадии загрузки, так как ядро может начать свой переход в пользовательское пространство. Блочное устройство, содержащее корневую файловую систему, может быть жёстко закодировано в ядре (при сборке ядра), или оно может быть передано в качестве аргумента командной строки из загрузчика с помощью параметра загрузчика "**root=**".

На встроенных системах обычно используются три вида корневых файловых систем:

- Начальный электронный диск
- Сетевая файловая система, использующая NFS
- Файловая система на основе флеш-памяти

Обратите внимание, что корневая файловая система на основе NFS используется в основном для целей отладки; две другие используются для сборок для производства. Электронный диск имитирует блочное устройство с помощью системной памяти, поэтому он может быть использован для монтирования файловых систем при условии, что образ файловой системы копируется на него. Электронный диск может использоваться в качестве корневой файловой системы; это использование электронного диска известно как ***initrd*** (краткая форма initial ram disk). **Initrd** - это очень мощная концепция и имеет широкое использование, особенно на начальной стадии разработки встраиваемого Linux, когда у вас нет готового драйвера флеш-памяти, но ваши приложения готовы для тестирования (часто это тот случай, когда есть отдельные команды, разрабатывающие драйверы и приложения, и работающие параллельно). Так как же обойтись без корневой файловой системы во флеш-памяти? Можно использовать сетевую файловую систему при условии, что ваш сетевой драйвер готов; если нет, то лучшим вариантом является **initrd**. Создание начального электронного диска более подробно объясняется в Главе 8 240. Этот же раздел объясняет, как ядро монтирует **initrd** в качестве корневой файловой системы. Если вы хотите, чтобы ядро загрузило **initrd**, необходимо настроить ядро в процессе сборки с помощью параметра **CONFIG_BVLK_DEV_INITRD**. Как объяснялось ранее, образ **initrd** загружается вместе с образом ядра и ядру должен быть передан начальный и конечный адрес **initrd** с помощью аргументов командной строки. Как только они стали известны, ядро смонтирует корневую файловую систему, загруженную на **initrd**. Обычно используемыми файловыми системами являются **romfs** и **ext2**.

У **initrd** есть много больше возможностей. **Initrd** является корневой файловой системой используй-и-выброси. Она может быть использована для монтирования другой корневой файловой системы. Почему это необходимо? Предположим, что корневая файловая система монтируется на устройство хранения, драйвер которого является модулем ядра. Так что он должен присутствовать в файловой системе. Это представляет собой проблему курицы и яйца; модуль должен быть в файловой системе, которая, в свою очередь, требует, чтобы сначала был загружен модуль. Чтобы обойти это, может быт использована **initrd**. Драйвер может быть выполнен в виде модуля в **initrd**; раз **initrd** смонтирована, то модуль драйвера может быть загружен и, следовательно, устройство хранения может быть доступно. Затем файловая система на этом устройстве хранения может быть смонтирована в качестве фактической корневой файловой системы и, наконец, **initrd** может быть отброшен. Ядро Linux предоставляет возможность для такого её использования и отбрасывания; оно ищет файл **linuxrc** в корне **initrd** и выполняет его. Если этот двоичный файл возвращается, то ядро предполагает, что в **initrd** больше необходимости нет, и оно переключается на фактическую корневую файловую систему (файл **linuxrc** может быть использован для загрузки модулей драйверов). NFS и файловые системы на базе флеш-памяти более подробно описаны в Главе 4 59.

Если корневая файловая система не смонтирована, то ядро остановит выполнение и

после выдачи сообщения в консоль перейдёт в режим паники:

```
Unable to mount root fs on device
```

Выполнение Initcall и освобождение памяти, используемой при инициализации

Если вы откроете скрипт компоновщика для любой архитектуры, он будет иметь раздел **init**. Начало этого раздела помечено с помощью **__init_begin**, а конец отмечен с помощью **__init_end**. Идея этого раздела в том, что он содержит текст и данные, которые могут быть отброшены после того, как они используются один раз во время запуска системы. Функции инициализации драйверов - это пример функции используй-и-выброси. Как только драйвер, статически скомпонованный с ядром, выполнит свою регистрацию и инициализацию, эта функция не будет вызываться снова и, следовательно, её можно отбросить. Идея состоит в том, чтобы положить все эти функции вместе, чтобы вся память, занимаемая всеми такими функциями, могла быть освобождена как большой кусок и, следовательно, будет доступна для диспетчера памяти в качестве свободных страниц. Учитывая, что память на встраиваемых системах является дефицитным ресурсом, читателю рекомендуется эффективно использовать эту концепцию. Функция или переменная используй-и-выброси объявляется с помощью директивы **__init**. После того, как будет сделана вся инициализация драйверов и подсистем, код запуск всю эту память освобождает. Это делается прямо перед переходом в пространство пользователя.

Linux также предоставляет способ группировки функций, которые должны быть вызваны при запуске системы. Это можно сделать, объявив функцию с директивой **__initcall**. Эти функции автоматически вызываются во время запуска ядра, так что не требуется вставлять их в код запуска системы.

Переход в пользовательское пространство

Ядро, которое выполняется в контексте процесса инициализации, переходит в пространство пользователя, перекрывая себя (при помощи **execve**) исполняемым образом специальной программы, также называемом как **init**. Этот исполняемый файл обычно находится в корневой файловой системе в каталоге **/sbin**. Обратите внимание, что пользователь может указать программу инициализации, используя аргументы командной строки ядра. Однако, если ядро не может загрузить указанную пользователем программу инициализации или заданную по умолчанию, после выдачи сообщения, оно входит в состояние паники:

```
No init found. Try passing init= option to the kernel.
```

2.4.3 Инициализация пользовательского пространства

Инициализация пользовательского пространства зависит от того, что поставляется. С переходом к процессу **init** зона ответственности ядра завершается. Что делает процесс **init** и как он запускает службы зависит от комплекта поставки. Сейчас мы изучим общую модель на Linux (которая предполагает, что процессом инициализации является **/sbin/init**); общая модель мало отличается от последовательности инициализации варианта Unix, System V UNIX.

Процесс /sbin/init и /etc/inittab

Процесс **init** - это особый процесс ядра; он имеет следующие возможности:

- Он никогда не может быть убит. Linux предлагает сигнал, называемый **SIGKILL**, который может прекратить выполнение любого процесса, но процесс **init** он убить не может.
- Когда какой-то процесс запускает другой процесс, последний становится потомком первого. Это отношение родитель-потомок имеет важное значение. В случае, если родительский процесс умирает раньше процесса потомка, **init** "усыновляет" осиротевшие процессы.
- Ядро сообщает **init**-у об особых событиях используя сигналы. Например: если на клавиатуре системы нажать **Ctrl-Alt-Del**, это заставит ядро послать сигнал процессу **init**, который обычно выполняет выключение системы.

Процесс **init** может быть настроен на любой системе с помощью файла **inittab**, который обычно находится в каталоге **/etc**. **init** читает файл **inittab** и соответственно в последовательном порядке выполняет указанные действия. **init** также определяет состояние системы, известное как ***режим работы*** (уровень выполнения, run level). Режим работы - это число, которое передаётся в **init** в качестве аргумента. В случае, если не указано ничего, **init** может взять режим работы по умолчанию из файла **inittab**. Используются следующие режимы работы:

- 0 Остановка системы
- 1 Однопользовательский режим (используется для административных целей)
- 2 Многопользовательский режим с ограниченными сетевыми возможностями
- 3 Полный многопользовательский режим
- 4 Не используется
- 5 Графический режим (X11 ™)
- 6 Состояние перезагрузки

Файл **inittab** имеет специальный формат. Как правило, он содержит следующую информацию (для получения дополнительной информации, пожалуйста, обратитесь к главной странице **inittab** на вашей системе):

- Режим работы по умолчанию.
- Действия, предпринимаемые, когда для **init** меняется режим работы. Обычно вызывается сценарий **/etc/rc.d/rc** с режимом работы в качестве аргумента.
- Процесс, который должен быть выполнен во время запуска системы. Как правило, это файл **/etc/rc.d/rc.sysinit**.
- **init** может возрождать любой процесс, если так настроено в файле **inittab**. Эта функция используется для перезапуска процесса входа в систему после выхода пользователя после предыдущего входа в систему.
- Действия для улавливания специальных сообщений, таких как Ctrl-Alt-Del или сбой питания.

Файл rc.sysinit

Этот файл выполняет инициализацию системы до запуска служб. Обычно на встроенной системе этот файл выполняет следующие действия:

- Монтирует специальные файловые системы, такие как proc, ramfs
- Создаёт при необходимости каталоги и ссылки
- Устанавливает имя системы (hostname)
- Настраивает сетевую конфигурацию системы

Запуск служб

Как упоминалось выше, за запуск служб отвечает сценарий **/etc/rc.d/rc**. Служба определяется как возможность управлять системным процессом. Используя службы, процесс может быть остановлен, перезапущен и может быть запрошено его состояние. Службы обычно организованы в каталогах основываясь на режимах работы; в зависимости от режима работы выбирается, остановить службу или запустить. После выполнения описанных выше действий, **init** запускает программу входа в систему по TTY или запускает оконный менеджер на графическом дисплее (в зависимости от режима работы).

2.5 Кросс-платформенные инструменты GNU

Одним из первых шагов в переносе встраиваемого Linux является создание набора инструментов для сборки ядра и приложений. Набор инструментов (toolchain), который используется во встраиваемых системах, известен как кросс-платформенный инструментарий. Что именно значит кросс-платформенный? Обычно для генерации кода для платформы x86 используется компилятор для x86. Однако, на встроенных системах это может быть не так. Целевая платформа, на которой должны работать приложения и ядро, может не иметь достаточно памяти и дискового пространства для размещения инструментов сборки. Кроме того, в большинстве случаев целевая платформа может не иметь собственного компилятора. В таких случаях решением является кросс-компиляция. Кросс-компиляция обычно происходит на настольном компьютере (как правило, на базе x86) с помощью компилятора, который работает на Linuxx86 (HOST) и генерирует код, исполняемый на встроенной целевой (TARGET) платформе. Этот процесс компиляции на HOST для генерации кода для системы TARGET называется ***кросс-компиляцией***, а компилятор, используемый с этой целью, называется ***кросс-компилятор***.

Любой компилятор требует большого числа вспомогательных библиотек (таких как **libc**) и двоичных файлов (таких как трансляторы и компоновщики). Можно было бы потребовать аналогичный набор инструментов также и для кросс-компиляции. Весь этот набор инструментов, двоичных файлов и библиотек имеет общее название ***кросс-платформенный инструментарий***. Самым надёжным инструментом для компиляции с открытым кодом, доступным для различных платформ, является компилятор GNU, а его вспомогательные инструменты называются инструментарий GNU (GNU toolchain). Эти компиляторы поддерживаются множеством разработчиков со всего Интернета и проверены миллионами людей по всему миру на различных платформах.

Кросс-платформенный инструментарий имеет перечисленные ниже компоненты:

- ***Утилиты*** (binutils): утилиты представляют собой набор программ, необходимых для компиляции/компоновки/ассемблирования и других операций отладки.
- ***Компилятор GNU языка Си*** (GNU C compiler): основной компилятор языка Си, используемый для генерации объектного кода (и ядра, и приложений).
- ***Библиотека GNU языка Си*** (GNU C library): эта библиотека реализует интерфейсы системных вызовов, таких как **open**, **read**, и других, а также других вспомогательных функций. Все разрабатываемые приложения должны быть скомпонованы с этой базовой

библиотекой.

Помимо GCC и **glibc**, важной частью набора инструментов также являются **binutils**. Некоторыми из утилит, которые составляют **binutils**, являются следующие:

- **addr2line**: она переводит адреса программы в имена файлов и номера строк. Зная адрес и исполняемый файл, она использует отладочную информацию в исполняемом файле, чтобы выяснить, какое имя файла и какой номер строки связаны с указанным адресом.
- **ar**: программа GNU **ar** создаёт, модифицирует и извлекает файлы из архивов. Архив представляет собой один файл, содержащий набор других файлов в структуре, которая позволяет извлекать отдельные оригинальные файлы (так называемые члены архива).
- **as**: GNU **as** является семейством ассемблеров. Если вы используете (или использовали) ассемблер GNU на одной архитектуре, вы должны найти аналогичной среду, когда вы используете её на другой архитектуре. Каждая версия имеет много общего с другими, включая формат объектных файлов, большинство директив ассемблера (часто называемых псевдо-операциями) и синтаксис ассемблера.
- **c++filt**: программа **c++filt** выполняет обратное действие: декодирует низкоуровневые имена в имена пользовательского уровня, так что компоновщик может избежать конфликтов таких перегруженных функций.
- **gasp**: макро-препроцессор ассемблера GNU.
- **ld**: компоновщик GNU **ld** объединяет несколько объектных и архивных файлов, перемещает их данные и связывает символьные ссылки. Вызов **ld** часто является последним шагом в сборке новой скомпилированной программы для запуска.
- **nm**: GNU **nm** выводит символы из объектных файлов.
- **objcopy**: утилита GNU **objcopy** копирует содержимое одного объектного файла в другой. Для чтения и записи объектных файлов **objcopy** использует библиотеку GNU BFD. Она может записать объектный файл назначения в формате, отличном от исходного объектного файла. Точное поведение **objcopy** управляется параметрами командной строки.
- **objdump**: утилита GNU **objdump** выводит информацию об одном или нескольких объектных файлах. Какую информацию отображать, указывается параметрами, например, таблицу символов, GOT, и тому подобное.
- **ranlib**: **ranlib** генерирует список содержимого архива и сохраняет его в этом архиве. Список содержит каждый символ, определяемый членом архива, который является перемещаемым объектным файлом.
- **readelf**: она интерпретирует заголовки файлов ELF.
- **size**: утилита GNU **size** выводит размеры частей и общий размер для каждого из объектных файлов в своём списке аргументов. По умолчанию для каждого объектного файла или модуля в архиве создаётся одна строка вывода.
- **strings**: GNU **strings** распечатывает последовательности печатных символов, которые по крайней мере являются символами и заканчиваются непечатаемым символом. По умолчанию она выводит только строки из проинициализированных и загружаемых частей объектных файлов; для других типов файлов она выводит строки из всего файла.
- **strip**: GNU **strip** отбрасывает все символы из целевого объектного файла(ов). Список объектных файлов может включать архивы. Должен быть задан по крайней мере один объектный файл. **strip** изменяет файлы с именами, указанными в своём аргументе, вместо записи модифицированных копий под другими именами.

2.5.1 Сборка набора инструментов

Сборка кросс-платформенного инструментария немного сложна и может очень раздражать, когда процесс создания не получается. Поэтому целесообразно просто скачать готовые кросс-платформенные инструменты для вашей целевой платформы. Отметим, что хотя **binutils** и компилятор языка Си могут быть использованы как для сборки ядра, так и приложений, библиотека языка Си используется только в приложениях.

Вот набор ссылок, где вы можете взять последний кросс-платформенный инструментарий:

- ARM: http://www.emdebian.org/
- PPC: http://www.emdebian.org/
- MIPS: http://www.linux-mips.org/
- M68K: http://www.uclinux.org/

Теперь, если вам не повезло и кросс-компилятор для вашей платформы с полки не доступен, то шаги, описанные ниже, помогут вам в сборке кросс-платформенного инструментария.

1. Укажите **TARGET**.
2. Укажите версию Kernel/GCC/Glibc/Binutils.
3. Наложите патчи для всего вышеперечисленного, если таковые имеются.
4. Укажите в переменной **PREFIX**, куда поместить образ.
5. Соберите **binutils**.
6. Получить подходящие заголовки ядра для вашей платформы.
7. Скомпилируйте минимальный GCC.
8. Соберите **glibc**.
9. Перекомпилируйте полнофункциональный GCC.

Выбор целевого имени

Имя цели определяет компилятор, который вы собираете и его результат работы. Вот некоторые из основных типов:

- ***arm-linux***: поддерживает процессора ARM, такие как armV4, armv5t и так далее.
- ***mips-linux***: поддерживает разнообразные ядра MIPS, такие как r3000, r4000 и так далее.
- ***ppc-linux***: комбинация Linux/PowerPC с поддержкой разнообразных чипов PPC.
- ***m68k-linux***: для сборки Linux, работающего на процессоре Motorola 68k.

Полный список можно найти на сайте http://www.gnu.org.

Выбор правильной комбинации версий

Это самая сложная часть из всех шагов и, скорее всего, причина всех проблем. Необходимо проделать некоторые исследования и принять решение о правильном сочетании версий, которое будет работать на вашей целевой платформе. Для этого прочитайте наиболее активные архивы списков рассылки. Например, в качестве хорошей комбинации для ARM-Linux известны ARM/Kernel 2.6/GCC 2.95.1, GCC 3.3/BINUTILS 2.10.x или ARM/Kernel 2.4/GCC 2.95.1/BINUTILS 2.9.5.

Доступны ли какие-то патчи?

После принятия решения о номере версии, поищите доступные патчи. Решение, применять патч или нет, зависит исключительно от ваших требований.

Выбор структуры каталогов и установка переменных

Перед созданием набора инструментов должно бы принято решение о дереве каталогов для хранения инструментов и заголовков ядра. Дерево каталогов также экспортируется с помощью определения соответствующих переменных, чтобы сделать работу по сборке более удобной. Используются следующие переменные:

- **TARGET**: эта переменная содержит имя целевой машины. Например, **TARGET=mips-linux**.
- **PREFIX**: это базовый каталог, содержащий все другие подкаталоги вашего набора инструментов; по умолчанию для родного набора инструментов любой системы это почти всегда **/usr**. Это означает, что вы найдёте (двоичный) gcc в **BINDIR = $PREFIX/bin** и заголовки в **INCLUDEDIR= $PREFIX/include**. Чтобы при сборке кросс-компилятора не задеть собственные средства и инструменты в своей системе, вы должны поместить набор инструментов для кросс-разработки в каталог, отличный от каталога по умолчанию **/usr**. Например, **PREFIX=/usr/local/mips/**.
- **KERNEL_SOURCE_DIR**: это место, где хранятся исходные тексты ядра (или хотя бы заголовки ядра). Они вполне могут быть отличными от родного набора файлов, особенно, если вы занимаетесь кросс-компиляцией. Хорошей практикой является хранение файлов ядра Linux в каталоге **PREFIX/TARGET**. Например, **KERNEL_SOURCE_DIR = PREFIX/TARGET/linux** (например, **/usr/local/mips/linux**).
- **NATIVE**: платформа рабочего компьютера (обычно x86).

Сборка binutils

Первый шаг заключается в сборке GNU binutils. Версия 2.9.5 является стабильной, но рекомендуется последняя версия. Шаги заключаются в следующем:

1. ***Загрузка и распаковка***: загрузите последнюю версию с ftp://ftp.gnu.org/gnu/binutils/. Распакуйте архив с помощью команд:

   ```
   cd $PREFIX/src
   tar -xzf binutils-2.10.1.tar.gz
   ```

   Для исправления различных ошибок binutils могут быть зависимые от платформы патчи; как правило, использовать эти исходники является хорошей идеей, если они существуют. Лучшим местом для получения последней информации является лист рассылки о наборе инструментов.
2. ***Конфигурирование***: конфигурационный скрипт настраивает множество параметров компиляции, устанавливаемые двоичные файлы и конфигурацию компьютера. Командами для кросс-компилятора на рабочем компьютере для другой целевой платформы являются:

   ```
   ./configure --target=$TARGET --prefix=$PREFIX
   ```

Например,

```
./configure --target=mips-linux --prefix=/usr/local/mips
```

3. ***Сборка***: для сборки binutils вызовите **make** в каталоге **binutils**.
4. ***Установка***: чтобы установить созданный набор инструментов, вызовите **make install** в каталоге **binutils**. Перед установкой убедитесь, что путь установки не заменяет существующие binutils, если у вас они уже есть, а делает это только если вы действительно этого хотите. Вы найдёте новый набор инструментов в **PREFIX/TARGET/**.

Получение подходящих заголовков ядра

Первым шагом в сборке GCC является настройка заголовков ядра. Кросс-компиляция набора инструментов требует заголовки ядра. Ниже приводится список шагов, которые необходимы для этого:

1. Загрузите ядро Linux с ftp.kernel.org. Распакуйте tar-архив. Мы предлагаем сделать это в каталог **$PREFIX/linux** (например, **/usr/local/mips/linux**).
2. Если необходимо, примените патчи ядра.
3. Укажите архитектуру, для которой вы извлекаете файлы заголовков. Это делается установкой переменной **ARCH** в **Makefile** верхнего уровня, например **ARCH=mips**.
4. Сконфигурируйте ядро, даже если вы не обязательно хотите собрать его. Дайте команду **make menuconfig** в каталоге верхнего уровня исходных текстов ядра:

   ```
   make menuconfig
   ```

   Это вызовет программу конфигурации сборки ядра. Если на вашей системе работает X, вы можете запустить **make xconfig**. Войдите в меню: **System and processor type** и выберите систему в соответствии с инструментами, которые вы создаёте.
5. Построение зависимостей (этот шаг необходим только для ядер версии 2.4). Выйдите из программы настройки, сохранив изменения, а затем выполните команду:

   ```
   make dep
   ```

   Эта команда на самом деле сделает ссылки (свяжет **/usr/local/mips/linux/include/asm/** с **/usr/local/mips/linux/include/asmarm** и так далее) и убедится, что ваши заголовки ядра находятся в пригодном состоянии.

Сборка минимального GCC

Минимальный GCC - это компилятор, который имеет базовую поддержку языка Си. После сборки минимального GCC для целевой платформы может быть собрана **glibc**. Затем **glibc** используется для сборки полнофункционального GCC с поддержкой C++. Теперь после настройки заголовков ядра мы можем собрать минимальный GCC. Для этого выполним следующие действия:

1. ***Загрузка и распаковка***: загрузите последнюю версию GCC с сайта http://gcc.gnu.org.

Распакуйте загруженный GCC. Затем можно решить, применять патчи или нет, если они существуют.

2. ***Конфигурирование***: это может быть сделано так же, как делалось для binutils.

```
./configure --target=$TARGET --prefix=$PREFIX
--with-headers=$KERNEL_SOURCE_DIR/include --enable-languages=c
```

Например,

```
./configure --target=mips-linux --prefix=/usr/local/mips/
--with-headers=/usr/local/mips/linux/include --enable-languages=c
```

Последний параметр данной настройки **-enable-languages=c** необходим потому, что минимальный GCC может поддерживать другие языки, помимо Си, а на данный момент нам необходима поддержка именно языка Си.

3. ***Сборка***: для компиляции минимального GCC необходимо просто вызвать команду **make**. Если компиляция выполняется первый раз, это может привести к следующей ошибке:

```
./libgcc2.c:41: stdlib.h: No such file or directory
./libgcc2.c:42: unistd.h: No such file or directory
make[3]: *** [libgcc2.a] Error 1
```

Эта ошибка может быть исправлена с помощью приёма, называемого ***inhibit-libc***. Процедура следующая:

a. Отредактируйте файл конфигурации, добавив к **TARGET_LIBGCC2_CFLAGS** строку **-Dinhibit_libc** и **-D__gthr_posix_h**. После этого измените строку **TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer –fPIC** на **TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc -D__gthr_posix_h**.

b. Перезапустите конфигурацию.

4. ***Установка***: Предположим, что создание кросс-компилятор прошло успешно, и теперь новый кросс-компилятор можно установить:

```
make install
```

Сборка glibc

Как упоминалось ранее, с каждым приложением компонуется **glibc**. Ядро, однако, не использует **glibc** (у него есть собственная собранная внутри него минимальная библиотека языка Си). Глава 4 59 рассматривает альтернативы **glibc**, в основном из-за того, что **glibc** является тяжеловесной для встраиваемых систем. Независимо от того, используется ли **glibc** на целевой платформе, **glibc** имеет важное значение для сборки компилятора языка Си.

1. ***Загрузка и распаковка***: из-за некоторых ограничений на экспорт программного обеспечения и зависимостей от внешнего исходного кода, **glibc** разделена на ядро **glibc** и набор пакетов. Эти пакеты называются дополнениями (add-ons). Наряду с исходным кодом **glibc**, дополнения должны быть распакованы и во время настройки дополнение должно быть включено. Встроенные системы в основном требуют

дополнение потоков Linux. Следовательно, загрузка и распаковка выполняется для ядра и потоков Linux. Выберите на ftp.gnu.org/gnu/glibc свежий архив **glibc** и соответствующий архив потоков Linux. Распакуйте главный архив **glibc** куда удобно, например, в **$PREFIX/src/glibc**. Потом распакуйте архив дополнения в каталог, созданный, когда вы распаковывали основной архив **glibc**.

2. ***Конфигурирование***: самое главное - это установить системную переменную **CC**, которая запрашивает у системы сборки компилятор, который будет использоваться. Так происходит потому, что вы хотите выполнить кросс-компиляцию **glibc** используя собранный недавно кросс-компилятор. Для этого установите значение переменной CC в оболочке, используя команду:

   ```
   export CC=$TARGET-linux-gcc
   ```

   Например, если целевой платформой является MIPS,

   ```
   export CC= mips-linux-gcc
   ```

   Выполните команду конфигурирования с соответствующими параметрами:

   ```
   ./configure $TARGET --build=$NATIVE-$TARGET
   --prefix=$PREFIX --enable-add-ons=linux threads, crypt
   ```

   Например, если целевой платформой является MIPS,

   ```
   configure mips-linux --build=i686-linux --prefix=/usr/local/mips/ --enable-
   add-ons=linux threads, crypt
   ```

3. ***Сборка***: чтобы собрать **glibc**, вызовите **make** в каталоге **glibc**.
4. ***Установка***: чтобы установить **glibc**, вызовите **make install** в каталоге **glibc**.

Пересборка полнофункционального GCC

Повторите эти шаги сборки GCC для добавления дополнительной языковой поддержки. Вы должны также удалить уловку для сборки **Dinhibit_libc**, если вам пришлось ранее её применить. Также не забудьте убрать установку переменной окружения CC, сделанную для кросс-компиляции, чтобы для окончательной сборки вашего набора инструментов использовался родной компилятор.

После этого на вашей системе должны быть доступны все средства компиляции. Как с помощью данного инструментария собирать ядро и приложения объясняет Глава 8 240.

2.5.2 Сборка набора инструментов для MIPS

Ниже перечислены шаги, которые необходимы для создания набора инструментов для MIPS в качестве целевой платформы. Это должно быть использовано в качестве эталона для сборки для других платформ.

Перечисление каталогов исходных текстов

Мы используем

```
TARGET=mips-linux
PREFIX=/usr/local/mips
```

binutils - **/usr/local/mips/src/binutils**
gcc - **/usr/local/mips/src/gcc**
glibc - **/usr/local/mips/src/glibc**
исходные тексты ядра - **/usr/local/mips/linux/**

Всегда безопаснее создавать отдельный каталог для сборки и запуска оттуда настройки.

```
# cd /usr/local/mips/
# mkdir build
# cd build
# mkdir binutils
# mkdir gcc
# mkdir glibc
```

Сборка binutils

```
# cd /usr/local/mips/build/binutils/
# /usr/src/local/mips/src/binutils/configure
--target=mips-linux --prefix=/usr/local/mips
# make
# make install
```

Настройка заголовков ядра

```
#cd /usr/local/mips/linux
```

Откройте **Makefile** и укажите **ARCH:=mips**

```
#make menuconfig
```

Выберите подходящую платформу MIPS и выйдите с сохранением настроек:

```
#make dep
```

Сборка минимального GCC

```
# cd /usr/local/mips/src/gcc/gcc/config/mips
```

Откройте файл **t-linux** и измените строку **TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer –fPIC** на **TARGET_LIBGCC2_CFLAGS = -fomit-frame-pointer -fPIC -Dinhibit_libc –D__gthr_posix_h**.

```
# cd /usr/local/mips/build/gcc

#/usr/local/mips/src/gcc/configure --target=mips-linux
--host=i386-pc-linux-gnu --prefix=/usr/local/mips/
--disable-threads --enable-languages=c
```

```
#make
#make install
```

Сборка glibc

```
#cd /usr/src/build/glibc/
#/usr/src/glibc/configure mips-linux --build=i386-pc-linux-gnu
--prefix=/usr/local/mips/ --enable-add-ons=linuxthreads,crypt
--with-headers=/usr/local/mips/linux/include/linux

#make
#make install
```

Сборка GCC с поддержкой потоков и дополнительных языков

```
#cd /usr/local/mips/src/gcc/gcc/config/mips
```

Откройте файл **t-linux** и верните обратно изменения, сделанные для трюка **inhibit_libc**. В строке **TARGET_LIBGCC2_CFLAGS** укажите **TARGET_LIBGCC2_CFLAGS = -fomitframe-pointer –fPIC**.

```
#cd /usr/local/mips/build/gcc/
#rm -rf *

#/usr/local/mips/src/gcc/configure --target=mips-linux
--host=i386-pc-linux-gnu --prefix=/usr/local/mips

#make
#make install
```

Глава 3, Пакет поддержки платформы

BSP или "Board Support Package, Пакет Поддержки Платформы" это набор программного обеспечения, используемого для инициализации на плате аппаратных устройств и реализации относящихся к данной плате процедур, которые могут быть использованы ядром, а также драйверами устройств. BSP, таким образом, представляет собой уровень абстрагирования оборудования, присоединяющий оборудование к ОС, скрывая детали, относящиеся к процессору и плате. BSP скрывает относящиеся к плате и процессору детали от остальной части ОС, поэтому перенос драйверов между разными платами и процессорами становится чрезвычайно простым. Ещё одним термином, который часто используется вместо BSP, является Hardware Abstraction Layer, Уровень Абстрагирования Оборудования, или HAL. У пользователей UNIX более известным является HAL, в то время как сообщество разработчиков RTOS чаще использует BSP, особенно те, кто используют VxWorks. BSP имеет два компонента:

1. Поддержку микропроцессора: Linux имеет широкую поддержку всех ведущих процессоров на рынке встраиваемых систем, таких как MIPS, ARM, и вскоре ожидается PowerPC.
2. Процедуры, относящиеся к плате: типичный HAL для оборудования платы будет включать в себя:
   a. Поддержку загрузчика
   b. Поддержку карты памяти
   c. Системные таймеры
   d. Поддержку контроллера прерываний
   e. Часы реального времени, Real-Time Clock (RTC)
   f. Поддержку последовательных устройств (для отладки и консоли)
   g. Поддержку шины (PCI/ISA)
   h. Поддержку DMA
   i. Управление питанием

В этой главе не рассматривается перенос Linux на какой-либо микропроцессор или микроконтроллер, потому что это огромная тема сама по себе; переносу Linux на разные процессоры и микроконтроллеры должна быть посвящена отдельная книга. Вернее, эта книга предполагает, что читатель имеет плату на основе одного из уже поддерживаемых процессоров. Так что она целиком посвящена вопросам, относящимся к плате. Для прояснения терминологии, мы ссылаемся на HAL, как на уровень, который сочетает в себе программное обеспечение, относящееся к плате и процессору, а на BSP, как на уровень, который имеет только код, относящийся к данной плате. Поэтому, когда мы говорим о HAL для MIPS, это означает поддержку процессоров MIPS и плат с процессорами MIPS. Когда мы говорим о BSP, мы имеем в виду программное обеспечение, которое не имеет программного обеспечения поддержки процессора, а только дополнительное программное обеспечение для поддержки данной платы. HAL может быть понят как надмножество всех поддерживаемых BSP и дополнительно включает в себя программное обеспечение, относящееся к процессору.

Как уже упоминалось в Главе 2, ни HAL Linux, ни BSP, не имеет какого-либо стандарта. Следовательно, очень трудно объяснять HAL для нескольких архитектур. Эта глава погружает в BSP Linux и вопросы переноса для архитектуры на основе MIPS; где это необходимо, обсуждение может перекинуться на другие процессоры. Для упрощения мы используем вымышленную плату EUREKA на базе MIPS, имеющую следующий набор аппаратных компонентов:

- 32-х разрядный процессор MIPS
- 8 Мб SDRAM
- 4 Мб флеш-памяти
- Программируемый контроллер на основе 8259
- Шину PCI с такими подключенными к ней устройствами, как сетевая и звуковая карта
- Микросхему-таймер для генерации тактовой частоты системы
- Последовательный порт, который может быть использован для консоли и удалённой отладки

3.1 Включение BSP в процедуру сборки ядра

Исходный код HAL Linux находится в каталогах **arch/** и **include/<asm-XXX>** (XXX = имя процессора, такое как PowerPC, MIPS). Таким образом, **arch/ppc** будет содержать исходные файлы для плат на основе PPC, а **include/asm-ppc** будет содержать файлы заголовков для таких плат.

Внутри каталога для каждого процессора все платы на основе этого процессора также находятся в подкаталогах. Важными каталогами внутри каждого подкаталога являются:

- ***kernel***: этот каталог содержит относящиеся к процессору процедуры для инициализации, настройки линий прерываний, прерываний и процедуры программных прерываний.
- ***mm***: содержит зависимый от оборудования код настройки TLB (Translation LookUp Buffer, буфер быстрого преобразования адреса, аппаратную таблицу, используемую для перевода физического адреса в виртуальный и наоборот) и обработки исключений.

Например, HAL для MIPS имеет два подкаталога **arch/mips/kernel** и **arch/mips/mm**, которые содержат описанный выше код. Наряду с этими двумя каталогами существует множество других подкаталогов; это каталоги BSP, содержащие только код, относящийся к плате. Пользователь должен создать внутри каталога соответствующего процессора дерево подкаталогов, которое содержит файлы, необходимые для BSP. Следующий шаг заключается в интеграции BSP с процессом сборки, чтобы при сборке образа ядра можно было выбрать относящиеся к плате файлы. Это может требовать, чтобы процесс выбора компонентов ядра (выполненный с помощью команды **make menuconfig** во время сборки ядра) знал о плате. Почему этот шаг является необходимым? Помимо упрощения процедуры сборки, выполнение этого даёт дополнительные преимущества:

- Переключатели настроек часто делают много изменений определённой конфигурации. Некоторыми примерами таких настроек являются скорость процессора, скорость UART, и так далее. Вместо такой правки кода, как изменение файлов заголовков, все такие относящиеся к плате детали могут быть сделаны в виде параметров конфигурации и централизованно храниться в хранилище конфигураций (например, файле **.config**, используемом для сборки ядра); это делает процесс сборки ядра проще, а также позволяет избежать загромождения исходного кода.
- Часто поставщики OEM являются покупателями встроенного решения и они, возможно, захотят добавлять в ядро свои собственные компоненты. Они могут быть не заинтересованы в выборе компонентов ядра для платы, поставляемых вами; они могут захотеть, чтобы это было уже зафиксировано как часть процесса сборки. Это делается

путём добавления вашей платы как пункта конфигурации при сборке ядра; когда выбирается данная плата, все программные компоненты, необходимые для платы, автоматически становятся выбранными. OEM поставщику не нужно беспокоиться о подробностях вашей платы и о том, какие программные компоненты необходимы для её сборки.

Два вышеописанных шага могут быть осуществлены за счёт подключения BSP к процессу конфигурации. Компоненты ядра Linux выбираются с помощью команды **make config** (или **make menuconfig/make xconfig**). Сердцем процесса настройки является конфигурационный файл, находящийся внутри каталога используемого процессора. Более подробно это рассматривается в Главе 8 240. Например, для включения компонентов платы EUREKA в процесс сборки ядра вам необходимо отредактировать файл **arch/mips/config.in** (для ядра версии 2.4) или **arch/mips/Kconfig** (для ядра версии 2.6), как показано на Рисунке 3.1.

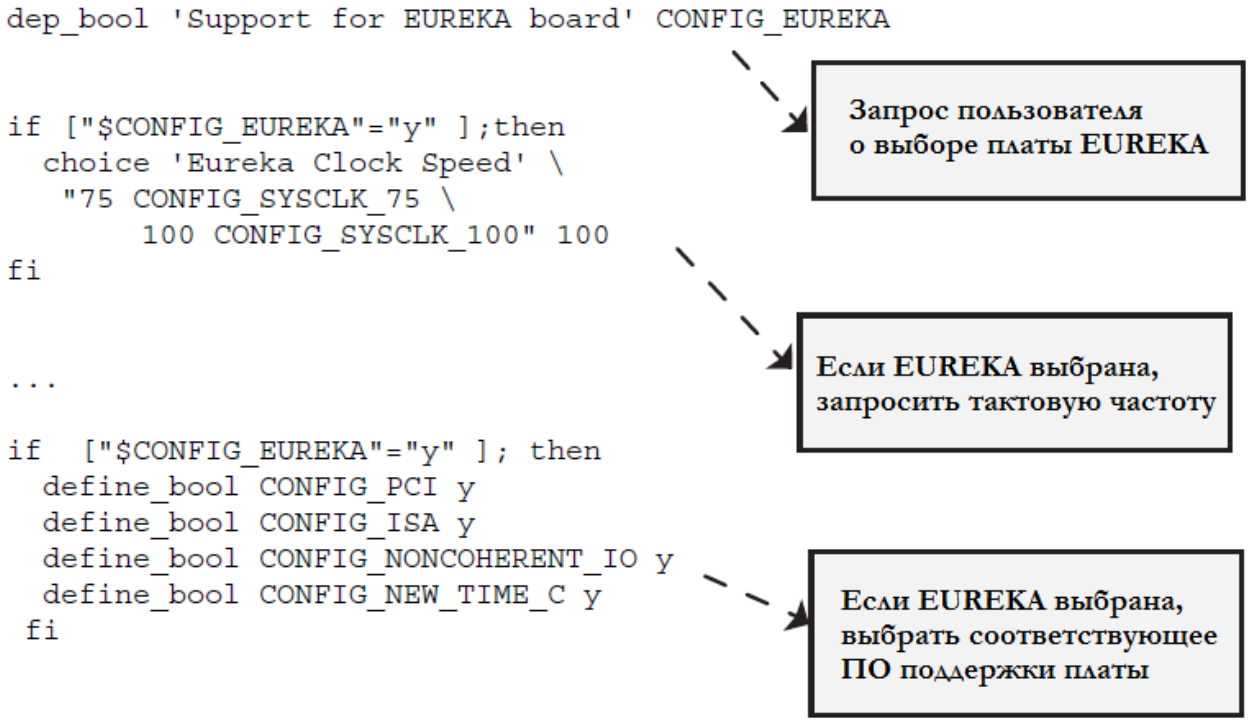

Рисунок 3.1 Опции сборки платы EUREKA.

Связующим звеном между конфигурацией и процессом сборки является **CONFIG_EUREKA**. Для приведённого выше примера в файл **arch/mips/Makefile** должны быть добавлены следующие строки:

```
ifdef CONFIG_EUREKA
LIBS        += arch/mips/eureka/eureka.o
SUBDIRS     += arch/mips/eureka
LOADADDR    := 0x80000000
endif
```

Последняя строка, **LOADADDR**, определяет начальный адрес ядра. Компоновщик забирает его, используя скрипт компоновщика, так что вы можете снова увидеть ссылку на этот адрес в скрипте компоновщика. Таким образом, когда пользователь выбрал во время

конфигурации плату EUREKA, список таких настроек, относящихся к данной плате, как тактовая частота, становится выбранным. Кроме того, когда ядро собрано, процесс сборки осведомлён о таких опциях сборки, относящихся к плате EUREKA, как подкаталоги, через которые он должен пройти для сборки программного обеспечения.

3.2 Интерфейс системного загрузчика

Загрузчик представляет собой часть программного обеспечения, которая начинает выполняться сразу после включения питания системы. Загрузчик является важной частью процесса разработки, а также одной из самых сложных. Большинство вопросов загрузки, зависящих от процессора и платы, вынесены за рамки обсуждения. Многие процессоры, такие как ARM, x86 и MIPS, при сбросе начинают выполнение с определённых векторов. Некоторые другие, такие как M68K, читают стартовый адрес из загрузочного ПЗУ. Таким образом, возникает такой вопрос, как может ли первым и единственным загруженным образом быть сам Linux, что исключает использование загрузчика. Устранение загрузчика и запись в ПЗУ ядра, которое загружает само себя, является подходом, предоставляемым многими RTOS, в том числе VxWorks, которая предоставляет процедуры инициализации загрузки, чтобы выполнить POST (Power-On Self-Test, самотестирование при включении питания), настроить сигналы выбора микросхем, проинициализировать память и кэши памяти, и переместить образ из ПЗУ в ОЗУ.

Большинство настроек при сбросе зависимы от платы и обычно производители плат предоставляют на плате PROM, которая выполняет всё вышеописанное. Лучше использовать для загрузки образа ядра или временного загрузчика PROM и таким образом спасти разработчиков от работы по программированию платы. Даже если PROM не доступна, лучше отделить процесс загрузки, чтобы затем загрузчик позволил ядру загрузиться. Преимущества такого подхода приведены ниже.

- Кроме загрузок из ROM, могут быть реализованы несколько методов загрузки ядра, такие как через последовательный порт (Kermit) или через сеть (TFTP).
- Это обеспечивает защиту от небезопасных перезаписей образа ядра в случае, когда образ ядра хранится во флеш-памяти. Предположим, что когда ядро обновлялось, было отключение электроэнергии; после этого плата находится в подвешенном состоянии. Безопасным способом будет поместить загрузчик в какие-то защищённые секторы флеш-памяти (обычно называемые загрузочными секторами) и оставить его неприкасаемым, чтобы всегда был доступен путь для восстановления.

По опыту, Linux всегда предполагает, что он выполняется из памяти (некоторые патчи для выполнения на месте [eXecute In Place, XIP] позволяют Linux выполняться непосредственно из ПЗУ; это обсуждается позже). Загрузчики являются независимыми частями программного обеспечения, которые должны быть собраны независимо от ядра Linux. Если ваша плата поддерживает PROM, загрузчик выполнит инициализацию процессора и платы. Поэтому загрузчик является сильно зависимым от платы и процессора. Функциональность загрузчика можно разделить на две части: обязательную и необязательную. Дополнительная функциональность загрузчика разнообразна и зависит от потребностей потребителя. Обязательными функциями загрузчика являются:

1. ***Инициализация оборудования***: это включает в себя процессор, внутренние контроллеры, такие как контроллер памяти, и аппаратные устройства, необходимые для загрузки ядра, такие как флеш-память.

2. ***Загрузка ядра***: необходимое программное обеспечение для загрузки ядра и копирование его в соответствующие ячейки памяти.

Ниже приведён список шагов, которым обычно следует любой загрузчик; это общие шаги и в зависимости от использования могут быть исключения. Обратите внимание, что процессоры x86 обычно поставляются с находящейся на плате BIOS, которая помогает с базовым включением и загрузкой вторичного загрузчика для загрузки операционной системы; так что следующий набор шагов предназначен для не x86 процессоров, таких как MIPS и ARM.

1. ***Загрузка***: большинство загрузчиков стартуют из флеш-памяти. Они выполняют начальную инициализацию процессора, такую как конфигурирование кэша, настройку некоторых основных регистров, и проверяют находящееся на плате ОЗУ. Также они запускают процедуры POST, чтобы сделать проверку оборудования, необходимую для процедуры загрузки, такую как проверка памяти, флеш-памяти, шин, и так далее.
2. ***Перемещение в памяти***: загрузчики перемещают себя в ОЗУ. Это выполняется, потому что оперативная память быстрее, чем флеш-память. Также шаг перемещения может включать в себя распаковку, так как загрузчики могут храниться в сжатом формате, чтобы сохранить дорогостоящее место для хранения.
3. ***Инициализация устройств***: затем загрузчик инициализирует основные устройства, необходимые для взаимодействия с пользователем. Это обычно означает настройку консоли, чтобы предоставить пользователю интерфейс. Он также инициализирует устройства, необходимые для запуска ядра (и, может быть, корневую файловую систему). Это может включать флеш-память, сетевую карту, USB, и так далее.
4. ***Пользовательский интерфейс***: после этого пользователю предоставляется интерфейс, чтобы он выбрал образ ядра для загрузки на плату. Здесь может быть указано время, предоставляемое пользователю для ввода своего выбора; в случае его превышения может быть загружен образ по умолчанию.
5. ***Загрузка образа***: загружается образ ядра. В случае, если пользователю был дан выбор для загрузки корневой файловой системы с помощью механизма **initrd**, в память также загружается образ **initrd**.
6. ***Подготовка загрузки ядра***: далее, в случае, если ядру должны быть переданы аргументы, заполняются аргументы командной строки и помещаются в известные для ядра Linux места.
7. ***Загрузка ядра***: наконец, выполняется перемещение ядра. Как только запускается ядро Linux, загрузчик становится больше не нужен. Обычно память, занимаемая им, забирается ядром; настройке карты памяти ядра необходимо позаботиться об этом.

Общую последовательности работы загрузчика показывает Рисунок 3.2.

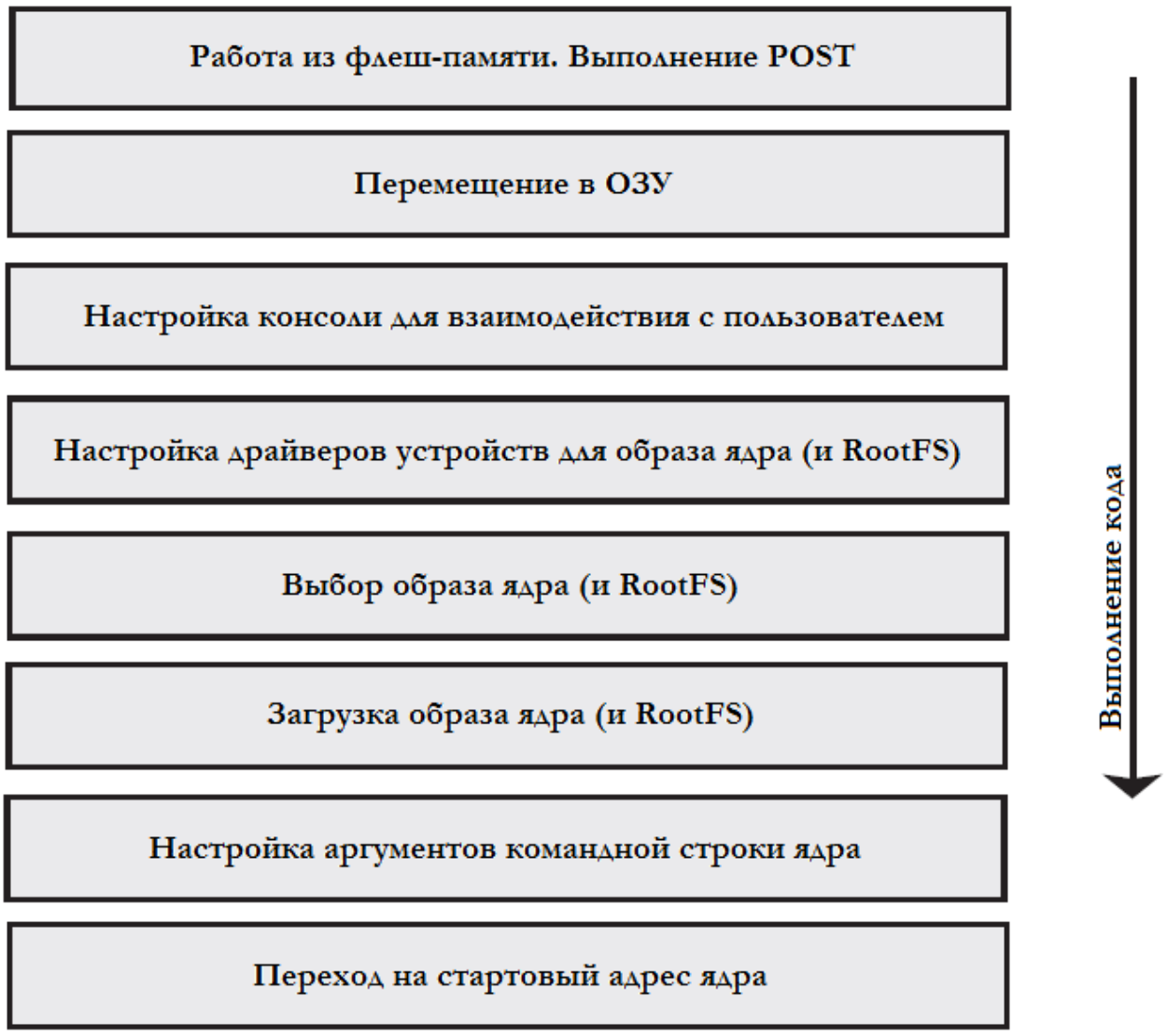

Рисунок 3.2 Последовательность запуска загрузчика.

Есть много загрузчиков для Linux в свободном доступе; системный архитектор может оценить существующие, прежде чем принять решение написать с нуля новый загрузчик. Каковы критерии при выборе загрузчика для данной встраиваемой платформы?

- ***Поддержка встроенного оборудования***: это должно быть главным критерием. Есть много загрузчиков для настольных компьютеров, таких как LILO, которые не могут быть использованы во встраиваемых системах из-за их зависимости от BIOS ПК. Тем не менее есть и доступны некоторые универсальные встраиваемые загрузчики: в частности, U-Boot и Redboot. Ниже показаны некоторые из неуниверсальных загрузчиков, доступных для наиболее часто используемых встраиваемых процессоров:
  – MIPS – PMON2000, YAMON
  – ARM – Blob, Angel boot, Compaq bootldr
  – X86 – LILO, GRUB, Etherboot
  – PowerPC – PMON2000
- ***Вопросы лицензирования***: они подробно рассмотрены в Приложении Б 367.
- ***Занимаемое в памяти место***: многие загрузчики поддерживают сжатие для

сохранения пространства флеш-памяти. Это может быть важным критерием, особенно когда хранится несколько образов ядра.
- ***Поддержка сетевой загрузки***: загрузка по сети может быть необходима, особенно для отладочных сборок. Большинство популярных загрузчиков поддерживают загрузку по сети и могут поддерживать популярные сетевые протоколы, связанные с загрузкой, такие как BOOTP, DHCP и TFTP.
- ***Поддержка загрузки из флеш-памяти***: загрузка из флеш-памяти состоит из двух компонентов, связанных с ней: программное обеспечение для чтения из флеш-памяти и программное обеспечение для чтения файловой системы. Последнее требуется в случае, если образ ядра хранится во флеш-памяти в файловой системе.
- ***Доступность консольного интерфейса***: консольный пользовательский интерфейс является практически обязательным для большинства современных загрузчиков. Консольный пользовательский интерфейс обычно предоставляет пользователю следующие варианты:
  – Выбор образа ядра и местоположения
  – Установка режима загрузки ядра (сеть, последовательный порт, флеш-память)
  – Настройка аргументов, которые передаются ядру
- ***Доступность обновления решений***: обновление решения требует наличия в загрузчике стирания флеш-памяти и программного обеспечения записи во флеш-память.

Ещё одной важной областью обсуждения загрузчиков является интерфейс от загрузчика к ядру, который включает в себя следующие компоненты:

- ***Передача аргументов от загрузчика ядру Linux***: ядру Linux, как и любому приложению, могут быть переданы аргументы в хорошо описанной форме, которые ядро анализирует, и либо само их обрабатывает, либо передаёт соответствующим драйверам или приложениям. Это очень мощное средство и оно может быть использовано для реализации обходных путей для некоторых аппаратных проблем. После того, как система полностью загрузилась, список аргументов ядра Linux во время загрузки может быть проверен чтением **proc** файла **/proc/cmdline**.
  – Передача аргументов команды загрузки: аргумент команды загрузки может иметь несколько значений, разделённых запятыми. Несколько аргументов должны быть разделены пробелами. Как только весь набор построен, загрузчик должен поместить его в известные ядру Linux адреса памяти.
  – Разбор аргументов команды загрузки: команда загрузки типа **foo** требует, чтобы в ядре была зарегистрирована функция **foo_setup()**. Ядро при инициализации перебирает каждый аргумент команды и вызывает соответствующую зарегистрированную функцию. Если функция не зарегистрирована, то аргумент используется как переменная окружения или передаётся задаче **init**.
- Некоторыми важными параметрами загрузки являются:
  – **root**: определяет имя устройства, которое будет использоваться в качестве корневой файловой системы.
  – **nfsroot**: Указывает сервер NFS, каталог и параметры, которые будут использоваться в качестве корневой файловой системы. (NFS является очень мощным шагом при сборке системы Linux на начальных стадиях.)
  – **mem**: задаёт объём памяти, доступный для ядра Linux.
  – **debug**: задаёт уровень отладки для печати сообщений на консоль.
- ***Карта памяти***: на многих платформах, особенно Intel и PowerPC, загрузчики создают карту памяти, которая может быть подхвачена ОС. Это позволяет легко переносить ОС на разные платформы. Подробнее карта памяти обсуждается в следующем разделе.

- ***Вызов из ядра подпрограмм в PROM***: на многих платформах загрузчик, который выполняется из PROM, можно рассматривать как библиотеку, так что могут быть сделаны вызовы из PROM. Например, на станции DEC на основе MIPS находящиеся в PROM процедуры ввода-вывода используются для реализации консоли.

3.3 Карта памяти

Карта памяти определяет схему адресного пространства процессора. Определение карты памяти является одним из наиболее важных этапов и должно быть сделано в начале процесса переноса. Карта памяти необходима по следующим причинам:

- Она фиксирует адресное пространство, выделенное для разных аппаратных компонентов, таких как ОЗУ, флеш-память и отображённые на память периферийные устройства ввода-вывода.
- Она отмечает распределение памяти на плате для разных программных компонентов, таких как загрузчик и ядро. Это имеет решающее значение для сборки программных компонентов; эта информация обычно передаётся во время сборки через скрипт компоновщика.
- Она определяет связь виртуального и физического адреса для данной платы. Это отображение сильно зависит от процессора и платы; это отображение определяет конструкция различных контроллеров встроенной памяти и шин на плате.

Есть три адреса, которые видны во встраиваемой системе на Linux:

- ***Не транслируемый процессором или физический адрес***: это адрес, который виден на реальной шине памяти.
- ***Транслируемый процессором адрес или виртуальный адрес***: это диапазон адресов, который распознаётся процессором как действительный диапазон адресов. Например, основной распределитель памяти ядра **kmalloc()** возвращает виртуальный адрес. Для перевода в физический адрес виртуальный адрес проходит через MMU.
- ***Адрес шины***: это адрес памяти с точки зрения устройства, кроме процессора. Этот адрес может меняться в зависимости от шины.

Карта памяти связывает схему памяти системы с точки зрения процессора, устройства памяти (ОЗУ, флеш-память и так далее) и внешние устройства; эта карта показывает, как должны взаимодействовать устройства, имеющие разные взгляды на адресное пространство. На большинстве платформ шинный адрес совпадает с физическим адресом, но это не является обязательным. Для уверенности, что драйверы устройств переносимыми между всеми платформами, ядро Linux предоставляет макросы.

Определение карты памяти требует понимание следующего:

- Понимание адресации памяти и ввода-вывода аппаратными компонентами на плате. Это часто требует понимания того, как сконфигурированы контроллеры шин памяти и ввода-вывода.
- Понимание того, как процессор выполняет управление памятью.

Создание карты памяти для системы может быть разбито на следующие задачи:

- ***Карта памяти процессора***: это первая карта памяти, которая должна быть создана. Она объясняет политики управления памятью процессора, например, как процессор

обрабатывает различные адресные пространства (пользовательский режим, режим ядра), каковы политики кэширования для различных областей памяти, и так далее.

- ***Карта встроенной памяти***: раз есть представление о том, как процессор видит различные области памяти, следующим шагом будет соотнести различные встроенные устройства с областью памяти процессора. Это требует понимания работы различных встроенных устройств и контроллеров шины.
- ***Карта памяти программного обеспечения***: затем следует выделить часть памяти для различных компонентов программного обеспечения, таких как загрузчик и ядро Linux. Ядро Linux настраивает свою собственную карту памяти и решает, где будут располагаться различные разделы ядра, такие как код и "куча".

В следующих разделах подробно объясняется каждая из этих карт памяти по отношению к плате EUREKA.

3.3.1 Карта памяти процессора — модель памяти MIPS

Адресное пространство процессора для 32-х разрядных процессоров MIPS (4 Гб) разделено на четыре области, как это показано на Рисунке 3.3.

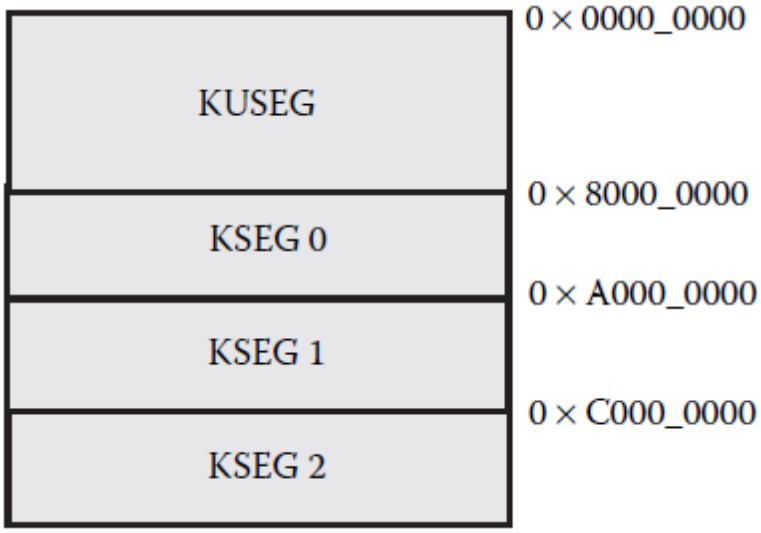

Рисунок 3.3 Карта памяти MIPS.

- ***KSEG0***: диапазон адресов этого сегмента от 0x80000000 до 0x9fffffff. Эти адреса отображаются на 512 Мб физической памяти. Трансляция виртуального в физический адрес происходит простым выключением самого старшего бита виртуального адреса. Это адресное пространство всегда идёт в кэш и оно должно быть создано только после того, как кэш правильно проинициализирован. Также это адресное пространство не доступно через TLB; поэтому ядро Linux использует это адресное пространство.
- ***KSEG1***: адресный диапазон этого сегмента от 0xA0000000 до 0xBFFFFFFF. Эти адреса также отображаются на 512 Мб физической памяти; они отображаются путём сбрасывания последних трёх битов виртуального адреса. Тем не менее, разница между KSEG0 и KSEG1 в том, что KSEG1 не использует кэш. Следовательно, это адресное пространство используется сразу после сброса, когда системные кэши находятся в неопределённом состоянии. (Вектор сброса MIPS 0xBFC00000 лежит в этом диапазоне адресов.) Также это адресное пространство используется для привязки периферии ввода-вывода, поскольку оно обходит кэш.

- ***KUSEG***: адресное пространство этого сегмента от 0x00000000 до 0x7FFFFFFF. Это адресное пространство, выделенное для пользовательских программ. Они транслируются в физические адреса через TLB.
- ***KSEG2***: адресное пространство этого сегмента от 0xC0000000 до 0xFFFFFFFF. Это адресное пространство, которое доступно через ядро, но получает трансляцию через TLB.

3.3.2 Карта памяти платы

Ниже приведена карта памяти, предназначенная для платы EUREKA, которая имеет 8 Мб встроенной SDRAM, 4 Мб флеш-памяти и устройства ввода-вывода, которые требуют 2-х Мб диапазона карты памяти.

- с 0x80000000 до 0x80800000: используется для отображения 8 Мб SDRAM
- с 0xBFC00000 до 0xC0000000: используется для отображения 4 Мб флеш-памяти
- с 0xBF400000 до 0xBF600000: используется для отображения 2 Мб периферии ввода-вывода

3.3.3 Карта памяти программного обеспечения

8 Мб SDRAM становится доступным для работы как загрузчика, так и ядра Linux. Обычно загрузчик заставляют работать с конца доступной памяти, так что как только он передаёт управление ядру Linux, оно может легко вернуть себе эту память. Если это не так, то вам, возможно, придётся использовать некоторые уловки, чтобы освободить память загрузчика, если память загрузчика не является смежной с адресным пространством Linux или если его адресное пространство располагается перед адресным пространством ядра. Настройка карты памяти Linux разделена на четыре этапа:

- Схема памяти ядра Linux - скрипт компоновщика
- Распределитель памяти при загрузке
- Создание отображения памяти и ввода-вывода в виртуальное адресное пространство
- Создание ядром различных зон распределения памяти

Схема памяти ядра Linux

Схема памяти ядра Linux задаётся в момент сборки ядра с помощью файла сценария компоновщика **ld.script**. Для архитектуры MIPS сценарием компоновщика по умолчанию является **arch/mips/ld.script.in**; сценарий компоновщика, предоставляемый платформой, может его переопределить. Сценарий компоновщика написан с использованием командного языка компоновщика и описывает, как должны быть упакованы различные разделы ядра и какие адреса необходимо им дать. Образец сценария компоновщика, который определяет следующее распределение памяти, можно найти в Распечатке 3.1 40:

- Текстовый раздел начинается с адреса 0x8010_0000. Все другие разделы неизменно следуют за текстовым разделом. Началом текстового раздела является адрес **_ftext**, а концом текстового раздела является адрес **_etext**.
- После **_etext** выровненное по 8 Кб пространство отведено для присвоения дескриптора процесса и стека для процесса 0, называемого также процессом подкачки.
- Следующим за ним является пространство для раздела **init**. Адреса **_init_begin** и

_init_end обозначают начало и конец этих разделов, соответственно.

- После этого инициализируется раздел данных ядра. Символы **_fdata** и **_edata** обозначают начало и конец этих разделов, соответственно.
- Последний раздел представляет собой неинициализированные данные ядра, или BSS. В отличие от других разделов, этот раздел не является частью образа ядра, а только пространством, используемым им и заданным с помощью адресов **_bss_start** и **_end**. Процедура запуска ядра использует эти символы, чтобы получить диапазон адресов BSS и обнулить пространство BSS.

Рисунок 3.4 показывает расположение разделов ядра, как это определено сценарием компоновщика.

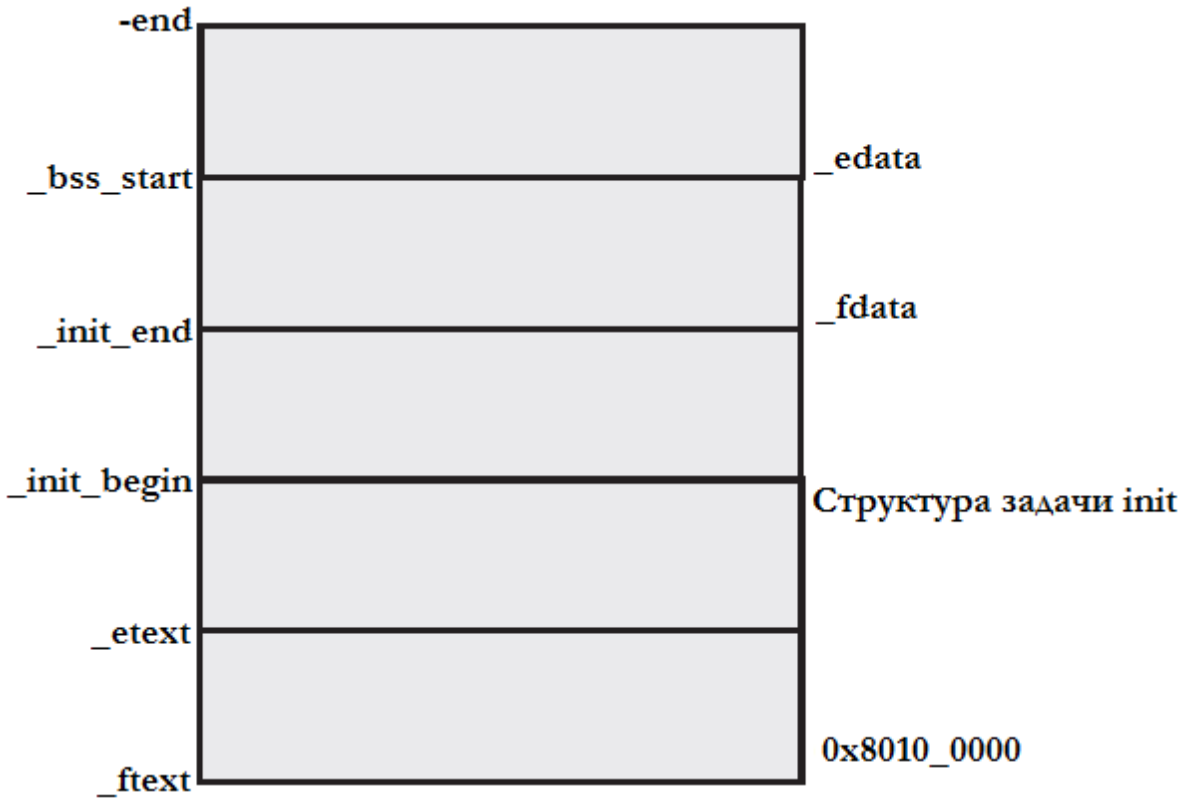

Рисунок 3.4 Разделы образа ядра.

Распределители загрузочной памяти

Распределители загрузочной памяти являются динамическими распределителями памяти ядра на ранних стадиях запуска ядра (до настройки подкачки); после настройки подкачки за динамическое распределение памяти несут ответственность зонные распределители. Распределители загрузочной памяти вызываются из функции **setup_arch()**. Распределитель загрузочной памяти работает с картой памяти платы, которая обычно передаётся загрузчиком; это обычная процедура для серии x86, и именно поэтому мы рассмотрим карту памяти, которая передаётся загрузчиком в ПК на Pentium®.

```
0000000000000000 - 000000000009f800 (используется)
000000000009f800 - 00000000000a0000 (зарезервировано)
00000000000e0000 - 0000000000100000 (зарезервировано)
```

```
0000000000100000 - 000000001f6f0000 (используется)
000000001f6f0000 - 000000001f6fb000 (данные ACPI)
000000001f6fb000 - 000000001f700000 (ACPI NVS)
000000001f700000 - 000000001f780000 (используется)
000000001f780000 - 0000000020000000 (зарезервировано)
00000000fec00000 - 00000000fec10000 (зарезервировано)
00000000fee00000 - 00000000fee01000 (зарезервировано)
00000000ff800000 - 00000000ffc00000 (зарезервировано)
00000000fffffc00 - 0000000100000000 (зарезервировано)
```

Ядро начинает использовать память с физического адреса 1 Мб. Так что распределитель загрузочной памяти резервирует следующие регионы:

- ***От 640 К до 1 Мб***: они зарезервированы для ПЗУ видео и расширения.
- ***Свыше 1 Мб***: они зарезервированы для кода ядра и разделов данных.

Настройка связи памяти и ввода-вывода

Это должно быть сделано на процессорах, которые не имеют включённой при сбросе виртуальной памяти. Примерами таких процессоров являются Intel и PowerPC. С другой стороны, MIPS работают после сброса в среде виртуальной памяти. Для Intel и PowerPC виртуальные отображения памяти должны быть настроены так, чтобы они могли обращаться к памяти и отображаемому на память вводу-выводу. Например, на PowerPC во время ранней стадии инициализации ядра виртуальный адрес связан с физическим адресом один к одному. Как только карта памяти стала известной, настраивается таблица виртуального отображения.

Настройка зонных распределителей

Зонные распределители делят память на три зоны:

- ***Зона DMA***: эта зона предназначена для распределения памяти, которую можно использовать для передач DMA. Для платформ MIPS и PowerPC в зону DMA добавляются вся динамическая нижняя память. Однако, на ПК на базе i386™ она имеет максимальный размер 16 Мб из-за ограничений адресации памяти на шине ISA.
- ***Зона NORMAL***: распределители памяти ядра пытаются выделять память из этой зоны, либо откатываются обратно к зоне DMA.
- ***Зона HIGHMEM***: многие процессоры не могут получить доступ ко всей физической памяти из-за небольшого размера линейного адресного пространства. Эта зона используется для отображения такой памяти. Она обычно не используется на встраиваемых системах и больше используется на настольных компьютерах и серверах.

Распечатка 3.1 Пример скрипта компоновщика

Распечатка 3.1

```
OUTPUT_ARCH(mips)
ENTRY(kernel_entry)
SECTIONS
{
```

```
/* Только читаемые секции, объединённые в текстовый сегмент: */
. = 0x80100000;
.init : { *(.init) } = 0

.text :
{
  _ftext = . ;                /* Начало текстового сегмента */
  *(.text)
  *(.rodata)
  *(.rodata.*)
  *(.rodata1)
  /* секции .gnu.warning специально обрабатываемые elf32.em. */
  *(.gnu.warning)
} =0

.kstrtab : { *(.kstrtab) }

. = ALIGN(16);                /* Таблица запуска */
__start___ex_table = .;
__ex_table : { *(__ex_table) }
__stop___ex_table = .;

__start___dbe_table = .;
__dbe_table : { *(__dbe_table) }
__stop___dbe_table = .;

__start___ksymtab = .;        /* Таблица символов ядра */
__ksymtab : { *(__ksymtab) }
__stop___ksymtab = .;

_etext = .;                   /* Конец текстового сегмента */

. = ALIGN(8192);
.data.init_task : { *(.data.init_task)

. = ALIGN(4096);
__init_begin = .;             /* Начало кода запуска */
.text.init : { *(.text.init) }
.data.init : { *(.data.init) }

. = ALIGN(16);
__setup_start = .;
.setup.init : { *(.setup.init) }
__setup_end = .;

__initcall_start = .;
.initcall.init : { *(.initcall.init) }
__initcall_end = .;

. = ALIGN(4096); /* Выравнивание двойной страницы для init_task_union */
__init_end = .;  /* Конец кода запуска */

. = .;
.data :
```

```
  {
    _fdata = . ;        /* Начало сегмента данных */
    *(.data)
    /* Выравнивание для начального образа электронного диска (INITRD) */
    . = ALIGN(4096);

    __rd_start = .;
    *(.initrd)
    . = ALIGN(4096);
    __rd_end = .;

    CONSTRUCTORS
  }

  .data1   : { *(.data1) }
  _gp = . + 0x8000;
  .lit8 : { *(.lit8) }
  .lit4 : { *(.lit4) }
  .ctors          : { *(.ctors) }
  .dtors          : { *(.dtors) }
  .got            : { *(.got.plt) *(.got) }
  .dynamic        : { *(.dynamic) }
  .sdata     : { *(.sdata) }

  . = ALIGN(4);
  _edata = .;           /* Конец сегмента данных */
  PROVIDE (edata = .);

  __bss_start = .;      /* Начало сегмента bss */
  _fbss = .;
  .sbss      : { *(.sbss) *(.scommon) }
  .bss       :
  {
    *(.dynbss)
    *(.bss)
    *(COMMON)
    . = ALIGN(4);
    _end = . ;          /* Конец bss */
    PROVIDE (end = .);
  }
}
```

3.4 Управление прерываниями

Каждая плата с её оборудованием управления прерываниями уникальна, в основном из-за интерфейса PIC (Programmable Interrupt Controller, программируемый контроллер прерываний). В этом разделе подробно описаны этапы программирования контроллера прерываний в Linux. Прежде чем углубляться в подробности программирования контроллера прерываний, познакомимся с основными функциональными возможностями PIC:

- Микропроцессор, как правило, имеет ограниченное число прерываний, которых может не хватать, если на плате есть много устройств, и всем им необходимо прерывать процессор. В таком случае на помощь приходит контроллер прерываний. Он расширяет

возможности прерываний процессора, позволяя мультиплексировать на одной линии много прерываний.

- PIC обеспечивает аппаратное управление приоритетом прерываний. Это может быть полезной функцией в случае, если сам процессор не поддерживает аппаратно приоритеты прерываний. Когда процессор обрабатывает прерывание высокого приоритета, PIC не доставляет процессору прерывания более низкого приоритета. Также, когда два устройства одновременно вызывают прерывание, PIC будет проверять регистр приоритета, чтобы распознать прерывание с более высоким приоритетом, а затем доставить его в процессор.
- Выполнение преобразований при переключениях. Оборудование обеспечивает прерывания двумя способами: уровнем и переключением уровня. Прерывания при переключении происходят на переходе сигнала из одного состояния в другое (как правило, от высокого к низкому); обычно это короткий импульс, указывающий на возникновение прерывания. Прерывание, работающее по уровню, с другой стороны, удерживают линию прерывания на высоком уровне, пока процессор не выключит её. Прерывания, вызываемые переключением, это старый метод прерываний и используется архитектурами шины ISA. Тем не менее, основным недостатком прерываний при переключении является то, что они не позволяют совместное использование прерываний. Срабатывания по уровню позволяют разделять прерывания, но должны использоваться осторожно, потому что неправильно спроектированный обработчик прерывания может привести к зависанию системы навсегда в цикле обработки прерываний. Некоторые процессоры позволяют сконфигурировать обработку прерываний или по переходу, или по уровню, в то время как другие процессоры распознают только прерывания по уровню. Если устройство, генерирующее прерывания переключением уровня, подключено к последнему множеству, то прерывания при переходе будут рассматриваться как ложные прерывания, поскольку причина прерывания будет снята до начала обработки. В таком случае будет удобен PIC, потому что PIC может перевести прерывания по переходу в прерывания по уровню, запоминая входное прерывание.

В качестве примера PIC для понимания вопросов программирования мы возьмём 8259A. Почему 8259A? Потому что это очень популярное и мощное аппаратное обеспечение и есть много вариантов его использования на встраиваемых платах. Прежде чем мы сможем понять подключение 8259A в BSP, нам требуется общее представление о 8259A. Основные функции контроллера 8259A:

- Он имеет восемь контактов прерываний. При использовании каскадного режима он может быть использован для обслуживания до 64-х прерываний.
- Он может быть запрограммирован для различных режимов:
  – Полностью вложенный режим, который вводится сразу после инициализации. В этом режиме приоритеты являются уникальными и неизменными. В этом режиме прерывание с более высоким приоритетом может прервать процессор, когда он обслуживает более низко приоритетное прерывание.
  – Режим автоматической ротации для поддержки прерываний, имеющих одинаковый приоритет.
  – Режим настраиваемой ротации, когда приоритет прерывания можно запрограммировать, изменяя самый низкий приоритет и тем самым назначать все другие приоритеты.
- PIC может быть запрограммирован для работы по переходу или срабатывания по уровню.
- Каждое прерывание может быть замаскировано.

В обработку прерываний 8259A вовлечены 3 регистра: IRR или interrupt request register, регистр запроса прерывания, IMR или interrupt mask register, регистр маски прерываний, и ISR или interrupt service register, регистр обслуживания прерываний. Когда внешнее устройство требует прерывания, устанавливается соответствующий бит в IRR. Если прерывание не замаскировано, что зависит от того, установлен ли бит в IMR, прерывание поступает в логику арбитража приоритета. Она проверяет содержимое регистра ISR, чтобы выяснить, обслуживается ли в настоящее время прерывание с более высоким приоритетом; если нет, поднимается флаг прерывания, чтобы процессор увидел прерывание. Процессоры x86 формируют цикл INTA, с тем, чтобы PIC управлял вектором прерывания на шине; однако процессоры не имеют такой функциональности в оборудовании; это явным образом должно сделать программное обеспечение. Бит ISR остаётся установленным, пока не выдан сигнал EOI (End Of Interrupt, Завершение Прерывания). Если PIC сконфигурирован для автоматического режима завершения прерывания, то подтверждение прерывания само вызывает очистку данного бита.

Устройство может быть подключено непосредственно к процессору и в этом случае оно прерывает процессор по одной из линий прерываний процессора; в противном случае его прерывания могут маршрутизироваться через PIC. Но маршрут захваченного прерывания не должен быть заботой программного драйвера устройства. BSP должен экранировать драйвер от фактической маршрутизации прерываний. Например, Рисунок 3.5 показывает сетевую карту, подключенную непосредственно через PIC к процессору MIPS (MIPS поддерживает шесть входов прерываний).

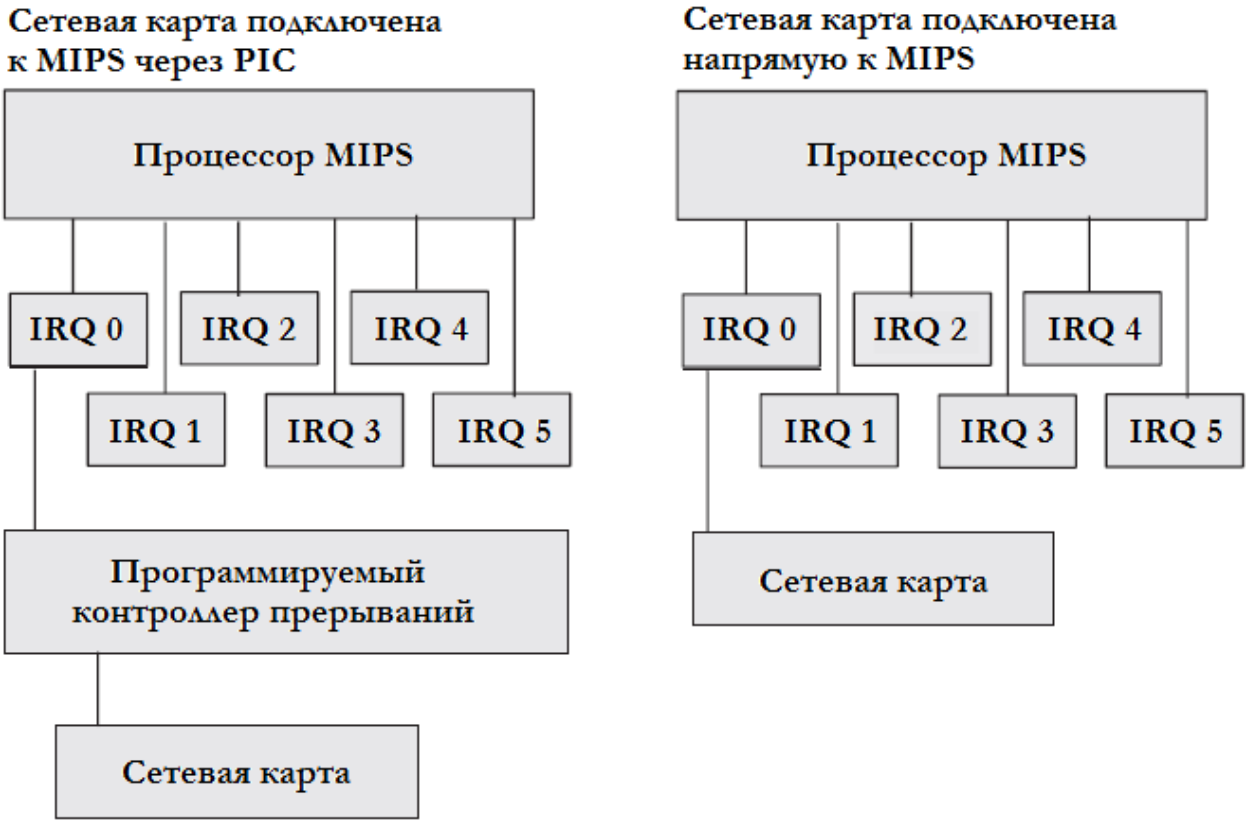

Рисунок 3.5 Подключения прерывания сетевой карты.

Ядро Linux обрабатывает все прерывания как логические прерывания; логические прерывания непосредственно подключены к процессору или могут идти через PIC. В обоих

случаях, когда прерывание зарегистрировано (через функцию **request_irq()**), драйвер устройства должен передать номер прерывания как аргумент, который определяет номер прерывания; передаваемый номер прерывания также является логическим номером прерывания. Количество логических прерываний варьируется в зависимости от процессора; для процессора MIPS оно определено как 128. За связь логических прерываний с реальными физическими прерываниями ответственнен BSP. Таким образом, в приведённом выше примере в обоих случаях драйвер может использовать один и тот же номер прерывания, но как номер прерывания маршрутизируется на реальном оборудовании решает BSP; это упрощает переносимость драйверов устройств.

Основой интерфейса прерываний BSP являются две структуры данных:

- ***Описание контроллера прерываний*** **hw_interrupt_type**: эта структура задекларирована в **include/linux/irq.h**. Любой механизм аппаратного контроля прерываний должен использовать эту структуру. Важными полями этой структуры данных являются:
  – **start-up**: указатель на функцию, которая вызывается, когда прерывания зондируются или когда они запрашиваются (с использованием функции **request_irq()**)
  – **shutdown**: указатель на функцию, которая вызывается, когда прерывание освобождается (с использованием функции **free_irq()**)
  – **enable**: указатель на функцию, которая включает линию прерывания
  – **disable**: указатель на функцию, которая выключает линию прерывания
  – **ack**: указатель на зависимую от контроллера функцию, которая подтверждает прерывания
  – **end**: указатель на функцию, которая вызывается после того, как прерывание было обслужено
- ***Описание прерывания*** **irq_desc_t**: оно также задекларировано в **include/linux/irq.h**. С помощью этой структуры определяется каждое логическое прерывание. Важными полями этой структуры данных являются:
  – **status**: состояние источника прерываний
  – **handler**: указатель на описание контроллера прерываний (описано выше)
  – **action**: список событий прерывания

Использование этих структур данных может быть лучше всего объяснено на примере. Рисунок 3.6 показывает архитектуру прерываний на плате EUREKA.

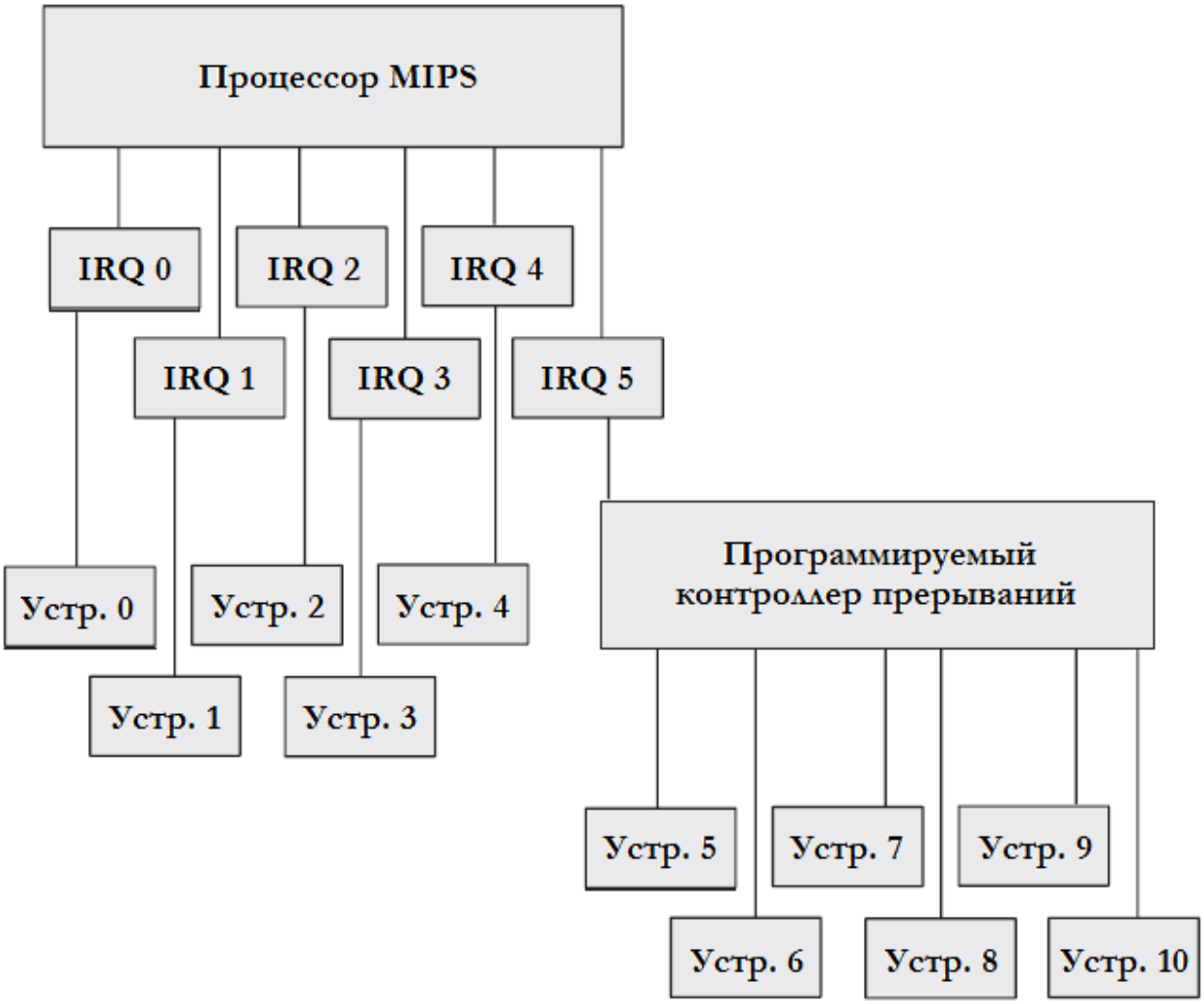

Рисунок 3.6 Подключения прерываний на плате EUREKA.

MIPS поддерживает шесть аппаратных прерываний. На плате на Рисунке 3.6 пять из них подключены непосредственно к аппаратным устройствам, в то время как шестое прерывание подключено к PIC, который в свою очередь используется для подключения ещё пяти аппаратных устройств. Таким образом, на плате есть десять источников прерываний, которые должны быть связаны с десятью логическими прерываниями для десяти устройств для использования драйверами устройств. Шаги, которые должны быть сделаны BSP для реализации логических прерываний:

- Создание **hw_interrupt_type** для прерываний, которые подключены непосредственно к процессору. Назовём это **generic_irq_hw**.
- Создание **hw_interrupt_type** для PIC. Назовём это **pic_irq_hw**.
- Определение десяти структур **irq_desc_t**, соответствующих десяти логическим прерываниям. Первые пять привязываем к полю **handler** в **generic_irq_hw**, а последние пять привязываем к **pic_irq_hw**.
- Написание кода запуска прерывания, который вызывается, когда процессор прерывается. Эта процедура должна проверять состояние регистра прерываний MIPS, а затем регистр состояния PIC, чтобы оценить, было ли прерывание прямым прерыванием, или же было смаршрутизировано через PIC. Если прерывание относится

к PIC, код должен прочитать регистр состояния PIC и узнать логический номер прерывания. Затем должна быть вызвана функция универсального обработчика прерывания **do_IRQ()** с логическим номером прерывания. Она будет выполнять соответствующую обработку PIC до и после вызова фактического обработчика.

Каждая поддержка контроллера прерываний требует вызовов BSP. Следующие функции, заполняющие **hw_interrupt_type**, были описаны выше, и вызываются на различных этапах обработки прерываний.

- ***Процедура инициализации***: она должна быть вызвана один раз (в функции **init_IRQ ()**). Функция выдаёт команды ICW (Initialization Command Word, Слова Команд Инициализации), которые должны быть выполнены до того, как 8259 сможет обрабатывать и принимать запросы прерываний.
- ***Процедура запуска***: эта функция вызывается, когда запрошено прерывание или когда прерывание зондируется. В 8259 эта процедура просто разрешает прерывание, демаскируя его в PIC.
- ***Процедура выключения***: это дополнение процедуры запуска; она запрещает прерывание.
- Процедура включения: эта процедура демаскирует указанное прерывание в 8259. Она вызывается из функции ядра **enable_irq()**.
- ***Процедура запрета***: эта процедура устанавливает бит в IMR. Эта функция вызывается из функции ядра **disable_irq()**.
- ***Процедура подтверждения***: процедура подтверждения вызывается на начальной стадии обработки прерывания. Когда происходит прерывание, последующие срабатывания этого же прерывания запрещены до тех пор, пока работает обработчик прерывания. Это сделано для предотвращения проблемы повторного входа для ISR. Так что эта процедура маскирует прерывание, которое обслуживается в настоящее время. Кроме того, для 8259, если не установлен режим автоматического завершения прерывания, эта процедура посылает в PIC команду EOI.
- ***Процедура завершения прерывания***: она вызывается на заключительной стадии обработки прерывания. Эта процедура должна разрешить прерывание, которое было запрещено в процедуре подтверждения.

3.5 Подсистема PCI

Архитектура PCI на Linux имеет свои корни в модели x86. Linux предполагает, что за настройку каждого PCI устройства отвечает BIOS или системное ПО, так что эти ресурсы (ввод-вывод, память и прерывания) уже выделены. В то время, когда драйвер устройства обращается к устройству, его области памяти и ввода-вывода должны быть переведены в адресное пространство процессора. Многие из плат не приходят с BIOS или прошивкой, которая выполняет инициализацию PCI. Даже если это выполняется, диапазон адресов, предложенный устройствам, может не соответствовать точно требованиям Linux. Таким образом, обязанностью BSP становится делать зондирование и конфигурирование устройств PCI. Мы обсуждаем BSP для устройств PCI на базе MIPS.

3.5.1 Уникальность архитектуры PCI

Linux определяет три объекта PCI: шину, устройство и функцию. В системе может быть до 256 шин, каждая шина имеет 32 слота, которые могут содержать устройства. Устройства могут быть одно или многофункциональными. Несколько шин PCI связаны между собой

через мост PCI. Подключение подсистемы PCI на плате может быть уникальным; это, в свою очередь, может сделать архитектуру PCI зависимой от платы. Некоторые из этих особенностей могут проистекать из следующего:

Вопросы карты памяти

Часть оборудования, которая подключает шину процессора к шине PCI называется ***северным мостом***. Северный мост имеет встроенный контроллер памяти, так что он может обращаться к памяти процессора. Кроме того, некоторые северные мосты имеют возможность сделать адресное пространство PCI устройства частью адресного пространства процессора; они делают это захватывая адреса на шине процессора и выдавая циклы чтения/записи PCI. На платформе ПК северный мост имеет такую возможность и, следовательно, адресное пространство PCI отображается в адресное пространство процессора.

Тем не менее, не исключено, что возможность отобразить адресное пространство PCI в виртуальное адресное пространство процессора доступна не на всех платах. Драйверы устройств Linux принимают это во внимание и, следовательно, ни один из драйверов не обращается к вводу-выводу и памяти PCI через прямые указатели; скорее, они используют команды вида **inb()/outb()**. Они могут транслироваться к прямым ссылкам на память в случае, если PCI отображается непосредственно в виртуальное адресное пространство процессора. И MIPS, и PowerPC позволяют пространству PCI быть отображённым в адресное пространство процессора, при условии, что плата поддерживает это. В таком случае BSP должен предоставить ***начало ввода-вывода***; это начальный адрес в виртуальной карте процессора для доступа к устройствам PCI. Рассмотрим, например, схему платы, показанную на Рисунке 3.7, где мост PCI взаимодействует с процессором через FPGA и двухпортовое ОЗУ. FPGA предоставляет специальные регистры для начала выполнения конфигурации, работы с памятью и вводом-выводом. Такая плата имеет две аномалии по сравнению с обычными платами и, следовательно, два последствия для BSP:

- Память и адресное пространство ввода-вывода PCI не может быть отображено на адресное пространство процессора напрямую, потому что операция по вводу-выводу и работы с памятью требует программирования FPGA. Так что BSP должен обеспечить процедуры для выполнения операций с памятью и вводом-выводом на шине PCI.
- Используется двухпортовая оперативная память, так что PCI устройства могут выполнять DMA к памяти. Но поскольку основная память не доступна для контроллера PCI, драйверам PCI устройств, требующим способную к DMA память, должна быть обеспечена память в пределах области двухпортовой оперативной памяти.

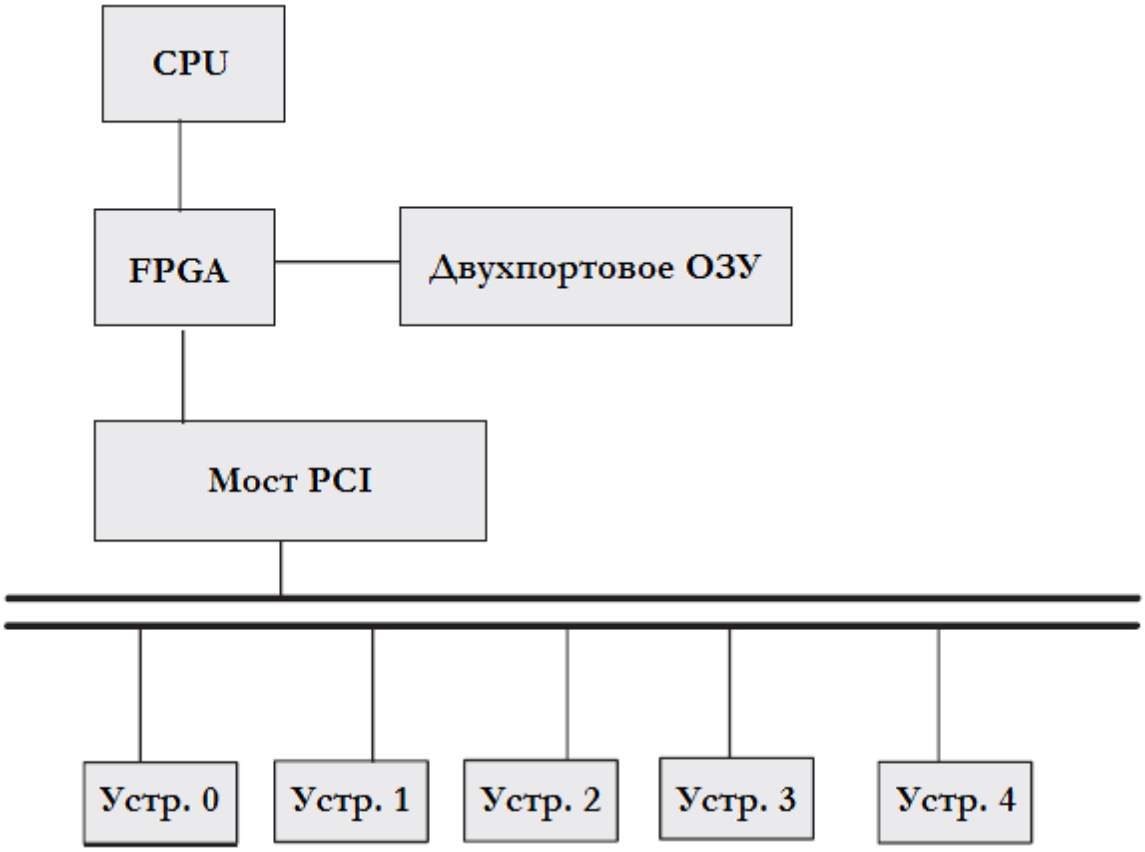

Рисунок 3.7 Подключение PCI через FPGA.

Доступ к пространству конфигурации

Каждое устройство PCI имеет пространство конфигурации, которое должно быть прочитано и запрограммировано перед тем, как устройство действительно может быть использовано. Процессор не имеет прямого доступа к пространству конфигурации, а зависит для выполнения этого от контроллера PCI. Контроллер PCI обычно обеспечивает регистры, которые должны быть запрограммированы для конфигурирования устройства. Поскольку это зависит от платы, процедуры для этого должен предоставить BSP.

Маршрутизация прерываний на плате

Оборудование PCI предоставляет четыре логических прерывания, A, B, C, и D, которые должны быть жёстко прописаны в каждом устройстве PCI. Эта информация хранится в поле ***контакт прерывания*** в заголовке конфигурации. То, как логические прерывания PCI на самом деле подключены к линиям прерываний процессору, зависит от платы. Пространство конфигурации имеет ещё одно поле, называемое ***линией прерывания***, которая должна быть заполнена фактической линией прерывания, которую использует устройство PCI. BSP должен просмотреть заголовок конфигурации для информации о контакте прерывания, а затем исходя из карты маршрутизации прерываний заполнить поле ***линия прерывания***. Например, на Рисунке 3.8 все контакты, помеченные как A и C, подключены к IRQ0, а контакты, помеченные как B и D, подключены к IRQ1. Поэтому BSP должен запрограммировать первый набор карт (Устройство A и Устройство C) на линию IRQ0, а последний на линию IRQ1 (Устройство B и Устройство D).

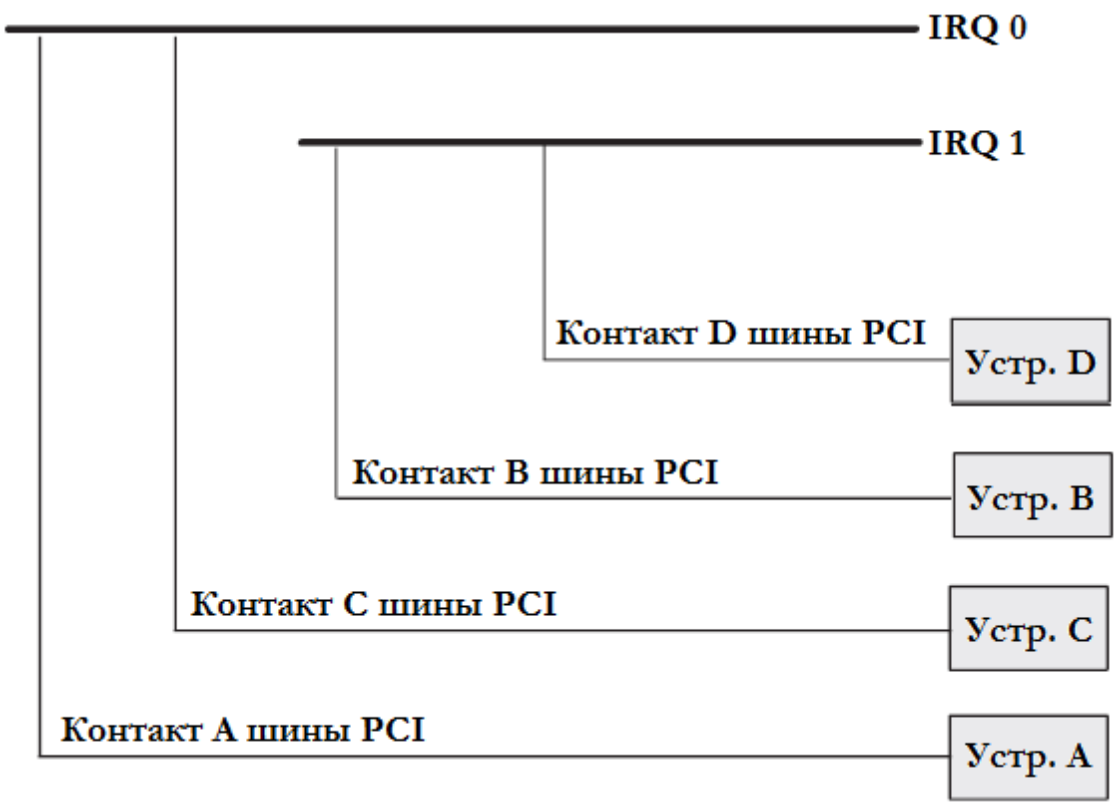

Рисунок 3.8 Маршрутизация прерываний на шине PCI.

3.5.2 Архитектура программного обеспечения шины PCI

Архитектуру программного обеспечения PCI для MIPS можно разделить на четыре уровня:

- ***BSP***: BSP даёт программному обеспечению следующую информацию:
  – Начальный адрес ввода-вывода для доступа к областям памяти и ввода-вывода PCI
  – Процедуры для доступа к пространству конфигурации PCI
  – Карту маршрутизации прерываний на плате
- ***HAL***: этот уровень реализует функциональность BIOS, назначая ресурсы PCI (память и ввод-вывод) для различных устройств. HAL использует информацию, предоставленную BSP для назначения ресурсов. Поддержка HAL включает в себя функции для сканирования шины, построения дерева устройств PCI и управления ресурсами. Таким образом, разработчикам BSP не нужно беспокоиться о деталях HAL; это часть HAL для MIPS. Разработчик BSP должен беспокоиться об интерфейсе между BSP и HAL. Это подробно обсуждается ниже, в разделе "BSP для шины PCI".
- ***Библиотека PCI***: она предоставляет интерфейсы для HAL и драйверов устройств. Библиотека находится в исходных текстах ядра в каталоге **drivers/pci**.
- ***Драйверы устройств PCI***: они должны использовать функции, экспортируемые библиотекой PCI.

В этом разделе мы обсудим BSP PCI.

BSP для шины PCI

Фактическое осуществление обмена информацией между BSP и HAL зависит от архитектуры. MIPS определяет структуру канала PCI для этой цели; эта структура данных содержит начало и конец памяти и регионы ввода-вывода, а также имеет указатель на структуру **pci_ops**, содержащую процедуры для доступа к пространству конфигурации. Эта структура данных указывает на следующие структуры данных, которые должны быть заполнены:

- **pci_ops**: это указатель на структуру, содержащую процедуры для доступа к пространству конфигурации PCI. Определение этой структуры берётся из **include/linux/pci.h**. Поскольку доступ к конфигурации зависит от платы, BSP необходимо заполнить эти процедуры так, чтобы остальные подсистемы PCI могли обращаться к пространству конфигурации.
- **io_resource** *и* **mem_resource**: эти структуры данных задают начальный и конечный адреса ввода-вывода и пространства памяти, которые назначены устройствам PCI. Хотя HAL выполняет сканирование и распределение ресурсов, всем устройствам пространство памяти или ввода-вывода назначается внутри диапазона адресов, указанного BSP.

Определение структуры канала PCI отличается в версиях ядра 2.4 и 2.6. В реализации версии 2.4 канал определяется статически с помощью **mips_pci_channels**; это подхватывается HAL. В реализации версии 2.6 структуры канал PCI реализует структура **pci_controller**; BSP необходимо заполнить эту структуру и зарегистрировать её специально для использования HAL с помощью интерфейса **register_pci_controller()**.

Наряду с заполнением вышеописанной структуры, BSP для PCI несёт дополнительную ответственность, и это выполняется различными коррекциями, которые описаны ниже.

- Наиболее важной коррекцией является та, которая выполняет маршрутизацию прерываний. Поскольку маршрутизация прерываний сильно зависит от платы, BSP должен прочитать номер контакта прерывания для каждой обнаруженной шины и назначить линию прерывания такому устройству; это выполняется в стандартном API **pcibios_fixup_irqs()**, который вызывается в HAL.
- Другой набор коррекций включает в себя все общесистемные и зависимые от устройств коррекции. Первые должны быть реализованы BSP с помощью функции **pcibios_fixup()**. Зависимые от устройства коррективы регистрируются в таблице **pcibios_fixups[]**. Когда на шине PCI обнаружено устройство, коррективы применяются для данного устройства, используя его ID в качестве идентификатора. Это особенно полезно для убирания каких-либо аномалий в отношении реализации PCI для данного устройства.

3.6 Таймеры

В BSP должны быть запрограммированы два таймера:

- Программируемый интервальный таймер (Programmable Interval Timer, PIT): этот таймер подключён к таймерному прерыванию, чтобы обеспечивает системные сигналы времени или тики. Значение для тика по умолчанию на системе Linux на MIPS составляет 10 мс.
- Часы реального времени (Real-Time Clock, RTC): они не зависят от процессора, так как это отдельная микросхема на плате.RTC питается от специальной батарейки, которая питает их даже когда плата выключена; поэтому однажды запрограммированные, они

могут обеспечить работу службы времени.

Первый таймер является обязательным в любой системе Linux; RTC, с другой стороны, не является обязательным.Аппаратная реализация PIT также варьируется в зависимости от аппаратных архитектур. На PowerPC это регистр вычитающего счётчика, который является регистром обратного отсчёта и может быть использован для генерации периодических прерываний; поэтому он может быть использован в качестве PIT. Тем не менее, подобные регистры счётчиков не доступны на всех процессорах MIPS и, следовательно, они должны полагаться на внешнее оборудование. На MIPS для настройки и разрешения работы обработчика прерывания таймера используется **board_timer_setup()**.

3.7 UART

Последовательный порт на вашей плате может быть использован для трёх целей:

- Системная консоль для вывода всех сообщений при загрузке
- Стандартное устройство TTY
- Интерфейс отладчика ядра KGDB

3.7.1 Реализация консоли

Консоль настраивается в функции **console_init()** в файле **init/main.c**. Это делается во время начальных стадий запуска системы, чтобы она могла быть использована как инструмент ранней отладки. Вся печать в ядре выполняется через функцию **printk()**; эта функцию принимает переменное число аргументов (как и **printf()**). Первым аргументом **printk()** может быть приоритет для строки, которая должна быть напечатана; чем меньше число, тем выше приоритет строки. Номер приоритета 0 используется для печати аварийных сообщений, в то время как 7 (самый высокий) используется для вывода сообщений отладочного уровня. **printk()** сравнивает приоритет строки с приоритетом консоли, устанавливаемым с помощью функции **syslog()**; если приоритет строки меньше или равен приоритету консоли, то сообщения печатаются.

printk() сохраняет список сообщений, которые должны быть напечатаны, в кольцевом буфере журнала и вызывает список зарегистрированных обработчиков консольных устройств для распечатки находящихся в очереди сообщений. Регистрация консоли происходит с помощью функции **register_console()**, которая берётся из структуры данных консоли; это сердце консольной подсистемы. Любые устройства, такие как UART, принтер или сеть могут использовать эту структуру данных для взаимодействия с консолью и захвата вывода от **printk()**.

Консоль реализована в виде структуры данных; её определение можно найти в заголовочном файле **include/linux/console.h**. Важными элементами структуры являются:

- **name**: название консольного устройства.
- **write()**: это основная функция; она вызывается **printk()**, чтобы выдавать сообщения. Функция **printk** может быть вызвана из любой части ядра, включая обработчики прерываний, поэтому вы должны быть осторожны при проектировании обработчика записи, чтобы он не засыпал. Обычно обработчик записи это простая процедура, которая для передачи символов в устройство (UART, принтер и т.д.) использует адресацию.
- **device()**: возвращает номер устройства для нижележащего устройства TTY, которое в

настоящее время выступает в качестве консоли.
- **unblank()**: эта функция, если она определена, используется для включения экрана.
- **setup()**: эта функция вызывается, когда аргумент командной строки **console=** совпадает с именем этой структуры консоли.

3.7.2 Интерфейс KGDB

KGDB представляет собой отладчик ядра на уровне исходных текстов. Он позволяет использовать GDB для отладки ядра на уровне исходных текстов; поскольку GDB является широко используемым протоколом, KGDB является весьма популярным инструментом отладки ядра. Более подробная информация о фактическом использовании KGDB представлена в Главе 8 240. KGDB используется в основном через последовательный интерфейс (доступны некоторые патчи для использования KGDB через интерфейс Ethernet). Архитектуру KGDB можно разделить на две части: заглушка GDB и драйвер последовательного порта. Заглушка GDB доступна в ядре на уровне HAL; он реализует протокол GDB и настраивает обработчики исключений. Связывание последовательного интерфейса и заглушки GDB является зоной ответственности BSP; BSP необходимо реализовать две простые функции для отправки и получения символов через последовательный интерфейс.

3.8 Управление питанием

Многие из типов встраиваемых устройств имеют разные требования к питанию, зависящие от их использования. Например, сетевые маршрутизаторы должны иметь минимальное потребление энергии, чтобы избежать нагрева, особенно при использовании в неблагоприятных условиях. Такие устройства, как КПК и мобильные телефоны, должны потреблять меньше энергии, чтобы не слишком укоротить жизнь батареи. В этом разделе описываются схемы управления питанием, доступные в Linux для встраиваемых систем и структуру управления питанием на Linux, которая охватывает уровни BSP, драйверов и приложений. Но перед этим мы обсудим связь между конструкцией оборудования и управлением питанием.

3.8.1 Управление оборудованием и питанием

Для того, чтобы понять, как встраиваемая система потребляет энергию, очень полезно узнать энергопотребление различных устройств, составляющих её. Давайте рассмотрим портативное устройство, которое состоит из следующих компонентов:

- Процессор, такой как MIPS или StrongArm
- Память (и DRAM, и флеш-память)
- Сетевая карта, например, карта для беспроводного доступа
- Звуковая карта
- Блок ЖК дисплея

Обычно блок ЖК дисплея будет крупнейшим потребителем электроэнергии в системе; затем CPU; затем звуковой карта, память и сетевая карта. Как только стало возможным идентифицировать устройства, потребляющие максимальную мощность, могут быть изучены методы для поддержания устройств в режимах потребления малой мощности. Много аппаратных устройств, доступных сегодня на рынке, имеют различные режимы работы,

чтобы удовлетворять различным требованиям по энергопотреблению; они имеют возможность работать с очень малой мощностью, если не используются, и переключиться на нормальный режим энергопотребления, когда они используются обычным образом. Драйверы устройств для таких устройств должны принять управление питанием во внимание. Из различных аппаратных устройств наиболее важным является центральный процессор. Многие процессоры для рынка встраиваемых систем обеспечивают надёжные схемы энергосбережения, которые мы сейчас проанализируем.

Двумя базовыми фактами управления питанием процессора являются:

- Мощность, потребляемая процессором, прямо пропорционально тактовой частоте.
- Мощность, потребляемая процессором, прямо пропорционально квадрату напряжения.

Новые встраиваемые процессоры принимают это во внимание и предлагают две схемы: ***динамическое изменение частоты*** и ***динамическое изменение напряжения***. Примером процессора, который поддерживает динамическое изменение частоты, является SA1110, а примером процессора, который поддерживает динамическое управление напряжением, является процессор Crusoe от Transmeta. Режимы, предлагаемые процессорами, как правило находятся под управлением ОС, которая может выбрать режим в зависимости от загрузки системы. Обычно встраиваемые системы являются системами, управляемыми событиями, и процессор тратит много времени в ожидании событий от пользователя или от внешнего мира. ОС, работающая на таких системах, может настроить энергопотребление процессора в зависимости от загрузки системы. В случае, если процессор простаивает в ожидании пользовательских событий, ему надо уделять внимание минимальным задачам, необходимым для системы, таким как обслуживание прерываний от таймера. Если процессор поддерживает режим ожидания, то в таких условиях ОС может перевести процессор в режим ожидания. Режим ожидания представляет собой режим, в котором различные тактовые частоты процессора остановлены (тактовые частоты для периферийных устройств всё же могут быть активны). Некоторые процессоры идут ещё дальше и предлагают другой режим, называемый ***спящим режимом***, в котором питание процессора и большинства периферийных устройств отключено. Это режим с самым низким энергопотреблением; однако, использование этого режима является очень сложным и чтобы использовать спящий режим, ОС должна учитывать следующие факторы:

- Состояние системы, такое как состояние центрального процессора и периферийных устройств, должно быть сохранено в памяти так, чтобы контекст сохранения мог быть восстановлен при возвращении системы из состояния сна.
- Время для выхода из состояния сна должно быть достаточно быстрым, чтобы отвечать характеристикам системы реального времени (с аппаратной и программной точки зрения).
- События, которые должны пробудить систему, должна быть оценены и надлежащим образом внесены в программное обеспечение.
- Отслеживание времени, когда система переходит в спящий режим, очень сложная задача. Помимо того факта, что система должна продолжать отслеживание времени, когда спит, система должна также учитывать тот факт, что могут быть задачи, которые спят и должны быть пробуждены после того, как система переходит в спящий режим. Обычно это могут быть внешние аппаратные часы (например, RTC), которые не будут выключены в спящем режиме; RTC могут быть запрограммированы на пробуждение системы для задач, находящихся в режиме ожидания. RTC может также использоваться для поддержания внешнего времени, которое система сможет засинхронизировать после возвращения из состояния сна.

Операционная система играет очень важную роль в реализации системы управления питанием. Роль операционной системы является многогранной:

- Как уже говорилось выше, она принимает решения относительно того, когда процессор может переключаться в другие режимы, такие как режим ожидания и спящий режим. Также в ОС должны быть разработаны соответствующие механизмы пробуждения.
- Предоставление поддержки динамического изменения частоты и напряжения в зависимости от загрузки системы.
- Предоставление драйверного ядра, чтобы драйверы различные устройств могли быть написаны с возможностью использования энергосберегающих режимов периферийных устройств.
- Экспорт системы управления питанием для специальных приложений. Этот шаг является очень важным, потому что требования к питанию каждого встраиваемого устройства очень уникальны и, следовательно, очень трудно поместить политики в ОС, чтобы они подходили различным встраиваемым устройствам. Вместо это ОС должна просто создать базовую систему управления питанием, но оставить политики приложениям, которые смогут настроить данную систему в зависимости от требований.

Мы покажем, как Linux предлагает каждую из них разработчикам встраиваемых систем, но перед этим мы должны разобраться в действующих стандартах управления питанием.

3.8.2 Стандарты управления питанием

Существуют два стандарта управления питанием, поддерживаемых Linux: стандарт APM и ACPI. Оба эти стандарта имеют свои корни в архитектуре x86. Основное различие между этими двумя стандартами в том, что при управлении питанием ACPI позволяет больше управлять изнутри ОС, тогда как APM больше зависит от BIOS. Управление питанием встраивается в ядро при выборе во время конфигурирования ядра опции **CONFIG_PM**; пользователю будет предложено выбрать стандарт APM или ACPI. В зависимости от выбранного варианта внутри ядра, должны быть выбраны соответствующие приложения пользовательского пространства; например, если в качестве стандарта управления питанием в ядре был выбран APM, то также должно быть выбрано приложение **apmd**.

Стандарт APM, введённый Microsoft и Intel, выделяет большую часть контроля управления питанием в BIOS. BIOS контролирует список устройств и знает, как определить бездействие системы, чтобы переводить систему в режимы пониженного энергопотребления. Решение позволить BIOS выполнять такое управление имело много недостатков:

- BIOS может решить перевести систему в режим пониженного энергопотребления, когда система на самом деле участвует в интенсивной вычислительной задаче. Так происходит потому, что BIOS определяет состояние системы смотря только на активность портов ввода-вывода, например, клавиатуры. Таким образом, в середине огромной задачи компиляции, когда активность клавиатуры может быть нулевой, BIOS может решить перевести систему в режим пониженного энергопотребления.
- BIOS определяет только активность устройств, расположенных на материнской плате. Устройства, находящиеся не на материнской плате, например, подключенные к шине USB, не могут участвовать в управлении питанием.
- Поскольку APM зависела от BIOS, а каждая BIOS имела свой набор ограничений и интерфейсов (и ошибок!), получить работающее управление питанием на всех системах было слишком сложным.

Когда стало ясно, что APM не был идеальным стандартом управления питанием, был разработан новый стандарт, названный ACPI. Обоснованием APCI было то, что большая часть политик управления питанием должны обрабатываться в ОС, поскольку она может принять лучшие решения в отношении нагрузки на систему и, следовательно, она может управлять питанием процессора и периферийных устройств. Стандарт ACPI всё ещё делает систему зависимой от BIOS, но в меньшей степени. При использовании интерпретируемого языка, названного AML (ACPI Machine Language, Машинный Язык ACPI), ОС может работать с устройствами, не имея большой информации об устройствах.

3.8.3 Поддержка энергосберегающих режимов процессора

Одной из первых вещей, которые имели место в ядре Linux в отношении управления питанием, была интеграция состояния ожидания процессора в цикл ожидания. Цикл ожидания ядра Linux представляет собой задачу с идентификатором процесса 0; эта задача становится запланированной, когда процессор простаивает или ждёт, когда произойдёт прерывание. Переход в состояние ожидания в течение такого цикла и пробуждение, когда происходит внешнее прерывание, снижает энергопотребление, поскольку тактовые частоты остановлены.

Следующей важной вещью, которая произошла в отношении управления питанием, была поддержка в Linux модели APM. За этим последовала поддержка ACPI. Теперь возникает вопрос: если эти модели управления питанием предназначены для архитектуры x86 и зависят от BIOS, могут ли они быть использованы для других процессоров? Использование этих стандартов будет гарантировать, что набор интерфейсов легко доступен даже на не x86 платформах; следовательно, приложения управления питанием могут быть использованы непосредственно. Методом, выбранным для не x86 платформ, было представить доступ внутрь ядра Linux, чтобы предоставить интерфейсы типа APM/ACPI в пространство пользователя, распределяя работу BIOS x86 между ядром и загрузчиком. Давайте посмотрим, как может это сделать платформа на базе процессора StrongArm.

- BSP для StrongArm на Linux предоставляет процедуры для приостановления и возобновления работы всего системного программного обеспечения; эту работа на платформах x86 выполняет BIOS (процедуры **sa1100_cpu_suspend()** и **sa1100_cpu_ resume()** в файле **arch/arm/mach-sa1100/sleep.S**). Эти процедуры используются в обработчике, который зарегистрирован в коде управления питанием в Linux и вызываются, когда система переходит в спящий режим. Тем не менее, перед переходом в сон выбирается источник пробуждения (например, активность на контакте GPIO или сигнал от RTC).
- Прежде чем система переходит в режим сна, память переводится в режим саморегенерации, для уверенности, что содержимое памяти сохранится при приостановках/возобновлениях работы. Память должна быть выведена из режима саморегенерации, когда система пробуждена. Это выполняет загрузчик. Если загрузчик был вызван из-за пробуждения после засыпания, он выведет память из режима саморегенерации и выполнит переход на адрес, хранящийся в регистре (регистре PSPR, состояние которого сохраняется в спящем режиме). Этот адрес предоставляется ядром и содержит процедуру для вывода ядра из спящего режима с помощью восстановления контекста и продолжения работы с того места, где ядро приостановило свою работу.

Ядро версии 2.6 имеет встроенные средства управления частотой. Эти средства обеспечивают метод для динамического изменения частоты на поддерживаемых архитектурах. Однако, ядро не реализует политики; наоборот, они оставлены приложениям,

использующим встроенные средства для управления частотой. Так происходит потому, что политики изменения частоты зависят от характера использования системы. Универсальное решение невозможно; поэтому реализацию политики оставляют приложениям или службам пользовательского пространства. Важными особенностями механизма управления частотой являются:

- Программное обеспечение управления частотой внутри ядра состоит из двух компонентов: скалярное ядро и драйвер управления частотой. Управление частотой скалярного ядра реализовано в файле **linux/kernel/cpufreq.c**. Это ядро является универсальным куском кода, который реализует основу, которая не зависит от аппаратного обеспечения. Однако фактическая работа управления оборудованием по изменению частоты оставлена драйверам управления частотой, которые зависят от платформы.
- Ядро делает важную работу по обновлению важной системной переменной **loops_per_jiffy**, которая зависит от частоты процессора. Эта переменная используется в различных устройствах, чтобы делать небольшие, но синхрониированные паузы с использованием функции ядра **udelay()**. Во время запуска системы значение этой переменной устанавливается с помощью функции **calibrate_delay()**. Тем не менее, всякий раз, когда позднее системная частота изменяется, эта переменная должна быть обновлена. Это делает ядро.
- Изменения тактовой частоты могут повлиять на аппаратные компоненты, которые зависят от частоты процессора. Все соответствующие драйверы устройств должны быть уведомлены об изменениях частоты, чтобы они смогли управлять оборудованием соответствующим образом. Ядро реализует механизм уведомления драйверов; драйвер, который заинтересован в получении событий изменения частоты должен быть зарегистрирован в данном ядре.
- Настройками частоты можно управлять из пользовательского пространства с помощью интерфейса **proc**. Это может использоваться приложениями для изменения тактовой частоты.
- В пространстве пользователя для управления частотой в зависимости от загрузки системы доступны различные приложения и инструменты. Например, приложение **cpufreqd** контролирует уровень заряда батареи, состояние питания и работающие программы, и управляет регулятором частоты в соответствии с набором правил, указанных в файле конфигурации.

3.8.4 Унифицированная драйверная платформа для управления питанием

Драйверы устройств являются центральным элементом в программном обеспечении управления питанием; важно обеспечить их совместную работу, особенно, если потребляемая мощность устройств, которыми они управляют, составляет значительную часть потребляемой мощности. Подобно методу, используемому механизмом управления частотой, ядро отделяет драйверы устройств от фактического программного обеспечения управления питанием в ядре, позволяя драйверам устройств зарегистрировать себя прежде, чем они примут участие в управлении питанием. Это делается с помощью вызова **pm_register()**; одним из аргументов этой функции является функция обратного вызова. Ядро хранит список всех драйверов, зарегистрированных в системе управления питанием; когда происходит событие, связанное с управлением питанием, выполняются обратные вызовы драйверов.

Если драйвер устройства участвует в управлении питанием, необходимо, чтобы с устройством не совершалось никаких операций, пока устройство находится в нерабочем

состоянии. Для этого Linux предлагает интерфейс **pmaccess**; драйвер устройства должен вызвать этот интерфейс, прежде чем он начнёт работать с оборудованием. Для выявления простаивающих устройств предоставляется дополнительный интерфейс **pm_dev_idle**, так что они могут быть помещены в сон.

Важным вопросом в реализации управления питанием для включения драйверов является вопрос о порядке. Когда одно устройство зависит от другого устройства, они должны быть включены или выключены в правильном порядке. Классическим примером этого является подсистема PCI. Если все устройства PCI на шине PCI выключены, может быть отключена сама шина PCI. Но если шина отключена в то время, когда устройства всё ещё работают, или если какое-то устройство пробуждается перед самой шиной, это может оказаться катастрофическим. То же самое верно, когда шины PCI связаны. Подсистема PCI заботится об этом обходя шины PCI снизу вверх и убеждаясь, что устройства выключены, прежде чем каждая из встреченных шин будет отключена. После выхода из сна шина обходится сверху вниз, чтобы убедиться, что шина восстановлена до того, как будут пробуждены устройства на этой шине.

3.8.5 Приложения управления питанием

Как упоминалось ранее, ядро Linux предоставляет механизмы для реализации управления питанием, но оставляет принятие решений пространству пользователя. И APM, и ACPI поставляются с приложениями, которые используются для запуска перехода системы в ждущий режим работы/ожидание (suspend/standby). Чтобы проверить, есть ли поддержка APM в ядре Linux, фоновая программа **apmd** для APM использует интерфейс **/proc/apm**; вместе с инициированием ожидания она регистрирует различные события управления питанием. Фоновая программа ACPI **acpid** прослушивает файл **/proc/acpi/event** и когда происходит событие, запускает программы для обработки события. Более подробную информацию об этих службах можно найти на:

- **apmd**: http://worldvisions.ca/
- **acpid**: http://acpid.sf.net/

Глава 4, Хранение данных во встраиваемых системах

Традиционно во встраиваемых системах хранение данных делалось с использованием постоянной памяти для хранения только читаемого кода и NVRAM (энергонезависимого ОЗУ) для хранения читаемых и записываемых данных. Тем не менее, они были заменены технологией флеш-памяти, которая обеспечивает энергонезависимую память высокой плотности. Эти преимущества в сочетании с низкой стоимостью флеш-памяти резко увеличили её использование во встраиваемых системах. В этой главе рассматриваются системы хранения данных в основном вокруг устройств флэш-памяти и различные файловые системы, существующие на Linux, предназначенные для встраиваемых систем. Глава состоит из четырёх частей:

- Флеш-память по отношению к встраиваемому Linux.
- Знакомство с подсистемой MTD (Memory Technology Drivers), предназначенной в первую очередь для устройств флеш-памяти.
- Знакомство с файловыми системами, предназначенными для встраиваемых систем. Существуют специализированные файловые системы для флеш и встроенной памяти встраиваемых систем.
- Тонкая настройка для увеличения пространства для хранения: методы, чтобы уместить больше программ во флеш-памяти.

4.1 Карта флеш-памяти

Во встраиваемой системе Linux флеш-память обычно будет использоваться для:

- Хранения загрузчика
- Хранения образа ОС
- Хранения приложений и образов библиотек приложений
- Хранения читаемых и записываемых файлов (содержащих данные о конфигурации)

Из этих четырёх вариантов, в первых трёх в течение большей части времени исполнения системы (за исключением моментов обновления) память доступна только для чтения. Таким образом, если используется загрузчик, следует иметь как минимум два раздела: один с загрузчиком, другой с корневой файловой системы. Такое разделение флеш-памяти может быть описано как карта флэш-памяти. Очень желательно, чтобы карта флеш-памяти была создана в начале проекта. Карта флеш-памяти, как и карта памяти, фиксирует то, как вы планируете разделить флеш-память для хранения вышеописанных данных, и как вы планируете получать доступ к данным.

Ниже приведены различные вопросы, которые возникнут, когда вы попытаетесь создать карту флеш-памяти:

- Как бы вы хотели разделить флеш-память? Вы можете иметь ОС, приложения и читаемые/записываемые файлы в одном разделе, но это увеличивает риск повреждения внутренних данных системы, потому что весь раздел является читаемым и записываемым. С другой стороны, можно поместить только читаемые данные в отдельный раздел, а читаемые и записываемые в другой раздел, так что только читаемые будут защищены от любых повреждений; но тогда придётся зафиксировать размер каждого раздела, убедившись, что когда-либо в будущем размер данного раздела не будет превышен.

- Как бы вы хотели получать доступ к разделам, как к необработанным данным, или вы хотели бы использовать файловую систему? Простые разделы могут быть полезны для загрузчика, потому что не будет необходимости в файловой системе; вы можете выделить сектор флеш-памяти для хранения данных конфигурации загрузки, а остальные секторы для хранения кода загрузчика. Тем не менее, для разделов, содержащих данные Linux, безопаснее идти через файловые системы. Выбор файловой системы для данных также играет важную роль в создании карты флеш-памяти.
- Как бы вы хотели делать обновления? Обновления на встроенной системе может быть выполнено на работающей системе или через загрузчик. В случае, если обновления предполагают изменение только данных для чтения (как это обычно бывает), то лучше разделить флеш-память на разделы только для чтения и только для записи, так что вам не придётся делать резервного копирования и восстановления для читаемых/записываемых данных.

Рисунок 4.1 показывает карту флеш-памяти для 4 Мб флеш-памяти, содержащей загрузчик, образ ОС и приложения. Как видно, только читаемые данные хранятся в файловой системе CRAMFS, которая является файловой системы только для чтения, а читаемые и записываемые данные хранятся в файловой системе JFFS2, которая является файловой системой для чтения и записи.

Простой раздел для загрузчика	256 K
Простой раздел для ядра	640 K
Раздел с CRAMFS для только читаемых данных	2 M
Раздел с JFFS2 для читаемых и записываемых данных	1.2 M

Рисунок 4.1 Карта памяти для 4-х Мб флеш-памяти.

4.2 MTD — Технологическое Устройство Памяти

MTD означает Memory Technology Device, Технологическое Устройство Памяти, и является подсистемой, используемой для управления устройствами хранения данных, находящихся на плате. Является ли MTD отдельным классом набора драйверов, как символьные или блочные? Простой ответ: нет. Тогда в чём заключается работа MTD и когда и как устройства флеш-памяти включаются в подсистему MTD? Как на MTD устройство будут помещаться файловые системы? На эти вопросы отвечают следующие подразделы.

4.2.1 Модель MTD

Хотя устройства флеш-памяти являются устройствами хранения, похожими на жёсткие диски, между ними существуют некоторые фундаментальные различия.

- Обычно жёсткие диски имеют сектор, который делится по размерам страницы (обычно 4096 байт). Стандартным значением является 512 байт. Модель файловой системы Linux, особенно буфер кэша (кэш памяти между файловой системой и уровнем блочного устройства), основана на этом предположении. Микросхемы флеш-памяти, с другой стороны, имеют большие размеры сектора; стандартный размер составляет 64 Кб.
- Секторы флеш-памяти обычно перед записью в них должны быть стёрты; операции записи и стирания могут быть независимыми, что зависит от программного обеспечения, использующего флэш-память.
- Микросхемы флеш-памяти имеют ограниченный срок службы, который определяется в терминах числа раз стирания сектора. Так что если какой-то сектор становится записываемым очень часто, его срок службы уменьшается. Чтобы предотвратить это, записи на флеш-память должны быть распределены по всем секторам. Это называется выравниванием износа и не поддерживается блочными устройствами.
- Обычные файловые системы не могут быть использованы поверх флеш-памяти, потому что это проходит через буферный кэш. Обычный дисковый ввод-вывод медленный; чтобы ускорить его, используется кэш в памяти, называемой кэш-буфером, который хранит данные для ввода-вывода на диск. Пока эти данные не записаны обратно на диск, файловая система находится в неустойчивом состоянии. (Это и является причиной, почему вы должны выключать ОС на ПК перед выключением его питания.) Тем не менее, встроенные системы могут быть выключены без надлежащего завершения работы и всё же иметь непротиворечивые данные; так что обычные файловые системы и модель блочного устройства не очень подходят для встраиваемых систем.

Традиционным методом, используемым для доступа к флеш-памяти, является FTL, то есть Flash Translation Layer, Уровень Трансляции Флеш-памяти. Этот уровень эмулирует на флеш-памяти поведение блочного устройства, чтобы обеспечить на них работу обычных файловых систем. Однако, создание новой файловой системы или нового драйвера флеш-памяти, работающего с FTL, является сложной задачей и это является причиной изобретения подсистемы MTD. (Владельцем подсистемы MTD является Дэвид Вудхаус, а разработки, связанные с MTD, можно получить на веб-сайте http://www.linux-mtd.infradead.org/. Подсистема MTD была создана как часть основного ядра версии 2.4.) Решение MTD вышеуказанных проблем простое: рассматривать устройства памяти как устройства памяти, а не как диски. Поэтому вместо изменения низкоуровневых драйверов или введения уровня трансляции, измените приложение, чтобы использовать устройства памяти такими, как они есть. MTD очень привязано к приложениям; подсистема MTD состоит из двух частей: драйверы и приложения.

Подсистема MTD не реализует новый вид драйвера, но скорее она связывает любое устройство с драйверами символьных и блочных устройств. Когда драйвер зарегистрирован в подсистеме MTD, она экспортирует такое устройство в оба этих вида драйверов. Почему это делается именно так? Символьное устройство может позволить непосредственный доступ к устройству памяти с использованием стандартных вызовов **open/read/write/ioctl**. Но в случае, если вы захотите смонтировать обычную файловую систему на такое устройство памяти с помощью традиционного метода, вы сможете также смонтировать её используя блочный драйвер.

Мы разберёмся с каждым уровнем на Рисунке 4.1, но перед этим давайте разберёмся с двумя устройствами, которые в настоящее время поддерживаются MTD: микросхемы флеш-памяти и флеш-диски.

4.2.2 Микросхемы флеш-памяти

Рассмотрим различные микросхемы флеш-памяти, поддерживаемые подсистемой MTD. Устройства флеш-памяти поставляются в двух вариантах: NAND и NOR флеш-память. Хотя оба варианта созданы примерно в одно время (NOR была представлена Intel, а NAND - Toshiba в конце 1980 года), NOR быстрее вошла в мир встраиваемых устройств, поскольку она проще в использовании. Однако, когда встраиваемые системы эволюционировали к необходимости иметь больше места для хранения (например, медиаплееры и цифровые камеры), для приложений хранения данных стала популярной NAND Flash. Уровень MTD также первоначально развивался вокруг NOR Flash, а поддержка NAND была добавлена позднее. Таблица 4.1 сравнивает два этих типа флеш-памяти.

Таблица 4.1 Сравнение NOR и NAND Flash

	NOR	***NAND***
Доступ к данным	К данным можно обращаться в случайном порядке, так же как к SRAM. Операциями с флеш-памятью могут быть: ***Процедура чтения***: чтение содержимого флеш-памяти. ***Процедура стирания***: стирание - это процесс установки всех битов флеш-памяти в 1. Стирание в микросхемах NOR происходит в терминах блоков (называемых областями стирания). ***Процедура записи***: запись - это процесс изменения во флеш-памяти 1 в 0. После того, как бит стал 0, он не может быть записан, пока блок не будет стёрт, что означает установку всех битов в блоке в 1.	В микросхемах NAND флеш пространство разделено на блоки, которые также разделены на страницы. Каждая страница разделена на обычные данные и дополнительные (out-of-band) данные. Дополнительные данные используются для того, чтобы хранить метаданные, такие как ECC (Error-Correction Code, Код для коррекции ошибок) данные и информацию о неисправном блоке. NAND флеш, как и NOR флеш, имеет три основные операции: **чтение**, **стирание** и **запись**. Однако, в отличие от NOR, к данным которой можно обращаться в произвольном порядке, чтение и запись в NAND флеш выполняются в терминах страниц, тогда как стирание происходит в терминах блоков.
Подключение к плате	Подключается как обычное устройство SRAM к шинам адреса и данных процессора.	Существуют несколько способов подключения NAND флеш к CPU, зависящих от производителя. Для доступа к

		NAND выполняется соединение выводов данных и команд к обычно 8 выводам ввода-вывода на микросхеме флеш-памяти.
Выполнение кода	Код может быть выполнен прямо из NOR, поскольку она подключена непосредственно к шинам адреса и данных.	Если код находится в NAND флеш, для запуска он нуждается в копировании в память.
Производительность	Флеш-память NOR характеризуется медленным стиранием, медленной записью и быстрым чтением.	Флеш-память NAND характеризуется быстрым стиранием, быстрой записью и быстрым чтением.
Неисправные блоки	Неисправные блоки в микросхемах NOR флеш не ожидаются, поскольку они были разработаны, чтобы хранить системные данные.	Эти микросхемы были разработаны в основном как устройства хранения медиа-данных с более низкой ценой, поэтому стоит ожидать, что они имеют неисправные блоки. Обычно такие микросхемы флеш-памяти поставляются с помеченными неисправными участками. Также сектора NAND флеш больше страдают от проблемы переключения битов, когда бит становится перевернутым во время записи; это обнаруживается алгоритмами коррекции ошибок, называемых ECC/EDC, которые выполняются либо в оборудовании, либо в программном обеспечении.
Применение	В основном используются для выполнения кода. На NOR флеш могут находиться загрузчики, потому что код из такой флеш-памяти может быть выполнен напрямую. Такая флеш-память довольно дорогая и она обеспечивает меньшие плотности памяти и имеет относительно более короткую продолжительность жизни (приблизительно 100 000 циклов стирания).	Используются главным образом в качестве устройств хранения для встраиваемых систем, таких как приставки к телевизорам и MP3-плееры. Если вы планируете использовать плату только с NAND, вам придётся добавить дополнительное ПЗУ загрузки. Они предлагают высокие плотности при более низких ценах и имеют более длительный срок службы (около 10 в 6-ой степени циклов стирания).

Микросхемы NOR флеш бывают двух видов: старые не-CFI и новые, CFI совместимые. CFI расшифровывается как Common Flash Interface (Стандартный Интерфейс Флеш-памяти) и является отраслевым стандартом для обеспечения совместимости микросхем флеш-памяти одного и того же поставщика. Микросхемы флеш-памяти, как и любое другое устройство памяти, всегда находятся в стадии эволюции, новые микросхемы заменяют старые очень быстро; это означало бы переписывание драйверов флеш-памяти. Часто такие изменения были бы изменениями в конфигурации, такими как время ожидания стирания, размеры блоков, и тому подобное. Чтобы избежать этих усилий, были введены стандарты CFI, которые позволяют поставщикам флеш-памяти сделать так, чтобы данные конфигурации считывались с устройств флеш-памяти. Так, чтобы системное ПО могло бы опросить устройства флеш-памяти и изменить свою конфигурацию. MTD поддерживает набор команд CFI от Intel и AMD.

Поддержка NAND флеш была добавлена в конце серии ядер версии 2.4; наряду с NFTL (NAND Flash Translation Layer, Уровнем Трансляции NAND Флеш) она может монтировать обычные файловые системы, но поддержка JFFS2 была добавлена только для ядра версии 2.6. Ядро версии 2.6 можно считать хорошим вариантом для использования флеш-памяти типа NAND.

4.2.3 Флеш диски

Флеш диски были представлены для приложений хранения больших объёмов информации. Как следует из их названия, флеш-диски означает локальные диски системы, основанные на технологии флеш-памяти. Флеш диски также бывают двух видов: на основе ATA и линейные.

Флеш диски на основе ATA используют для взаимодействия на материнской плате стандартный дисковый интерфейс, так что они выглядят в системе как диски IDE. Контроллер находится в той же микросхеме, что и флеш-память, но реализация FTL делит флеш-память на секторы. Кроме того, он реализует дисковый протокол так, чтобы флеш-память выглядела в системе как обычный диск. Это был подход, использованный разработчиками CompactFlash. Основным преимуществом использования данного подхода была совместимость программного обеспечения, но недостатком было то, что это дороже, поскольку общее решение было сделано на аппаратном уровне. Linux рассматривает эти устройства как обычные устройства IDE и драйвер для этих устройств можно найти в каталоге **drivers/ide**.

Линейный флеш-диск является механизмом, используемым системами M2000. Это устройства на основе NAND, которые имеют возможность загрузки (они имеют ПЗУ загрузки, которая распознаётся как расширение BIOS), простой контроллер, который использует алгоритмы исправления ошибок, и программное обеспечение trueFFFS, которое выполняет эмуляцию FTL. Таким образом, эти устройства могут быть использованы для прямой загрузки системы и могут быть использованы для запуска обычных файловых систем на устройстве, подобном блочному. Они менее дорогие по сравнению с CompactFlash, но в то же время обладают всеми функциями, необходимыми блочному устройству. Поскольку доступ к этим устройствам флеш-памяти похож на доступ к устройствам памяти, Linux реализует драйверы для них под моделью MTD.

4.3 Архитектура MTD

Когда встаёт задача заставить Linux работать на устройстве на базе флеш-памяти, обычно возникают следующие два вопроса:

- Есть ли в Linux драйвер для поддержки моей флеш-памяти; если же нет, как мне написать драйвер?
- Если Linux поддерживает драйвер для моей флеш-памяти, как можно обнаружить флеш-память на плате и сделать, чтобы драйвер устанавливался автоматически?

Ответы на эти вопросы даёт понимание архитектуры MTD. Архитектура MTD разделена на следующие компоненты:

- ***Ядро MTD***: оно обеспечивает интерфейс между низкоуровневыми драйверами флеш-памяти и приложениями. Оно реализует режим символьного и блочного устройства.
- ***Низкоуровневые драйверы флеш-памяти***: в этом разделе говорится только о микросхемах флеш-памяти NOR и NAND.
- ***BSP для флеш-памяти***: флеш-память может быть подключена к плате каким-то уникальным способом. Например, NOR Flash может быть подключена непосредственно на шину процессора, или же может быть подключена к внешней шине PCI. Доступ к флеш-памяти также может быть уникальным, в зависимости от типа процессора. Уровень BSP позволяет драйверу флеш-памяти работать с любой платой/процессором. Пользователь должен предоставить подробности того, как на плате подключена флеш-память; мы называем этот кусок кода как ***драйвер связи с флеш-памятью (flash-mapping driver)***.
- ***Приложения MTD***: это могут быть как субмодули ядра, такие как JFFS2 или NFTL, так и приложения в пространстве пользователя, такие как менеджер обновлений.

Как эти компоненты взаимодействуют друг с другом и другими частями ядра показывает Рисунок 4.2.

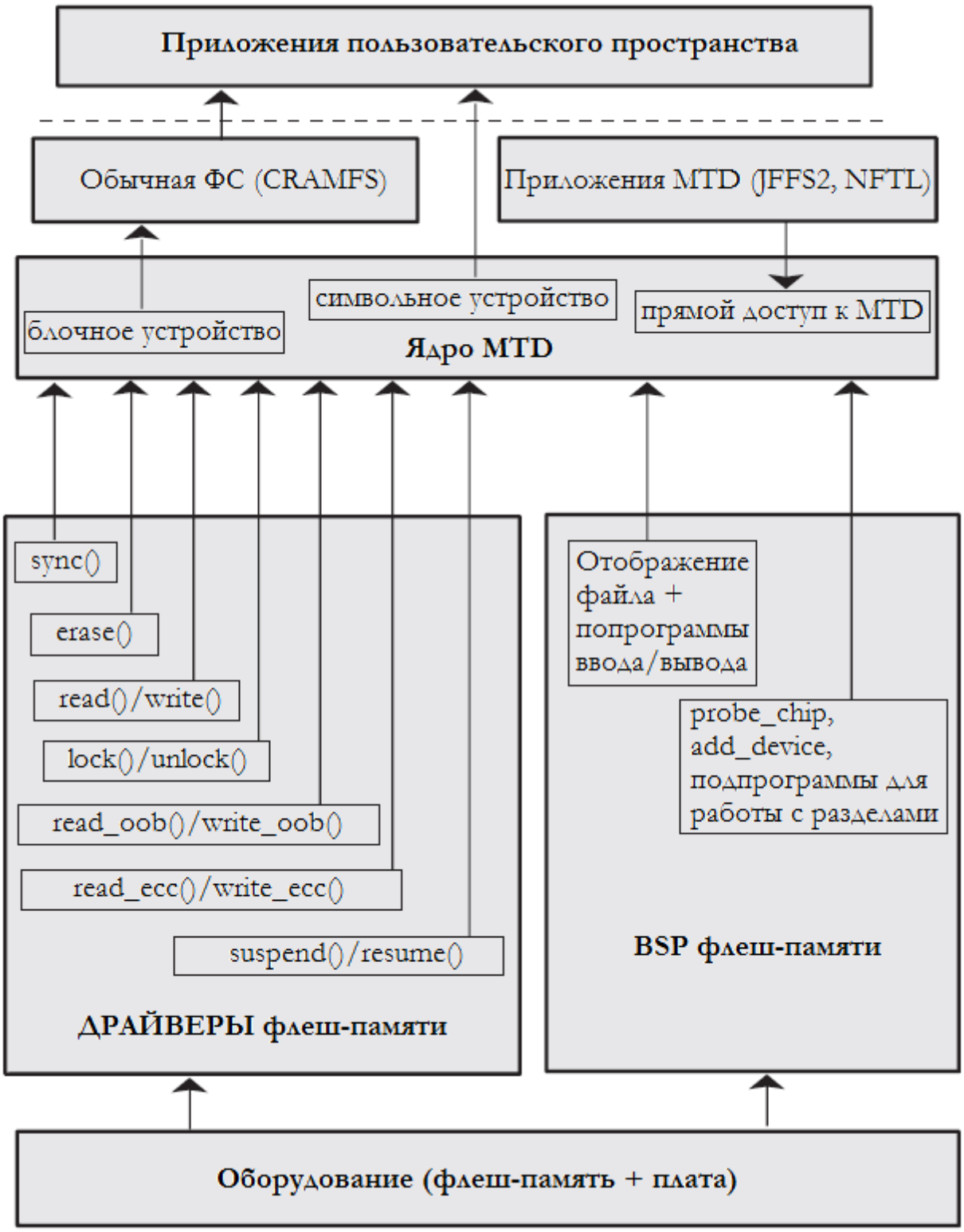

Рисунок 4.2 Архитектура MTD.

4.3.1 Структура данных mtd_info

mtd_info является сердцем программного обеспечения MTD. Она определена в файле **include/linux/mtd/mtd.h**. Программный драйвер при определении флеш-памяти заполняет эту структуру указателями на все необходимые методы (например, стирание, чтение, запись и так далее), которые используются ядром и приложениями MTD. Список структур **mtd_info** для всех добавленных устройств хранится в таблице с именем **mtd_table[]**.

4.3.2 Интерфейс между ядром MTD и низкоуровневым драйвером флеш-памяти

Как упоминалось выше, низкоуровневый драйвер флеш-памяти экспортирует следующие функции в ядро MTD:

- Функции, общие для обоих типов микросхем флеш-памяти, NAND и NOR
  - **read()/write()**
  - **erase()**
  - **lock()/unlock()**
  - **sync()**
  - **suspend()/resume()**
- Функции только для микросхем NAND
  - **read_ecc()/write_ecc()**
  - **read_oob()/write_oob()**

Если вы имеете флеш-память NOR с поддержкой CFI или стандартную, связанную с устройством ввода-вывода 8-ми разрядную микросхему NAND, то ваш драйвер уже готов. В противном случае вам необходимо реализовать драйвер MTD. Некоторые из процедур могут потребовать аппаратной поддержки; так что вы должны проверить спецификацию вашей флеш-памяти для реализации таких функций. Следующий раздел даёт описание процедур, кроме подпрограмм **read()**, **write()** и **erase()**.

- **lock()** и **unlock()**: они используются для реализации блокировки флеш-памяти; для части флеш-памяти может быть установлена защита от записи или стирания, чтобы предотвратить случайную перезапись данных. Например, можно заблокировать все разделы, на которых находятся только читаемые файловые системы, на большую часть времени работы системы за исключением случаев, когда выполняется обновление. Они экспортируются в пользовательские приложения, используя вызовы ioctl **MEMLOCK** и **MEMUNLOCK**.
- **sync()**: он вызывается, когда устройство получает запрос на закрытие или освобождение, и это гарантирует, что флеш-память находится в безопасном состоянии.
- **suspend()** и **resume()**: они полезны только когда вы включаете при сборке ядра опцию **CONFIG_PM**.
- **read_ecc()** и **write_ecc()**: эти процедуры применяются только для флеш-памяти NAND. ECC является кодом исправления ошибок, который используется для выявления на странице плохих битов. Эти процедуры ведут себя как обычные **read()/write()**, за исключением отдельного буфера, содержащего ECC, который читается и записывается вместе с данными.
- **read_oob()** и **write_oob()**: эти процедуры применяются только для флеш-памяти NAND. Каждая микросхема флеш-памяти NAND делится либо на 256-ти, либо на 512-ти байтовые страницы; каждая из этих страниц содержит дополнительную 8-ми или 16-ти байтовую резервную область, называемую дополнительными (out-of-band) данными,

которая хранит ECC, информацию о плохих блоках, и другие зависимые от файловой системы данные. Эти функции используются для обращения к таким дополнительным данным.

4.4 Пример драйвера MTD для NOR Flash

Погрузимся в подробности драйвера памяти NOR флеш для Linux. Файл **mtd.c** содержит код для простой NOR Flash, основанный на следующих предположениях:

- Устройство флеш-памяти имеет один регион стирания, так что все секторы имеют одинаковый размер. (Регион стирания определяется как площадь микросхемы, которая содержит секторы одного и того же размера.)
- Обращение к микросхеме флеш-памяти происходит с помощью 4-х байтовой шины.
- Функциональность блокировки, разблокировки, приостановления и возобновления работы не поддерживается.

Для простоты будем считать, что у нас в виде макросов или функций имеется следующая информация:

- **DUMMY_FLASH_ERASE_SIZE**: размер сектора стирания флеш-памяти
- **DUMMY_FLASH_SIZE**: размер флеш-памяти
- **PROBE_FLASH()**: функция, которая проверяет, присутствует ли NOR Flash по указанному адресу
- **WRITE_FLASH_ONE_WORD**: функция/макрос для записи слова по указанному адресу
- **ERASE_FLASH_SECTOR**: функция для стирания заданного сектора
- **DUMMY_FLASH_ERASE_TIME**: время стирания одного сектора в тиках (jiffies)

Сначала составим список всех заголовочных файлов, которые потребуются для нашего драйвера флеш-памяти.

```
/* mtd.c */
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/sched.h>
#include <linux/errno.h>
#include <linux/interrupt.h>
#include <linux/mtd/map.h>
#include <linux/mtd/mtd.h>
#include <linux/mtd/cfi.h>
#include <linux/delay.h>
```

Теперь определим все API/макросы, которые, как мы ожидаем, определит пользователь.

```
#define DUMMY_FLASH_ERASE_SIZE
#define PROBE_FLASH(map)
#define WRITE_FLASH_ONE_WORD(map, start, addr, data)
#define ERASE_FLASH_SECTOR(map, start, addr)
#define DUMMY_FLASH_ERASE_TIME
#define DUMMY_FLASH_SIZE
```

Краткое описание аргументов, которые передаются вышеописанному API:

- **map**: это указатель на структуру **map_info**, объявленную в заголовочном файле **include/linux/mtd/map.h**. Эта структура более подробно объясняется в Разделе 4.5 79.
- **start**: это начальный адрес микросхемы NOR флеш. Этот адрес обычно используется для программирования флеш-памяти с помощью команды стирания или записи данных.
- **addr**: это смещение от начального адреса микросхемы, куда должны быть записаны данные или где должен быть стёрт сектор.
- **data**: этот аргумент для API записи представляет собой 32-х разрядное слово, которое определяет, что должно быть записано по указанному адресу.

Далее мы определяем структуру, которая содержит информацию, относящуюся именно к этой флеш-памяти.

```
struct dummy_private_info_struct
{
  int number_of_chips; /* Количество микросхем флеш-памяти */
  int chipshift; /* Размер каждой флеш-памяти */
  struct flchip *chips;
} ;
```

Краткое описание каждого из полей этой структуры:

- **number_of_chips**: как следует из названия, оно указывает, сколько всего микросхем можно найти по адресу **probe**. Это число коду драйвера должен вернуть API **PROBE_FLASH()**.
- **chipshift**: это общее количество разрядов адреса для устройства, которое используется для вычисления адреса смещения, и общего числа байтов, которое вмещает устройство.
- **chips**: **struct flchip** можно найти в **include/linux/mtd/flashchip.h**. Дополнительное объяснение находится в функции **dummy_probe()**.

Далее идёт список статических функций, которые должны быть объявлены.

```
static struct mtd_info * dummy_probe(struct map_info *);
static void dummy_destroy(struct mtd_info *);
static int dummy_flash_read(struct mtd_info *, loff_t , size_t ,
                            size_t *, u_char *);
static int dummy_flash_erase(struct mtd_info *,
                             struct erase_info *);
static int dummy_flash_write(struct mtd_info *, loff_t ,
                             size_t , size_t *, const u_char *);
static void dummy_flash_sync(struct mtd_info *);
```

Структура **mtd_chip_driver** используется процедурой инициализации **dummy_flash_init()** и функцией выхода **dummy_flash_exit()**. Наиболее важным полем является **.probe**, которое вызывается, чтобы определить, присутствует ли на плате по указанному адресу флеш-память указанного типа. Уровень MTD хранит список таких структур. Обращение к процедурам **probe** происходит, когда драйвер связи с флеш-памятью вызывает процедуру **do_map_probe()**.

```
static struct mtd_chip_driver dummy_chipdrv =
{
  .probe   = dummy_probe,
  .destroy = dummy_destroy,
  .name    = "dummy_probe",
  .module  = THIS_MODULE
};
```

Теперь определим процедуру **probe**. Эта функция проверяет, может ли флеш-память быть найдена по адресу **map->virt**, который заполняется драйвером связи с флеш-памятью. Если он в состоянии обнаружить флеш-память, то он выделяет память для структуры **mtd** и структуры **dummy_private_info**. Структура **mtd** заполнена различными процедурами драйвера, такими как **read**, **write**, **erase** и так далее. Структура **dummy_private_info** заполнена информации о данной флеш-памяти. Реализацию процедуры **probe** можно посмотреть в Распечатке 4.1 71.

Наиболее интересными структурами данных, которые инициализируются в функции **probe**, являются очередь ожидания и мьютекс. Они используются для предотвращения одновременного доступа к флеш-памяти, что является необходимым условием почти для всех устройств флеш-памяти. Таким образом, когда должны быть выполнены такие операции, как чтение или запись, драйвер должен проверить, не используется ли флеш-память. Это выполняется с помощью поля **state**, которое установлено во время инициализации в **FL_READY**. Если флеш-память используется, то процесс должен заблокироваться на очереди ожидания и ждать пробуждения. Мьютекс (спин-блокировка) используется для предотвращения проблем с состояниями гонок на машинах с SMP (симметричной многопроцессорной обработкой) или в случае, если разрешено вытеснение.

Перейдём к процедуре **read**. Процедурой чтения, зарегистрированной в ядре MTD, является **dummy_flash_read()**, которая вызывается, чтобы прочитать **len** байтов из флеш-памяти со смещения **from**. Поскольку процедуры записи могут охватывать несколько микросхем, для чтения данных из одной микросхемы внутри вызывается функция **dummy_flash_read_one_chip()**. Реализацию можно посмотреть в Распечатке 4.2 72.

Теперь перейдём к процедуре записи. Процедурой, зарегистрированной в ядре MTD, является **dummy_flash_write()**. Поскольку запись может начинаться с невыровненного адреса, данная функция гарантирует, что она забуферирует данные в таких случаях и, в свою очередь, вызовет функцию **dummy_flash_write_oneword()** для записи 32-х разрядных данных по выровненным 32-х разрядным адресам. Реализацию можно посмотреть в Распечатке 4.3 74.

Функцией стирания, зарегистрированной в ядре MTD, является **dummy_flash_erase()**. Эта функция должна убедиться, что указанный адрес стирания является выровненным по сектору и количество байт, которое будет стираться, кратно размеру сектора. Внутри вызывается функция **dummy_flash_erase_one_block()**; она стирает один сектор по указанному адресу. Поскольку стирание сектора занимает много времени, эта функция вытесняет вызывающую задачу, отправляя её в сон на **DUMMY_FLASH_ERASE_TIME** тиков. По окончании стирания ядро MTD сигнализирует, что стирание завершено, устанавливая состояние стирания в **MTD_ERASE_DONE**, а затем перед возвращением выполняются все зарегистрированные обратные вызовы. Реализацию можно посмотреть в

Распечатке 4.4 76.

Когда устройство флеш-памяти закрывается, вызывается функция **sync**. Эта функция должна убедиться, что ни одна из микросхем флеш-памяти не используется на момент закрытия; если же они используются, функция заставляет вызывающий процесс ждать, пока все микросхемы не перейдут в неиспользуемое состояние. Реализацию имитации функции **sync** можно посмотреть в Распечатке 4.5 78.

В случае, если драйвер флеш-памяти загружен как модуль, вызывается функция **dummy_destroy**. Функции **dummy_destroy()** выполняет все очистку при выгрузке модуля.

```
static void dummy_destroy(struct mtd_info *mtd)
{
  struct dummy_private_info_struct *priv =
                   ((struct map_info *)mtd->priv)->fldrv_priv;
  kfree(priv->chips);
}
```

Следующими являются функции инициализации и выхода.

```
int __init dummy_flash_init(void)
{
  register_mtd_chip_driver(&dummy_chipdrv);
  return 0;
}
void __exit dummy_flash_exit(void)
{
  unregister_mtd_chip_driver(&dummy_chipdrv);
}

module_init(dummy_flash_init);
module_exit(dummy_flash_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Embedded Linux book");
MODULE_DESCRIPTION("Sample MTD driver");
```

Распечатка 4.1 Имитация функции probe

Распечатка 4.1

```
static struct mtd_info *dummy_probe(struct map_info *map)
{
  struct mtd_info * mtd = kmalloc(sizeof(*mtd), GFP_KERNEL);
  unsigned int i;
  unsigned long size;
  struct dummy_private_info_struct * dummy_private_info =
       kmalloc(sizeof(struct dummy_private_info_struct), GFP_KERNEL);

  if(!dummy_private_info)
  {
    return NULL;
```

```
  }
  memset(dummy_private_info, 0, sizeof(*dummy_private_info));

  /* Функция probe возвращает число опознанных микросхем */
  dummy_private_info->number_of_chips = PROBE_FLASH(map);
  if(!dummy_private_info->number_of_chips)
  {
    kfree(mtd);
    return NULL;
  }

  /* Инициализация структуры mtd */
  memset(mtd, 0, sizeof(*mtd));
  mtd->erasesize = DUMMY_FLASH_ERASE_SIZE;
  mtd->size = dummy_private_info->number_of_chips * DUMMY_FLASH_SIZE;
  for(size = mtd->size; size > 1; size >>= 1)
    dummy_private_info->chipshift++;
  mtd->priv = map;
  mtd->type = MTD_NORFLASH;
  mtd->flags = MTD_CAP_NORFLASH;
  mtd->name = "DUMMY";
  mtd->erase = dummy_flash_erase;
  mtd->read = dummy_flash_read;
  mtd->write = dummy_flash_write;
  mtd->sync = dummy_flash_sync;

  dummy_private_info->chips = kmalloc(sizeof(struct flchip) *
              dummy_private_info->number_of_chips, GFP_KERNEL);
  memset(dummy_private_info->chips, 0,
         sizeof(*(dummy_private_info->chips)));
  for(i=0; i < dummy_private_info->number_of_chips; i++)
  {
    dummy_private_info->chips[i].start = (DUMMY_FLASH_SIZE * i);
    dummy_private_info->chips[i].state = FL_READY;
    dummy_private_info->chips[i].mutex =
                         &dummy_private_info->chips[i]._spinlock;
    init_waitqueue_head(&dummy_private_info->chips[i].wq);
    spin_lock_init(&dummy_private_info->chips[i]._spinlock);
    dummy_private_info->chips[i].erase_time = DUMMY_FLASH_ERASE_TIME;
  }

  map->fldrv = &dummy_chipdrv;
  map->fldrv_priv = dummy_private_info;

  printk("Probed and found the dummy flash chip\n");
  return mtd;
}
```

Распечатка 4.2 Имитация функции read

Распечатка 4.2

```
static inline int dummy_flash_read_one_chip(struct map_info *map,
```

```
       struct flchip *chip, loff_t addr, size_t len, u_char *buf)
{
  DECLARE_WAITQUEUE(wait, current);

again:
  spin_lock(chip->mutex);

  if(chip->state != FL_READY)
  {
    set_current_state(TASK_UNINTERRUPTIBLE);
    add_wait_queue(&chip->wq, &wait);
    spin_unlock(chip->mutex);
    schedule();
    remove_wait_queue(&chip->wq, &wait);
    if(signal_pending(current))
      return -EINTR;
    goto again;
  }

  addr += chip->start;
  chip->state = FL_READY;
  map_copy_from(map, buf, addr, len);
  wake_up(&chip->wq);
  spin_unlock(chip->mutex);
  return 0;
}

static int dummy_flash_read(struct mtd_info *mtd, loff_t from,
                      size_t len,size_t *retlen, u_char *buf)
{
  struct map_info *map = mtd->priv;
  struct dummy_private_info_struct *priv = map->fldrv_priv;
  int chipnum = 0;
  int ret = 0;
  unsigned int ofs;
  *retlen = 0;

  /* Ищем номер и смещение для первой микросхемы */
  chipnum = (from >> priv->chipshift);
  ofs = from & ((1 << priv->chipshift) - 1);
  while(len)
  {
    unsigned long to_read;
    if(chipnum >= priv->number_of_chips)
      break;

    /* Проверяем, правильно ли переходит чтение на следующую микросхему */
    if( (len + ofs - 1) >> priv->chipshift)
      to_read = (1 << priv->chipshift) - ofs;
    else
      to_read = len;
    if( (ret = dummy_flash_read_one_chip(map, &priv->chips[chipnum],
                                         ofs, to_read, buf)))
      break;
```

```
    *retlen += to_read;
    len -= to_read;
    buf += to_read;
    ofs=0;
    chipnum++;
  }
  return ret;
}
```

Распечатка 4.3 Имитация функции write

Распечатка 4.3

```
static inline int dummy_flash_write_oneword(struct map_info *map,
                  struct flchip *chip, loff_t addr, __u32 datum)
{
  DECLARE_WAITQUEUE(wait, current);

again:
  spin_lock(chip->mutex);

  if(chip->state != FL_READY)
  {
    set_current_state(TASK_UNINTERRUPTIBLE);
    add_wait_queue(&chip->wq, &wait);
    spin_unlock(chip->mutex);
    schedule();
    remove_wait_queue(&chip->wq, &wait);
    if(signal_pending(current))
      return -EINTR;
    goto again;
  }

  addr += chip->start;
  chip->state = FL_WRITING;
  WRITE_FLASH_ONE_WORD(map, chip->start, addr, datum);
  chip->state = FL_READY;
  wake_up(&chip->wq);
  spin_unlock(chip->mutex);
  return 0;
}

static int dummy_flash_write(struct mtd_info *mtd, loff_t from,
      size_t len,size_t *retlen, const u_char *buf)
{
  struct map_info *map = mtd->priv;
  struct dummy_private_info_struct *priv = map->fldrv_priv;
  int chipnum = 0;
  union {
    unsigned int idata;
    char cdata[4]; }
  wbuf;
```

```
unsigned int ofs;
int ret;

*retlen = 0;
chipnum = (from >> priv->chipshift);
ofs = from & ((1 << priv->chipshift) - 1);

/* Сначала проверяем, выровнено ли первое слово для записи */
if(ofs & 3)
{
  unsigned int from_offset = ofs & (~3);
  unsigned int orig_copy_num = ofs - from_offset;
  unsigned int to_copy_num = (4 - orig_copy_num);
  unsigned int i, len;

  map_copy_from(map, wbuf.cdata, from_offset +
                priv->chips[chipnum].start, 4);

  /* Перезаписываем новым содержимым из buf[] */
  for(i=0; i < to_copy_num; i++)
    wbuf.cdata[orig_copy_num + i] = buf[i];

  if((ret = dummy_flash_write_oneword(map, &priv->chips[chipnum],
      from_offset, wbuf.idata)) < 0)
    return ret;

  ofs += i;
  buf += i;
  *retlen += i;
  len -= i;
  if(ofs >> priv->chipshift)
  {
    chipnum++;
    ofs = 0;
  }
}

/* Теперь записываем все выровненные слова */
while(len / 4)
{
  memcpy(wbuf.cdata, buf, 4);
  if((ret = dummy_flash_write_oneword(map, &priv->chips[chipnum],
                                      ofs, wbuf.idata)) < 0)
    return ret;

  ofs += 4;
  buf += 4;
  *retlen += 4;
  len -= 4;
  if(ofs >> priv->chipshift)
  {
    chipnum++;
    ofs = 0;
  }
```

```
  }

  /* Записываем последнее слово */
  if(len)
  {
    unsigned int i=0;

    map_copy_from(map, wbuf.cdata, ofs + priv->chips[chipnum].start,
                  4);
    for(; i<len; i++)
      wbuf.cdata[i] = buf[i];

    if((ret = dummy_flash_write_oneword(map, &priv->chips[chipnum],
                                        ofs, wbuf.idata)) < 0)
      return ret;
    *retlen += i;
  }

  return 0;
}
```

Распечатка 4.4 Имитация функции erase

Распечатка 4.4

```
static int dummy_flash_erase_one_block(struct map_info *map,
                   struct flchip *chip, unsigned long addr)
{
  DECLARE_WAITQUEUE(wait, current);
again:
  spin_lock(chip->mutex);

  if(chip->state != FL_READY)
  {
    set_current_state(TASK_UNINTERRUPTIBLE);
    add_wait_queue(&chip->wq, &wait);
    spin_unlock(chip->mutex);
    schedule();
    remove_wait_queue(&chip->wq, &wait);
    if(signal_pending(current))
      return -EINTR;
    goto again;
  }

  chip->state = FL_ERASING;
  addr += chip->start;
  ERASE_FLASH_SECTOR(map, chip->start, addr);

  spin_unlock(chip->mutex);
  schedule_timeout(chip->erase_time);
  if(signal_pending(current))
    return -EINTR;
```

```
  /* Мы пробуждены после превышения времени ожидания.
     Захватываем мьютекс для обработки */
  spin_lock(chip->mutex);

  /* Добавьте любые проверки на то, был ли стёрт сектор флеш-памяти. */

  /* Предполагаем, что здесь стирание флеш-памяти было завершено */
  chip->state = FL_READY;
  wake_up(&chip->wq);
  spin_unlock(chip->mutex);
  return 0;
}

static int dummy_flash_erase(struct mtd_info *mtd,
                             struct erase_info *instr)
{
  struct map_info *map = mtd->priv;
  struct dummy_private_info_struct *priv = map->fldrv_priv;
  int chipnum = 0;
  unsigned long addr;
  int len;
  int ret;

  /* Сначала проверяем на ошибки */
  if( (instr->addr > mtd->size) ||
      ((instr->addr + instr->len) > mtd->size) ||
      instr->addr & (mtd->erasesize -1)))
    return -EINVAL;

  /* Ищем номер для первой микросхемы */
  chipnum = (instr->addr >> priv->chipshift);
  addr = instr->addr & ((1 << priv->chipshift) - 1);
  len = instr->len;
  while(len)
  {
    if( (ret = dummy_flash_erase_one_block(map,
                       &priv->chips[chipnum], addr)) < 0)
      return ret;

    addr += mtd->erasesize;
    len -= mtd->erasesize;
    if(addr >> priv->chipshift)
    {
      addr = 0;
      chipnum++;
    }
  }

  instr->state = MTD_ERASE_DONE;
  if(instr->callback)
    instr->callback(instr);
  return 0;
}
```

Распечатка 4.5 Имитация функции sync

Распечатка 4.5

```
static void dummy_flash_sync(struct mtd_info *mtd)
{
  struct map_info *map = mtd->priv;
  struct dummy_private_info_struct *priv = map->fldrv_priv;
  struct flchip *chip;
  int i;

  DECLARE_WAITQUEUE(wait, current);

  for(i=0; i< priv->number_of_chips;i++)
  {
    chip = &priv->chips[i];
again:
    spin_lock(chip->mutex);

    switch(chip->state)
    {
    case FL_READY:
    case FL_STATUS:
      chip->oldstate = chip->state;
      chip->state = FL_SYNCING;
      break;
    case FL_SYNCING:
      spin_unlock(chip->mutex);
      break;
    default:
      add_wait_queue(&chip->wq, &wait);
      spin_unlock(chip->mutex);
      schedule();
      remove_wait_queue(&chip->wq, &wait);
      goto again;
    }
  }

  for(i--; i >=0; i--)
  {
    chip = &priv->chips[i];
    spin_lock(chip->mutex);
    if(chip->state == FL_SYNCING)
    {
      chip->state = chip->oldstate;
      wake_up(&chip->wq);
    }
    spin_unlock(chip->mutex);
  }
}
```

4.5 Драйверы связи с флеш-памятью

Независимо от типа устройства (NAND или NOR), основными операциями драйвера связи являются получение заполненной структуры **mtd_info** (вызовом соответствующей процедуры **probe**) и последующая регистрация её в ядре MTD. **mtd_info** будет иметь различные указатели на функции, зависящие от типа устройства. Структуры **mtd_info** помещаются в массив **mtd_table[]**. В этой таблице могут быть сохранены не более 16-ти таких устройств. Как записи в **mtd_table** экспортируются как символьные и блочные устройства, будет объясняться позже. Процесс работы драйвера связи с флеш-памятью можно разделить на:

- Создание и заполнение структуры **mtd_info**
- Регистрацию **mtd_info** в ядре MTD

4.5.1 Заполнение mtd_info для микросхемы NOR Flash

В случае, если ваша флеш-память подключена непосредственно к аппаратной шине процессора (как это обычно происходит), низкоуровневое оборудование NOR на плате зависит от следующего:

- Адреса, с которым связана флеш-память
- Размера флеш-памяти
- Размера шины; он может быть 8-и, 16-и или 32-х разрядным
- Подпрограммы для выполнения 8-и, 16-и и 32-х разрядного чтения и записи
- Подпрограммы выполнения массового копирования

Подключение NOR Flash определяется в структуре данных **map_info**, а база данных для плат различных конфигураций находится в каталоге **drivers/mtd/maps**. После того, как структура **map_info** заполнена, вызывается функция **do_map_probe()** с **map_info** в качестве аргумента. Эта функция возвращает указатель на структуру **mtd_info**, заполненную указателями на функции для работы с микросхемой флеш-памяти.

4.5.2 Заполнение mtd_info для микросхемы NAND Flash

Как упоминалось ранее, доступ к NAND Flash выполняется путём подключения линий данных и команд к линиям ввода-вывода процессора. Для микросхемы NAND Flash важными являются следующие контакты:

- контакт CE (Chip Enable, Выбор Микросхемы): микросхема NAND Flash выбрана, когда на этом выводе установлен низкой уровень.
- контакт WE (Write Enable, Разрешение Записи): микросхема NAND Flash принимает данные от процессора, когда на этом выводе установлен низкой уровень.
- контакт RE (Read Enable, Разрешение Чтения): микросхема NAND Flash посылает данные в процессор, когда на этом выводе установлен низкой уровень.
- контакт CLE (Command Latch Enable, Разрешение Фиксации Команды) и контакт ALE (Address Latch Enable, Разрешение Фиксации Адреса).

Эти выводы определяют назначение операции для микросхемы NAND. Как используются эти контакты, объясняет Таблица 4.2.

Таблица 4.2 Использование контактов ALE и CLE

ALE	*CLE*	*Регистр*
0	0	Регистр данных
0	1	Регистр команды
1	0	Регистр адреса

- контакт WP (Write Protect, Защита Записи): этот вывод может быть использован для защиты от записи.
- контакт RB (Ready Busy, Готов/Занят): используется в фазах передачи данных, чтобы указать, что микросхема занята.
- контакты IO (Ввода/Вывода): используются для передачи данных.

В отличие от микросхемы NOR Flash, драйвер которой для выделения памяти для структуры **mtd_info** вызывает **do_map_probe()**, драйверу для NAND Flash необходимо выделить память для структуры **mtd_info**. Ключом к этому является структура **nand_chip**, которая заполняется драйвером связи с NAND. Драйвером связи с NAND выполняются следующие шаги:

- Выделение памяти для структуры **mtd_info**.
- Выделение памяти для структуры **nand_chip** и заполнение необходимых полей.
- Сделать поле **priv** структуры **mtd_info** указывающим на структуру **nand_chip**.
- Вызов функции **nand_scan()**, которая будет пытаться опознать микросхему NAND, а также заполнить структуру **mtd_info** функциями для работы NAND.
- Регистрацию структуры **mtd_info** в ядре MTD.

Параметры для NAND, которые хранятся в структуре **nand_chip**, можно разделить на:

- Обязательные параметры:
  - **IO_ADDR_R**, **IO_ADDR_W**: адреса для доступа к линиям ввода-вывода микросхемы NAND.
  - **hwcontrol()**: эта функция реализует зависимый от платы механизм для установки и сброса выводов CLE, ALE и CE.
  - **eccmode**: используется для обозначения типа ECC для NAND Flash. Это включает отсутствие ECC, программное ECC и аппаратное ECC.
- Обязательные параметры, если ECC выполняется аппаратно. Некоторое оборудование обеспечивает генерацию ECC; в этом случае должны быть реализованы следующие функции. В случае программной реализации ECC, драйвер NAND предоставляет следующие функции как функции по умолчанию.
  - **calculate_ecc()**: функция для генерации ECC
  - **correct_data()**: функция для коррекции ECC
  - **enable_hwecc()**: функция для разрешения аппаратной генерации ECC
- Необязательные параметры. Драйвер предоставляет для следующих параметров функции/значения по умолчанию. Однако они могут быть заменены драйвером связи:
  - **dev_ready()**: эта функция используется для определения состояния флеш-памяти.
  - **cmdfunc()**: эта функция предназначена для отправки команд флеш-памяти.
  - **waitfunc()**: эта функция вызывается после выполнения записи или стирания.

Функцией по умолчанию, предоставляемой драйвером NAND, является функция опроса; в случае, если плата может подключить контакт RB к линии прерывания, эта функция может быть преобразована в функцию, управляемую прерываниями.
- **chip_delay**: это задержка для передачи данных из массива NAND в его регистры; значение по умолчанию составляет 20 мкс.

4.5.3 Регистрация mtd_info

Следующие шаги носят общий характер и применимы к обоим типам флеш-памяти, NAND и NOR. Основой регистрации является функция **add_mtd_device()**, которая добавляет устройство в массив **mtd_table[]**. Однако в большинстве случаев вам не потребуется использовать эту функцию напрямую, потому что вы захотите создать на микросхеме разделы.

Разбиение

Разбиение позволяет создать на флеш-памяти несколько разделов и добавить их в разные слоты массива **mtd_table[]**. Таким образом, разделы будут экспортироваться в приложение как несколько устройств. Разные разделы для доступа к массиву используют одни и те же функции. Например, вы, вероятно, хотели бы разделить 4-х Мб флеш-память на разделы по 1 Мб и 3 Мб, как показано на Рисунке 4.3.

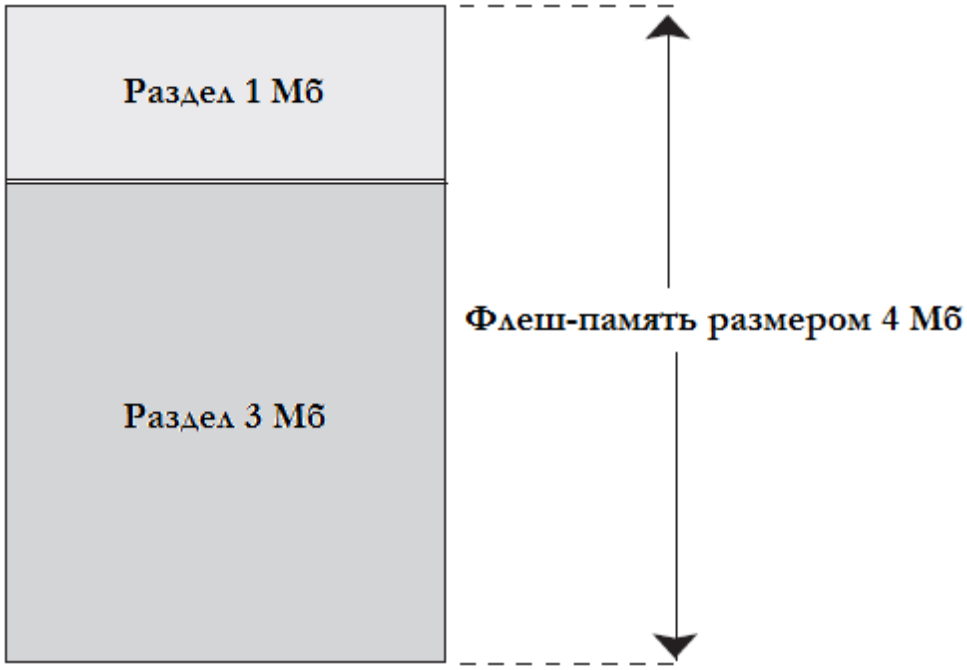

Рисунок 4.3 Флеш-память, разделённая на две части.

Ключом к разбиению является структура данных **mtd_partition**. Вероятно, для экспорта раздела вы бы определили массив из этой структуры данных.

```
struct mtd_partition partition_info[] =
{
  { .name="part1", .offset=0, .size= 1*1024*1024},
  { .name="part2", .offset=1*1024*1024, .size= 3*1024*1024}
```

```
}
```

Разделы добавляются с помощью функции **add_mtd_partition()**. Более подробную информацию об этом можно найти в примере драйвера связи в Разделе 4.5.4 83.

Объединение

Это мощная техника, которая позволяет объединить несколько отдельных устройств в одно устройство. Предположим, что у вас в системе есть два устройства флеш-памяти. На Рисунке 4.4 показана одна флеш-память, имеющая три раздела, и вторая флеш-память, имеющая один раздел.

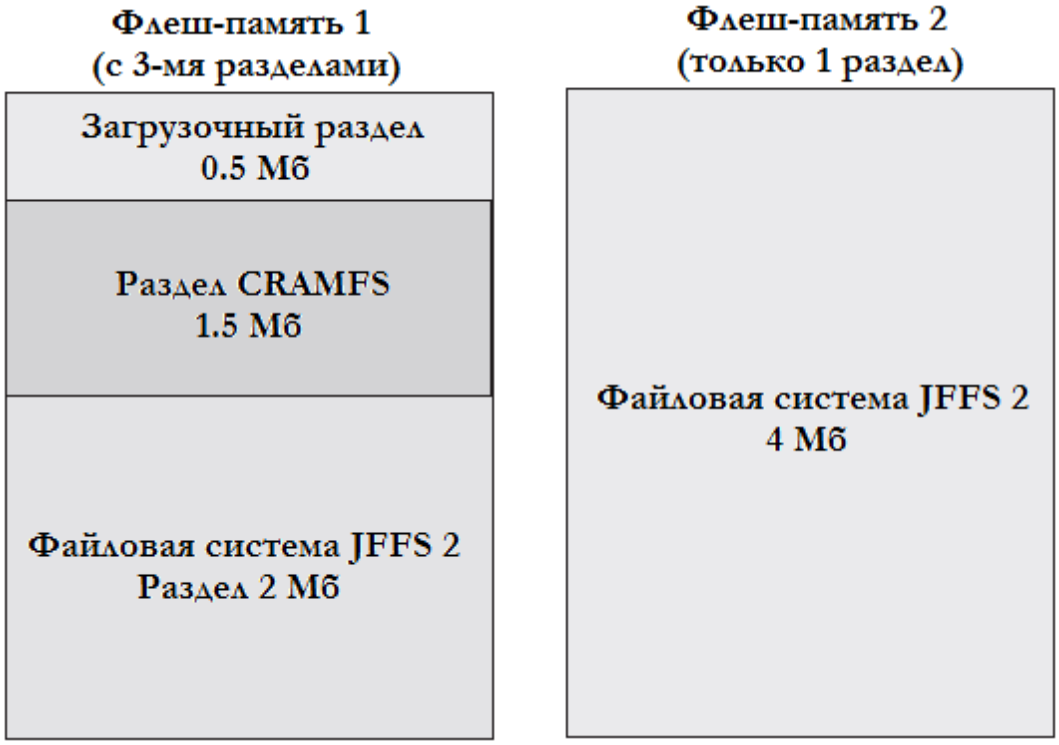

Рисунок 4.4 Флеш-память, содержащая несколько разделов.

Так как файловую систему необходимо распределить по двум микросхемам флеш-памяти, как правило, вам пришлось бы создавать две файловые системы на каждой из микросхем. Это громоздкая техника, потому что придётся поддерживать две файловые системы. Этого можно избежать объединением микросхем флеш-памяти в одно виртуальное устройство, как показано на Рисунке 4.5. Тогда в системе должен быть смонтирован только один экземпляр файловой системы.

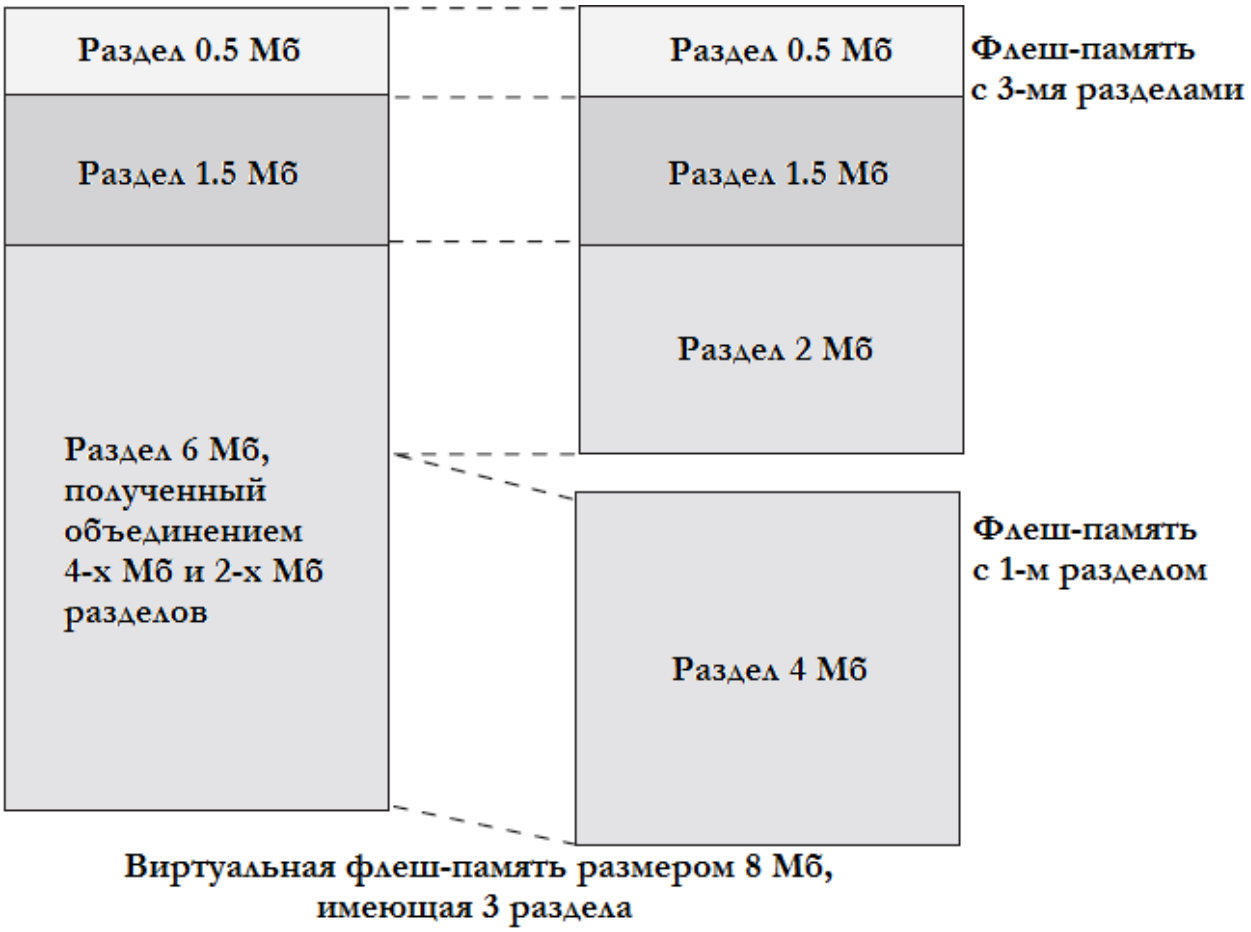

Рисунок 4.5 Два устройства флеш-памяти, объединённые в одно виртуальное устройство.

4.5.4 Пример драйвера связи для NOR Flash

Давайте рассмотрим пример платы на основе MIPS с двумя микросхемами флеш-памяти, имеющими адреса, показанные на Рисунке 4.6.

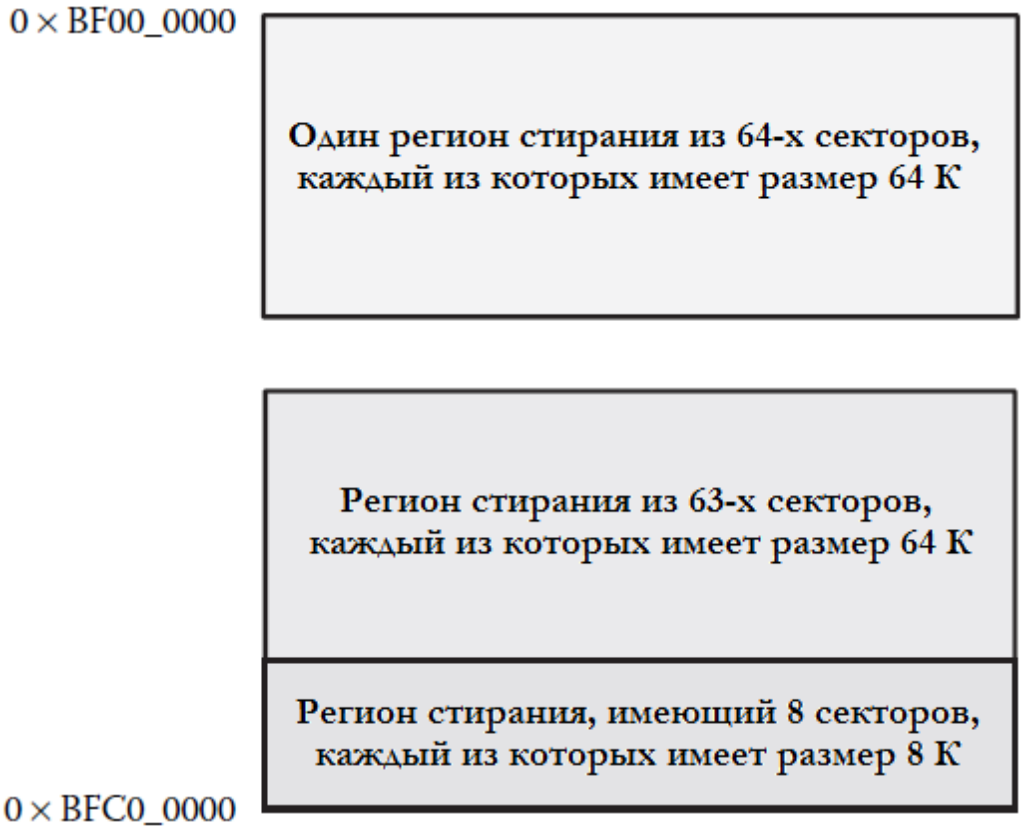

Рисунок 4.6 Карта флеш-памяти.

Ниже приведены сведения об использовании этих микросхем флеш-памяти.

- Первая микросхема флеш-памяти привязана к адресу 0xBFC00000 и имеет размер 4 Мб. Первая микросхема является загрузочной памятью и имеет два региона стирания. Первый регион стирания имеет восемь секторов, каждый из которых имеет размер 8 Кб; эта 64 Кб область используется для хранения загрузчика и параметров конфигурации загрузки. Вторая область стирания имеет секторы размером 64 Кб и полностью используется для хранения файловой системы JFFS2.
- Вторая микросхема флеш-памяти привязана к адресу 0xBF000000 и тоже имеет размер 4 Мб. Вторая микросхема содержит только один регион стирания, а все сектора имеют размер 64 Кб. Эта флеш-память полностью используется для хранения файловой системы JFFS2.

Требованием к нашему драйверу связи с флеш-памятью является разделение первой флеш-памяти, стартующей с адреса 0xBFC00000, на две части. Первый раздел размером 64 Кб будет загрузочным разделом. Второй раздел будет объединён с флеш-памятью, стартующей с адреса 0xBF000000, для хранения файловой системы JFFS2.

Давайте начнём с заголовочных файлов и определений.

```
/* mtd-bsp.c */
#include <linux/config.h>
#include <linux/module.h>
#include <linux/types.h>
#include <linux/kernel.h>
```

```
#include <asm/io.h>
#include <linux/mtd/mtd.h>
#include <linux/mtd/map.h>
#include <linux/mtd/cfi.h>
#include <linux/mtd/partitions.h>
#include <linux/mtd/concat.h>

#define WINDOW_ADDR_0 0xBFC00000
#define WINDOW_SIZE_0 0x00400000
#define WINDOW_ADDR_1 0xBF000000
#define WINDOW_SIZE_1 0x00400000
```

map_info содержит информацию о начальном адресе (физическом) каждой микросхемы, объёме памяти и размере шины. Они используются процедурами проверки подключения микросхемы.

```
static struct map_info dummy_mips_map[2] = {
 {
  .name      = "DUMMY boot flash",
  .phys      = WINDOW_ADDR_0,
  .size      = WINDOW_SIZE_0,
  .bankwidth = 4,
 },
 {
  .name      = "Dummy non boot flash",
  .phys      = WINDOW_ADDR_1,
  .size      = WINDOW_SIZE_1,
  .bankwidth = 4,
 }
};
```

Следующая структура используется для создания разделов на загрузочной флеш-памяти.

```
static struct mtd_partition boot_flash_partitions [] = {
 {
    .name = "BOOT",
    .offset = 0,
    .size = 0x00010000,
 },
 {
    .name = "JFFS2",
    .offset = 0x00010000,
    .size = 0x003f0000,
 },
};

/*
 * Следующая структура содержит указатели mtd_info для
 * разделов, которые будут объединены
 */
static struct mtd_info *concat_partitions[2];

/*
```

```
 * Следующая структура содержит указатели на структуры mtd_info для
 * каждого из устройств флеш-памяти
 */
static struct mtd_info * mymtd[2], *concat_mtd;
```

Основной функцией является функция **init_dummy_mips_mtd_bsp()**. Её реализацию можно увидеть в Распечатке 4.6 86. Функция делает следующее:

- Проверяет наличие флеш-памяти по адресу 0xBFC00000 и заполняет структуру MTD для этой флеш-памяти в **mymtd[0]**
- Проверяет наличие флеш-памяти по адресу 0xBF000000 и заполняет структуру MTD для этой флеш-памяти в **mymtd[1]**
- Создаёт два раздела на флеш-памяти с начальным адресом 0xBFC00000
- Объединяет второй раздел с флеш-памятью, которая стартует с адреса 0xBF000000, а затем создаёт новое устройство с помощью вызова функции **add_mtd_device()**

Завершает всё функция очистки:

```
static void __exit cleanup_dummy_mips_mtd_bsp(void)
{
    mtd_concat_destroy(concat_mtd);
    del_mtd_partitions(mymtd[0]);
    map_destroy(mymtd[0]);
    map_destroy(mymtd[1]);
}

module_init (init_dummy_mips_mtd_bsp);
module_exit (cleanup_dummy_mips_mtd_bsp);
MODULE_LICENSE ("GPL");
MODULE_AUTHOR ("Embedded Linux book");
MODULE_DESCRIPTION ("Sample Mapping driver");
```

Распечатка 4.6 Функция init_dummy_mips_mtd_bsp

Распечатка 4.6

```
int __init init_dummy_mips_mtd_bsp (void)
{

  /* Сначала проверяем загрузочную флеш-память */
  dummy_mips_map[0].virt =
    (unsigned long)ioremap(
                   dummy_mips_map[0].phys,dummy_mips_map[0].size);
  simple_map_init(&dummy_mips_map[0]);
  mymtd[0] = do_map_probe("cfi_probe", &dummy_mips_map[0]);
  if(mymtd[0])
    mymtd[0]->owner = THIS_MODULE;

  /* Повторяем для второй флеш-памяти */
  dummy_mips_map[1].virt =
    (unsigned long)ioremap(dummy_mips_map[1].phys,
                           dummy_mips_map[1].size);
```

```
  simple_map_init(&dummy_mips_map[1]);
  mymtd[1] = do_map_probe("cfi_probe", &dummy_mips_map[1]);
  if(mymtd[1])
    mymtd[1]->owner = THIS_MODULE;
  if (!mymtd[0] || !mymtd[1])
    return -ENXIO;

  /*
   * Теперь делим загрузочную флеш-память. Мы заинтересованы в
   * новом объекте mtd для второго раздела, так как мы будем
   * объединять его с другой флеш-памятью.
   */
  boot_flash_partitions[1].mtdp = &concat_partitions[0];
  add_mtd_partitions(mymtd[0], boot_flash_partitions, 2);

  /*
   * concat_partitions[1] должен содержать указатель mtd_info
   * для 2-го раздела. Выполняем объединение
   */
  concat_partitions[1] = mymtd[1];
  concat_mtd = mtd_concat_create(concat_partitions, 2,
                                 "JFFS2 flash concatenate");
  if(concat_mtd)
    add_mtd_device(concat_mtd);
  return 0;
}
```

4.6 Блочные и символьные устройства MTD

Как упоминалось ранее, MTD устройства экспортируются в пользовательское пространство двумя способами: как символьное и как блочное устройство. Символьные устройства представлены с использованием следующих имён устройств:

```
/dev/mtd0
/dev/mtdr0
/dev/mtd1
/dev/mtdr1
...
/dev/mtd15
/dev/mtdr15
```

Все символьные устройства имеют старший номер 90. Символьные устройства экспортируются как или символьные устройства для чтения и записи, или же как символьные устройства только для чтения. Это достигается с помощью младших номеров. Все MTD устройства, имеющие нечётные младшие номера (1, 3, 5, ...), экспортируются как устройства только для чтения. Таким образом, и **/dev/mtd1**, и **/dev/mtdr1** указывают на одно и то же устройство (устройство, содержащееся во втором слоте в **mtd_table[]**); первое может быть открыто в режиме чтения-записи, в то время как второе может быть открыто только в режиме чтения. Ниже приводится список команд ввода/вывода (ioctl), которые поддерживаются символьными устройствами MTD.

- **MEMGETREGIONCOUNT**: команда для передачи пользователю числа регионов

стирания
- **MEMGETREGIONINFO**: команда для получения информации о регионе стирания
- **MEMERASE**: команда стереть заданный сектор флеш-памяти
- **MEMWRITEOOB/MEMREADOOB**: команды, используемые для доступа к дополнительным данным
- **MEMLOCK/MEMUNLOCK**: команда, используемые для блокировки указанных секторов при условии наличия аппаратной поддержки

Блочные устройства имеют старший номер 31. Поддерживаются до 16-ти младших устройств. Блочные устройства используются для монтирования файловых систем поверх устройств флеш-памяти (смотрите Рисунок 4.7).

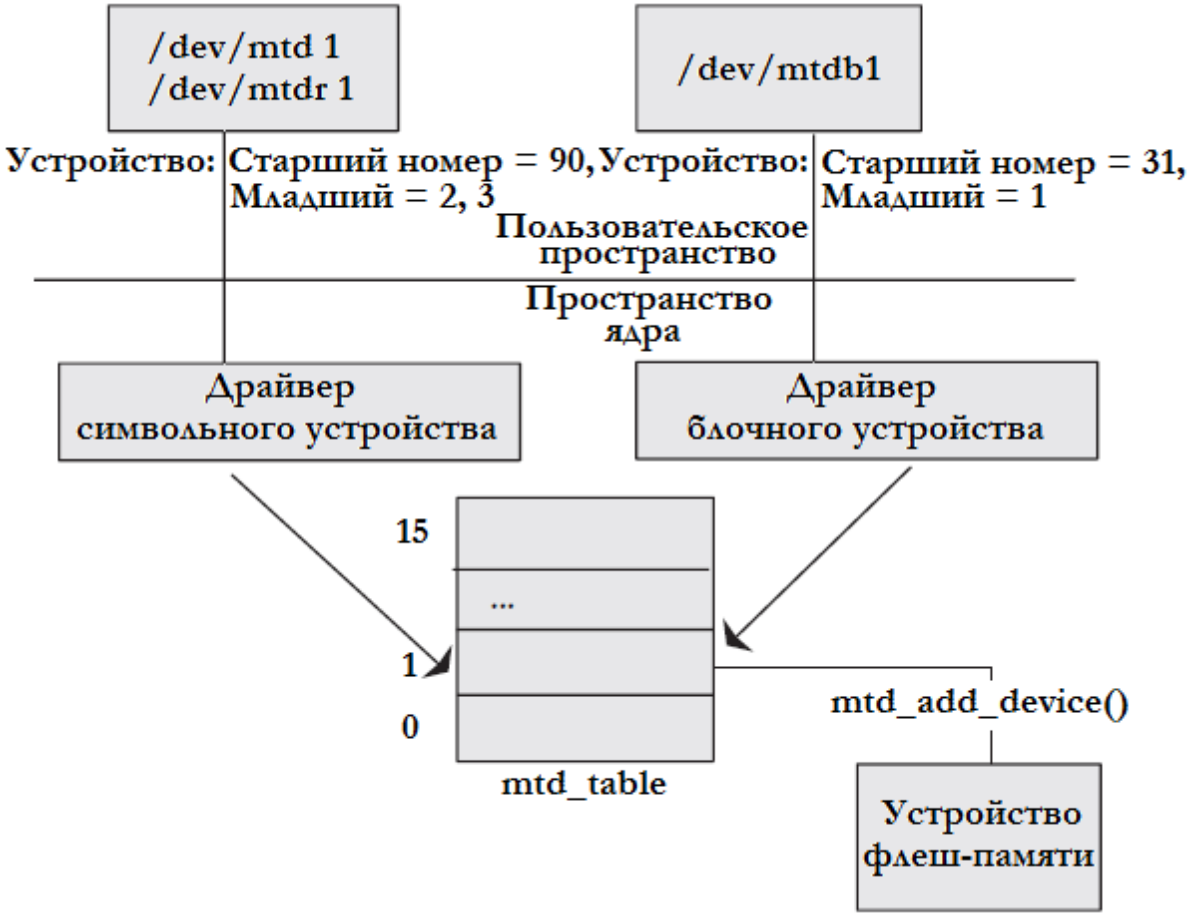

Рисунок 4.7 MTD устройство, экспортируемое как символьное и блочное устройство.

4.7 Пакет mtdutils

Пакет **mtdutils** представляет собой набор полезных программ, таких как создание файловых систем и тестирование целостности устройства флеш-памяти. Обратите внимание, что некоторые утилиты предназначены для использования на ПК (например, инструменты для создания файла образа JFFS2), а некоторые могут быть использованы на целевой платформе (например, утилиты для стирания устройства флеш-памяти). Программы для запуска на целевой платформе должны быть собраны с помощью кросс-компилятора. Ниже приводится описание отдельных утилит этого пакета.

- **erase**: эта утилита используется для стирания указанного количества блоков по заданному смещению.
- **eraseall**: эта утилита используется для стирания всего устройства (поскольку устройства разделены на разделы, с помощью этой программы могут быть стёрты целые разделы).
- **nftl_fromat**: эта утилита используется для создания на устройстве MTD раздела NFTL (NAND Flash Translation Layer, Уровень Трансляции NAND Flash). Она используется для систем диск-на-микросхеме (disk-on-chip) для форматирования диска на микросхеме.
- **nftldump**: эта утилита используется для вывода данных раздела NFTL.
- **doc_loadbios**: эта утилита используется для перепрограммирования диска-на-микросхеме новой прошивкой (например, GRUB).
- **doc_loadip1**: эта утилита используется для загрузки IPL (кода инициализации) во флеш-память DOC (Disk-On-Chip, Диск-На-Микросхеме).
- **ftl_fromat**: эта утилита используется для создания на флеш устройстве раздела FTL.
- **nanddump**: эта утилита выводит содержимое микросхем NAND (необработанные, или те, которые содержатся в DOC).
- **nandtest**: эта утилита тестирует NAND устройства (то есть записывает и читает обратно из NAND флеш-памяти и проверяет, удачна ли запись).
- **nandwrite**: эта утилита используется для записи двоичного образа во флеш-память NAND.
- **mkfs.jffs**: эта утилита создаёт образ JFFS, который можно записать во флеш-память, используя заданное дерево каталогов.
- **mkfs.jffs2**: эта утилита создает образ JFFS2, который можно записать в флеш-память, используя заданное дерево каталогов.
- **lock**: эта утилита блокирует один или несколько секторов флеш-памяти.
- **unlock**: эта утилита разблокирует все сектора устройства флеш-памяти.
- **mtd_debug**: это очень полезная утилита, которая может быть использована для получения информации об устройстве флеш-памяти, чтения, записи и стирании устройств флеш-памяти.
- **fcp**: это утилита используется для копирования файла на устройство флеш-памяти.

4.8 Встраиваемые файловые системы

Этот раздел рассказывает о популярных встраиваемых файловых системах. Большинство из них основано на флеш-памяти, остальные являются файловыми системами на основе памяти. Файловые системы на основе памяти могут быть использованы во время загрузки для хранения корневой файловой системы, а также для хранения изменяемых данных, которые не обязательно сохранять между перезагрузками; они обсуждаются в первую очередь.

4.8.1 Ramdisk

Ramdisk (или виртуальный диск), как следует из названия, это способ, которым Linux эмулирует жёсткий диск, используя память. Виртуальный диск мог бы потребоваться, если для хранения корневой файловой системы нет традиционного устройства хранения данных, такого как жёсткий диск или флеш-память. Обратите внимание, что виртуальный диск это не файловая система, а механизм, посредством которого можно загрузить настоящую файловую систему в память и использовать её в качестве корневой файловой системы.

Механизм, посредством которого будет загружен образ ядра вместе с корневой файловой системы в память, обеспечивает **initrd**. Для использования **initrd** потребуются следующие действия:

- Создание образа **initrd** и упаковка его вместе с образом ядра. Необходимо создать образ виртуального диска на машине для разработки (ПК). Затем необходимо упаковать образ виртуального диска вместе с ядром. На некоторых платформах эта опция доступна при сборке ядра. Обычно в ELF файле есть раздел под названием **.initrd**, который содержит образ виртуального диска, который может быть использован загрузчиком напрямую.
- Изменение загрузчика для загрузки **initrd**.

Initrd также предоставляет механизм, с помощью которого можно переключиться на новую корневую файловую систему на более позднем этапе во время работы системы. Таким образом, **initrd** можно использовать для восстановления системы, а также для процедуры обновления при выпуске новых версий. Шаги, необходимые для создания образа **initrd**, обсуждаются в Главе 8 240.

4.8.2 RAMFS

Часто встроенная система имеет файлы, которые не нужны после перезагрузки, и обычно хранимые в каталоге **/tmp**. Хранение таких файлов в памяти было бы лучшим вариантом, чем хранение их во флеш-памяти, поскольку запись во флеш-память стоит дорого. Вместо этого можно было бы использовать RAMFS (RAM File System, Файловую систему в Оперативной Памяти); она не имеет фиксированного размера и сжимается и растёт вместе с файлами, хранящимися в ней.

4.8.3 CRAMFS (Compressed RAM File System)

Это очень полезная файловая система для флеш-памяти, которая появилась в ядре версии 2.4. Это файловая система "только для чтения" с высокой степенью сжатия. CRAMFS является обычной файловой системой; то есть для общения с блочным устройством, содержащим фактические данные, данные проходят через буферный кэш. Следовательно, необходимо включить режим MTD драйвера блочного устройства, если вы хотите использовать CRAMFS. CRAMFS использует для сжатия вызовы **zlib** и выполняет сжатие для каждого блока размером 4 Кб (размер страницы). Образ CRAMFS, который должен быть записан во флеш-память, может быть создан с помощью программы под названием **mkcramfs**.

4.8.4 Журналирующие файловые системы для флеш-памяти — JFFS and JFFS2

Традиционные файловые системы не были предназначены ни для встраиваемых систем, ни для механизмов хранения на основе флеш-памяти. Давайте кратко перечислим то, что мы хотим от файловой системы на основе флеш-памяти:

- Выравнивание износа
- Отсутствие повреждения данных при внезапном прекращении подачи электроэнергии
- Прямое использование API на уровне MTD вместо уровней трансляции

В 1999 году компания Axis Communications выпустила JFFS для ядра Linux версии 2.0 со всеми вышеперечисленными возможностями. Она была быстро перенесена на серии ядер 2.2 и 2.3. Но тогда не хватало поддержки сжатия и таким образом был начат проект JFFS2. JFFS2 был выпущена для серии ядер версии 2.4 и быстро обогнала JFFS из-за расширенных возможностей. И JFFS, и JFFS2 являются журналирующими файловыми системами; любые изменения в файле записываются как данные в журнале, который непосредственно хранится во флеш-памяти (журналы также называются узлами).

Такой журнал будет содержать:

- Идентификатор файла, для которого ведётся журнал
- Версия, которая является уникальной в журнале, принадлежащем данному файлу
- Метаданные, например, временная метка
- Данные и размер данных
- Смещение данных в файле

Запись в файл создаёт журнал, в который записывается смещение в файле, где записываются данные, и размер записанных данных вместе с фактическими данными. Когда файл должен быть прочитан, журналы прочитываются, и используя размер данных и смещение, пересоздаются файлы. Со временем некоторые журналы устаревают частично или полностью; они должны быть очищены. Этот процесс называется уборкой мусора. Результат уборки мусора заключается в определении чистых стёртых блоков. Уборка мусора должна также предусматривать выравнивание износа, для уверенности, что все стёртые блоки имеют шанс появиться в списке свободных.

Основными особенностями JFFS2 файловой системы являются:

- ***Управление стиранием блоков***: в JFFS2 стёртые блоки размещены в трёх списках: чистом, грязном и свободном. Чистый список содержит только достоверные журналы (то есть журналы не ставшие недействительными из-за появления более новых). Грязный список содержит один или несколько журналов, которые являются устаревшими и, следовательно, могут быть удалены при вызове уборщика мусора. Свободный список не содержит журналы и, следовательно, может быть использован для сохранения новых журналов.
- ***Уборка мусора***: уборка мусора в JFFS2 происходит в контексте отдельного потока, который запускается, когда файловая система JFFS2 монтируется. Для обеспечения выравнивания износа в каждые 99 из 100 раз этот поток будет забирать блок из грязного списка и в один из 100 будет забирать блок из чистого списка. JFFS2 резервирует пять блоков для выполнения уборки мусора (это число, кажется, было выбрано эвристически).
- ***Сжатие***: отличительная особенность между JFFS и JFFS2 в том, что JFFS2 предоставляет несколько способов сжатия, в том числе zlib и rubin.

Образ файловой системы JFFS/JFFS2 для целевой платформы может быть создан с помощью команд **mkfs.jffs** и **mkfs.jffs2**. Обе эти команды принимают в качестве аргумента дерево каталогов и создают образ на основном компьютере, используемом для разработки. Эти образы должны быть загружены на целевую платформу и записаны в соответствующие разделы флеш-памяти; такая возможность предоставляется большинством загрузчиков Linux.

4.8.5 NFS — Сетевая файловая система

Для монтирования файловой системы по сети может быть использована сетевая файловая система. С помощью NFS можно экспортировать популярные файловые системы ПК EXT2 и EXT3; так что разработчик может использовать для хранения файловой системы EXT2 или EXT3 стандартный ПК с Linux и обращаться к ней на встроенной системе как к корневой файловой системе. Часто во время отладки разработчик хотел бы вносить изменения в корневую файловую систему. В таком случае запись во флеш-память может быть дорогостоящей (поскольку флеш-память имеет ограниченное число циклов записи) и отнимать много времени. NFS может быть хорошим выбором, при условии, что готов сетевой драйвер. Также с NFS у вас нет никаких ограничений по объёму, поскольку все хранение осуществляется на удалённом сервере.

Ядро Linux предоставляет механизм для автоматического монтирования файловой системы при помощи NFS во время загрузки системы при выполнении следующих действий:

- Во время сборки должна быть включена опция конфигурации **CONFIG_NFS_FS**, которая позволяет смонтировать файловую систему с удалённого сервера, и **CONFIG_ROOT_NFS**, которая позволяет использовать NFS в качестве корневой файловой системы. Также, если вы монтируете корневую файловую систему EXT2 или EXT3 удалённого ПК, в ядре должна быть включена поддержка этой файловой системы.
- Поскольку во время загрузки для подключения к серверу NFS становится необходимым IP адрес, вы можете включить автоматическую конфигурацию по сети с помощью BOOTP, RARP или DHCP.
- Должны быть указаны параметры командной строки ядра для указания сервера NFS.

Для более подробной информации по синтаксису аргументов командной строки обратитесь к **Documentation/nfsroot.txt** в дереве исходных текстов ядра.

4.8.6 Файловая система PROC

Файловая система **proc** является логической файловой системой, используемой ядром для экспорта своей информации во внешний мир. Файловая система **proc** может быть использована для мониторинга системы, отладки и настройки. Файлы в **proc** создаются на лету, когда они открываются. Файлы могут быть доступны только для чтения или для чтения и записи в зависимости от информации, экспортируемой файлами. Только читаемые файлы системы **proc** могут быть использованы для получения информации о состоянии ядра, которая не может быть изменена во время работы. Примером только читаемого файла в **proc** является информация о состоянии процесса. Читаемые и записываемые файлы в **proc** содержат информацию, которая может быть изменена во время выполнения; ядро использует изменяемые значения на лету. Примером этого является время поддержания tcp соединения. Файловая система **proc** монтируется стартовыми скриптами в стандартную точку монтирования **/proc**.

Эта файловая система требует использования оперативной памяти. Хотя ОЗУ является дорогим, всё же преимущества наличия файловой системы **proc** перевешивают недостатки. Файловая система **proc** используются стандартными программами Linux, такими как **ps** и **mount**. Поэтому читателям рекомендуется не удалять файловую систему **proc**, если они не уверены в том, что делают.

4.9 Оптимизация пространства хранения

В этом разделе рассматривается основная проблема, часто встречающаяся во встраиваемых системах: как эффективно использовать место для хранения. Так происходит потому, что микросхемы флеш-памяти являются дорогостоящими. И хотя за несколько последних лет их цены видели резкое снижение, они всё ещё являются основным компонентом спецификации оборудования (BOM, Bill Of Materials). Проблема становится более острой, когда для работы на вашей системе вы берёте открытый исходный код из Интернета; если программа была написана с учётом встроенных систем, есть очень маленький шанс, что программа была оптимизирована по размеру. Такая программа могла бы иметь много нежелательного кода, который может увеличить размер. Разобьём эту часть на три основные части:

- Оптимизация ядра Linux для эффективного хранения
- Оптимизация размера приложений
- Использование сжатых файловых систем для хранения ядра и приложений. Поскольку такие файловые системы, как CRAMFS и JFFS2, обеспечивающие сжатие, уже обсуждались, этот раздел игнорирует эту тему.

4.9.1 Оптимизация размера ядра

Основным способом уменьшения размера ядра является удаление из ядра нежелательного кода. Помимо этого, для создания образов ядра меньшего размера может быть использована методика оптимизации с помощью компилятора (с помощью опции **-Os**). (Это справедливо также и для приложений.) Ядро версии 2.6 имеет отдельную опцию сборки для встраиваемых систем; для создания более оптимизированного ядра используется опция **CONFIG_EMBEDDED**. Кроме того, во время сборки могут быть отключены некоторые субмодули ядра, такие как подсистема подкачки. Независимо от версии ядра, крайне важно убедиться, что во время настройки ядра при выборе различных подсистем включены только необходимые параметры, чтобы не раздувать размер ядра без необходимости.

Ядро версии 2.6 имеет проект с открытым исходным кодом, который нацелен на то, чтобы запускать его на встраиваемых системах. Это "Linux tiny kernel project, проект крошечного ядра Linux", начатый Маттом Маккалом. Целью данного проекта является поддержание дерева исходников ядра, которое включает в себя патчи, направленные на создание ядра меньшего размера и оптимальное использование ядром памяти. Некоторые из интересных патчей, направленных на уменьшение размера ядра:

- Опция для удаления **printk** и всех строк, передаваемых в **printk**
- Проверка компиляции, которая сообщает о слишком большом использовании встроенных функций, потому что они раздувают размер кода

Дерево очень маленького ядра можно найти на www.selenic.com.

Поскольку настройка памяти ядра использует много тех же трюков, что и оптимизация размера ядра, в последнем разделе этой главы обсуждаются методы для настройки памяти ядра.

4.9.2 Оптимизации пространства, занимаемого приложениями

Оптимизация пространства, занимаемого приложениями, может быть эффективно выполнена с помощью следующих шагов:

- Удаление лишнего кода в отдельных приложениях
- Использование таких инструментов, как оптимизатор библиотек
- Использование уменьшенных программ/дистрибутивов, нацеленных на использование во встраиваемых системах
- Использование уменьшенной библиотеки языка Си, такой как **uClibc**

Оптимизатор библиотек

Этот инструмент предназначен для удаления из общих библиотек нежелательного кода. Вспомним из обсуждения общих библиотек в Главе 2, что разделяемые библиотеки могут содержать нежелательный код, который может быть никогда не используемым, но всё же может расходовать драгоценное пространство для хранения. Оптимизатор библиотек представляет собой инструмент с открытым кодом, который используется в конце сборки для сканирования разделяемых библиотек и перестройки их так, чтобы они содержали только объектные файлы, необходимые для системы. Веб-сайт для разработки оптимизатора библиотек: http://libraryopt.sourceforge.net.

Тем не менее, оптимизатор библиотек не может быть использован в системах, в которых приложения должны загружаться и выполняться динамически (на встраиваемых системах это может случиться очень редко), так как библиотека языка Си может не содержать функций, необходимых для выполнения новых приложений.

Уменьшенные библиотеки языка Си

Библиотека языка Си является одним из важнейших компонентов пользовательского пространства; все приложения должны компоноваться с библиотекой языка Си для часто используемых функций. Стандартная библиотека языка Си **libc.so** и **libc.a**, которая доступна на веб-сайте GNU, часто называется **glibc**. Тем не менее, **glibc** предназначена в большей степени для настольных и серверных сред. Она содержит избыточный код, который не найдёт большого использования во встраиваемых системах, используя при этом дорогое место для хранения. По словам сопровождающего **glibc**, Ульриха Дреппера,

> Как правило, что-то вроде glib или утилит gnu не будет использоваться во встраиваемых системах. ... Эти варианты на самом деле предназначены не для встраиваемых сред, а для систем, на которых работает Linux (например, [S]VGA, жёсткий диск, мышь, ОЗУ 64 Мб и тому подобное).

Есть две популярные альтернативы использованию **glibc** на встраиваемых системах: **dietlibc** и **uclibc**. Обе будут рассмотрены ниже.

- ***Dietlibc***: **dietlibc** - это небольшая библиотека языка Си, которая может быть загружена с http://www.dietlibc.org/
- ***Uclibc***: **uclibc** является очень популярной встраиваемой библиотекой языка Си. Этот проект был начат и поддерживается Эриком Андерсеном на Веб-сайте www.uclibc.org. Одной из важных особенностей **uclibc** является то, что она может быть использована

как на процессорах с MMU, так и без MMU. Список процессоров, которые поддерживаются **uclibc**, включает в себя:
- x86
- ARM
- MIPS
- PPC
- M68K
- SH
- V850
- CRIS
- MicroBlaze ™

4.9.3 Приложения для встраиваемого Linux

Теперь обсудим некоторые популярные дистрибутивы и приложения, используемые в системах на встраиваемом Linux.

Busybox

Программа **Busybox** является программой, поддерживающей мультивызов. Это означает, что в одном небольшом исполняемом файле реализованы некоторые часто используемые во встроенных системах программы. **Busybox** нацелена на использование во встраиваемых системах. Она также имеет механизм конфигурирования, с помощью которого на этапе сборки могут быть выбраны только необходимые для системы программы. **Busybox** может быть загружена с http://busybox.net. **Busybox** содержит следующие основные программы, известные в терминологии **Busybox** как апплеты (минипрограммы):

- Оболочки, такие как **ash**, **lash**, **hush** и другие
- Основные утилиты, такие как **cat**, **chmod**, **cp**, **dd**, **mv**, **ls**, **pwd**, **rm** и другие
- Утилиты управления процессами и мониторинга, такие как **ps**, **kill** и другие
- Утилиты для загрузки модулей, такие как **lsmod**, **rmmod**, **modprobe**, **insmod** и **depmod**
- Системные программы, такие как **reboot**, **init** и другие
- Сетевые утилиты, такие как **ifconfig**, **route**, **ping**, **tftp**, **httpd**, **telnet**, **wget**, **udhcpc** (клиент dhcp) и другие
- Утилиты для входа в систему и управления паролями, такие как **login**, **passwd**, **adduser**, **deluser** и другие
- Утилиты для архивирования, такие как **ar**, **cpio**, **gzip**, **tar** и другие
- Утилиты для ведения системных журналов, такие как **syslogd**

Сборка **Busybox** делится на два этапа:

- Конфигурирование: для выбора апплетов, которые вы хотите собрать, дайте команду **make menuconfig**.
- Сборка **Busybox**: для сборки исполняемого файла **busybox** дайте команду **make**.

Следующим шагом является установка **Busybox** на целевой платформе. Это достигается вызовом **Busybox** с опцией в **--install** в сценарии запуска системы (например, скрипте **rc**).

```
busybox mount -n -t proc /proc /proc
```

```
busybox --install -s
```

Команда **install Busybox** создаёт мягкие ссылки всех апплетов, выбранных во время процесса конфигурации. Например, выполнение после установки **ls -l** в каталоге **/bin** даёт вывод, показанный ниже:

```
-rwxr-xr-x    1 0        0          1065308 busybox
lrwxrwxrwx    1 0        0                7 init -> busybox
lrwxrwxrwx    1 0        0               12 ash -> /bin/busybox
lrwxrwxrwx    1 0        0               12 cat -> /bin/busybox
lrwxrwxrwx    1 0        0               12 chmod -> /bin/busybox
lrwxrwxrwx    1 0        0               12 cp -> /bin/busybox
lrwxrwxrwx    1 0        0               12 dd -> /bin/busybox
lrwxrwxrwx    1 0        0               12 echo -> /bin/busybox
```

Как видно, для каждого выбранного апплета **install Busybox** создал мягкую ссылку по имени этого апплета на себя. Когда вызывается какая-либо программа (скажем **chmod**), **Busybox** получает название программы из первого аргумента командной строки и вызывает соответствующую функцию.

Tinylogin

Tinylogin является программой, поддерживающей мультивызов, схожей с **Busybox**, и используется для реализации UNIX-подобного входа в систему и доступа к приложениям. Ниже приводится список функциональных возможностей, реализуемых **Tinylogin**.

- Добавление и удаление пользователей
- Приложения **login** и **getty**
- Приложение для изменения пароля **passwd**

Tinylogin может быть загружена с www.tinylogin.org.

Ftp сервер

FTP сервер полезен для копирования файлов на и из встроенной системы. Доступны два FTP сервера, стандартный сервер **wu-ftpd** и более популярный сервер **proftpd**, который легко конфигурируется. Они могут быть загружены с www.wu-ftpd.org и www.proftpd.org, соответственно.

Веб сервер

Веб-серверы необходимы для удалённого управления встроенным устройством. Есть много веб-серверов, предназначенных для встраиваемого Linux, наиболее популярные из которых описаны ниже.

- ***BOA***: встраиваемый однозадачный HTTP сервер, доступный на http://www.boa.org/
- ***mini_httpd***: небольшой веб-сервер, предназначенный для небольшого и среднего веб-трафика. Он может быть загружен с http://www.acme.com/
- ***GoAhead***: этот популярный веб-сервер с открытым кодом предназначен для встраиваемых систем и может быть загружен с http://www.goahead.com

4.10 Уменьшение размера памяти, занимаемого ядром

В этом разделе описываются методы для уменьшения размера памяти, используемого ядром. Ядро Linux не участвует в подкачке и, следовательно, всё ядро (код, данные и стек) всегда находится в оперативной памяти. Прежде чем углубляться в методы оптимизации, давайте разберёмся, как оценить объём памяти, используемый ядром. Размер статической памяти, которая будет использоваться ядром, можно определить используя команду **size**; эта утилита выводит список размеров различных секций и общий размер объектного файла. Ниже приведён вывод для ядра, скомпилированного для процессора MIPS.

```
bash >mips-linux-size vmlinux
text        data     bss     dec      hex    filename
 621244   44128     128848  794220   c1e6c    vmlinux
```

Показанная выше распечатка показывает, что 621 Кб памяти используется текстом ядра, 44 Кб данными и 128 Кб BSS. Обратите внимание, что BSS не является частью образа хранения ядра; код запуска выделяет память для BSS и заполняет его 0, таким образом, эффективно создавая его во время выполнения.

Следующая часть полезной информации отображается во время запуска Linux:

```
Memory: 61204k/65536k available (1347k kernel code, 4008k
reserved, 999k data, 132k init, 0k highmem)
```

Данное сообщение показывает, что из 65 536 Кб памяти, присутствующей в системе, около 4 Мб было использовано для хранения разделов текста ядра, кода и инициализации, и для создания структур данных для управления памятью. Остальные 61 Мб доступны ядру для динамического распределения памяти.

Отчёт о расходе памяти в системе во время работы можно получить через **/proc/meminfo**. Пример вывода для ядра версии 2.4:

```
# cat /proc/meminfo

        total:    used:    free:  shared: buffers:  cached:
Mem:  62894080 47947776 14946304        0  4964352 23674880
Swap:        0        0        0
MemTotal:         61420 Kb
MemFree:          14596 Kb
MemShared:            0 Kb
Buffers:           4848 Kb
Cached:           23120 Kb
SwapCached:           0 Kb
Active:           32340 Kb
ActiveAnon:       10760 Kb
ActiveCache:      21580 Kb
Inact_dirty:       6336 Kb
Inact_clean:        236 Kb
Inact_target:      7780 Kb
HighTotal:            0 Kb
HighFree:             0 Kb
LowTotal:         61420 Kb
```

```
LowFree:          14596 Kb
SwapTotal:            0 Kb
SwapFree:             0 Kb
```

Важными являются поля **used** и **free**. Остальная информация о том, как расходуют память различные кэши системы (буферы, страницы и так далее). В случае, если читателю интересно узнать эти подробности, он может обратиться к документации, доступной вместе с ядром Linux. Файл **Documentation/proc.txt** имеет раздел, который объясняет каждое поле, отображаемое **/proc/meminfo**.

Теперь давайте рассмотрим методы оптимизации использования памяти ядром.

- ***Сокращение структур со статическим выделением памяти для данных***: структуры со статическим выделением памяти для данных находятся либо в секции **.data**, либо в секции **.bss**. Многие из таких задействованных структур данных не имеют опции для настройки (это сделало бы процесс сборки очень сложным) и, следовательно, остаются с размером по умолчанию, который следовало бы изменить для встраиваемых систем. Некоторые из них перечислены ниже.
  - Количество консолей TTY по умолчанию (**MAX_NR_CONSOLES** и **MAX_NR_USER_CONSOLES**), определённое в **include/linux/tty.h** как 63.
  - Размер буфера журнала консоли **LOG_BUF_LEN**, определённый в **kernel/printk.c** как 16 Кб.
  - Количество символьных и блочных устройств (**MAX_CHRDEV** и **MAX_BLKDEV**), определённое в включать **include/linux/major.h**
- ***Файл System.map***: файл **System.map**, сгенерированный при сборке ядра, может быть полезным источником информации в этом отношении. Этот файл содержит адреса для каждого символа; разница между последовательными адресами даст размер символа, который является либо текстом, либо данными. Все крупные структуры данных представляют собой цели для исследования. Для получения размеров различных символов в образе ядра можно также использовать команду **nm** с опцией **--size**.
- ***Сокращение неиспользуемого кода в ядре***: код ядра может быть проверен, чтобы удалить неиспользуемые модули и функции. Обратите внимание, что методы, обсуждаемые для уменьшения ядра в разделе, посвящённом оптимизации пространства, занимаемого ядром, справедливы и здесь.
- ***Неправильное использование* kmalloc**: **kmalloc** представляет собой универсальный распределитель памяти ядра. Драйверы устройств обычно используют **kmalloc()** для динамического выделения памяти. **kmalloc()** работает с кэшем объектов, которые кратны 32-м байтам; так что любое выделение памяти, размер которой лежит между двумя последовательными кратными 32-м байтам величинами, приведёт к внутренней фрагментации. Предположим, что субмодуль выполняет **kmalloc** размером 80 байт; он получает их выделенными из объекта размером 128 байт и, следовательно, 48 байт при выделении памяти тратятся впустую. Если ваш субмодуль или драйвер делает много таких выделений, то много памяти тратится впустую. Средством решения этой проблемы является создание собственных кэшей, из которых можно выделять объекты точно требуемого размера. Это делается с помощью следующих двух шагов:
  - Создание кэша, связанного с объектом slab (кусок памяти), с помощью **kmem_cache_create()**. (Чтобы уничтожить кеш, вызовите **kmem_cache_destroy()**.)
  - Создание объекта, связанного с кэшем, с помощью **kmem_cache_alloc()**. (Функцией освобождения памяти является **kmem_cache_free()**.)
- ***Использование директивы* __init**: раздел **.init** содержит все функции, которые могут быть выброшены после инициализации ядра. Как правило, все функции инициализации

находятся в секции **.init**. Если вы включаете в ядро свой собственный драйвер или модуль, определите разделы, которые должны быть использованы только один раз во время запуска системы и поместите их в секцию **.init** с помощью директивы **__init**. Это будет гарантировать, что когда все функции секции **.init** выполнены, некоторая часть памяти, занимаемая ядром, вернётся обратно в систему.

- ***Сокращение дыр в физической памяти***: дыры в физической памяти являются обычным явлением во встроенных системах. Иногда дизайн платы или процессора не позволяет всей физической памяти быть смежной, создавая тем самым дыры. Однако, большие дыры в физической памяти могут приводить к потере памяти. Так происходит потому, что каждые 4 Кб физической памяти (страница) требуют для обслуживания 60-ти байтовой структуры данных **page_struct**. Если есть большая дыра, то память для этих структур выделена без необходимости, и страницы, которые находятся в области дыры, помечены как неиспользуемые. Чтобы предотвратить это, может быть использована поддержка **CONFIG_DISCONTIGMEM**, предоставляемая ядром Linux .
- ***XIP***: XIP или "eXecute In Place, выполнение на месте" является технологией, благодаря которой программа выполняется непосредственно из флеш-памяти; нет необходимости копировать программу из флеш-памяти, чтобы выполнить её. Помимо уменьшения требований к памяти, это также уменьшает время запуска, поскольку ядро не должно быть распаковано или скопировано в оперативную память. Обратной стороной является то, что в файловой системе, где хранится ядро, не может быть использовано сжатие и, следовательно, требуется иметь много флеш-памяти. Однако, использование XIP в файловых системах имеет ограниченное применение для приложений, так как код обычно разделён и загружается по требованию (подкачка по требованию). XIP более популярен в uClinux из-за нехватки виртуальной памяти в системах на нём. XIP подробно обсуждается в Главе 10 322.

Глава 5, Драйверы встраиваемых устройств

Перенос драйверов устройств из других RTOS (Real-Time Operating System, Операционных Систем Реального Времени) для встраиваемого Linux является сложной работой. Драйверы устройств являются частью подсистемы ввода-вывода Linux.Подсистема ввода-вывода обеспечивает доступ для приложений к низкоуровневому оборудованию с использованием чётко определённого интерфейса системных вызовов. Рисунок 5.1 даёт общий обзор того, как приложения используют драйверы устройств.

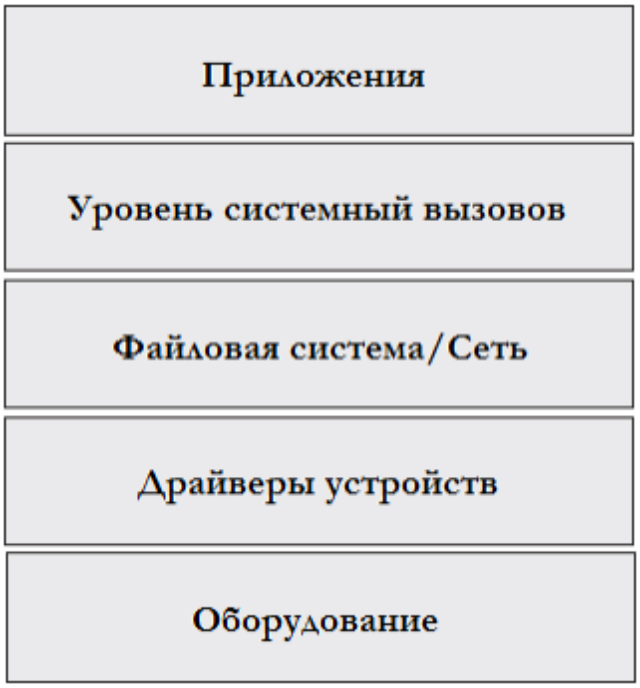

Рисунок 5.1 Обзор архитектуры драйверов устройств Linux.

Драйверы устройств в Linux делятся на три типа:

- ***Драйверы символьных устройств***: они используются для управления устройствами с последовательным доступом. Объём адресуемых данных не ограничен по размеру. Приложения обращаются к драйверам символьных устройств с помощью стандартных вызовов, таких как **open**, **read**, **write**. Например, драйвер последовательного порта является драйвером символьного устройства.
- ***Драйверы блочных устройств***: они используются для управления устройствами с произвольным доступом. Обмен данными происходит в терминах блоков. Драйверы блочных устройств используются для хранения файловых систем. В отличие от символьных драйверов, приложения не могут обращаться напрямую к драйверам блочных устройств; они могут быть доступны только через файловую систему. Файловая система монтируется на блочное устройство, в результате чего драйвер блочного устройства становится посредником между накопителем информации и файловой системой. Например, дисковый драйвер является драйвером блочного устройства.
- ***Драйверы сетевых устройств***: драйверы сетевых устройств рассматриваются как отдельный класс драйверов устройств, потому что они взаимодействуют со стеком

сетевых протоколов. Приложения не обращаться к ним напрямую; с ними взаимодействует только сетевая подсистема.

Эта глава объясняет некоторые из наиболее часто используемых подсистем драйверов устройств на встраиваемых платформах. Мы обсуждаем последовательные, сетевые, I2C драйверы, а также драйверы периферийных устройств (gadget) USB и сторожевого таймера.

5.1 Драйвер последовательного порта в Linux

Драйвер последовательного порта в Linux тесно связан с подсистемой TTY. Уровень TTY является отдельным классом символьного устройства. На встраиваемых системах, имеющих последовательный порт, уровень TTY используется для предоставления доступа к низкоуровневому последовательному порту. Часто встраиваемая плата может иметь больше, чем один последовательный порт; обычно другие порты могут использоваться для коммутируемого доступа с использованием таких протоколов, как PPP или SLIP. Часто задаётся вопрос, должны ли в таком случае быть предоставлены разные драйверы последовательных портов. Ответ: нет, так как TTY отделяет драйвер последовательного порта от приложения, так что может быть предоставлен один драйвер последовательного порта, вне зависимости от того, как он используется.

Пользовательский процесс не общается с драйвером последовательного порта напрямую. TTY представляет собой стек программного обеспечения над драйвером и экспортирует всю функциональность через устройства TTY. Подсистема TTY разделяется на три слоя, как показано на Рисунке 5.2. Как показывает Рисунок 5.2, каждое устройство, взаимодействующее с подсистемой TTY, связано с дисциплиной линии, которая решает, как передаваемые или принимаемые данные обрабатываются низкоуровневым драйвером. Linux предлагает дисциплину линии по умолчанию N_TTY, которая может быть использована для того, чтобы использовать в качестве стандартного терминала последовательный порт. Но дисциплины линий также могут использоваться для реализации более сложных протоколов, таких как X.25 или протокол PPP/SLIP.

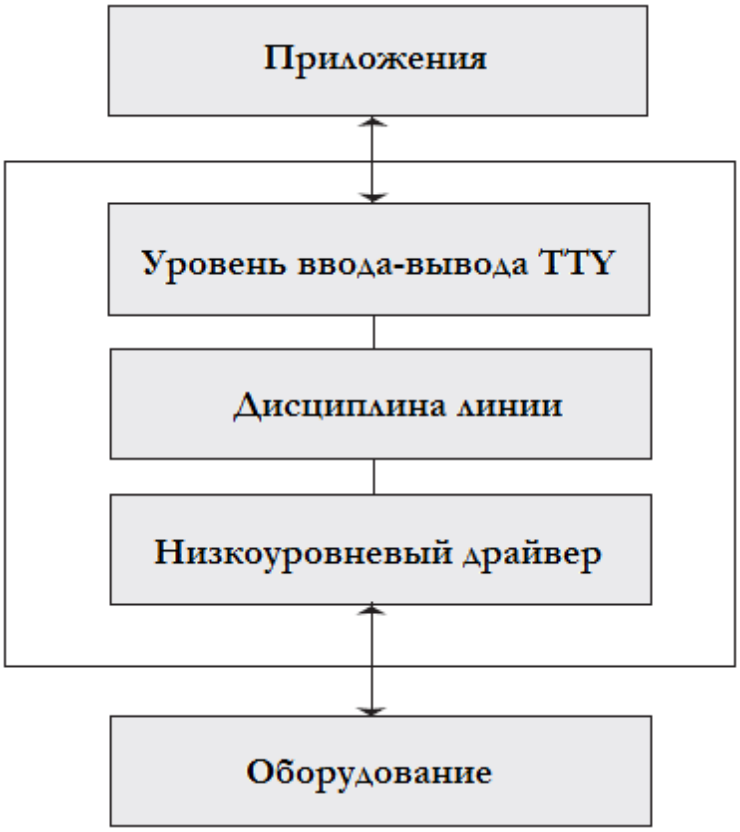

Рисунок 5.2 Подсистема TTY.

В Linux пользовательские процессы обычно имеют управляющий терминал. Управляющий терминал это то, где процесс принимает ввод, и то, где осуществляет стандартный вывод и куда перенаправляются ошибки. TTY и управление процессами автоматически заботятся о назначении и управлении управляющими терминалами. (* Процессы могут предпочесть работать без управляющего терминала. Такие процессы называются службами (daemon, демонами). Службы используются для запуска задач в фоновом режиме после отключения от управляющего терминала, чтобы они не пострадали, когда терминал закрывается.)

Существует ещё один набор устройств TTY, которые используются во встраиваемых системах. Это виртуальные или псевдо-TTY устройства (PTY). PTY являются мощным средством межпроцессного взаимодействия. Процессы, использующие псевдо-TTY получают все преимущества межпроцессного взаимодействия и подсистемы TTY. Например, подсистема Telnet на Linux использует псевдо-терминал для связи между **telnetd** (главной службой Telnet) и процессом, который является порождением **telnetd**. По умолчанию количество псевдо-терминалов TTY установлено в 256; оно может быть изменено на меньшее число из-за его ограниченного использования во встраиваемых системах.

Теперь обсудим реализацию в Linux драйвера последовательного порта. В ядре версии 2.4 структурой данных, используемой для подключения последовательного драйвера к подсистеме TTY, является **tty_driver**. Драйвер последовательного порта заполняет эту структуру такой информацией, как название устройства, старший/младший номера и всеми необходимыми интерфейсами, требуемыми вводом-выводом TTY и уровнем дисциплины линии для обращения к драйверу последовательного порта. В версии 2.4 функции, экспортируемые в уровень TTY от драйвера последовательного порта, содержит файл **drivers/char/generic_serial.c**; он может быть использован для подключения к уровню TTY вашего низкоуровневого драйвера последовательного порта.

В ядре версии 2.6 уровень драйвера последовательного порта был очищен, так что перенос нового драйвера последовательного порта в Linux становится проще. Драйвер последовательного порта больше не должен беспокоиться о вызовах TTY; вернее, это выполняет уровень абстракции. Это делает работу по написанию драйвера последовательного порта легче. Этот раздел объясняет, как на новой платформе может быть написан драйвер последовательного порта.

Путь есть вымышленное оборудование UART MY_UART со следующими функциональными возможностями:

- Простая логика передачи и приёма; один регистр для передачи данных и один регистр для получения данных
- Допустимые настройки скорости 9600 или 19200 бод
- Для оповещения о конце передачи или приёма данных используется прерывание
- Оборудование имеет только один порт UART (то есть, оно однопортовое)

Мы предполагаем, что макрос, показанный в Распечатке 5.1 103, уже доступен для доступа к оборудованию. Также эти макросы предполагают, что регистры и буферы отображаются начиная с базового адреса **MY_UART_BASE**. Мы также предполагаем, что BSP для этой конкретной платы уже сделал это отображение, так что мы можем начать эффективно использовать адрес **MY_UART_BASE**. Тем не менее, мы не обсуждаем поддержку драйвером модема; это выходит за рамки данного раздела.

Сначала мы обсудим конфигурацию устройства. В файл **drivers/serial/Kconfig** добавляем следующие строки:

```
config MY_UART
  select SERIAL_CORE
  help
  Test UART driver
```

Затем в **drivers/serial/Makefile** добавляем следующую строку:

```
obj-$(CONFIG_MY_UART)+= my_uart.o
```

Опция конфигурации выбирает, чтобы файл **my_uart.c** был скомпилирован вместе с **drivers/serial/serial_core.c**. Файл **serial_core.c** содержит общие процедуры UART, которые взаимодействуют с TTY и модулями дисциплины линии. В дальнейшем универсальный уровень UART, реализованный в **serial_core.c**, упоминается как ***ядро UART***.

Распечатка 5.1 Макросы доступа к оборудованию MY_UART

Распечатка 5.1

```
/* my_uart.h */

/*
 * Указывает оборудованию настроить регистры, необходимые для
 * отсылки данных
 */
#define START_TX()
```

```
/*
 * Указывает оборудованию, что мы больше не выполняем отсылку
 * данных.
 */
#define STOP_TX()

/* Связанный с оборудованием макрос для передачи символа */
#define SEND_CHAR()

/*
 * Макрос, который указывает на наличие данных в приёмном
 * регистре UART
 */
#define CHAR_READY()

/* Макрос, который читает символ из оборудования UART */
#define READ_CHAR()

/* Макрос для чтения регистра состояния приёма */
#define READ_RX_STATUS

/* Макросы, которые показывают бит ошибок */
#define PARITY_ERROR
#define FRAME_ERROR
#define OVERRUN_ERROR
#define IGNORE_ERROR_NUM

/*
 * Макрос, который указывает оборудованию прекратить приём
 * символов
 */
#define STOP_RX()

/*
 * Макросы для прерывания обработки; чтение маски прерывания и проверка
 * типа прерывания
 */
#define READ_INTERRUPT_STATUS
#define TX_INT_MASK
#define RX_INT_MASK

/*
 * Макрос, который показывает, является ли буфер передачи пустым
 */
#define TX_EMPTY()

/* Макросы для установки скорости, числа стоповых битов, чётности и числа
битов */
#define SET_SPEED()
#define SET_STOP_BITS
#define SET_PARITY
#define SET_BITS
```

5.1.1 Инициализация и старт драйвера

Теперь давайте обсудим функции инициализации для драйвера. Функция инициализации регистрирует устройство TTY, а затем задаёт путь между ядром UART и драйвером. Основными структурами данных, участвующими в этом процессе и объявленными в файле **include/linux/serial_core.h**, являются следующие:

- **struct uart_driver**: эта структура данных содержит информацию об имени, старшем и младшем номерах, и количестве портов данного драйвера.
- **struct uart_port**: эта структура данных содержит все данные о конфигурации низкоуровневого оборудования.
- **struct uart_ops**: эта структура данных содержит указатели на функции, которые работают с оборудованием.

Для устройства UART с двумя аппаратными портами, эти три структуры данных связаны друг с другом так, как показано на Рисунке 5.3. Для данного примера мы используем двухпортовое оборудование; однако, наш пример оборудования однопортовый.

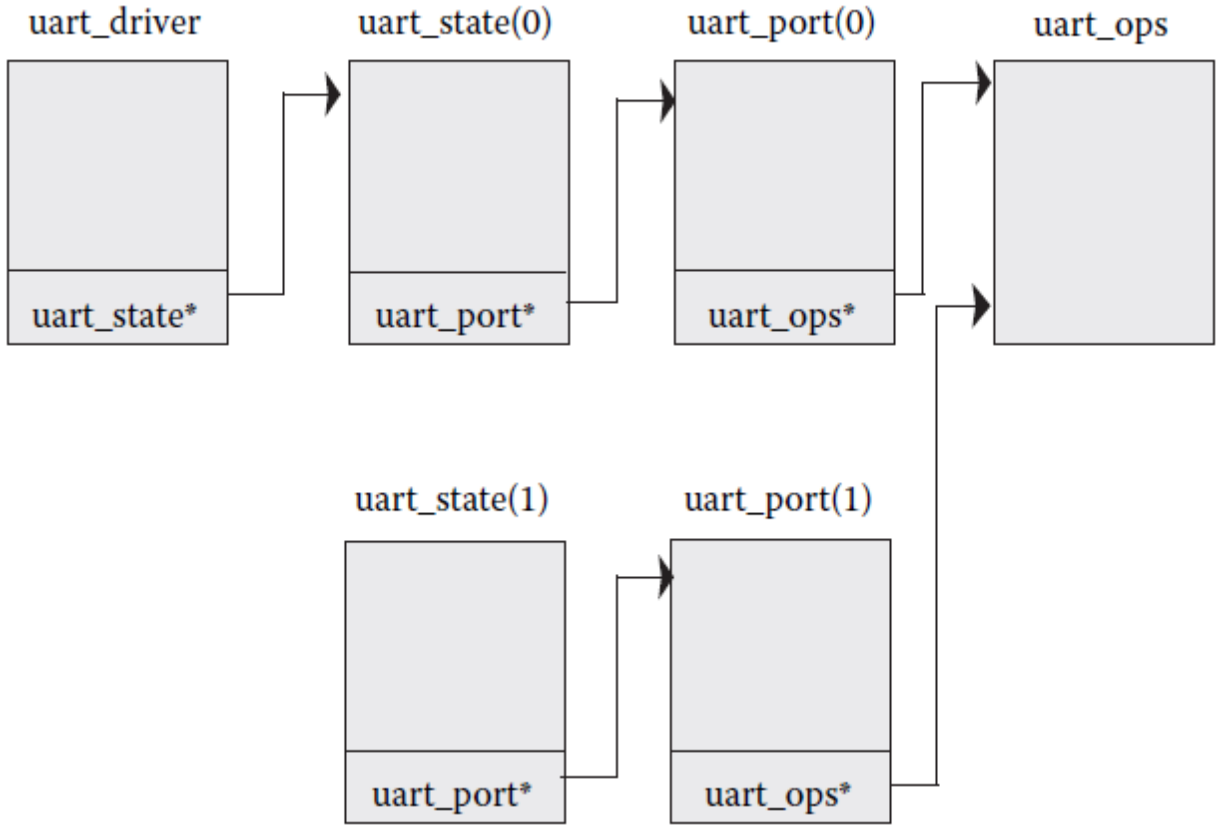

Рисунок 5.3 Связь структур данных для UART.

Существует одна закрытая структура, удерживаемая ядром, - **uart_state**. Число **uart_state** равно числу аппаратных портов, которые доступны через драйвер. Каждая **uart_state** содержит указатель на структуру настройки данного порта **uart_port**, которая, в свою очередь, содержит структуру **uart_ops**, содержащую процедуры для доступа к оборудованию.

Структуры данных для MY_UART определены как показано в Распечатке 5.2 106.

Сначала мы пишем процедуру инициализации.

```
int __init my_uart_init(void)
{
  /*
   * uart_register_driver связывает низкоуровневый драйвер с
   * последовательным ядром, которое в свою очередь регистрируется
   * в уровне TTY с помощью функции tty_register_driver(). Также
   * создаются структуры uart_state (количество таких структур
   * равно числу аппаратных портов) и указатель на этот массив
   * сохраняется в my_uart_driver.
   */
  uart_register_driver (&my_uart_driver);

  /*
   * Как показано на Рисунке 5.3, эта функция соединяет
   * uart_state с uart_port. Также с помощью функции
   * tty_register_device() эта функция даёт знать
   * уровню TTY, что было добавлено устройство.
   */
  uart_add_one_port (&my_uart_driver, &my_uart_port);

  return 0;
}
```

Теперь обсудим функции в структуре **my_uart_ops**. Функции **request_port()** и **release_port()** обычно используются для запроса регионов ввода-вывода и памяти, используемых портом. Функции включения и выключения **my_uart_startup()** и **my_uart_shutdown()** выполняют настройку и отключение прерывания, соответственно.

```
static int my_uart_startup(struct uart_port *port)
{
  return(request_irq(MY_UART_IRQ, my_uart_irq_handler, 0,
                     "my uart", port));
}

static void my_uart_shutdown(struct uart_port *port)
{
  free_irq(MY_UART_IRQ, port);
}
```

Распечатка 5.2 Структуры данных MY_UART

Распечатка 5.2.

```
static struct uart_ops my_uart_ops= {
  .tx_empty     = my_uart_tx_empty,
  .get_mctrl    = my_uart_get_mctrl,
  .set_mctrl    = my_uart_set_mctrl,
  .stop_tx      = my_uart_stop_tx,
  .start_tx     = my_uart_start_tx,
  .stop_rx      = my_uart_stop_rx,
  .enable_ms    = my_uart_enable_ms,
```

```
  .break_ctl    = my_uart_break_ctl,
  .startup      = my_uart_startup,
  .shutdown     = my_uart_shutdown,
  .set_termios  = my_uart_set_termios,
  .type         = my_uart_type,
  .release_port = my_uart_release_port,
  .request_port = my_uart_request_port,
  .config_port  = my_uart_config_port,
  .verify_port  = my_uart_verify_port,
};

static struct uart_driver my_uart_driver = {
  .owner       = THIS_MODULE,
  .driver_name = “serial”,
  .dev_name    = “ttyS%d”,
  .major       = TTY_MAJOR,
  .minor       = MY_UART_MINOR,
  .nr          = 1
};

static struct uart_port my_uart_port = {
  .membase  = MY_UART_MEMBASE,
  .iotype   = SERIAL_IO_MEM,
  .irq      = MY_UART_IRQ,
  .fifosize = 1,
  .line     = 0,
  .ops      = &my_uart_ops
}
```

5.1.2 Передача данных

Функции, участвующие в передаче данных, показаны в Распечатке 5.3 107. Передача начинается с функции **my_uart_start_tx()**; эта функция вызывается дисциплиной линии для начала передачи. После того, как передан первый символ, остальная часть передачи осуществляется из обработчика прерывания, пока не будут переданы все символы в очереди уровня дисциплины линии. Это реализуется универсальной функцией передачи **my_uart_char_tx()**. Ядро последовательного порта обеспечивает механизм кругового буфера для хранения символов, которые должны быть переданы. Ядро последовательного порта предоставляет макросы для работы с этим буфером, следующие из которых используются в этом драйвере:

- **uart_circ_empty()** используется для проверки, пуст ли буфер.
- **uart_circ_clear()** используется для очистки буфера.
- **uart_circ_chars_pending()** используется, чтобы узнать число символов, которые ещё не отосланы.

Распечатка 5.3 Функции передачи

Распечатка 5.3.

```
static void my_uart_char_tx(struct uart_port *port)
{
```

```
  struct circ_buf *xmit = &port->info->xmit;

  /*
   * Если должен быть передан символ XON/XOFF, ядром
   * последовательного порта устанавливается поле x_char
   * данного порта
   */
  if(port->x_char)
  {
    SEND_CHAR(port->x_char);
    port->x_char = 0; /* Reset the field */
    return;
  }

  if(uart_tx_stopped(port) || uart_circ_empty(xmit))
  {
    my_uart_stop_tx(port, 0);
    return;
  }

  SEND_CHAR(xmit->buf[xmit->tail]);

  /*
   * UART_XMIT_SIZE определено include/linux/serial_core.h
   */
  xmit->tail = (xmit->tail + 1) & (UART_XMIT_SIZE - 1);

  /*
   * Теперь проверяем, есть ли ещё символы для передачи
   * и есть ли достаточно места в буфере передачи, которое
   * определено макросом WAKEUP_CHARS в файле
   * include/linux/serial_core.h как 256. Предоставляемая ядром
   * последовательного порта функция uart_write_wakeup в конце
   * концов заканчивается функцией обработчика пробуждения TTY,
   * которая в свою очередь информирует дисциплину линии, что
   * низкоуровневый драйвер готов к приёму других данных.
   */
  if(uart_circ_chars_pending(xmit) < WAKEUP_CHARS)
    uart_write_wakeup(port);
  if(uart_circ_empty(xmit))
    my_uart_stop_tx(port, 0);
}

static void
my_uart_stop_tx(struct uart_port *port, unsigned int c)
{
  STOP_TX();
}

static void
my_uart_start_tx(struct uart_port *port, unsigned int start)
{
  START_TX();
  my_uart_char_tx(port);
```

```
}

/* Возвращает 0, если не пусто */
static unsigned int my_uart_tx_empty(struct uart_port *port)
{
  return (TX_EMPTY()? TIOCSER_TEMT : 0);
}
```

5.1.3 Приём данных

Приём данных происходит в контексте обработчика прерываний. Путь приёма данных поясняется блок-схемой, показанной на Рисунке 5.4.

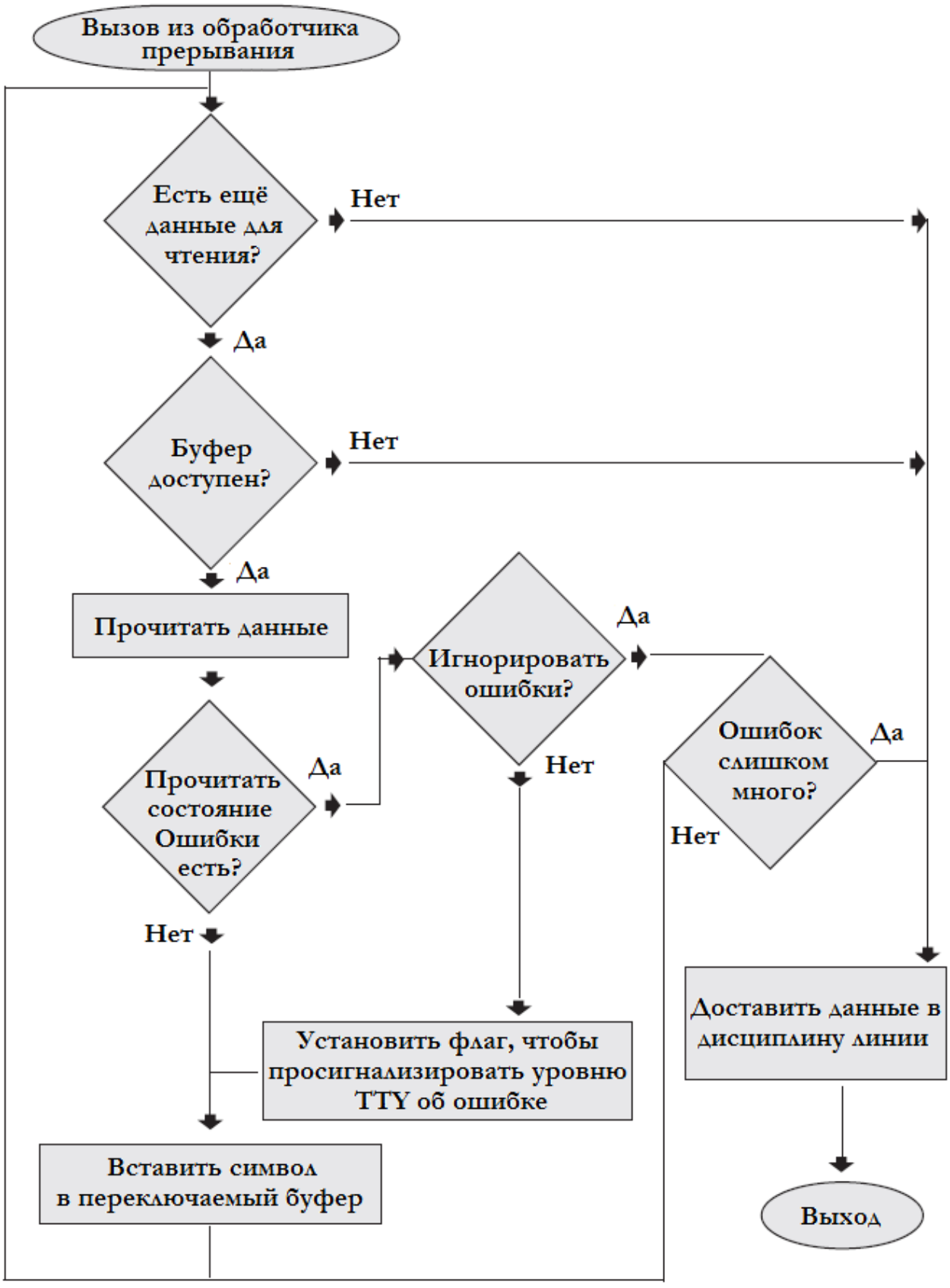

Рисунок 5.4 Блок-схема пути приёма данных.

Основой операции приема является переключаемый буфер TTY. Это пара буферов, которые предоставляются уровнем TTY. Пока один буфер занят дисциплиной линии для обработки принятых символов, другой буфер доступен для записи. Уровень TTY предоставляет стандартные интерфейсы для доступа к переключаемым буферам. Мы заинтересованы только в функциях для вставки полученного символа внутрь доступного переключаемого буфера и последующей передачи полученных символов из переключаемого буфера в дисциплину линии. Это выполняется с помощью функций **tty_insert_flip_char** и **tty_flip_buffer_push**, соответственно. Функции **my_uart_char_rx** и **my_uart_stop_rx** показаны в Распечатке 5.4 111.

Распечатка 5.4 Функции приёма

Распечатка 5.4.

```
void my_uart_char_rx(struct uart_port *port)
{
  struct tty_struct *tty = port->info->tty;
  struct uart_count *icount = &port->icount;
  unsigned int i=0;

  while(CHAR_READY())
  {
    unsigned char c;
    unsigned char flag = TTY_NORMAL;
    unsigned char st = READ_RX_STATUS();

    if(tty->flip.count >= TTY_FLIPBUF_SIZE)
      break;

    c = READ_CHAR();

    icount->rx++;

    if(st & (PARITY_ERROR | FRAME_ERROR | OVERRUN _ERROR) )
    {
      if(st & PARITY_ERROR)
        icount->parity ++;
      if(st & FRAME_ERROR)
        icount->frame ++;
      if(st & OVERRUN_ERROR)
        icount->overrun ++;

      /*
       * Если нас просили игнорировать ошибки, то делаем следующее
       */
      if(st & port->ignore_status_mask)
      {
        if(++i > IGNORE_ERROR_NUM)
          break;
        goto ignore;
      }
```

```
      /*
       * Сообщаем об ошибках, которые нас не просили игнорировать
       */
      st &= port->read_status_mask;
      if(st & PARITY_ERROR) flag = TTY_PARITY;
      if(st & FRAME_ERROR) flag = TTY_FRAME;
      /*
       * Переполнение это специальный случай; оно не влияет на
       * чтение символа
       */
      if(st & OVERRUN_ERROR)
      {
        tty_insert_flip_char(tty, c, flag);
        c = 0;
        flag = TTY_OVERRUN;
      }
    }
    tty_insert_flip_char(tty,c,flag);

ignore:
  }
  tty_flip_buffer_push(tty);
}

static void my_uart_stop_rx(struct uart_port *port)
{
  STOP_RX();
}
```

5.1.4 Обработчик прерываний

Теперь напишем обработчик прерываний, который использует функции приёма и передачи.

```
static irqreturn_t
my_uart_irq_handler(int irq, void *dev_id,
                    struct pt_regs *regs)
{
  unsigned int st = READ_INT_STATUS;
  if(st & TX_INT_MASK) my_uart_char_tx(my_uart_port);
  if(st & RX_INT_MASK) my_uart_char_rx(my_uart_port);

  return IRQ_HANDLED;
}
```

5.1.5 Настройка termios

Наконец, обсудим функцию, которая выполняет настройки интерфейса **termios**. Настройки **termios** представляют собой набор настроек терминала, которые включают различные параметры; они грубо делятся на:

- Параметры управления, такие как скорость передачи, число битов данных, контроль чётности и стоп-биты

- Параметры линии, входные параметры и параметры вывода

Реализация настроек **termios** выполняется в различных уровнях TTY; низкоуровневый драйвер должен беспокоиться только о параметрах управления. Эти параметры задаются с помощью поля **c_cflag** структуры **termios**. Функция **my_uart_set_termios**, которая устанавливает эти параметры, показана в Распечатке 5.5 113.

Распечатка 5.5 Настройка termios

Распечатка 5.5.

```
static void
my_uart_set_termios(struct uart_port *port,
                    struct termios *termios, struct termios *old)
{
  unsigned int c_cflag = termios->c_cflag;
  unsigned int baud=9600, stop_bits=1, parity=0, data_bits=8;
  unsigned long flags;

  /* Считаем число битов данных */
  switch (c_cflag & CSIZE) {
    case CS5: data_bits = 5; break;
    case CS6: data_bits = 6; break;
    case CS7: data_bits = 7; break;
    case CS8: data_bits = 8; break;
    default: data_bits = 8;
  }

  if(c_cflag & CSTOPB) stop_bits = 2;

  if(c_cflag & PARENB) parity = 1;
  if(c_cflag & PARODD) parity = 2;

  /*
   * Мы поддерживаем только 2 скорости, 9600 и 19200. Создаём
   * настройки termios для одной из них
   */
  baud = uart_get_baud_rate(port, termios, old_termios, 9600,
                                                        19200)

  spin_lock_irqsave(&port->lock, flags);
  SET_SPEED(baud);
  SET_STOP_BITS(stop_bits);
  SET_PARITY(parity);
  SET_BITS(data_bits);

  port->read_status_mask = OVERRUN_ERROR;

  if(termios->c_iflag & INPCK)
    port->read_status_mask |= PARITY_ERROR | FRAME_ERROR;
  port->ignore_status_mask = 0;

  if(termios->c_iflag & IGNPAR)
```

```
    port->ignore_status_mask |= PARITY_ERROR | FRAME_ERROR;

  spin_lock_irqrestore(&port->lock, flags);
}
```

5.2 Сетевой драйвер

В Linux драйвер сетевого устройства рассматривается как отдельный класс драйверов. Сетевые драйверы не привязаны к файловой системе, а скорее связаны с интерфейсом подсистемы (таким, как интерфейс Ethernet). Прикладная программа не общается с драйвером сетевого устройства напрямую, а использует сокеты и IP адреса. Сетевой уровень направляет сделанные поверх сокета запросы в сетевой драйвер. В этом разделе описывается процесс написания сетевого драйвера для ядра Linux версии 2.4.

Чтобы объяснить архитектуру сетевого драйвера, этот раздел предполагает использование выдуманного сетевого оборудования. Сетевая карта подключена непосредственно к адресному пространству процессора и, следовательно, её регистры и внутренняя память отображаются в адресное пространство процессора непосредственно. Мы предполагаем два банка памяти: один для передачи, другой для приёма. Для простоты мы предполагаем отсутствие DMA; для передачи данных из системной оперативной памяти в сетевую карту и наоборот драйвер должен вызывать процедуру **memcpy**.

Снова мы предполагаем, что доступны следующие функции/макросы:

- **NW_IOADDR**: это базовый адрес для доступа к вводу-выводу на карте. Мы предполагаем, что инициализация система предоставила достоверный базовый адрес.
- **NW_IRQ**: линия прерывания, используемая для сетевой карты.
- **FILL_ETHER_ADDRESS**: макрос, который программирует оборудование сетевым адресом.
- **INIT_NW**: процедура, которая инициализирует сетевую карту.
- **RESET_NW**: процедура, которая выполняет сброс сетевой карты.
- **READ_INTERRUPT_CONDITION**: этот макрос указывает, что вызвало прерывание. В нашем случае есть две причины: одна это получение входящих данных, а другая - конец передачи.
- **FILL_TX_NW**: процедура для копирования данных из сетевых буферов в память оборудования. Она используется в передающем тракте.
- **READ_RX_NW**: процедура, которая копирует данные из памяти оборудования в сетевые буферы. Она используется в тракте приёма.

5.2.1 Инициализация и закрытие устройства

Linux поддерживает структуру **net_device**, объявленную в **include/linux/netdevice.h**. Эта управляющая структура включает в себя всю информацию, необходимую для данного устройства, от высокоуровневых деталей, таких как настройки драйвера и указатели на функции, предлагаемые драйвером, до низкоуровневых деталей, таких как дисциплина очереди и указатели протокола, используемые внутри ядра. В данном разделе объясняется использование драйвером этой структуры.

Во время сборки ядра включите опции конфигурации **CONFIG_NET**, **CONFIG_NETDEVICES** и **CONFIG_NET_ETHERNET**. Есть два способа осуществления регистрации; один метод используется, когда сетевой драйвер загружен как модуль, а другой метод используется, когда сетевой драйвер компонуется как часть ядра. Ниже

описаны оба метода.

Когда драйвер устройства скомпонован напрямую с адресным пространством ядра, память для структуры **struct net_device** выделяется ядром. Драйвер должен предоставить процедуру зондирования, которая вызывается ядром во время запуска. Процедуры зондирования для различных аппаратных устройств содержит файл **drivers/net/space.c**; так что вам необходимо добавить сюда поддержку вашего сетевого устройства. Каждое сетевое устройство связано со своим уникальным списком зондирования, который связывает данное устройство с его архитектурой и шиной. После определения списка зондирования, функция зондирования добавляется в этот список так:

```
#ifdef TEST_HARDWARE
  {lxNWProbe, 0},
#endif
```

Во время инициализации устройства ядро вызывает функции зондирования. В нашем случае вызывается функция **lxNWProbe**; аргумент функции включает в себя **struct net_device**, которая инициализируется значениями по умолчанию, в том числе именем устройства. (* Сетевые устройства инициализируются по умолчанию именами от "eth0" до "eth7".) За заполнение остальных полей в структуре **net_device** отвечает функция зондирования. Мы предполагаем, что это единственная сетевая карта в системе и, следовательно, нет необходимости делать какую-либо аппаратную проверку. **lxNWprobe** показана в Распечатке 5.6 116.

В случае, когда драйвер пишется в виде модуля ядра, память для структуры **net_device** выделяется модулем и регистрируется явным образом с помощью функции **register_netdev**. Эта функция присваивает устройству имя, вызывает функцию инициализации (в данном случае это **lxNWprobe**), добавляет его в цепочку сетевых устройств и уведомляет протоколы верхнего уровня, что появилось новое устройство.

```
#ifdef MODULE

static struct net_device lxNW_dev;

static int init_module(void)
{
  dev->init = lxNWprobe;

  register_netdev(dev);
  return 0;
}

static void cleanup_module(void)
{
  unregister_netdev(dev);
}

module_init(init_module);
module_exit(cleanup_module);

#endif
```

Функция открытия вызывается всякий раз, когда устройство переводится из состояния ВЫКЛЮЧЕНО (DOWN) во ВКЛЮЧЕНО (UP).

```
static int LXHWopen(struct net_device *dev)
{
  RESET_NW(); INIT_NW();

  /* Запуск передающей очереди устройства */
  netif_start_queue(dev);
}
```

Функция закрытия вызывается для перевода интерфейса из состояния ВКЛЮЧЕНО (UP) в ВЫКЛЮЧЕНО (DOWN).

```
static int LXHWclose(struct net_device *dev)
{
  RESET_NW();

  /* Остановка передающей очереди устройства */
  netif_stop_queue(dev);
}
```

Распечатка 5.6 Функция зондирования

Распечатка 5.6

```
int __init lxNWprobe(struct net_device *dev)
{
  /*
   * Эта функция используется только в случае, если драйвер
   * используется как модуль. В этом случае эта функция
   * инициализирует владельца данного устройства
   */
  SET_MODULE_OWNER(dev);

  /*
   * Настройка стартового адреса для доступа к вводу-выводу;
   * Это будет использоваться семейством команд inb()/outb()
   */
  dev->base_addr = NW_IOADDR;

  dev->irq = NW_IRQ;

  /*
   * Заполняем сетевой адрес; обычно он получен от
   * какой-либо настройки при загрузке
   */
  FILL_ETHER_ADDRESS(dev->dev_addr);

  /* Запрашиваем прерывание */
  request_irq(dev->irq, &LXNWIsr, 0, "NW", dev);

  /* Выполняем инициализацию микросхемы */
  RESET_NW();
```

```
  /* Заполняем структуру устройства важными функциями */
  dev->open = lxNW_open;
  dev->hard_start_xmit = lxNW_send_packet;
  dev->stop = lxNW_close;
  dev->get_stats = lxNW_get_stats;
  dev->set_multicast_list = lxNW_set_multicast_list;
  dev->watchdog_timeo = HZ;
  dev->set_mac_address = lxNW_set_mac_address;

  /*
   * Для заполнения полей сетевого устройства значениями по умолчанию
   * предоставляется ether_setup. Здесь одним из важных полей является
   * длина очереди передачи, поддерживаемой для устройства. Значением
   * по умолчанию является 100. Также в dev->flags установлены
   * IFF_BROADCAST и IFF_MULTICAST, что означает, что это устройство
   * имеет поддержку широковещательных (broadcasting) и групповых
   * (multicast) запросов. В случае, если ваше устройство не
   * поддерживает групповые запросы, необходимо явно очистить этот
   * флаг.
   */
  ether_setup(dev);

  return 0;
}
```

5.2.2 Передача и приём данных

Передача пакетов от драйвера в оборудование осложняется тем, что это включает в себя контроль потока данных между ядром (уровень 3 стека), процедуру передачи драйвера, обработчик прерываний и оборудование. Реализация зависит от возможностей оборудования по передаче. Наш пример устройства имеет только один встроенный буфер передачи. Так что программное обеспечение должно убедиться, что буфер передачи оборудования защищён от перезаписи, когда данные из буфера всё ещё передаются из оборудования в сеть. Буферы, используемые Linux для передачи и приёма, называются **skbuff**.

Ядро поддерживает очередь передачи для каждого устройства с размером по умолчанию 100. Существуют две операции с этой очередью: (* Эта очередь больше известна в народе как qdisc, дисциплина (потому что каждая очередь может иметь дисциплину, связанную с ней, определяющую механизм, посредством которого пакеты помещаются в очередь и извлекаются из очереди).)

1. Добавление пакетов в очередь. Это выполняется стеками протоколов.
2. Удаление пакетов из очереди. Операция вывода данных представляет собой буфер, который передаётся устройству интерфейсом передачи драйвера.

На втором этапе интерфейс передачи копирует буфер в оборудование. В случае, если оборудование имеет ограниченное пространство для хранения, эта функция не должна вызываться, когда оборудование обрабатывает буфер. Вызывать эту функцию следует только после того, как прерывания дают понять, что передача завершена и что использовать оборудование безопасно. Ядро Linux обеспечивает такой контроль с помощью трёх функций:

- **netif_start_queue**: используется драйвером, чтобы просигнализировать верхним уровням, что вызов интерфейса драйвера для отправки дополнительных данных является безопасным.
- **netif_stop_queue**: используется драйвером, чтобы просигнализировать верхним уровням, что буферы передачи заполнены и, следовательно, интерфейс передачи драйвера не должен быть вызван.
- **netif_wake_queue**: Linux обеспечивает программное прерывание для автоматической отправки пакетов, когда произошло прерывание ***завершение передачи*** и стеку верхнего уровня запрещено отправлять пакеты в драйвер устройства. Программное прерывание вызывает интерфейс передачи драйвера устройства, чтобы избавиться от следующего пакета в очереди. Программное прерывание срабатывает при вызове из обработчика прерываний функции **netif_wake_queue**.

Так что то, как вы должны вызывать вышеописанные функции для управления потоком данных, зависит от оборудования. Ниже приводится эмпирическое правило:

- Если интерфейс передачи вашего драйвера останавливает извлечение из очереди **qdisc** из-за ограниченного размера буфера, то обработчик прерывания должен организовать программное прерывание для передачи пакетов из **qdisc** с помощью **netif_wake_queue()**.
- Тем не менее, если в оборудовании при вызове интерфейса передачи ещё есть место, так что он может быть снова вызван верхним уровнем, функция **netif_stop_queue** вызываться не должна.

В нашем примере драйвера мы должны остановить вызовы функции передачи драйвера устройства, пока не произошло прерывание ***конец передачи***. Таким образом, в течении времени, пока передаётся пакет, стек более высокого уровня может только помещать пакеты в очередь **qdisc**; пакеты не могут быть удалены из **qdisc**.

Приём сравнительно проще; он выделяет память для **skbuff** и вызывает функцию **netif_rx**, которая осуществляет планирование программного прерывания для обработки пакета (смотри Распечатку 5.7 118).

Распечатка 5.7 Функции приёма и передачи

Распечатка 5.7.

```
void LXNWIsr(int irq, void *id, struct pt_regs *regs)
{
  struct net_device *dev = id;

  switch (READ_INTERRUPT_CONDITION())
  {
    case TX_EVENT:
      netif_wake_queue(dev);
      break;
    case RX_Event:
      LXHWReceive(nCS);
      break;
  }
}
```

```
int LXNWSendPacket(struct sk_buff *skb, struct net_device *dev)
{
  /* Запрещаем прерывание, так как это может вызвать программное
   * прерывание
   */
  disable_irq(dev->irq);
  netif_stop_queue(dev);
  FILL_TX_NW(skb);
  enable_irq(dev->irq);
  dev_kfree_skb(skb);

  return 0;
}

void LXHWReceive(struct net_device *dev)
{
  struct sk_buff *skb;

  /*
   * После получения длины фрейма выделяем память для skb
   */
  skb = dev_alloc_skb(READ_RX_LEN + 2);

  /* Это выполняется для выравнивания по границе 16 байт */
  skb_reserve(skb,2);

  skb->dev = dev;
  READ_RX_NW(skb);
  skb->protocol = eth_type_trans(skb,dev);
  netif_rx(skb);
}
```

5.3 Подсистема I2C в Linux

Шина I2C (inter IC, шина обмена данными между микросхемами) представляет собой двухпроводную последовательную шину, разработанную Philips Semiconductor в начале 1980-х. Когда она была изобретена, основной целью было соединение различных микросхем на плате телевизора. Однако её простота в использовании и низкие накладные расходы при разработке платы сделали её универсальным стандартом и в настоящее время она используется для подключения различных периферийных устройств в широком спектре конфигураций. Первоначально она была низкоскоростной шиной; она развилась, чтобы предлагать различные скорости от 100 Кб/с до 3.4 Мб/сек. Шина I2C предлагает различные преимущества, такие как экономия места на плате, уменьшение общей стоимости оборудования, а также предлагает средства упрощённой отладки.

Сегодня шина I2C активно используется во встраиваемых системах и платы без шины I2C можно встретить очень редко. В этом разделе объясняется подсистема I2C в Linux. Прежде чем углубляться в детали, рассмотрим, как работает шина I2C.

5.3.1 Шина I2C

Шина I2C имеет две линии: линию SDA (данные) и линию SCL (тактовая частота). Линия SDA несёт такую информацию, как адрес, данные и подтверждение, один бит на такт. Приемник и

отправитель синхронизируются между собой с помощью линии тактовой частоты. На одной шине I2C могут находиться множество устройств I2C; устройства классифицируются как ведущие или ведомые. Ведущее - это устройство, которое стартует и останавливает передачу и генерирует сигналы на линии тактирования. Ведомое - это устройство, которое адресуется ведущим. Ведущим может быть как передатчик, так и приёмник; то же самое относится к ведомым устройствам. Каждое устройство на шине I2C имеет уникальный 7-ми или 10-ти разрядный адрес, который используется для идентификации этого устройства. Пример реализации показан на Рисунке 5.5.

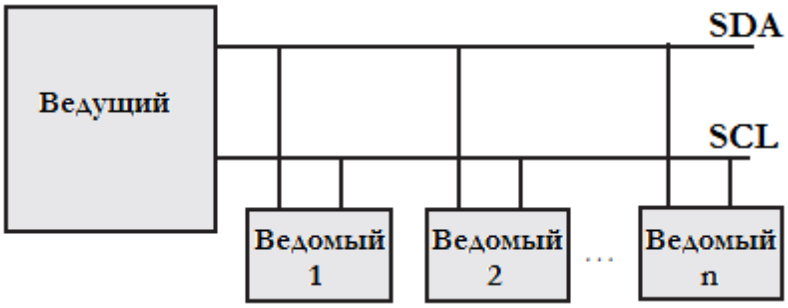

Рисунок 5.5 Шина I2C.

Передача данных по I2C делится на следующие фазы:

- ***Фаза ожидания (idle)***: когда шина I2C не используется, обе линии, SDA и SCL, находятся в состоянии ВЫСОКОГО уровня.
- ***Фаза старта (start)***: когда линия SDA меняет уровень с ВЫСОКОГО на НИЗКИЙ и когда SCL сохраняет ВЫСОКИЙ уровень, это означает начало фазы данных. Она инициируется ведущим устройством.
- ***Фаза адресации (address)***: в этой фазе ведущий посылает адрес ведомого целевого устройства и режим передачи данных (чтение или запись). Ведомое устройство должно ответить подтверждением, чтобы могла начаться фаза передачи данных.
- ***Фаза передачи данных (data transfer)***: по шине I2C данные передаются побитово. В конце передачи каждого байта приемником к передатчику посылается один бит подтверждения.
- ***Фаза останова (стоп)***: ведущий указывает это, переводя линию SDA из НИЗКОГО в ВЫСОКИЙ уровень в то время, когда линия SCL сохраняет ВЫСОКИЙ уровень.

Когда на шине I2C ведущий должен послать данные ведомому устройству, выполняются следующие шаги, показанные на Рисунке 5.6.

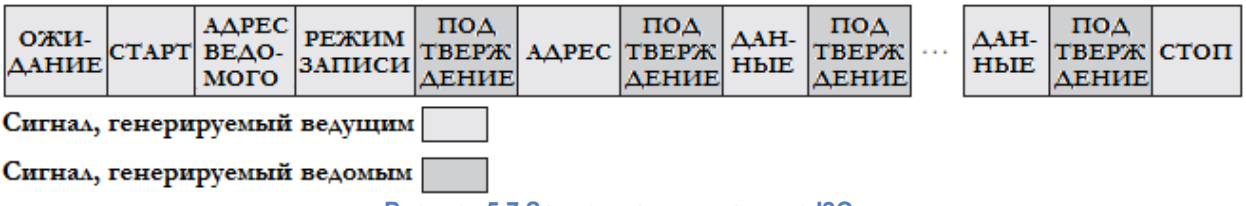

Рисунок 5.7 Запись данных на шине I2C.

1. Ведущий сигнализирует о состоянии СТАРТ.
2. Ведущий посылает адрес ведомого устройства, которому он хочет послать данные, и посылает режим передачи как запись.
3. Ведомый посылает ведущему устройству подтверждение.

4. Ведущий посылает адрес, по которому данные должны быть записаны в ведомом устройстве.
5. Ведомый посылает ведущему устройству подтверждение.
6. Ведущий посылает по шине SDA данные для записи.
7. В конце байта передачи ведомое устройство отправляет бит подтверждения.
8. Два вышеописанных шага выполняются вновь, пока записываются все необходимые байты. Адрес записи автоматически увеличивается на единицу.
9. Ведущий сигнализирует о состоянии СТОП.

Когда на шине I2C ведущему необходимо прочитать данные из ведомого устройства, выполняются следующие шаги, показанные на Рисунке 5.7.

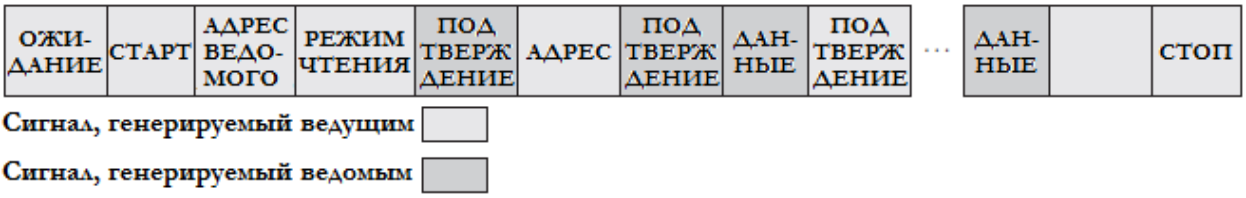

Рисунок 5.7 Чтение данных на шине I2C.

1. Ведущий сигнализирует о состоянии СТАРТ.
2. Ведущий посылает адрес ведомого устройства, которому он хочет послать данные, и посылает режим передачи как чтение.
3. Ведомый посылает ведущему устройству подтверждение.
4. Ведущий посылает адрес, по которому данные должны быть прочитаны в ведомом устройстве.
5. Ведомый посылает ведущему устройству подтверждение.
6. Ведомый посылает по шине SDA данные для чтения.
7. В конце байта передачи ведомое устройство отправляет бит подтверждения.
8. Два вышеописанных шага выполняются вновь, пока записываются все необходимые байты. Адрес чтения автоматически увеличивается на единицу. Однако, для последнего байта ведущий не посылает подтверждение. Это предотвращает отправку ведомым устройством дополнительных данных по шине.
9. Ведущий сигнализирует о состоянии СТОП.

5.3.2 Архитектура программного обеспечения I2C

Подсистема I2C появилась после большой реконструкции, с выходом ядра версии 2.6. В этом разделе мы обсудим архитектуру I2C в ядре версии 2.6. Хотя эта шина сама по себе очень проста, архитектура подсистемы I2C в Linux является довольно сложной и лучше всего её можно понять на примере.

Предположим, что ваша плата использует I2C, как показано на Рисунке 5.8, который показывает на плате две шины I2C; каждая шина I2C находится под управлением адаптера шины I2C, подобного PCF8584, который действует как ведущее устройство I2C для данной шины и, кроме того, работает как интерфейс между шиной процессора и шиной I2C. Таким образом, процессор может обращаться к любому из устройств I2C на шинах I2C путём программирования этих адаптеров. К первой шине I2C подключены две микросхемы EEPROM, а к другой шине I2C подключена микросхема RTC.

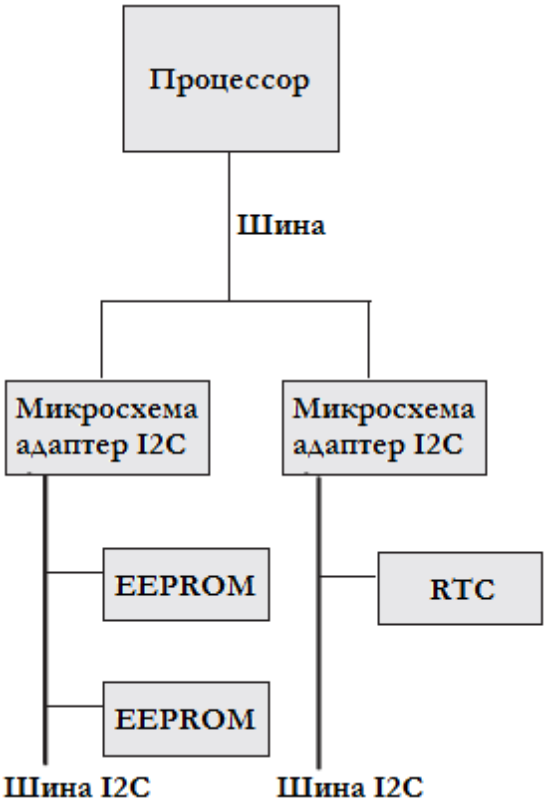

Рисунок 5.8 Пример топологии шины I2C.

Подсистемой I2C в Linux определены следующие логические компоненты программного обеспечения:

- ***Драйвер алгоритма I2C***: каждый адаптер шины I2C имеет свой собственный способ взаимодействия с процессором и шиной I2C. В приведённом выше примере оба шинных адаптера используют стиль сопряжения микросхемы PCF, который определяет регистры, которые должны быть реализованы адаптером шины и реализацию алгоритмов для передачи и приёма данных. Драйвер алгоритма реализует основные процедуры обмена данными (передача и приём). Например, на Рисунке 5.8 показан только один драйвер алгоритма, драйвер алгоритма PCF8584.
- ***Драйвер адаптера I2C***: это можно рассматривать как уровень BSP для подсистемы I2C. Драйвер адаптера I2C и драйвер алгоритма вместе управляют шинами I2C в системе. В приведённом выше примере мы определяем два драйвера адаптера I2C для каждой из двух шин в системе. Оба эти драйвера адаптера связаны с драйвером алгоритма PCF8584.
- ***Драйвер ведомого устройства I2C***: драйвер ведомого устройства содержит процедуры для доступа к определённому виду ведомого устройства на шине I2C. В нашем примере мы предоставляем два драйвера: один для доступа к EEPROM на первой шине I2C, а другой для доступа к микросхеме RTC на второй шине I2C.
- ***Драйвер клиента I2C***: один драйвер клиента является представлением одной единицы оборудования, которое должно быть доступно через шину I2C. Драйверы ведомого устройства и клиента связаны друг с другом. В нашем примере мы должны определить три клиентских драйвера: два клиентских драйвера EEPROM и один клиентский драйвер RTC.

Почему подсистема разделена таким образом? Это делается для как можно большего повторного использования программного обеспечения и чтобы дать возможность переносить код. Это достигается за счёт сложности. Подсистема I2C находится в каталоге **drivers/i2c** дерева исходных текстов ядра. В этом каталоге подкаталог **buses** содержит различные драйверы адаптера шины, **algos** содержит различные драйверы микросхем алгоритма, а каталог **chips** содержит различные драйверы ведомых устройств и клиентов. Общая часть всей подсистемы I2C называется ядром I2C и реализована в файле **drivers/ic2/i2c-core.c**.

Драйвер алгоритма и адаптера шины

Для того, чтобы лучше понять, как писать эти драйверы, мы рассмотрим реализацию драйвера алгоритма для PCF8584 для Linux и шинный адаптер, который использует этот алгоритм. Прежде чем мы углубимся в исходный код, давайте сделаем очень общий обзор микросхемы интерфейса I2C PCF8584. PCF8584 представляет собой интерфейсное устройство между стандартными высокоскоростными параллельными шинами и шиной I2C. Микросхема осуществляет передачу данных между шиной I2C и широко распространённой параллельный шиной микроконтроллера, используя либо прерывание, либо опрос. PCF8584 определяет следующие регистры адаптера шины I2C.

- S0: регистр буфера данных/сдвига, который выполняет параллельно-последовательное преобразование между процессором и шиной I2C.
- S0': это внутренний регистр адрес и он заполняется во время инициализации.
- S1: управляющий регистр и регистр состояния, используемый для доступа к шине и управления.
- S2: регистр тактовой частоты.
- S3: регистр вектора прерывания.

Спецификация PCF8584 содержит более подробную информацию о том, как программировать регистры для инициализации, передачи и приёма. Спецификацию можно загрузить с веб-сайта компании Philips Semiconductor.

Каждый драйвер алгоритма связан со структурой данных **i2c_algorithm**, объявленной в файле **include/linux/i2c.h**. Эта структура данных имеет указатель на функцию **master_xfer**, который указывает на функцию, которая реализует фактический алгоритм передачи и приёма по I2C. Другими важными полями этой структуры являются:

- **name**: название алгоритма.
- **id**: каждый алгоритм определяется с помощью уникального номера. Различные типы алгоритмов определены в заголовочном файле **include/linux/i2c-id.h**.
- **algo_control**: это указатель на функцию, подобную ioctl.
- **functionality**: это указатель на функцию, которая возвращает функции, поддерживаемые адаптером, например, какие типы сообщений поддерживаются драйвером I2C.

```
static struct i2c_algorithm pcf_algo = {
  .name          = "PCF8584 algorithm",
  .id            = I2C_ALGO_PCF,
  .master_xfer   = pcf_xfer,
  .functionality = pcf_func,
};
```

Драйвер алгоритма сам по себе не имеет смысла, если он не связан с драйвером адаптера шины I2C. Драйвер алгоритма PCF обеспечивает для этой цели функцию привязки: **i2c_pcf_add_bus()**. Каждый драйвер адаптер связан со структурой данных **i2c_adapter** (объявленной в файле **include/ linux/i2c.h**), экземпляр которой создаётся драйвером адаптера. Драйвер адаптера вызывает функцию **i2c_pcf_add_bus** с указателем на структуру **i2c_adapter**. Важными полями структуры **i2c_adapter**, которые настраиваются драйвером адаптера, являются:

- **name**: имя для адаптера.
- **class**: указывает тип класса устройств I2C, который поддерживает этот драйвер.
- **algo**: указатель на структуру данных **i2c_algorithm**. **i2c_pcf_add_bus()** устанавливает **algo** указывающим на **pcf_algo**.
- **algo_data**: указатель на закрытую, зависимую от алгоритма структуру данных. Например, драйвер алгоритма PCF присваивает этому полю внутренний указатель на структуру данных **i2c_algo_pcf_data**. Эта структура данных содержит указатель на процедуры для доступа к различным регистрам адаптера. Таким образом, драйвер алгоритма отделён от деталей конструкции платы; драйвер адаптера экспортирует особенности конструкции платы используя эту структуру данных. Драйвер адаптера определяет различные процедуры, которые определены в структуре данных **i2c_algo_pcf_data** следующим образом:

```
static struct i2c_algo_pcf_data pcf_data = {
  .setpcf     = pcf_setbyte,
  .getpcf     = pcf_getbyte,
  .getown     = pcf_getown,
  .getclock   = pcf_getclock,
  .waitforpin = pcf_waitforpin,
  .udelay     = 10,
  .mdelay     = 10,
  .timeout    = 100,
};
```

Драйверу адаптера шины необходимо выполнить следующие действия, чтобы связать себя с драйвером алгоритма.

- Определить структуру типа **i2c_adapter** следующим образом:

```
static struct i2c_adapter pcf_ops = {
  .owner     = THIS_MODULE,
  .id        = I2C_HW_P_ID,
  .algo_data = &pcf_data,
  .name      = "PCF8584 type adapter",
};
```

- Определить функции инициализации, которые делают следующее:
  – Запрашивают различные ресурсы, необходимые для драйвера адаптера, такие как линия прерывания.
  – Вызывают функцию **i2c_pcf_add_bus** для связи алгоритма PCF с этим драйвером адаптера. Функция **i2c_pcf_add_bus** выполняет внутренний вызов функции ядра I2C **i2c_add_adapter**, которая регистрирует в ядре новый драйвер адаптера. После этого

адаптер доступен для регистрации клиентами.

I2C драйверы ведомых устройств и клиентов

Чтобы понять модель драйвера клиента I2C, предположим, что есть вымышленное устройство, подключенное по шине I2C, которое содержит один 32-х разрядный регистр. Функциональность драйвера заключается в предоставлении процедур для выполнения чтения и записи регистра. Мы также предполагаем наличие программного обеспечения драйвера алгоритма и драйвера адаптера. Это включает в себя создание драйвера ведомого устройства и клиента. Драйвер ведомого устройства использует структуру данных **i2c_driver**, объявленную в заголовочном файле **include/linux/i2c.h**. Важными полями этой структуры данных являются:

- **name**: название клиента.
- **id**: уникальный идентификатор этого устройства. Список всех идентификаторов может быть найден в файле **include/linux/i2c-id.h**.
- **flags**: устанавливается в **I2C_DF_NOTIFY**, что разрешает уведомления при обнаружении устройств на шине, так что драйвер сможет обнаруживать новые устройства.
- **attach_adapter**: указывает на функцию, которая определяет наличие устройств I2C на шине I2C. Если устройство найдено, то она вызывает функцию для создания нового экземпляра клиента и подключения клиента к ядру I2C.
- **detach_client**: указывает на функцию, которая удаляет экземпляр клиента и уведомляет ядро I2C о его удалении.
- **command**: это подобная ioctl команда, которая может быть использована для специальных функций в устройстве.

В нашем примере драйвера мы определяем структуру **i2c_driver** следующим образом:

```
static struct i2c_driver i2c_test_driver = {
  .owner          = THIS_MODULE,
  .name           = "TEST",
  .id             = I2C_DRIVERID_TEST,
  .flags          = I2C_DF_NOTIFY,
  .attach_adapter = i2c_test_scan_bus
  .detach_client  = i2c_test_detach,
  .command        = i2c_test_command
};
```

Сначала рассмотрим функцию **i2c_test_scan_bus**, которая вызывается, когда добавляется новый адаптер или новое устройство. Аргументом этой функции является указатель на структуру **i2c_adapter** для шины, на которой обнаружено и добавлено ведомое устройство.

```
static int i2c_test_scan_bus(struct i2c_adapter *d)
{
  return i2c_probe(d, &addr_data, i2c_test_attach);
}
```

Функция **i2c_probe** предоставляется ядром I2C; эта функция использует информацию в

структуре данных **addr_data**, чтобы вызвать функцию **i2c_test_attach**; последняя создаёт нового экземпляр клиента и регистрирует его в подсистеме. Структура **addr_data** объявлена в файле **include/linux/i2c.h**. Функция **addr_data** используется для выполнения следующего:

- Принудительной регистрации устройства I2C по указанному адресу в качестве клиента без проверки
- Игнорирования устройства I2C по указанному адресу
- Зондирования устройства I2C по указанному адресу с помощью адаптера и обнаружения его присутствия
- Функционирования в нормальном режиме, просто обращаясь к устройству I2C по указанному адресу и проверяя его присутствие

Функция **i2c_test_attach** создаёт структуру данных **i2c_client** для клиента и заполняет её. Структура данных **i2c_client** также объявлена в файле **include/linux/i2c.h** и её важными полями являются:

- **id**: идентификация
- **addr**: I2C адрес, по которому было обнаружено ведомое устройство
- **adapter**: указатель на структуру **i2c_adapter** для шины, на которой был обнаружен клиент
- **driver**: указатель на структуру **i2c_driver**

```
static int
i2c_test_attach(struct i2c_adapter *adap, int addr,
                int type)
{
  struct i2c_client *client =
      kmalloc(sizeof(struct i2c_client), GFP_KERNEL);
  client->id = TEST_CLIENT_ID:
  client->addr = addr;
  client->adapter = adapter;
  client->driver = &i2c_test_driver;

  return(i2c_attach_client(client));
}
```

Наконец, командная функция, которая реализует функциональные возможности чтения и записи одного регистра микросхемы, выглядит в следующим образом:

```
static int
i2c_test_command(struct i2c_client *client,
                 unsigned int cmd, void *arg)
{
  if(cmd == READ)
    return i2c_test_read(client, arg);
  else if(cmd == WRITE)
    return i2c_test_write(client, arg); return -EINVAL;
}

static int
i2c_test_read(struct i2c_client *client, void *arg)
```

```
{
  i2c_master_recv(client,arg,4);
}

static int
i2c_test_write(struct i2c_client *client, void *arg)
{
  i2c_master_send(client, arg ,4);
}
```

Функции **i2c_master_recv** и **i2c_master_send** читают и записывают байты от данного клиента. Внутри они вызывают функцию драйвера **master_xfer**. Также доступна ещё одна функция, **i2c_transfer**; она посылает серию сообщений, которые могут быть смесью операций чтения и записи, приводя к комбинированным операциям.

5.4 Периферийные USB устройства

Универсальная последовательная шина (Universal Serial Bus, USB) представляет собой коммуникационную шину ведущий-ведомый для подключения к ПК разных периферийных устройств. Топология шины похожа на дерево, где корнем является USB ***хост*** (host, контроллер узла). USB хост/корневое устройство обслуживается ***драйвером USB контроллера***, отвечающего за управление ***устройствами***, подключенными к шине. Новые устройства могут подключаться или отключаться к или от шины на лету. Хост отвечает за определение этих событий и настройку шины так, как это требуется. Кроме того, устройства, подключенные к шине, используют уникальный идентификатор для передачи данных к или от хоста. Все передачи данных выполняются по сигналу хоста. Это означает, что даже если устройство имеет некоторые данные, они не могут быть переданы, пока хост не попросит сделать это. Система также позволяет настраивать пропускную способность шины в зависимости от устройств. Это позволяет хосту резервировать полосу пропускания для каждого устройства. Например, видео или аудио устройство может запросить другие параметры пропускной способности по сравнению с устройствами для взаимодействия с человеком, такими как клавиатура или мышь. Последними стандартами разрешаются высокоскоростные USB соединения с теоретической скоростью передачи 480 Мбит/с.

Драйверная платформа USB в Linux обеспечивает поддержку как для хоста, так и для ведомых устройств. Часть, связанная с хостом, обычно используется, когда к системе Linux необходимо подключить устройство. Например, КПК на базе Linux может обнаружить USB устройство хранения данных, подключённое к его шине. Часть, связанная с ведомым устройством, используется для встраиваемых устройств, которые работают под управлением Linux и подключены к шине другого хоста. Рассмотрим, например, портативный MP3 плеер, работающий под управлением Linux, с USB интерфейсом для передачи песен с ПК.

Эта глава описывает инфраструктуру/драйверную платформу, предоставляемую ядром Linux на стороне ведомого (или периферийного) устройства. Эта драйверная платформа называется ***драйверами периферийных устройств USB*** (gadget-ов, гаджетов). Прежде чем углубляться в детали этих драйверов, сделаем общий обзор архитектуры устройств USB.

5.4.1 Основы USB

Как уже говорилось ранее, USB является высокоскоростной последовательной шиной, способной работать с максимальной скоростью передачи 480 Мбит/с. Устройства подключаются к корневому узлу, называемому хостом. Хост-устройство представляет собой контроллер шины USB, подключённый к системной шине через PCI. Устройства классифицируются так, чтобы подпадать под различные стандартные классы устройств, такие как:

- ***Класс устройств хранения (Storage Class)***: жёсткий диск, флеш-диски, и так далее
- ***Класс устройств для взаимодействия с человеком (Human Interface Class)***: мышь, клавиатура, сенсорная панель, и так далее
- ***Расширитель/Концентратор/Хаб (Extender / Hub)***: концентраторы (используемые для предоставления на шине дополнительных точек подключения)
- ***Класс устройств коммуникации (Communication Class)***: модем, сетевые карты, и так далее

Типичная топология шины представлена на Рисунке 5.9.

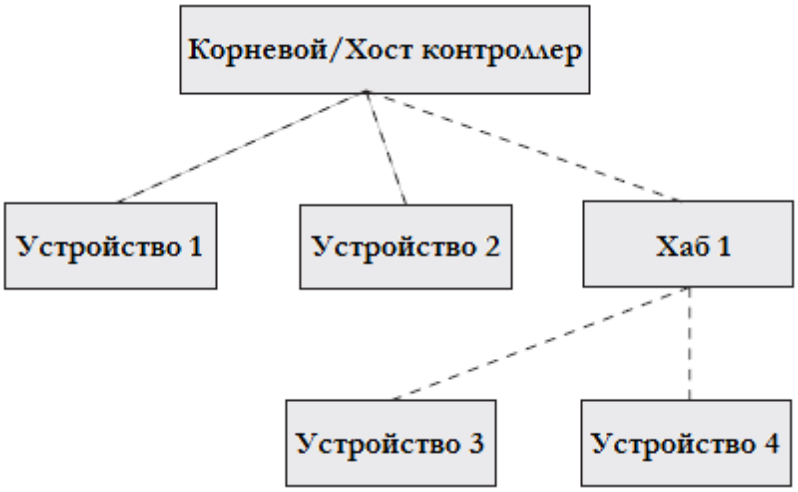

Рисунок 5.9 Топология шины USB.

Взаимодейтсвие с устройством USB происходит через однонаправленные трубы, называемые ***оконечными точками (endpoints)***. Каждое логическое устройство USB представляет собой набор оконечных точек. Каждому логическому устройству на шине хост присваивает уникальный номер, когда устройство подключается к шине. Каждая оконечная точка устройства связана с зависимым от устройства номером оконечной точки. Сочетание устройства и номера оконечной точки позволяет каждой оконечной точке быть идентифицированной однозначным образом.

USB определяет четыре типа передачи:

- ***Передача сигналов управления (Control transfers)***: передача настроек/параметров конфигурации, обычно происходит через оконечную точку конфигурации. Каждое устройство должно иметь как минимум одну оконечную точку, необходимую для настройки устройства при обнаружении. Она называется ***оконечной точкой 0***.
- ***Передача прерываний (Interrupt transfers)***: используется для передачи низкочастотных данных о прерываниях драйвера. HID (Human Interface Device, Устройства для Взаимодействия с Человеком) устройства используют передачу

прерываний.

- ***Поточные передачи (Bulk transfers)***: используются для передачи больших блоков данных, которые не ограничены пропускной способностью. Поточные передачи используют принтеры и коммуникационные устройства.
- ***Изохронные передачи (Isochronous transfers)***: используются для периодических, непрерывных передач данных. Изохронные передачи используют видео, аудио и другие потоковые устройства.

Оконечная точка обычно реализуется с помощью некоторых регистров в памяти или буферной зоны, экспортируемой оборудованием. Драйвер USB устройства записывает данные в эти регистры, чтобы запрограммировать оконечную точку. Высокоскоростные устройства передачи могут предоставить передачи с помощью DMA для своих оконечных точек. Чтобы сформировать ***интерфейс устройства***, одна или несколько оконечных точек группируются вместе. Интерфейс представляет собой логическое устройство, например, мышь или клавиатуру. Каждое логическое устройство должно иметь соответствующий интерфейс USB драйвера, доступный на хосте. Рисунок 5.10 показывает устройство, объединяющее клавиатуру и мышь, и соответствующие ему драйверы на ведущем устройстве.

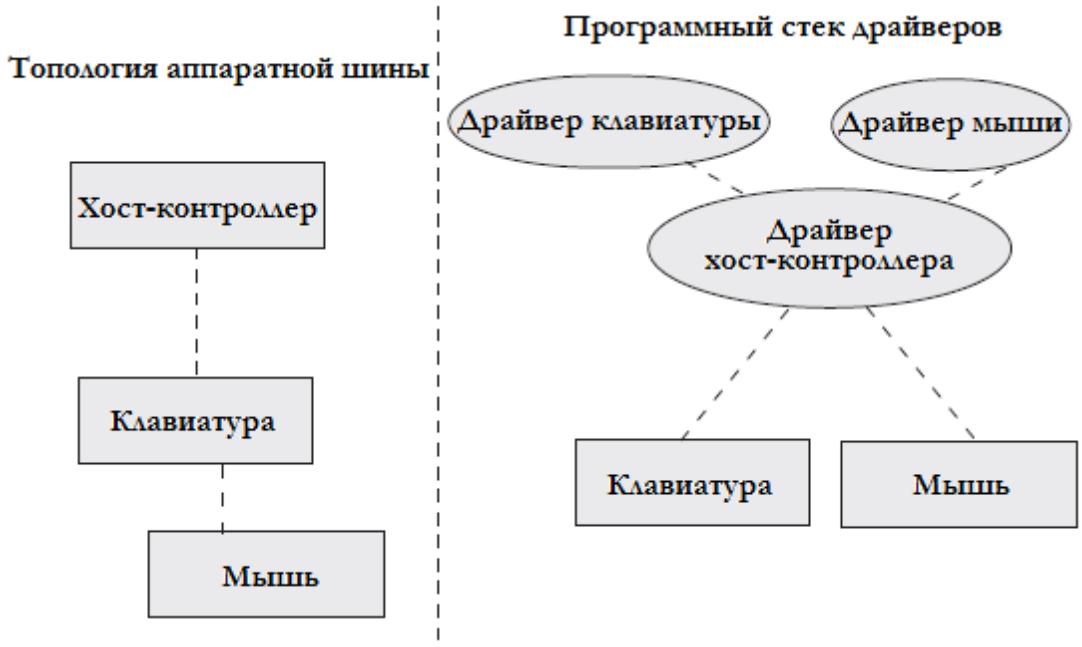

Рисунок 5.10 Стек драйверов USB.

Интерфейсы сгруппированы для формирования ***конфигураций***. Каждая конфигурация настраивает устройство в определённый режим. Например, модем может быть настроен для работы с двумя линиями по 64 Кбит/с или в другой конфигурации для одной линии 128 Кбит/с. Драйвер хоста обращается к хост-контроллеру для управления шиной, в то время как драйвер в устройстве делает устройство видимым для шины. Драйверная платформа периферийных USB устройств обеспечивает необходимые интерфейсы и структуры данных для реализации функции USB устройства или интерфейса. Структура состоит из двух уровней:

- ***Драйвер контроллера***: этот уровень обеспечивает аппаратную абстракцию для USB устройства и реализует интерфейсы периферийного устройства. Драйвер реализует

аппаратно-зависимую часть и предоставляет аппаратно-независимый уровень интерфейса периферийного устройства, используемый более высокоуровневыми драйверами.

- ***Драйвер периферийного устройства***: этот уровень представляет собой фактическую реализацию устройства с функцией USB устройства с помощью интерфейса периферийного устройства. Каждая функция USB устройства требует написания отдельного драйвера периферийного устройства. Поддерживаемые функции USB устройства зависят от возможностей низкоуровневого оборудования.

Драйвер контроллера обрабатывает только ограниченный набор стандартных управляющих запросов USB, относящихся к устройству и состоянию оконечной точки. Все остальные запросы управления, в том числе относящиеся к конфигурации устройства, обрабатываются драйвером периферийного устройства. Драйвер контроллера также управляет очередью ввода/вывода оконечной точки и передачей данных между оборудованием и буферами драйвера периферийного устройства с использованием DMA, где это возможно.

Как обсуждалось ранее, драйвер периферийного устройства реализует определённую функцию устройства. Например, драйвер сетевого периферийного устройства реализует такие функции, как передача и приём сетевых пакетов. Для этого драйвер периферийного устройства должен связать себя с ядром Linux и врезаться в соответствующий стек драйверов. Драйвер вызывает функции более высокого уровня, подобные **netif_rx** и **netdev_register**, а для выполнения необходимых аппаратно-зависимых действий он вызывает с более низкого уровня драйвер контроллера через уровень интерфейса периферийного устройства. Уровни драйверов периферийных устройств USB и то, как они взаимодействуют с остальной частью системы Linux, показывает Рисунок 5.11.

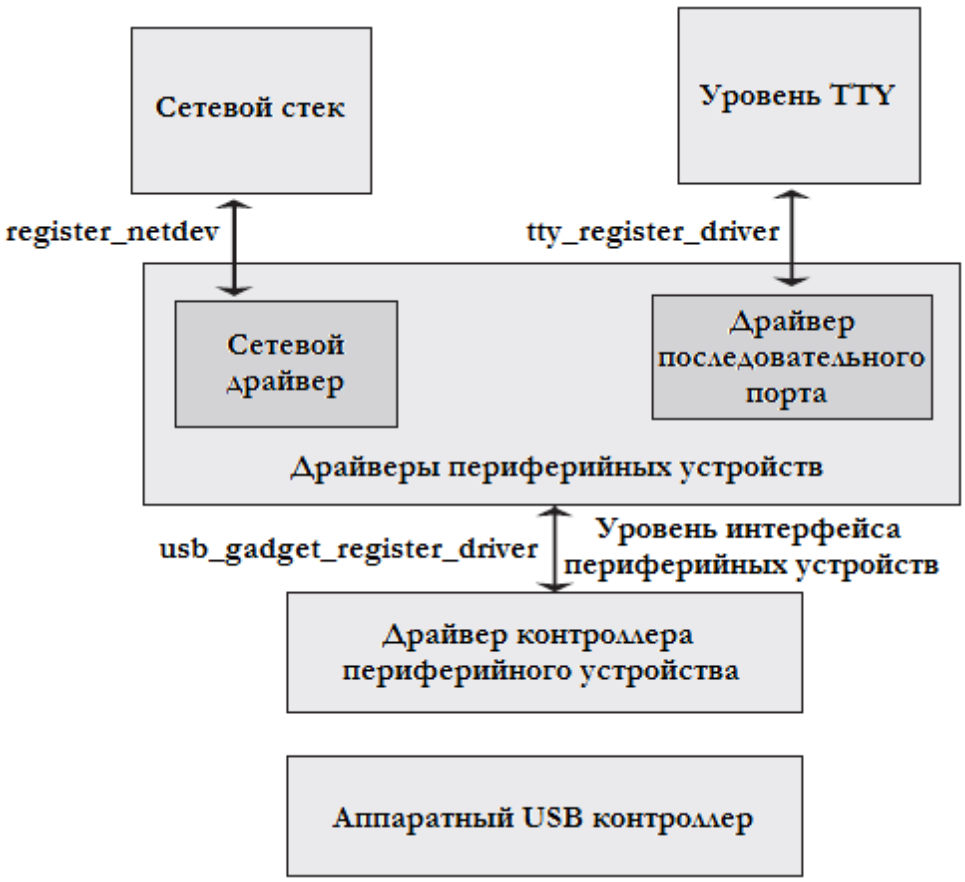

Рисунок 5.11 Архитектура драйвера периферийного USB устройства.

Как правило, контроллер подключает себя к ядру как обычное устройство PCI с помощью метода **pci_driver.probe**. Для регистрации устройства драйвер контроллера предоставляет интерфейс **usb_gadget_register_driver**. Логическими этапами, выполняемыми с функции зондирования контроллера, являются:

1. Регистрация и выделение памяти ресурсам PCI с помощью **pci_enable_device()** и **request_mem_region()**.
2. Инициализация оконечных точек оборудования USB контроллера, такая как установка оконечных точек и структур данных оконечных точек в начальное состояние.
3. Регистрация обработки прерываний контроллера с помощью **request_irq()**.
4. Инициализация регистров контроллера для DMA и выделение памяти для DMA.
5. Регистрация контроллера устройства в ядре с помощью **device_register()**.

Наиболее важной структурой данных является **usb_gadget_driver**, которая связывает периферийное устройство с контроллером. Ещё одним важным аспектом драйвера контроллера является то, что он заботится о взаимодействии различных оконечных точек. Для реализации этого уровень периферийного устройства предоставляет структуру **struct usb_ep_ops**. Обе структуры данных показаны в Распечатке 5.8 132.

Большинство указателей на функции в **usb_ep_ops** являются обёртками вызовов интерфейса периферийных устройств, которые используются драйверами периферийных устройств. Например, интерфейс драйвера периферийного устройства **usb_ep_enable** разрешает начать работу с оконечной точкой.

```
static inline int
usb_ep_enable (struct usb_ep *ep,
               const struct usb_endpoint_descriptor *desc)
{
  return ep->ops->enable (ep, desc);
}
```

Чтобы передавать любые USB данные устройству, драйвер периферийного устройства аналогичным образом использует функцию **usb_ep_queue**.

```
static inline int
usb_ep_queue (struct usb_ep *ep, struct usb_request *req,
              int gfp_flags)
{
  return ep->ops->queue (ep, req, gfp_flags);
}
```

ops->queue это зависимая от контроллера функция очереди. Полный список интерфейсов периферийного устройства находится в файле **include/linux/usb_gadget.h**.

Распечатка 5.8 Структуры данных драйвера периферийного USB устройства

Распечатка 5.8.

```
struct usb_gadget_driver {

  /* Строка, описывающая функции периферийного устройства */
  char *function;

  /*
   * Самая высокая скорость, которую поддерживает драйвер.
   * (Одна из USB_SPEED_HIGH, USB_SPEED_FULL, USB_SPEED_LOW)
   */
  enum usb_device_speed speed;

  /*
   * Функция подключения устройства, вызываемая из драйвера
   * периферийного устройства во время регистрации
   */
  int (*bind)(struct usb_gadget *);

  /*
   * setup вызывается для обработки запросов управления ep0,
   * которые не обрабатываются драйвером контроллера
   */
  int (*setup)(struct usb_gadget *,
               const struct usb_ctrlrequest *);

  /* Этого вызов вызывает отключение устройства от хоста */
  void (*disconnect)(struct usb_gadget *);

  /* Вызывается после отключения устройства */
```

```
  void (*unbind)(struct usb_gadget *);

  /* Вызывает приостановку работы USB устройства */
  void (*suspend)(struct usb_gadget *);

  /* Вызывает возобновление работы USB устройства */
  void (*resume)(struct usb_gadget *);
}

struct usb_ep_ops {

  /* Включение или Выключение оконечной точки */
  int (*enable) (struct usb_ep *ep,
              const struct usb_endpoint_descriptor *desc);
  int (*disable) (struct usb_ep *ep);

  /* Процедуры выделения и освобождения памяти для URB */
  struct usb_request * (*alloc_request) (struct usb_ep *ep,
                                         int gfp_flags);
  void (*free_request) (struct usb_ep *ep,
                        struct usb_request *req);
  void *(*alloc_buffer)(struct usb_ep *ep, unsigned bytes,
                        dma_addr_t *dma, int gfp_flags);
  void (*free_buffer) (struct usb_ep *ep, void *buf,
                       dma_addr_t dma, unsigned bytes);

  /* Функции управления очередью оконечной точки */
  int (*queue) (struct usb_ep *ep, struct usb_request *req,
                int gfp_flags);
  int (*dequeue) (struct usb_ep *ep, struct usb_request *req);
  int (*set_halt) (struct usb_ep *ep, int value);
  int (*fifo_status) (struct usb_ep *ep);
  void (*fifo_flush) (struct usb_ep *ep);
};
```

5.4.2 Драйвер сетевого периферийного устройства

Чтобы объяснить модель драйвера периферийного устройства, возьмём в качестве примера сетевое устройство. Оно реализовано в файле **drivers/usb/gadget/ether.c**.

Функция **init** драйвера любого драйвера периферийного устройства должна вызывать интерфейс **usb_gadget_register_driver**.

```
static int __init init (void)
{
  return usb_gadget_register_driver (&eth_driver);
}
module_init (init);
```

eth_driver является структурой **usb_gadget_driver**, заполненной соответствующими обработчиками.

```
static struct usb_gadget_driver eth_driver = {
```

```
#ifdef CONFIG_USB_GADGET_DUALSPEED
  .speed      = USB_SPEED_HIGH,
#else
  .speed      = USB_SPEED_FULL,
#endif

  .function   = (char *) driver_desc,
  .bind       = eth_bind,
  .unbind     = eth_unbind,
  .setup      = eth_setup,
  .disconnect = eth_disconnect,
};
```

При вызове регистрации из драйвера контроллера вызывается **usb_gadget_driver.bind**. Ожидается, что **bind()** драйвера периферийного устройства выполнит следующие действия:

- Проинициализирует зависимые от устройства структуры данных.
- Присоединится к необходимой подсистеме драйверов ядра (например, драйверу последовательного порта, сетевому драйверу, драйверу устройств хранения и так далее).
- Проинициализирует блок запроса оконечной точки 0.

eth_bind представляет собой функцию привязки.

```
static int
eth_bind (struct usb_gadget *gadget)
{
      …
      …
  net = alloc_etherdev (sizeof *dev);
      …
  net->hard_start_xmit = eth_start_xmit;
  net->open = eth_open;
  net->stop = eth_stop;
  net->do_ioctl = eth_ioctl;

  /* Выделение памяти для EP0 */
  dev->req = usb_ep_alloc_request (gadget->ep0,
                                   GFP_KERNEL);
      …
  dev->req->complete = eth_setup_complete;
  dev->req->buf = usb_ep_alloc_buffer (gadget->ep0,
                  USB_BUFSIZ, &dev->req->dma, GFP_KERNEL);
      …
  status = register_netdev (dev->net);
      …
}
```

После завершения настройки, устройство полностью сконфигурировано и функционирует как обычное устройство в стеке драйверов ядра. В данном примере сетевой USB драйвер подключает себя к стеку сетевых драйверов с помощью **register_netdev()**. Это позволяет приложениям использовать этот интерфейс в качестве стандартного сетевого интерфейса.

Каждый блок USB запроса (URB) требует, чтобы блоку была выделена память, и чтобы он был связан с процедурой завершения. Все запросы оконечной точки ставятся в очередь, потому что они ждут опроса данных от контроллера хоста/корня шины. Как только запрос был обработан оборудованием, драйвер контроллера вызывает связанную с ним процедуру завершения.

Функция передачи данных в любом драйвере по существу делает следующее:

- Создаёт новый URB или получает его из предварительно созданного пула
- Связывает данные и размер данных URB с данными и размером, предоставляемыми драйвером верхнего уровня
- Помещает URB в очередь для передачи данных в соответствующей оконечной точке

Функцией передачи драйвера сетевого периферийного устройства является **eth_start_xmit**.

```
static int eth_start_xmit (struct sk_buff *skb,
                           struct net_device *net)
{
  struct eth_dev*dev = (struct eth_dev *) net->priv;
  int length = skb->len;
    …
  req->buf = skb->data;
  req->context = skb;
  req->complete = tx_complete;
  req->length = length;
    …
  retval = usb_ep_queue (dev->in_ep, req, GFP_ATOMIC);
    …
}
```

Приём данных также требует использования URB и делается следующим образом:

- Создаётся список пустых URB.
- Инициализируется каждый из них с надлежащей процедурой завершения. Процедура завершения для приёма указывает верхним уровням стека сетевых протоколов о прибытии данных.
- Блоки помещаются в очередь в соответствующей оконечной точке.

Во время старта драйвер сетевого периферийного устройства заполняет очередь оконечной точки URB-ами с помощью функции **rx_submit**.

```
static int
rx_submit (struct eth_dev *dev, struct usb_request *req,
           int gfp_flags)
{
  struct sk_buff *skb;
  size_t size;

  size = (sizeof (struct ethhdr) + dev->net->mtu +
                                   RX_EXTRA);
```

```
  skb = alloc_skb (size, gfp_flags);
    …
  req->buf = skb->data;
  req->length = size;
  req->complete = rx_complete;
  req->context = skb;
  retval = usb_ep_queue (dev->out_ep, req, gfp_flags);
    …
}
```

Индикация приёма данных для сетевого уровня осуществляется в процедуре завершения.

```
static void rx_complete (struct usb_ep *ep,
                         struct usb_request *req)
{
  struct sk_buff *skb = req->context;
  struct eth_dev *dev = ep->driver_da
    …
  skb_put (skb, req->actual);
  skb->dev = dev->net;
  skb->protocol = eth_type_trans (skb, dev->net);
    …
  netif_rx (skb);
}
```

5.5 Сторожевой таймер

Сторожевые таймеры (watchdog) представляют собой аппаратные компоненты, которые используются, чтобы помочь системе восстановиться от аномалий программного обеспечения путём сброса процессора. Сторожевой таймер должен быть загружен начальным числом, после чего сторожевой таймер начинает отсчёт от заданного значения до нуля. Если счётчик достигает нуля до того, как программное обеспечение перезагрузит его начальным значением, предполагается, что система работает нестабильно и требуется перезагрузка системы. Некоторые сторожевые таймеры имеют встроенные в них дополнительные возможности, такие как мониторинг температуры и напряжения питания.

Обычно сторожевые таймеры обеспечивают четыре набора операций:

- Старт работы сторожевого таймера
- Настройка времени работы сторожевого таймера
- Остановка сторожевого таймера
- Перезагрузка сторожевого таймера начальным значением

В Linux сторожевой таймер экспортируется приложениям в виде символьного устройства. Устройства сторожевого таймера регистрируются в качестве второстепенных устройств специального символьного устройства, названного ***вспомогательным устройством (miscellaneous device)***. Драйвер сторожевого таймера использует младший номер 130.

Символьное устройство сторожевого таймера может быть использовано какой-либо службой, чтобы перезагружать сторожевой таймер после заданного интервала времени. Многие дистрибутивы предоставляют службу, называемую службой сторожевого таймера (* Busybox тоже имеет простую реализацию сторожевого таймера), которая выполняет эту работу. Этот подход может быть использован в ядре версии 2.6 из-за улучшения

возможностей реального времени ядра. В ядре версии 2.4, или если сторожевой таймер имеет очень маленький интервал перезагрузки, для перезагрузки сторожевого таймера лучше использовать таймер ядра. (5* Некоторые драйверы используют этот подход; посмотрите код драйвера сторожевого таймера процессора AMD Elan SC520.)

Типичный драйвер сторожевого таймера должен реализовать следующие функции:

- ***Функцию инициализации***: она включает в себя
  – Регистрацию драйвера сторожевого таймера в качестве вспомогательного символьного драйвера (с помощью функции **misc_register**)
  – Регистрацию функции, которая отключает сторожевой таймер при оповещении о перезагрузке системы (используя функцию **register_boot_notifier**). Эта зарегистрированная функция вызывается до перезагрузки системы. Это гарантирует, что после перезагрузки системы сторожевой таймер не работает, чтобы не вызвать повторный сброс системы.
- ***Функцию открытия***: она вызывается, когда открывается устройство */dev/watchdog*. Эта функция должна запускать сторожевой таймер.
- ***Функции освобождения ресурсов***: закрытие драйвера должно вызывать остановку сторожевого таймера. Однако, в Linux, когда задача завершает работу, все файловые дескрипторы автоматически закрываются, независимо от того, было ли завершение работы благополучным или же произошло в результате отказа. Так что, если служба сторожевого таймера не завершает работу благополучно, есть шанс, что сторожевой таймер отключён. Чтобы предотвратить такую ситуацию, в ядре версии 2.6 до благополучного завершения работы службы сторожевого таймера, она должна подать сигнал драйверу, что намерена явным образом отключить сторожевой таймер. Обычно в драйвер сторожевого таймера записывается системный символ "V", после чего вызывается эта функция. В качестве альтернативы вы можете выбрать вообще не реализовывать отключение сторожевого таймера в вашем драйвере. Существующие драйверы сторожевого таймера в Linux предоставляют такую возможность при выборе опции конфигурации **CONFIG_WATCHDOG_NOWAYOUT**.
- ***Функция записи***: эта функция вызывается приложением для перезагрузки сторожевого таймера начальным значением.
- **Ioctl**: однако, для перезагрузки сторожевого таймера начальным значением вы также можете использовать **ioctl**. Это выполняет команда **WDIOC_KEEPALIVE**. Также вы можете установить время работы таймера с помощью **WDIOC_SETTIMEOUT**.

В случае отсутствия поддержки сторожевого таймера в оборудовании, Linux предоставляет программный сторожевой таймер. Программный сторожевой таймер использует внутренние таймеры, однако, программные сторожевые таймеры не всегда работают; их работоспособность зависит от состояния системы и состояния прерываний.

5.6 Модули ядра

И наконец, обсудим вкратце модули ядра. Модули ядра добавляются динамически в работающее ядро. Это уменьшает размер ядра при обеспечении того, что модули ядра загружаются только тогда, когда они используются. С интерфейсом модулей ядра связаны три компонента:

- ***Интерфейс/API модуля***: как написать модуль?
- ***Сборка модуля***: как собрать модуль?
- ***Загрузка и выгрузка модуля***: как можно загружать и выгружать модули?

Все три компонента претерпели значительные изменения при переходе между версиями 2.4 и 2.6 ядра. Сборка модуля подробно описана в Главе 8 240. В этом разделе объясняются два других компонента.

5.6.1 Интерфейсы модуля

Пример модуля ядра для версии 2.4 и 2.6 показан в Распечатке 5.9 138. Модуль выводит строку **Hello world** каждый раз, когда он загружается и **Bye world** каждый раз, когда он выгружается. Сколько раз распечатается первая строка зависит от параметра модуля, определённого здесь как **excount**.

Некоторые моменты, которые следует отметить:

- ***Функции входа и выхода***: модуль должен иметь функции входа и выхода, которые автоматически вызываются ядром, когда модуль загружается и выгружается, соответственно. В ядре версии 2.4, функциями входа и выхода являются функции ***init_module()*** и ***cleanup_module()***. Однако, в ядре версии 2.6 они регистрируются специальным образом с помощью макросов ***module_init()*** и ***module_exit()***.
- ***Передача параметров***: каждому модулю могут быть переданы параметры; они передаются в качестве аргументов командной строки при загрузке модуля. В ядре версии 2.4 для предоставления аргументов модулю используется макрос **MODULE_PARM**. В ядре версии 2.6 параметры объявляются с макросом **module_param()** (* MODULE_PARAM в ядре версии 2.6 является устаревшим.), как показано в Распечатке 5.9 138.
- ***Поддержка счётчика использования модуля***: каждый модуль имеет счётчик использования, который показывает число ссылок на этот модуль. Значение счётчика ссылок 0 означает, что модуль может быть выгружен безопасно. В ядре версии 2.4 счётчик модуля хранился каждым модулем. Этот подход имел дефект при работе кода выгрузки модуля на системах с симметричной многопроцессорной обработкой. Поэтому в системах версии 2.6 модуль не должен поддерживать счётчик использования, вместо этого его поддерживает ядро. Однако, это приводит к проблеме, если происходит обращение к функции модуля после выгрузки модуля. Если вы делаете вызов через указатель на функцию в другом модуле, вы должны иметь ссылку на этот модуль. В противном случае, вы рискуете заснуть в модуле, если он выгружен. Для решения этой проблемы ядро предоставляет интерфейсы для доступа к модулю; ссылку на модуль можно получить при помощи интерфейса **try_module_get()** и освободить ссылку на него с помощью интерфейса **module_put()**.
- ***Объявление лицензии***: каждый модуль должен объявить, является ли он чьей-то собственностью или же непатентованным модулем. Это делается с помощью макроса **MODULE_LICENSE**. Если аргументом является **GPL**, это означает, что он будет выпущен под лицензией GPL.

Распечатка 5.9 Модули ядра

Распечатка 5.9.

```
/* Модуль, базирующийся на ядре версии 2.4 */

static int excount = 1;
MODULE_PARM(excount,”i”);
```

```
static int init_module(void)
{
  int i;
  if(excount <= 0) return -EINVAL;
  for(i=0; i<excount;i++)
    printk("Hello world\n");
  return 0;
}

static void cleanup_module(void)
{
  printk("Bye world\n");
}
```

```
/* Код модуля, базирующегося на ядре версии 2.6 */

MODULE_LICENSE("GPL");
module_param(excount, int, 0);
static int init_module(void)
{
  int i;
  if(excount <= 0) return -EINVAL;
  for(i=0; i<excount;i++)
    printk("Hello world\n");
  return 0;
}

static void cleanup_module(void)
{
  printk("Bye world\n");
}

module_init(init_module);
module_exit(cleanup_module);
```

5.6.2 Загрузка и выгрузка модуля

Ядро предоставляет системные вызовы для загрузки, выгрузки, а также доступа к модулям ядра. Тем не менее, доступны стандартные программы, которые выполняют работу по загрузке и выгрузке модуля. Программа **insmod** устанавливает загружаемый модуль в работающее ядро. **insmod** пытается связать модуль с работающим ядром путём разрешения всех символов из экспортируемой таблице символов ядра. Иногда загрузка модуля зависит от загрузки других модулей. Эта зависимость хранится в файле **modules.dep**. **modprobe** анализирует этот файл и загружает все необходимые модули до загрузки данного модуля. Наконец, **rmmod** отсоединяет модуль от ядра. Этот шаг успешен, если у модуля нет пользователей в ядре и его счётчик ссылок равен нулю.

Глава 6, Перенос приложений

Разработчик сталкивается с трудной задачей при переносе приложений из традиционной RTOS (real time operating system, операционная система реального времени, ОСРВ), такой как VxWorks, pSoS, Nucleus и других, на встроенный Linux.Трудности возникают из-за совершенно другой модели программирования Linux по сравнению с другими RTOS. В этой главе обсуждается способ переноса приложений из традиционной RTOS на встроенный Linux. В ней также рассматриваются разные методы, которые обычно используются для облегчения переноса.

6.1 Сравнение архитектур

В этом разделе мы сравним архитектуру традиционной RTOS со встроенным Linux.Традиционная RTOS обычно базируется на плоской модели памяти. Все приложения вместе с ядром являются частью единого образа, который затем загружается в память.Службы ядра, такие как планировщики, управление памятью, таймеры и тому подобные, работают в том же физическом адресном пространстве, что и пользовательские приложения. Приложения запрашивают любую службу ядра с помощью простого интерфейса вызова функции. Пользовательские приложения также делят общее адресное пространство между собой. Рисунок 6.1 показывает плоскую модель памяти традиционной RTOS.

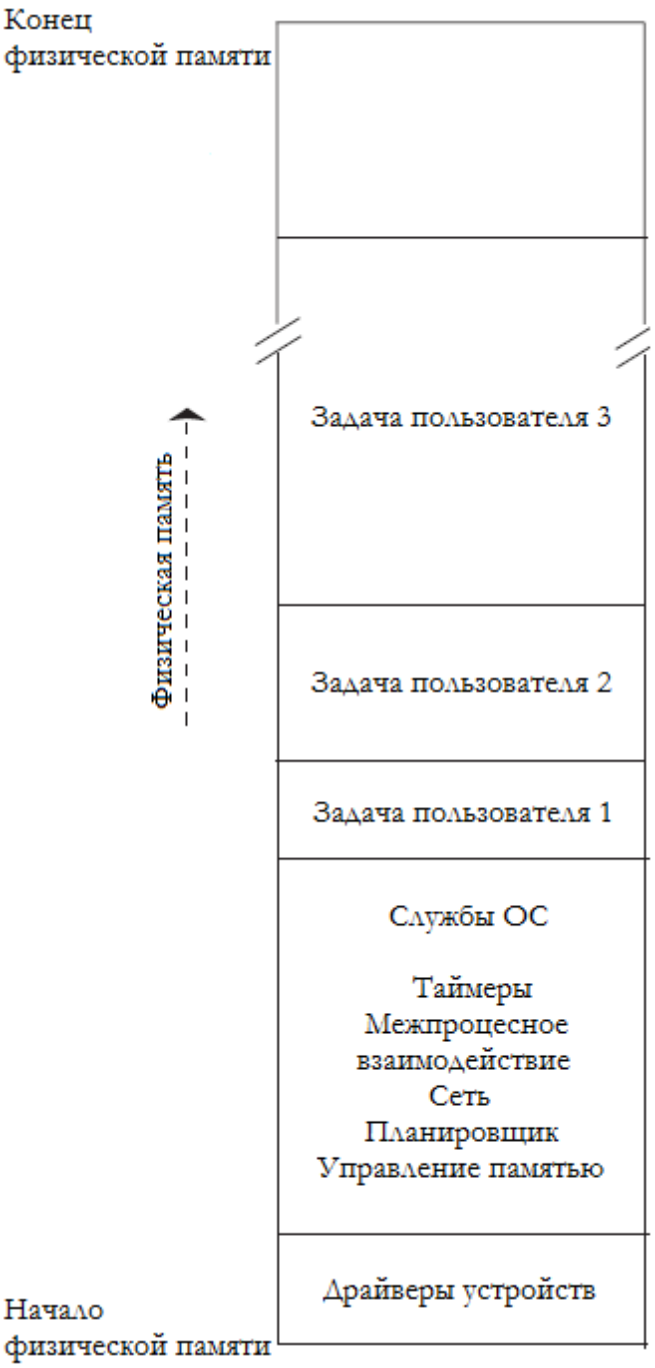

Рисунок 6.1 Плоская модель памяти ОСРВ.

Основным недостатком такой RTOS является то, что она основана на плоской модели памяти. В целях защиты памяти не использовался MMU. Следовательно, любое пользовательское приложение может повредить код ядра или данных. Это может также повреждать структуры данных другого приложения.

Linux, с другой стороны, для предоставления отдельного виртуального адресного пространства для каждого процесса использует MMU. Виртуальное адресное пространство защищено; это означает, что процесс не может получить доступ к любой структуре данных, принадлежащей другому процессу. Код ядра и структуры данных также защищены. Доступ к службам ядра пользовательскими приложениями осуществляется через чётко определенный

интерфейс системных вызовов. Рисунок 6.2 показывает модель памяти Linux на основе MMU.

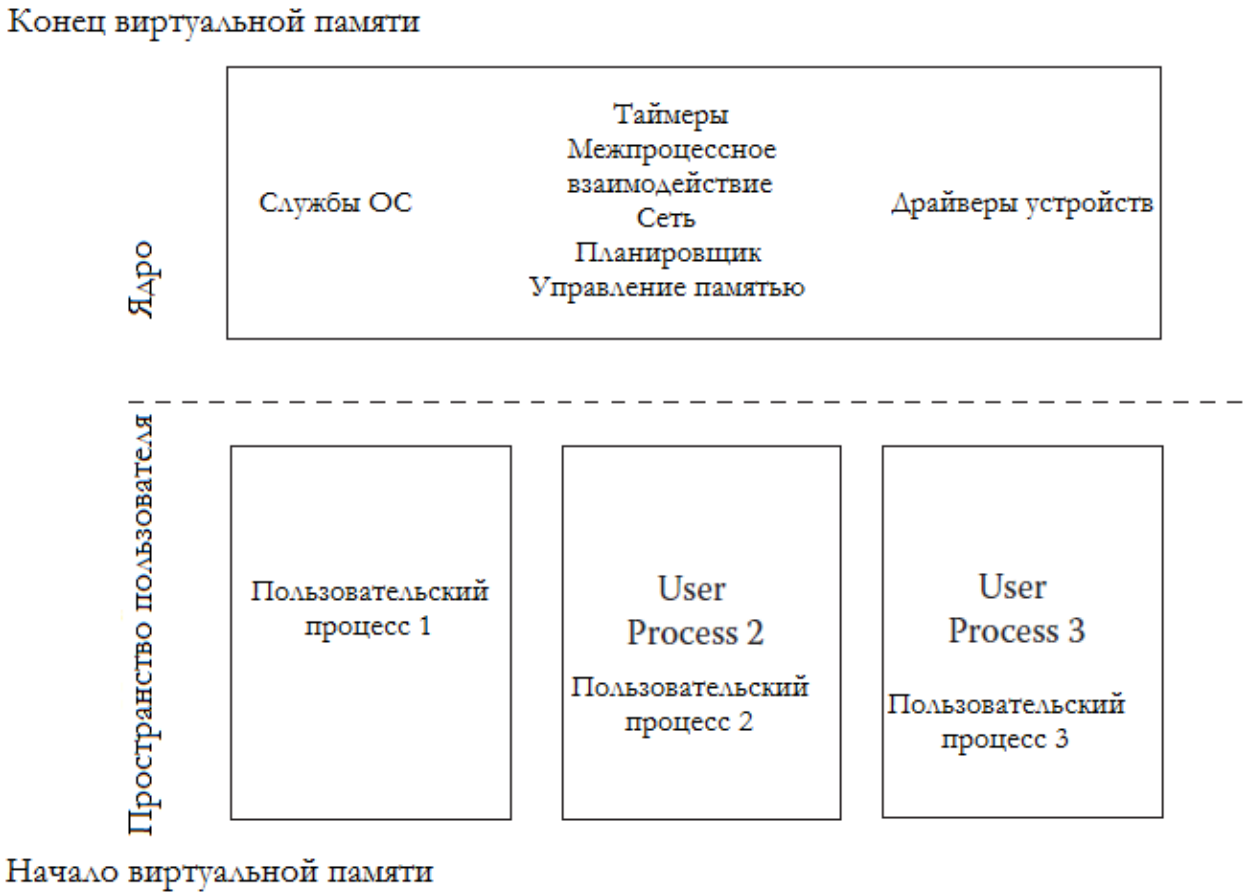

Рисунок 6.2 Модель памяти Linux.

С этого места любое упоминание RTOS относится к традиционной RTOS с плоской моделью памяти и без защиты памяти, если не указано иное.

Проблемами переноса, возникающими из вышеописанного сравнения являются:

- Приложения, которые "делят" единое адресное пространство в RTOS должны быть перенесены на модель защищённого виртуального адресного пространства Linux.
- RTOS обычно предоставляет свой родной набор программных интерфейсов для различных служб, таких как создание задачи, IPC (interprocess communicaton, межпроцессное взаимодействие), таймеры, и так далее. Таким образом, должно быть определено соответствие каждого такого родного API с эквивалентным API Linux.
- Интерфейс ядра в Linux не является простым интерфейсом вызова функции. Так что пользовательские приложения не могут делать какие-либо прямые вызовы драйвера или ядра.

6.2 План переноса приложений

В этом разделе мы рассмотрим общий план переноса приложений с RTOS на встроенный Linux. Следующие разделы посвящены подробностям плана по переносу.

6.2.1 Выбор стратегии переноса

Разделите все ваши задачи RTOS на две большие категории: задачи пространства пользователя и задачи ядра. Например, любая задача пользовательского интерфейса является задачей пространства пользователя, а любая задача инициализации оборудования является задачей ядра. Вы должны также определить список функций пространства пользователя и ядра. Например, любая функция, которая управляет регистрами устройств, является функцией ядра, а любая функция, которая считывает данные из файла, является функцией пользовательского пространства.

Могут быть приняты две стратегии переноса. Обратите внимание, что в обоих подходах задачи ядра переносятся как потоки ядра Linux. Следующее обсуждение касается только задач пользовательского пространства.

Модель с одним процессом

При таком подходе задачи RTOS пользовательского пространства мигрируют как отдельные потоки в одном процессе Linux, как показано на Рисунке 6.3. Преимуществом такого подхода является уменьшение усилий при переносе, так как требуется меньше изменений в существующем коде. Самым большим недостатком является отсутствие защиты памяти между потоками внутри процесса. Однако, службы ядра, драйверы, и так далее, полностью защищены.

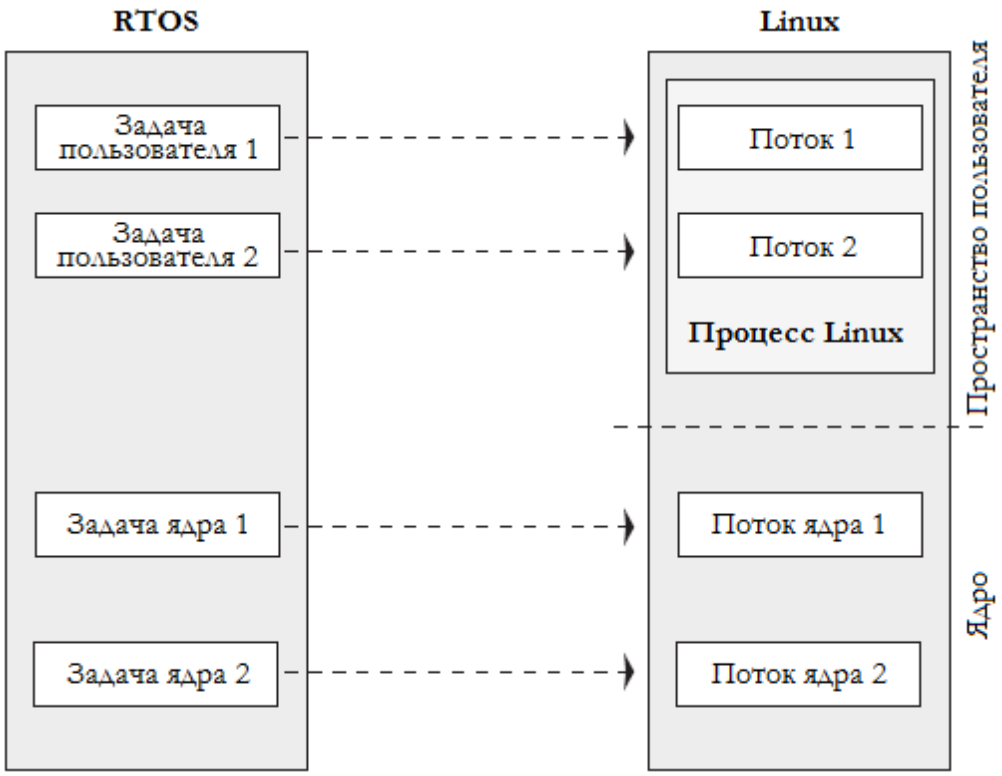

Рисунок 6.3 Миграция в модель одним процессом.

Модель с несколькими процессами

Поделите задачи на не связанные, связанные и ключевые задачи.

- ***Не связанные задачи***: слабосвязанные задачи, использующие механизмы межпроцессного взаимодействия, предлагаемые RTOS для связи с другими задачами, или автономные задачи (* Например, DHCP клиент является автономной задачей. Он может быть легко перенесён в качестве отдельного процесса Linux.), которые не связаны с другими задачами, могут быть перенесены в виде отдельных процессов Linux.
- ***Связанные задачи***: в эту категорию попадают задачи, которые совместно используют глобальные переменные, и функции обратных вызовов. Они могли бы быть перенесены в виде отдельных потоков в одном процессе Linux.
- ***Ключевые задачи***: задачи, которые выполняют ключевые действия, такие как системные сторожевые задачи, должны быть перенесены в виде отдельных процессов Linux. Это гарантирует, что ключевые задачи защищены от повреждения памяти со стороны других задач.

Рисунок 6.4 иллюстрирует данный подход.

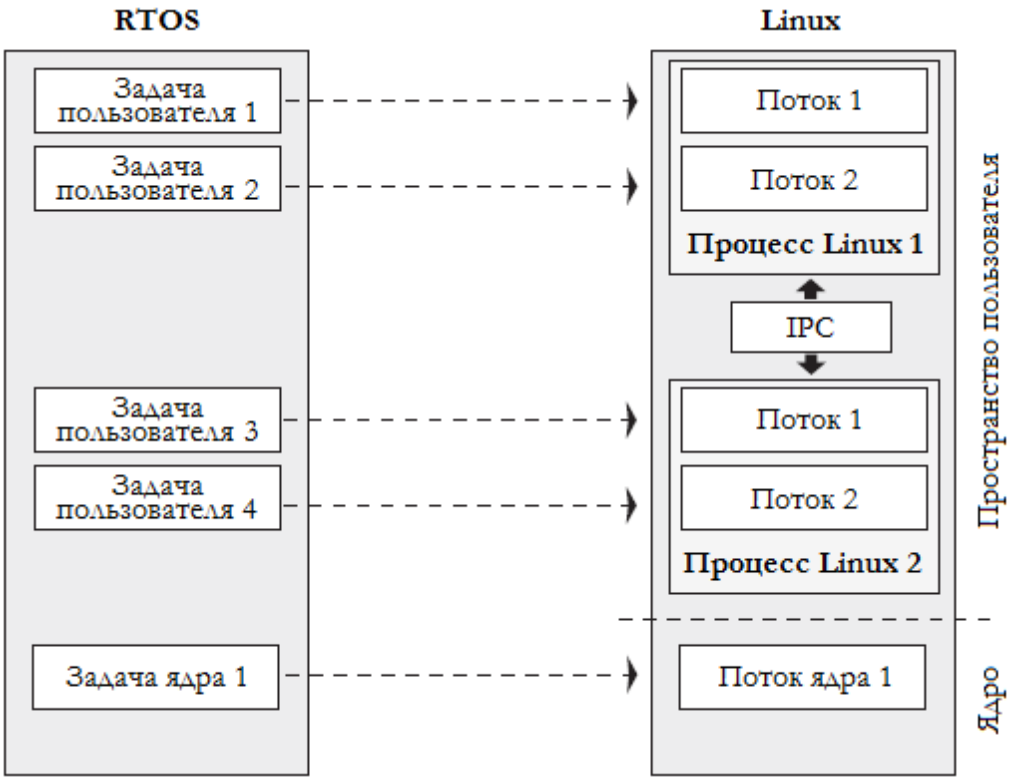

Рисунок 6.4 Миграция в модель с несколькими процессами.

Преимущества этой модели:

- Достигается защита памяти для каждого процесса. Задача не может повредить адресное пространство другого процесса.
- Она расширяема. Используя эту модель могут быть добавлены новые возможности.
- Приложения могут в полной мере использовать преимущества модели программирования Linux.

Самый большой недостаток такого подхода заключается в том, что миграция на Linux с использованием этой модели является трудоёмким процессом. Возможно, вам потребуется переписать большинство приложений. Одной из таких трудоёмких задач является перенос

библиотек пользовательского пространства. Трудности приходят, когда библиотека поддерживает глобальные переменные, которыми манипулируют несколько задач. Предположим, вы решили перенести библиотеку как библиотеку совместного доступа в Linux. Таким образом вы получаете преимущество при общем доступе к тексту из нескольких процессов. Что происходит с глобальными данными в библиотеке? В совместно используемой библиотеке совместно используется только текст, а данные являются своими для каждого процесса. Таким образом, все глобальные переменные в библиотеке теперь становятся глобальными для каждого процесса. Вы не можете изменить их в одном процессе и ожидать, что изменения станут видимыми в другом. Таким образом, для поддержки этого необходимо переписать приложения, использующие библиотеку, для использования между собой надлежащих механизмов межпроцессного обмена. Вы можете также испытать желание поместить такие задачи в категорию связанных задач, но в этом случае вы потеряете преимущества модели с использованием нескольких процессов.

6.2.2 Написание уровня переноса операционной системы (OSPL)

Этот уровень эмулирует API RTOS используя API Linux, как показано на Рисунке 6.5.

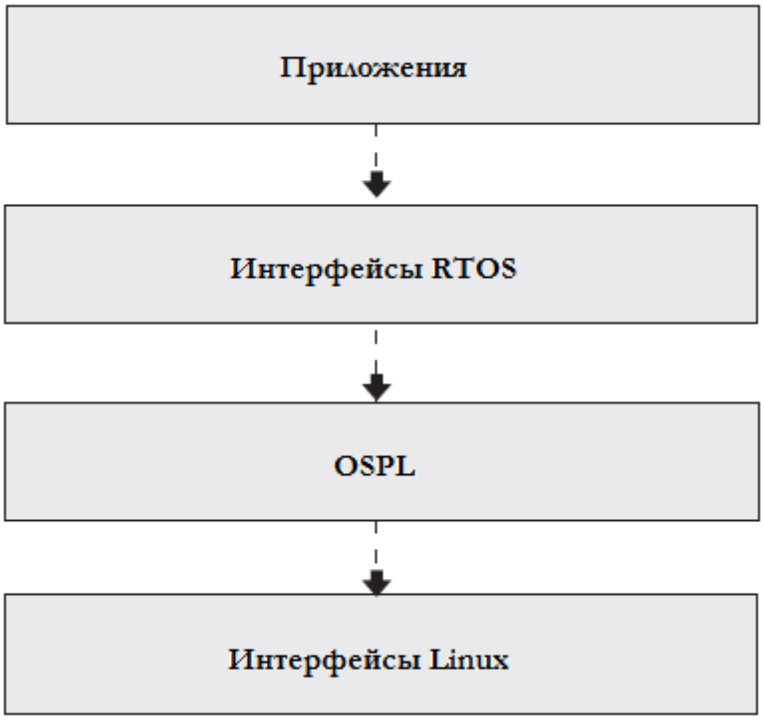

Рисунок 6.5 Уровень переноса операционной системы.

Хорошо написанный OSPL (Operating System Porting Layer, уровень для переноса операционной системы) минимизирует изменения в существующем коде. Для достижения этого должны быть определены соответствия между API RTOS и API Linux. Соответствия подпадают под следующие две категории:

- ***Соответствие один-к-одному***: каждый API RTOS можно эмулировать с помощью одного API Linux. Аргументы или возвращаемое значение эквивалентного API Linux могут отличаться, но ожидается такое же поведение функции.
- ***Соответствие один-ко-многим***: для эмуляции API RTOS необходимо более одного API

Linux.

Для многих API RTOS вы также должны определить соответствия с API ядра Linux, так как эти API могут быть использованы задачами ядра. Вы можете или иметь отдельные OSPL для ядра и пользовательского пространства, или одну библиотеку, которая компонуется и в пространстве пользователя, и пространстве ядра. API OSPL для последнего случая выглядит следующим образом.

```
void rtosAPI(void){
  #ifndef __KERNEL__
    /* Эквивалентный интерфейс пользовательского пространства Linux */
  #else
    /* Эквивалентный интерфейс ядра Linux */
  #endif
}
```

Обратите внимание, что при определении соответствия API RTOS и API Linux вы можете натолкнуться на некоторые API RTOS, которые не могут быть съэмулированы с использованием API Linux без внесения изменений в существующий код. В таких случаях вам может понадобиться переписать некоторые части существующего кода.

6.2.3 Написание драйвера API ядра

Иногда вы столкнётесь с трудностями при принятии решения о переносе задачи в пространство пользователя или пространство ядра, так как она вызывает как функции пользователя, так и функции ядра. Та же проблема возникает с функцией, которая вызывает функции как в пользовательском пространстве, так и в пространстве ядра. Например, рассмотрим функцию **func**, вызывающую функции **func1** и **func2**. **func1** является функцией пользовательского пространства, а **func2** - функцией ядра.

```
void func(){
  func1(); <-- Функция пользовательского пространства
  func2(); <-- Функция ядра
}
```

Куда должна быть перенесена функция **func**? В пространство пользователя или пространство ядра? Вам необходимо написать драйвер API ядра для поддержки таких случаев. В модели драйверов API ядра функция **func** переносится в пространство пользователя, предоставляя интерфейс для функции **func2** в пространстве пользователя. Драйвер API ядра подробно обсуждается в Разделе 6.5 167.

В этом разделе мы обсудили план по переносу приложения с ОС реального времени на Linux.Оставшаяся часть главы разделена на три части:

- В первой части мы вкратце обсудим pthread-ы (потоки POSIX). Pthread-ы представляют собой модель многопоточности в Linux.Раздел охватывает все операции с pthread-ами, которые надо понимать, прежде чем начинать процесс переноса.
- Во второй части мы напишем небольшой OSPL, поддерживающий только интерфейсы создания задачи, уничтожения задачи и мьютекс.
- В конце мы обсудим драйвер API ядра.

6.3 Программирование с помощью pthread-ов

Для обсуждения различных операций с pthread-ами мы взяли очень простой MP3-плеер, код которого находится в файле **player.c**. Плеер имеет два основных компонента.

- ***Инициализация***: она включает в себя инициализацию звуковой подсистемы в отдельном потоке. Используется для демонстрации создания потока и выхода из процедуры.
- ***Декодирование***: это ядро приложения. Участвуют два рабочих потока. Основной поток читает данные из MP3-файла и добавляет их в очередь. Поток декодера извлекает данные, декодирует их и проигрывает. Очередь представляет собой общую структуру данных между основным потоком и потоком декодера. Рисунок 6.6 показывает различные объекты, которые участвуют в фазе декодирования. Идея заключается в подробной демонстрации различных примитивов синхронизации потоков.

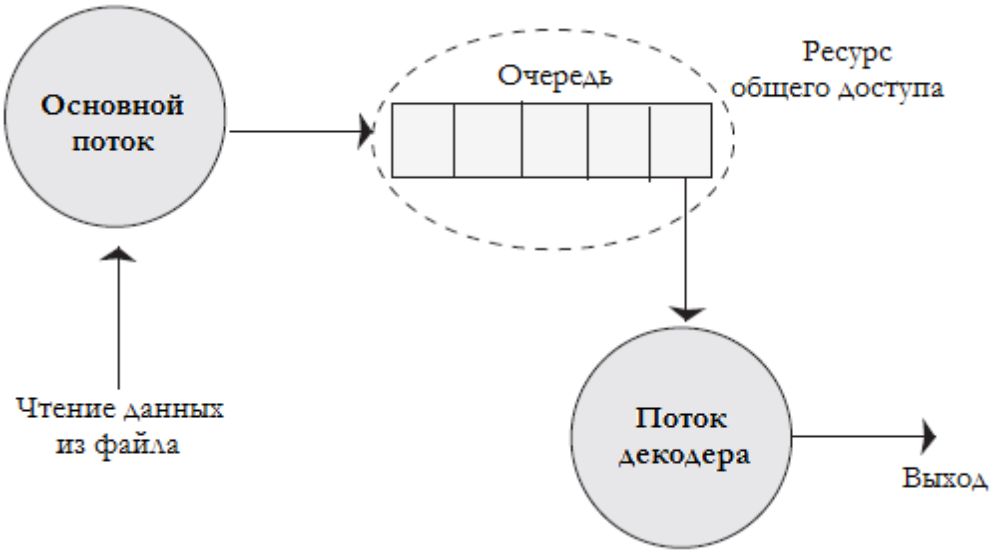

Рисунок 6.6 Простой звуковой плеер.

Пожалуйста, обратите внимание, что этот раздел не является полным справочником по работе с потоками pthread. Нашей целью является дать вам достаточное количество деталей для быстрого начала работы с pthread-ми. Также в нашем примере плеера намеренно опущены специфические подробности, касающиеся декодирования и воспроизведения. Это сделано, чтобы уделить больше внимания операциям с pthread-ми в плеере.

6.3.1 Создание потока и выход из него

Новый поток выполнения создается вызовом функции **pthread_create**. Прототип функции:

```
int pthread_create (pthread_t * thread_id,
                    pthread_attr_t *thread_attributes,
                    void * (*start_routine)(void *),
                    void * arg);
```

Функция возвращает ноль в случае успеха, а идентификатор созданного потока хранится в первом аргументе, **thread_id**. Новый поток начинает свое выполнение с функции

start_routine. **arg** является аргументом **start_routine**. **thread_attributes** представляет собой различные атрибуты потока, такие как политика планирования, приоритет, размер стека, и тому подобное. В случае неудачи функция возвращает ненулевое значение.

Давайте обратимся к нашему MP3-плееру в **player.c**. При запуске плеера для инициализации всех подсистем вызывается функция **system_init**. Функция **system_init** выполняется в контексте основного потока.

```
int system_init(){
  pthread_t audio_tid;
  int sample = 1;
  void * audio_init_status;
  /* Инициализируем звуковую подсистему в отдельном потоке */
  if (pthread_create(&audio_tid, NULL, audio_init,
                        (void *)sample) != 0){
    printf("Audio thread creation failed.\n");
    return FAIL;
  }
  /*
   * Инициализируем остальное приложение, структуры данных и т.д.
   */

  ....
  ....
}
```

Чтобы осуществить инициализацию звуковой подсистемы в новом потоке, **system_init** вызывает **pthread_create**. В случае успеха идентификатор потока созданного потока сохраняется в **audio_tid**. Новый поток выполняет функцию **audio_init**. **audio_init** принимает целочисленный аргумент **sample**. Вторым аргументом **pthread_create** является **NULL**, поток **audio_tid** начинает работать со стандартным набором атрибутов. (Например, политика планирования и приоритет потока наследуются от вызывающего.)

Новый поток инициализирует декодер и подсистему вывода звука. При необходимости он также проигрывает в течение двух секунд образец звука, чтобы проверить, что инициализация прошла успешно.

```
void* audio_init(void *sample){
  int init_status = SUCCESS;
  printf("Audio init thread created with ID %d\n",
                                  pthread_self());
  /*
   * Инициализируем подсистему MP3 декодера.
   * При неудаче устанавливаем init_status = FAIL.
   */
  /*
   * Инициализируем подсистему вывода звука.
   * При неудаче устанавливаем init_status = FAIL.
   */

  if ((int)sample){

  /*
   * Проигрываем звук в течение 2-х секунд.
```

```
    * При неудаче при воспроизведении
    * устанавливаем init_status = FAIL.
    */
  }
   printf("Audio subsystem initialized\n");

   pthread_exit((void *)init_status);
}
```

Возникают два вопроса:

- Как поток **audio_init** может отправить своё выходное состояние в основной поток?
- Может ли функция **system_init** подождать окончания потока **audio_init** перед выходом? Как можно передать выходное состояние потока **audio_init**?

Поток устанавливает своё выходное состояние используя функцию **pthread_exit**. Эта функция также завершает выполнение вызывающего потока.

```
void pthread_exit(void *return_val);
```

audio_init вызывает **pthread_exit**, чтобы прекратить его выполнение, а также установить его состояние завершения. **pthread_exit** аналогичен системному вызову **exit**. С точки зрения разработчика приложения есть только одна разница: **exit** завершает весь процесс, а **pthread_exit** прекращает только вызывающий поток.

Поток может получить выходное состояние другого потока вызовом функции **pthread_join**.

```
int pthread_join(pthread_t tid, void **thread_return_val);
```

pthread_join приостанавливает выполнение вызывающего потока до завершения потока **tid**. Когда **pthread_join** возвращается, выходное состояние потока **tid** сохраняется в аргументе **thread_return_val**. **pthread_join** аналогичен системному вызову **wait4**. **wait4** приостанавливает выполнение родительского процесса до тех пор, пока потомок, указанный в его аргументе, не завершится. Аналогично, **pthread_join** также приостанавливает выполнение вызывающего потока, пока поток, указанный в его аргументе, не завершится. Как вы можете видеть, **system_init** вызывает **pthread_join**, чтобы дождаться перед выходом завершения потока **audio_init**. Она также печатает сообщение об ошибке, если **audio_init** заканчивается неудачей.

```
int system_init(){
    ...
  void * audio_init_status;
    ...
    ...
  /* Ожидаем завершения потока audio_init */
  pthread_join(audio_tid, &audio_init_status);

  /* Если инициализация звука не удалась, возвращаем ошибку */
  if ((int)audio_init_status == FAIL){
    printf("Audio init failed.\n");
    return FAIL;
```

```
    }

    return SUCCESS;
}
```

Обратите внимание, что поток, создаваемый с помощью **pthread_create** со стандартным набором атрибутов (вторым аргументом **pthread_create** является **NULL**), является потоком, допускающим объединение. Ресурсы, выделяемые на объединяемые потоки, не освобождаются, пока какой-нибудь другой поток вызывает **pthread_join** для такого потока. Он становится зомби.

6.3.2 Синхронизация потоков

Pthread-ы обеспечивают синхронизацию потоков в форме взаимного исключения (мьютексов) и условных переменных.

Мьютекс представляет собой двоичный семафор, который обеспечивает монопольный доступ к структуре с общими данными. Он поддерживает две основные операции: блокировку и разблокировку. Поток должен заблокировать мьютекс перед входом в критическую секцию и разблокировать его по завершении работы. Поток блокируется, если он пытается заблокировать мьютекс, который уже заблокирован. Он пробуждается, когда мьютекс разблокируется. Операция блокировки мьютекса является атомарной. Если два потока пытаются захватить мьютекс в одно и то же время, гарантируется, что одна операция будет завершена или заблокирована перед началом другой. Также поддерживается неблокирующая версия операции блокировки, trylock (пробная блокировка). Пробная блокировка возвращает состояние успеха, если мьютекс приобретается; она возвращает состояние неудачи, если мьютекс уже заблокирован.

Общая последовательность для защиты структуры общих данных с помощью мьютекса:

```
блокируем мьютекс
работаем с общими данными
разблокируем мьютекс
```

Условная переменная представляет собой механизм синхронизации, который является более полезным для ожидания событий, чем для блокировки ресурса. Условная переменная связана с предикатом (логическим выражением, которое имеет значение ИСТИНА, TRUE, или ЛОЖЬ, FALSE), основанным на некоторых общих данных. Функции отправляются спать на условной переменной и пробуждают один или все потоки, если результат предиката изменяется.

В нашем примере плеера структурой общих данных между основным потоком и потоком декодера является очередь. Основной поток читает данные из файла и помещает их в очередь. Поток декодера извлекает данные и обрабатывает их. Если очередь пуста, поток декодера спит, пока в очередь не поступят данные. Основной поток после помещения данных в очередь пробуждает поток декодера. Вся логика синхронизации осуществляется с помощью привязки к очереди условной переменной. Общие данные являются очередью и предикат представляет собой условие "очередь не пуста". Поток декодера спит на условной переменной, если предикат ЛОЖЬ (то есть очередь пуста). Он пробуждается, когда основной поток "изменяет" предикат добавлением в очередь данных.

Давайте подробно обсудим реализацию мьютекса и условной переменной для pthread-ов.

Мьютекс для pthread-ов

Мьютекс инициализируется во время определения:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Он также может быть проинициализирован во время выполнения вызовом функции **pthread_mutex_init**.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                const pthread_mutexattr_t *mutexattr);
```

Первым аргументом является указатель на мьютекс, который инициализируется, а вторым аргументом - атрибуты мьютекса. Если **mutexattr** является NULL, устанавливаются атрибуты по умолчанию (подробнее об атрибутах мьютекса позже).

Мьютекс захватывается вызовом функции **pthread_mutex_lock**. Он освобождается вызовом функции **pthread_mutex_unlock**. **pthread_mutex_lock** или захватывает мьютекс, или приостанавливает выполнение вызывающего потока, пока владелец мьютекса (то есть поток, который захватил мьютекс вызовом функции **pthread_mutex_lock** ранее) не освободит его вызовом **pthread_mutex_unlock**.

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Общие данные могут быть защищены с помощью функций блокировки и разблокировки мьютекса следующим образом:

```
pthread_mutex_lock(&lock);
/* работаем с общими данными */
pthread_mutex_unlock(&lock);
```

Есть три типа мьютекса.

- Быстрый мьютекс
- Рекурсивный мьютекс
- Мьютекс проверки ошибки

Поведения этих трёх типов мьютекса похожи; они отличаются только когда владелец мьютекса вновь вызывает **pthread_mutex_lock**, что снова захватить его.

- Для быстрого мьютекса возникает тупиковая ситуация, поскольку поток теперь ждёт самого себя, чтобы разблокировать мьютекс
- Для рекурсивного мьютекса функция возвращается сразу и счётчик захвата мьютекса увеличивается на единицу. Мьютекс разблокируется только если счетчик дойдёт до нуля; то есть поток должен вызвать **pthread_mutex_unlock** для каждого вызова **pthread_mutex_lock**.
- Для мьютекса проверки ошибки **pthread_mutex_lock** возвращает ошибку с кодом ошибки **EDEADLK**.

Быстрый, рекурсивный мьютекс и мьютекс проверки ошибки инициализируются во время определения следующим образом:

```
/* Быстрый мьютекс */
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

/* Рекурсивный мьютекс */
pthread_mutex_t lock =
   PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

/* Мьютекс проверки ошибки */
pthread_mutex_t lock =
   PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP;
```

Они также могут быть проинициализированы во время выполнения вызовом функции **pthread_mutex_init**. Напомним, что передача NULL в качестве второго аргумента **pthread_mutex_init** устанавливает для мьютекса атрибуты по умолчанию. По умолчанию мьютекс инициализируется как быстрый мьютекс.

```
/* Быстрый мьютекс */
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);
```

Рекурсивные мьютекс инициализируется во время выполнения следующим образом:

```
pthread_mutex_t lock;
pthread_mutexattr_t mutex_attr;
pthread_mutexattr_init(&mutex_attr);
pthread_mutexattr_settype(&mutex_attr,
   PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP);
pthread_mutex_init(&lock, &mutex_attr);
```

Мьютекс проверки ошибки инициализируется во время выполнения аналогично показанному выше; только в вызове **pthread_mutexattr_settype** изменяется тип мьютекса на **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP**.

Условная переменная pthread-ов

Как и мьютекс, условная переменная инициализируется либо во время определения, либо во время выполнения вызовом функции **pthread_cond_init**.

```
pthread_cond_t cond_var = PTHREAD_COND_INITIALIZER;
```

или

```
pthread_cond_t cond_var;
pthread_cond_init(&cond_var, NULL);
```

Давайте вернёмся к нашему MP3-плееру, чтобы понять различные операции с условной переменной pthread-ов. В них участвуют три объекта, показанные на Рисунке 6.6.

- Основной поток: он порождает поток декодера звука. Он также поставляет данные для потока декодера.
- Поток декодера: он декодирует и воспроизводит звук из данных, предоставленных

основным потоком.
- Очередь: это общая структура данных между основным потоком и потоком декодера. Основной поток читает данные из файла и помещает их в очередь, а поток декодера извлекает данные и работает с ними.

Основной поток порождает поток декодера при запуске приложения.

```
int main(){

  pthread_t decoder_tid;
    ...

  /* Создаём поток декодера звука */
  if (pthread_create(&decoder_tid, NULL, audio_decoder,
                     NULL ) != 0){
    printf("Audio decoder thread creation failed.\n");
    return FAIL;
  }
  ...
  ...
}
```

Возникают три вопроса:

- Как поток декодера узнаёт, что в очереди есть данные? Должен ли он делать опрос? Есть ли более эффективный механизм?
- Есть ли способ для основного потока проинформировать поток декодера о наличии в очереди данных?
- Как можно защитить очередь от одновременного доступа со стороны основного потока и потока декодера?

Чтобы ответить на эти вопросы, давайте сначала посмотрим на детали потока декодера звука.

```
void* audio_decoder(void *unused){

  char *buffer;
  printf("Audio Decoder thread started\n");

  for(;;){
    pthread_mutex_lock(&lock);
    while(is_empty_queue())
      pthread_cond_wait(&cond, &lock);

    buffer = get_queue();

    pthread_mutex_unlock(&lock);

    /* декодируем данные в буфере */
    /* посылаем декодированные данные на выход для воспроизведения */

    free(buffer);
  }
```

```
}
```

Обратите, пожалуйста, внимание на следующий кусок кода в функции **audio_decoder**.

```
while(is_empty_queue())
  pthread_cond_wait(&cond, &lock);
```

Здесь мы ввели условную переменную **cond**. Предикатом для этой условной переменной является "очередь не пуста". Таким образом, если предикат является ложным (то есть очередь пуста), поток засыпает на условной переменной, вызвав функцию **pthread_cond_wait**. Поток будет оставаться в состоянии ожидания, пока какой-нибудь другой поток не просигнализирует об изменении условия (то есть изменит предикат путём добавления в очередь данных, что сделает её не пустой). Прототип этой функции:

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
```

Вы можете увидеть в приведённой выше декларации, что с условной переменной также связан мьютекс. Мьютекс необходим для защиты предиката, когда поток проверяет её состояние. Это позволяет избежать состояния гонок, когда один поток готовится ждать на условной переменной, а другой поток просигнализировал о состоянии как раз перед тем, как первый поток на самом деле впал в состояние ожидания.

```
while(is_empty_queue())
   <--- О состоянии сигнализирует другой поток --->
  pthread_cond_wait(...);
```

В нашем примере если бы не было мьютекса, поток декодера ждал бы на условной переменной, даже если бы в очереди были данные. Так что правило: проверка предиката и сон на условной переменной должны быть атомарной операцией. Эта атомарность достигается введением мьютекса вместе с условной переменной. Таким образом, шагами являются:

```
pthread_mutex_lock(&lock);  <-- Получаем мьютекс
while(is_empty_queue())     <-- проверяем предикат
  pthread_cond_wait(&cond,&lock);<-- засыпаем на условной переменной
```

Чтобы сделать эту работу, необходимо, чтобы все участвующие потоки должны были захватывать мьютекс, изменять/проверять состояние, а затем освобождать мьютекс.

Теперь, что происходит с мьютексом, когда поток переходит в спящий режим в **pthread_cond_wait**? Если мьютекс остаётся в состоянии блокировки, то никакой другой поток не сможет просигнализировать о состоянии, так как этот поток тоже попытается захватить тот же мьютекс перед изменением предиката. У нас проблема: один поток удерживает блокировки и спит на условной переменной, а другой поток ожидает блокировку для установки состояния. Для того чтобы избежать такой проблемы, соответствующий мьютекс должен быть разблокирован после засыпания потока на условной переменной. Это делается в функции **pthread_cond_wait**. Функция помещает поток в состояние сна и автоматически освобождает мьютекс.

Поток, спящий на условной переменной, пробуждается и функция **pthread_cond_wait** возвращается, когда другой поток устанавливает условие (вызовом функции **pthread_cond_signal** или **pthread_cond_broadcast**, как описано далее в этом разделе).

pthread_cond_wait также перед возвратом повторно захватывает мьютекс. Поток теперь может проверить условие и освободить мьютекс.

```
pthread_mutex_lock(&lock); <-- Получаем мьютекс
while(is_empty_queue()) <-- Проверяем условие
  pthread_cond_wait(&cond,&lock); <-- ждём на условной переменной
<-- Мьютекс повторно захватывается внутри pthread_cond_wait -->
buffer = get_queue(); <-- Обрабатываем условие
pthread_mutex_unlock(&lock);<-- По завершении освобождаем мьютекс
```

Давайте посмотрим, как поток может просигнализировать о состоянии. Шагами являются:

- Получение мьютекса.
- Изменение состояния.
- Освобождение мьютекса.
- Пробуждение одного или всех потоков, которые спали на условной переменной.

Основной поток нашего плеера пробуждает поток декодера звука после добавления в очередь данных.

```
fp = fopen("song.mp3", "r");
while (!feof(fp)){
  char *buffer = (char *)malloc(MAX_SIZE);
  fread(buffer, MAX_SIZE, 1, fp);

  pthread_mutex_lock(&lock); <-- Получаем мьютекс
  add_queue(buffer); <-- изменяем условие. Добавление
                         буфера в очередь делает её
                         не пустой

  pthread_mutex_unlock(&lock); <-- Освобождаем мьютекс
  pthread_cond_signal(&cond); <-- Пробуждаем поток декодера

  usleep(300*1000);
}
```

pthread_cond_signal пробуждает единственный поток, спящий на условной переменной. Чтобы пробудить все потоки, которые спят на условной переменной, также доступна функция **pthread_cond_broadcast**.

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

6.3.3 Завершение потока

Как один поток может прекратить выполнение другого потока? В нашем примере плеера после завершения воспроизведения основной поток должен остановить поток декодера перед тем, как приложение завершит работу. Это достигается с помощью функции **pthread_cancel**.

```
int pthread_cancel(pthread_t thread_id);
```

pthread_cancel посылает запрос на завершение потока **thread_id**. В нашем плеере основной поток вызывает функцию **pthread_cancel**, чтобы отправить запрос на остановку потока декодера, и ждёт перед выходом его завершения.

```
int main(){
    ...
    ...
  pthread_cancel(decoder_tid); <-- посылаем запрос
                                   на завершение
  pthread_join(decoder_tid, NULL);
}
```

Поток, который получает запрос на прекращение работы, может проигнорировать его, выполнить его немедленно, или отложить выполнение запроса. Чтобы определить, какое действие выполняется, когда потоком получен запрос на прекращение работы, существуют две функции:

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
```

pthread_setcancelstate вызывается, чтобы проигнорировать или принять запрос на завершение. Запрос игнорируется, если аргументом **state** является **PTHREAD_CANCEL_DISABLE**. Завершение разрешено, если **state** имеет значение **PTHREAD_CANCEL_ENABLE**. Если завершение разрешено, вызывается **pthread_setcanceltype**, чтобы установить либо немедленный, либо отложенный тип завершения. Завершение выполняется немедленно, если аргумент **type** имеет значение **PTHREAD_CANCEL_ASYNCHRONOUS**. Если **type** имеет значение **PTHREAD_CANCEL_DEFERRED**, запрос на завершение откладывается до следующей точки завершения.

По умолчанию поток всегда начинает работать с разрешённым завершением и с отложенным типом завершения. В примере плеера поток декодера вызывает функцию **pthread_setcanceltype**, чтобы установить завершение немедленного типа.

```
void* audio_decoder(void *unused){
    ...
    ...
  pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
    ...
}
```

Как уже упоминалось ранее, выполнение запроса на завершение может быть отложено до следующей точки завершения. Так что же это за точки завершения? Точками завершения являются такие функции, где выполняется проверка на наличие ожидающего обработки запроса на завершение. Если такой запрос ждёт обработки, завершение выполняется сразу же. В общем случае любая функция, которая приостанавливает выполнение текущего потока в течение длительного времени, должна быть точкой завершения. Такие функции для работы с pthread, как **pthread_join**, **pthread_cond_wait**, **pthread_cond_timedwait** и **pthread_testcancel** служат точками завершения. Пожалуйста, обратите внимание, что выполнение запроса завершения в любой данной точке эквивалентно вызову в данной точке **pthread_exit(PTHREAD_CANCELED)**.

Поток может проверить, есть ли ожидающий обработки запрос на завершение, вызвав

функцию **pthread_testcancel**.

```
void pthread_testcancel(void);
```

Завершение выполняется сразу же, если запрос на завершение находится в состоянии ожидания обработки, когда вызывается эта функция.

6.3.4 Отдельные потоки

Как уже говорилось ранее, поток, создаваемый с помощью **pthread_create** со стандартным набором атрибутов, является объединяемым потоком. Для освобождения ресурсов, выделенных объединяемому потоку, необходимо вызвать **pthread_join**. Иногда мы хотим создавать "независимые" потоки. Они должны завершиться, когда они захотят, и не нуждаются в другом потоке для присоединения к ним. Для достижения этого мы должны поместить их в состояние "отдельный" (detached). Это можно сделать двумя способами:

- Установкой атрибута **DETACH** во время создания потока

```
pthread_attr_t attr;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, routine, arg);
```

- Функцией **pthread_detach**

```
int pthread_detach(pthread_t tid);
```

Любой поток может поместить поток **tid** в состояние "отдельный", вызвав функцию **pthread_detach**. Поток также может поместить самого себя в состояние "отдельный", вызвав

```
pthread_detach(pthread_self());
```

6.4 Уровень переноса операционной системы (OSPL)

OSPL (Operating System Porting Layer, уровень переноса операционной системы) эмулирует интерфейсы RTOS, используя API Linux. Хорошо написанный OSPL должен свести к минимуму изменения в существующем коде. В этом разделе мы рассмотрим структуру OSPL. Для этой цели мы определили интерфейсы нашей собственной RTOS. Эти интерфейсы похожи на интерфейсы, имеющиеся в традиционной RTOS. Мы обсудим интерфейсы создания задачи, завершения задачи и мьютекса. Наш OSPL является одной библиотекой, которая компонуется и в пространстве ядра и в пользовательском пространстве. Определения находятся в файле **ospl.c**. Файл заголовка **ospl.h** эмулирует типы данных RTOS используя типы данных Linux. Сначала мы обсуждаем интерфейсы RTOS для мьютекса, так как в нашей реализации они имеют связь один-к-одному с интерфейсами мьютекса в Linux. Интерфейсы в RTOS для задачи имеют связь с эквивалентными интерфейсами Linux один-ко-многим.

В этом разделе мы обсудим OSPL для задач, которые реализованы в виде потоков в процессе Linux. В Разделе 6.4.3 166 мы покажем эмуляцию интерфейсов таймеров и межпроцессных взаимодействий для использования в процессах Linux.

6.4.1 Эмуляция интерфейсов мьютекса RTOS

Прототипами интерфейсов мьютекса в нашей RTOS являются:

- `rtosError_t rtosMutexInit(rtosMutex_t *mutex)`: инициализация мьютекса для операций блокировки, разблокировки и пробной блокировки.
- `rtosError_t rtosMutexLock(rtosMutex_t *mutex)`: захват мьютекса, если он не заблокирован; засыпание, если мьютекс уже заблокирован.
- `rtosError_t rtosMutexUnlock(rtosMutex_t *mutex)`: разблокировка мьютекса, захваченного ранее вызовом **rtosMutexLock**.
- `rtosError_t rtosMutexTrylock(rtosMutex_t *mutex)`: захват мьютекса, если он не заблокирован. Возвращает **RTOS_AGAIN**, если мьютекс уже заблокирован.

Обратите внимание, что все вышеперечисленные интерфейсы возвращают одно из значений **enum rtosError_t**, находящегося в файле **rtosTypes.h**. Этот файл RTOS включён в **ospl.h**.

```
typedef enum {
  RTOS_OK,
  RTOS_AGAIN,
  RTOS_UNSUPPORTED,
    ...
  RTOS_ERROR,
}rtosError_t;
```

Эти функции в качестве аргумента принимают указатель на объект **rtosMutex_t**. **rtosMutex_t** эмулируется в файле **ospl.h** следующим образом:

```
#ifndef __KERNEL__
  typedef pthread_mutex_t rtosMutex_t; <-- определение
                              в пространстве пользователя
#else
  typedef struct semaphore rtosMutex_t; <-- определение
                                          в ядре
#endif
```

В пользовательском пространстве интерфейсы мьютекса RTOS эмулируются с использованием мьютексных операций pthread-ов. В ядре они эмулируются с использованием семафоров ядра. Давайте поговорим о реализации этих интерфейсов в нашем OSPL.

Функция **rtosMutexInit** эмулируется в пользовательском пространстве с помощью функции **pthread_mutex_init**. В ядре используется функция **init_MUTEX**. **init_MUTEX** инициализирует семафор значением 1.

```
rtosError_t rtosMutexInit(rtosMutex_t *mutex){
#ifndef __KERNEL__
  pthread_mutex_init(mutex, NULL);
#else
  init_MUTEX(mutex);
#endif
```

```
  return RTOS_OK;
}
```

Версия **rtosMutexLock** для пользовательского пространства использует функцию **pthread_mutex_lock**. В ядре используется функция **down**.

```
rtosError_t rtosMutexLock(rtosMutex_t *mutex){
  int err;
#ifndef __KERNEL__
  err = pthread_mutex_lock(mutex);
#else
  down(mutex);
  err = 0;
#endif
  return (err == 0) ? RTOS_OK : RTOS_ERROR;
}
```

down автоматически уменьшает счётчик семафора. Семафор захвачен, если после уменьшения счётчик становится нулевым; если счётчик становится отрицательным, то текущая задача помещается в непрерываемый сон на очереди ожидания семафора.Задача пробуждается только когда владелец семафора освобождает его, вызвав функцию **up**. Недостатком использования **down** является её врождённый непрерываемый сон. Вы никак не может прекратить выполнение задачи, которая спит в **down**. Чтобы полностью контролировать выполнение задачи, вместо неё должна использоваться функция **down_interruptible**. Функция аналогична **down**, за исключением того, что возвращает **-EINTR**, если сон прерван. Таким образом новой реализацией **rtosMutexLock** является:

```
rtosError_t rtosMutexLock(rtosMutex_t *mutex){
  int err;
#ifndef __KERNEL__
  err = pthread_mutex_lock(mutex);
#else
  err = down_interruptible(mutex);
#endif
  return (err == 0) ? RTOS_OK : RTOS_ERROR;
}
```

Функция **rtosMutexUnlock** реализована в пользовательском пространстве с помощью функции **pthread_mutex_unlock**. В ядре используется функция **up**. **up** атомарно увеличивает счётчик семафора и пробуждает задачи, спящие (прерываемым или непрерываемым сном) на очереди ожидания семафора.

```
rtosError_t rtosMutexUnlock(rtosMutex_t *mutex){
  int err;
#ifndef __KERNEL__
  err = pthread_mutex_unlock(mutex);
#else
  up(mutex);
  err = 0;
#endif
  return (err == 0) ? RTOS_OK : RTOS_ERROR;
}
```

Наконец, **rtosMutexTryLock** использует **pthread_mutex_trylock** в пространстве пользователя и функцию **down_trylock** в ядре. **down_trylock** не блокируется и немедленно возвращается, если семафор уже захвачен.

```
rtosError_t rtosMutexTrylock(rtosMutex_t *mutex){
  int err;
#ifndef __KERNEL__
  err = pthread_mutex_trylock(mutex);
  if (err == 0)
    return RTOS_OK;
  if (errno == EBUSY)
    return RTOS_AGAIN;
  return RTOS_ERROR;
#else
  err = down_trylock(mutex);
  if (err == 0)
    return RTOS_OK;
  return RTOS_AGAIN;
#endif
}
```

Чтобы подвести итог, в Таблице 6.1 перечислены соответствия один-к-одному между интерфейсами мьютекса RTOS и интерфейсами мьютекса Linux.

Таблица 6.1 Интерфейсы мьютекса RTOS и Linux

RTOS	*Linux*	
	User Space	*Kernel Space*
Инициализация мьютекса	**pthread_mutex_init**	**init_MUTEX**
Блокировка мьютекса	**pthread_mutex_lock**	**down_interruptible**
Разблокировка мьютекса	**pthread_mutex_unlock**	**up**
Пробная блокировка мьютекса	**pthread_mutex_trylock**	**down_trylock**

6.4.2 Эмуляция интерфейсов задачи RTOS

Давайте посмотрим на довольно сложную часть OSPL, связь один-ко-многим между API RTOS и API Linux. Возьмём в качестве примера API RTOS для создание и уничтожения задачи. Прототипами API в нашей RTOS для создания и уничтожения задачи являются:

```
rtosError_t rtosCreateTask
                  (char *name, <-- имя задачи
                   rtosEntry_t routine, <-- точка входа
```

```
                    char * arg1, <-- аргумент для точки входа
                    int arg, <-- аргумент для точки входа
                    void * arg3, <-- аргумент для точки входа
                    int priority, <-- приоритет новой задачи
                    int stackSize, <-- размер стека
                    rtosTask_t *tHandle); <-- описание задачи
```

Функция **rtosCreateTask** порождает новую задачу, которая начинает своё выполнение с функции **routine**. Приоритет новой задачи устанавливается с помощью аргумента **priority**. В случае успешного завершения функция возвращает **RTOS_OK** и сохраняет описание задачи для созданной задачи в аргументе **tHandle**.

```
void rtosDeleteTask(rtosTask_t tHandle <-- описание задачи
                    );
```

Функция **rtosDeleteTask** завершает выполнение задачи **tHandle** и ждёт её завершения. Когда функция **rtosDeleteTask** возвращается, это гарантирует, что задача **tHandle** мертва. Сначала мы обсудим реализацию этих API в пользовательском пространстве, а затем реализацию в ядре.

Эмуляция интерфейсов задачи в пользовательском пространстве

Перед переходом к деталям реализации **rtosCreateTask** и **rtosDeleteTask**, давайте обсудим два участвующих типа данных: **rtosEntry_t** и **rtosTask_t**. **rtosEntry_t** определён в **ospl.h** и он должен быть использован без переопределения, как в нашей реализации. Изменение этого типа будет означать изменение описания всех функций, основанных на этом типе, что мы хотим избежать. **rtosEntry_t** определяется как

```
typedef void (*rtosEntry_t) (char *arg1, int arg2,
                             void *arg3);
```

Внутренности **rtosTask_t** понятны только интерфейсам задач. Для других API это просто непрозрачный тип данных. Так как мы реализуем API задач RTOS с помощью API Linux, мы имеем свободу переопределить этот тип в соответствии с нашими потребностями. Новое определение в файле **ospl.h**:

```
typedef struct {
  char name[100]; <-- имя задачи
  pthread_t thread_id; <-- идентификатор потока
}rtosTask_t;
```

Сначала мы обсудим **rtosCreateTask**.

```
rtosError_t
rtosCreateTask(char *name, rtosEntry_t routine,
               char * arg1, int arg2, void *arg3,
               int priority, int stackSize,
               rtosTask_t *tHandle){
#ifndef __KERNEL__

  int err;
```

```
  uarg_t uarg;
  strcpy(tHandle->name, name);
  uarg.entry_routine = routine;
  uarg.priority = priority;
  uarg.arg1 = arg1;
  uarg.arg2 = arg2;
  uarg.arg3 = arg3;
  err = pthread_create (&tHandle->thread_id, NULL,
                        wrapper_routine, (void *)&uarg);
  return (err) ? RTOS_ERROR : RTOS_OK;

#else
  ...
}
```

Мы определим новую структуру **uarg_t**, которая содержит все аргументы входной процедуры задачи RTOS, указатель на входную процедуру и приоритет задачи.

```
typedef struct _uarg_t {
  rtosEntry_t entry_routine;
  int priority;
  char *arg1;
  int arg2;
  void *arg3;
}uarg_t;
```

Для каждого вызова **rtosCreateTask** новый поток выполнения создаётся путём вызова функции **pthread_create**. Идентификатор созданного потока хранится в аргументе **tHandle->thread_id**. Новый поток выполняет функцию **wrapper_routine**.

```
void wrapper_routine(void *arg){
  uarg_t *uarg = (uarg_t *)arg;
  nice(rtos_to_nice(uarg->priority));
  uarg->entry_routine(uarg->arg1, uarg->arg2,
                      uarg->arg3);
}
```

В **wrapper_routine** приоритет потока регулируется с помощью системного вызова **nice**. Функция **rtos_to_nice** выполняет преобразование параметров приоритетов RTOS в параметры Linux. Вам необходимо переписать эту функцию в зависимости от вашего RTOS. Наконец, она передаёт управление фактической входной процедуре с соответствующими аргументами.

Вы можете быть удивлены тем, что мы проигнорировали в данной реализации аргумент **stackSize**. В Linux указывать размер стека потока или задачи нет необходимости. О выделении стека заботится ядро. Оно увеличивает стек динамически по мере необходимости. Если по каким-либо причинам вы захотите задать размер стека, вызовите **pthread_attr_setstacksize** до вызова **pthread_create**.

Теперь обсудим **rtosDeleteTask**.

```
void rtosDeleteTask(rtosTask_t tHandle){
#ifndef __KERNEL__
  pthread_cancel(tHandle.thread_id);
```

```
  pthread_join(tHandle.thread_id, NULL);
#else
  ...
}
```

Чтобы отправить запрос на завершение потока **tHandle**, функция вызывает **pthread_cancel**. Затем, чтобы подождать его завершения, она вызывает **pthread_join**.

Обратите внимание в нашем предыдущем обсуждении завершения потоков, что **pthread_cancel** не прекращает выполнение данного потока. Она просто посылает запрос на завершение. Поток, который получает запрос, имеет возможность либо принять его, либо проигнорировать. Таким образом, чтобы успешно проэмулировать **rtosTaskdelete**, все потоки должны принять запрос на завершение к исполнению. Они могут сделать это добавлением в код явных точек завершения, например, так:

```
static void rtosTask (char * arg1, int arg2, void * arg3){
  while(1){
    /*
     * Thread body
     */
    pthread_testcancel(); <-- Добавляем явную точку
                              завершения
  }
}
```

На самом деле, чтобы принять запрос на завершение, поток может использовать любой метод, о котором говорилось ранее. Но вы должны быть осторожны при использовании немедленного типа завершения потока (**PTHREAD_CANCEL_ASYNCHRONOUS**). Поток, использующий немедленное завершение, может быть прекращён, когда он держит какую-нибудь блокировку и находится в критической секции. При завершении блокировка не освобождается, и это может привести к тупиковой ситуации, если другой поток попытается получить блокировку.

Эмуляция интерфейсов задачи в ядре

Давайте обсудим реализацию интерфейсов задачи **rtosCreateTask** и **rtosDeleteTask** в ядре. Перед переходом к деталям реализации мы сначала рассмотрим две функции ядра: **wait_for_completion** и **complete**. Эти две функции используются в ядре для синхронизации выполнения кода. Они также используются для уведомления о событиях. Поток ядра может ждать пока произойдёт событие, вызывая **wait_for_completion** на специальной переменной. Он пробуждается другим потоком с помощью вызова **complete** на этой переменной, когда событие происходит.

Где мы нуждаемся в функциях ядра **wait_for_completion** и **complete**? В нашем OSPL задача RTOS реализуется с использованием потоков ядра. Поток ядра создаётся с помощью функции **kernel_thread**. Он может быть завершён посылкой ему сигнала. Наша реализация **rtosDeleteTask** посылает сигнал потоку ядра и вызывает для ожидания завершения **wait_for_completion**. Поток ядра при получении сигнала вызывает **complete**, чтобы перед выходом пробудить вызывающего **rtosDeleteTask**.

Первой является **rtosCreateTask**.

```
rtosError_t
```

```
rtosCreateTask(char *name, rtosEntry_t routine, void * arg,
               int priority, int stackSize,
               rtosTask_t *tHandle){
#ifndef __KERNEL__
    ...
#else
  struct completion *complete_ptr =
        (struct completion *)kmalloc(sizeof(structcompletion),
                                     GFP_KERNEL);
  karg_t *karg = (karg_t *)kmalloc(sizeof(karg_t),
                                   GFP_KERNEL);
  strcpy(tHandle->name, name);
  init_completion(complete_ptr); <-- Initialize a completion
                                     variable
  tHandle->exit = complete_ptr;
  karg->entry_routine = routine;
  karg->priority = priority;
  karg->arg1 = arg1;
  karg->arg2 = arg2;
  karg->arg3 = arg3;
  karg->exit = complete_ptr;
  tHandle->karg = karg;
  tHandle->thread_pid =
   kernel_thread(wrapper_routine, (void *)karg, CLONE_KERNEL);
  return RTOS_OK;

#endif

}
```

Каждая задача ядра RTOS связана с переменной **complete_ptr** для обработки его завершения. Она инициализируется вызовом функции ядра **init_completion**. **complete_ptr** обёрнута вместе с аргументами входной процедуры **arg** и приоритетом **priority** в структуру **karg** типа **karg_t**. Наконец, для создания потока ядра, который начинает исполняться с функции **wrapper_routine**, вызывается функция **kernel_thread**. Аргументом **wrapper_routine** является **karg**. **kernel_thread** возвращает для созданного потока идентификатор потока, который хранится в **tHandle->thread_pid**. Обратите внимание, что для последующего использования в функции **rtosDeleteTask** структура **tHandle** также хранит **complete_ptr** и **karg**.

wrapper_routine устанавливает приоритет используя **sys_nice** и вызывает фактическую процедуру входа.

```
void wrapper_routine(void *arg){
  karg_t *karg = (karg_t *)arg;
  sys_nice(rtos_to_nice(karg->priority));
  karg->entry_routine(karg->arg1, karg->arg2,
                      karg->arg3,karg->exit);
}
```

Обратите внимание, что мы внесли изменения в прототип входной функции для размещения ещё одного аргумента, указателя на переменную завершения.

```
typedef void (*rtosEntry_t) (char *arg1, int arg2,
               void *arg3, struct completion * exit);
```

Изменения, необходимые для обработки завершения потока ядра в **rtosDeleteTask**, обсуждаются далее.

Наконец, мы обсуждаем **rtosDeleteTask**.

```
void rtosDeleteTask(rtosTask_t tHandle){
#ifndef __KERNEL__
    ...
#else
  kill_proc(tHandle.thread_pid, SIGTERM, 1);
  wait_for_completion(tHandle.exit);
  kfree(tHandle.exit);
  kfree(tHandle.karg);
#endif
}
```

Чтобы послать сигнал **SIGTERM** потоку **tHandle.thread_pid**, реализация вызывает функцию ядра **kill_proc**. Затем вызывающий **rtosDeleteTask** ожидает завершения потока вызовом **wait_for_completion** на переменной завершения **tHandle.exit**. При пробуждении он освобождает ресурсы, выделенные в **rtosCreateTask**, и возвращается.

Отправки сигнала потоку ядра не достаточно для его прекращения. Поток, который получает сигнал, должен проверять наличие ожидающих сигналов на своём пути исполнения. Если сигнал прекращения (в нашем случае **SIGTERM**) находится в состоянии ожидания, то он должен перед выходом вызвать **complete** на переменной завершения. Поток ядра в данной реализации выглядел бы так:

```
static int
my_kernel_thread (char *arg1, int arg2, void *arg3,
                  struct completion * exit)
{
  daemonize("%s", "my_thread");
  allow_signal(SIGTERM);

  while (1) {

    /*
     * Тело потока
     */

    /* Проверка на завершение */
    if (signal_pending (current)) {
      flush_signals(current);
      break;
    }

  }

  /* посылаем сигнал и завершаем работу */
  complete_and_exit (exit, 0);
}
```

Чтобы подвести итог, в Таблице 6.2 перечислены соответствия между интерфейсами задачи RTOS и Linux.

Таблица 6.2 Интерфейсы задачи RTOS и Linux

RTOS	*Linux*	
	User Space	*Kernel Space*
Создание задачи	**pthread_create**	**kernel_thread**
Удаление задачи	**pthread_cancel**	**kill_proc**

6.4.3 Эмуляция интерфейсов межпроцессного взаимодействия и таймеров

Следующими большими наборами API после успешной эмуляции интерфейсов задачи являются межпроцессные взаимодействия и таймеры. Связь интерфейсов межпроцессного взаимодействия и таймеров с эквивалентными интерфейсами Linux показана в Таблице 6.3. Как вы можете видеть, большинство функций таймеров и межпроцессного взаимодействия могут быть реализованы с использованием расширения реального времени POSIX.1b. Мы обсудим подробности поддержки в Linux POSIX.1b в Главе 7 176.

Таблица 6.3 Интерфейсы таймеров и межпроцессного взаимодействия RTOS и Linux

RTOS	*Linux*	
	User Space	*Kernel Space*
Таймеры	Таймеры POSIX.1b, таймеры BSD	Интерфейсы таймеры ядра — **add_timer**, **mod_timer** и **del_timer**
Память общего доступа	Память общего доступа SVR4, память общего доступа POSIX.1b	Реализация зависит от задачи
Очереди сообщений и почтовые ящики	Очереди сообщений SVR4, очереди сообщений POSIX.1b	Реализация зависит от задачи
Семафоры	Семафоры SVR4, семафоры POSIX.1b	Функции семафоров ядра:**down**, **up** и друзья
События и сигналы	Сигналы реального времени POSIX.1b	Функции сигналов ядра: **kill_proc**, **send_signal** и друзья

6.5 Драйвер API ядра

Одной из основных сложных задач, с которой сталкивается разработчик при переносе приложений на встроенный Linux, является наличие в Linux разных режимов программирования: пространство ядра/пространство пользователя. В Linux, поскольку адресное пространство ядра является защищённым, приложение не может непосредственно вызвать какую-любо функцию ядра или получить доступ к какой-любо структуре данных ядра. Все обращения к объектам ядра должны происходить через чётко определённые интерфейсы, называемые системными вызовами. Защищённое адресное пространство ядра значительно увеличивает усилия по переносу приложений на встроенный Linux из традиционной ОС реального времени, которая имеет плоскую модель памяти.

Давайте рассмотрим пример, чтобы понять трудности, с которыми сталкивается разработчик при переносе приложений с ОС реального времени на Linux. Изделие, работающее под управлением RTOS имеет RTC (Real Time Counter, счётчик реального времени). Для установки RTC доступна функция **rtc_set**. **rtc_set** изменяет регистры RTC для установки нового значения.

```
rtc_set(new_time){
  Получаем из new_time год, месяц, день, час, минуту и секунду;
  программируем регистры RTC новыми значениями;
}
```

Также доступна функция **rtc_get_from_user**, которая принимает от пользователя новое значение RTC и вызывает функцию **rtc_set**.

```
rtc_get_from_user(){
  читаем значение времени, введённое пользователем в new_time;
  вызываем rtc_set(new_time);
}
```

При переносе вышеописанного приложения разработчик под Linux сталкивается с дилеммой. Функция **rtc_set** непосредственно изменяет регистров RTC, поэтому она должна перейти в ядро. С другой стороны, функция **rtc_get_from_user** читает пользовательский ввод, поэтому она должна перейти в пользовательское пространство. В Linux функция **rtc_get_from_user** не сможет вызывать функцию **rtc_set**, так как последняя является функцией ядра.

При переносе такого приложения на Linux возможны следующие решения.

- ***Перенести всё в ядро***: Это решение сможет работать, но это уничтожает преимущества перехода на Linux, защиту памяти.
- ***Написать новые системные вызовы***: при таком подходе для каждой функции, перенесённой в ядро, для вызова этой функции в пространстве пользователя мог бы быть предоставлен интерфейс системного вызова. Недостатки такого подхода:
  - Все системные вызовы регистрируются в системной таблице вызовов ядра. Эту таблицу трудно поддерживать, если число функций, перенесённых в ядро, велико.
  - Обновление до новой версии ядра потребует пересмотра таблицы системных вызовов для проверки, не выделило ли ядро запись в системной таблицы вызовов для вашей функции какой-то другой функции ядра.

В этом разделе мы обсудим эффективный метод переноса таких приложения на Linux. Мы называем это ***драйвером API ядра (kapi, kernel API driver)***. При таком подходе для каждой

функции ядра, которая должна быть экспортирована в пространство пользователя, в пользовательском пространстве пишется функция-заглушка. При вызове этой заглушки происходит вызов драйвера API ядра, который затем вызывает требуемую функцию в ядре. Драйвер API ядра (или драйвер kapi) реализован в виде символьного драйвера **/dev/kapi**. Он обеспечивает интерфейс **ioctl** для всех функций, которые должны быть экспортированы в пространство пользователя. Рисунок 6.7 поясняет случай нашего примера RTC.

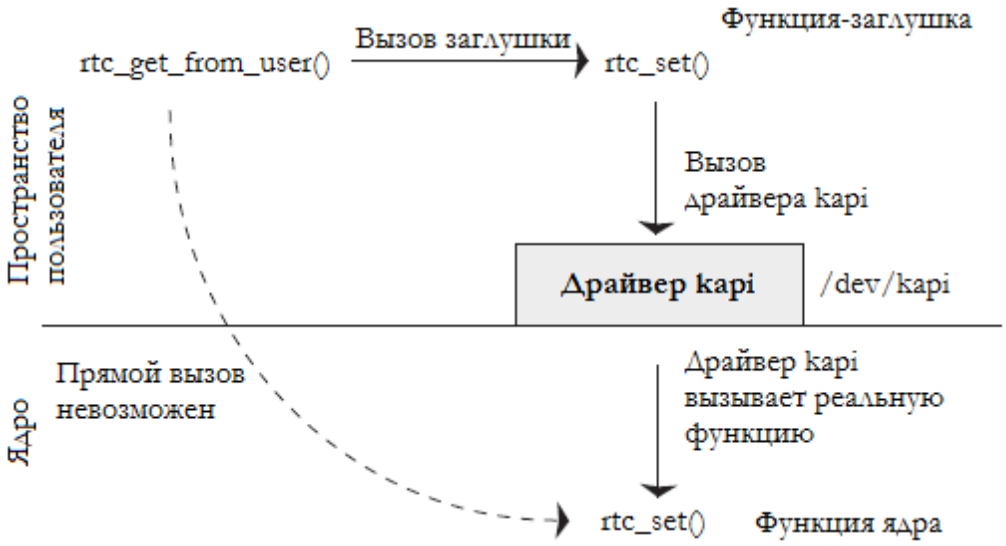

Рисунок 6.7 Экспорт функций ядра с помощью kapi.

Таким образом, чтобы перенести вышеописанное приложения для работы с RTC на Linux, разработчик должен:

- ***Обеспечить в kapi драйвере ioctl* RTC_SET**: реализация этого ioctl в драйвере вызывает функцию ядра **rtc_set** с необходимыми аргументами.
- ***Написать в пространстве пользователя заглушку* rtc_set**: заглушка вызывает ioctl **RTC_SET** для **/dev/kapi**. Она также передаёт необходимые параметры для этого ioctl.

Таким образом, используя драйвер kapi, функция пользовательского пространства **rtc_get_from_user** вызывает функцию-заглушку **rtc_set**, как будто это вызов реальной функции ядра.

В этом разделе мы обсудим:

- Шаги для написания функций-заглушек пользовательского пространства
- Реализацию драйвера kapi
- Как добавить свою функцию ioctl в драйвер kapi

Примером драйвера kapi является модуль ядра, написанный для ядра 2.6. Исходные тексты находятся в следующих файлах.

- **kapi-user.c**: содержит пример заглушки в пользовательском пространстве
- **kapi-kernel.c**: содержит исходный код драйвера kapi
- **kapi.h**: заголовочный файл, который должен быть включён в **kapi-user.c** и **kapi-kernel.c**

В этом примере мы экспортируем в пространство пользователя функцию ядра **my_kernel_func**. Прототип функции:

```
int my_kernel_func(int val, char* in_str, char *out_str);
```

Эта функция принимает три аргумента, один целочисленный и два указателя на char. Первый указатель на char является входным для функции (copy-in), а второй указатель используется для вывода данных из функции (copy-out). Возвращаемым значением функции является целое число.

6.5.1 Написание функций-заглушек пользовательского пространства

Распечатка 6.1 170 показывает структуры данных, участвующие в написании функции-заглушки, они находятся в файле **kapi.h**.

Заглушка должна заполнить соответствующие структуры данных перед вызовом соответствующего ioctl драйвера kapi.Структурами данных являются:

- **dir_t**: каждый аргумент, передаваемый в функцию, имеет направление, связанное с ним. Аргумент с направлением **DIR_IN** является для функции входным. Аргумент с направлением **DIR_OUT** является для функции выходным.
- **arg_t**: это объект для хранения одного аргумента в функции. Настоящий аргумента приводится к типу указателя **void ***, а его размер и направление копирования хранятся в полях **size** и **dir**, соответственно.
- **kfunc_t**: это основная структура данных. Указатель на объект типа **kfunc_t** передаётся в качестве аргумента для ioctl. Драйвер kapi также использует эту структуру, чтобы отправить обратно в функцию-заглушку возвращаемое значение функции ядра.
  – **num**: количество аргументов
  – **args**: объекты типа **arg_t**, заполненные на каждого аргумента
  – **ret**: возвращаемое значение функции
- **function_id**: перечисление, содержащее команды ioctl для каждой функции, которая должна быть экспортирована в пространство пользователя.

Итак, функция-заглушка **my_kernel_func**, представляющая настоящую **my_kernel_func** в ядре, находится в файле **kapi-user.c**, текст её можно увидеть в Распечатке 6.2 170.

Перед вызовом функций-заглушек следует убедиться, что драйвер kapi открыт успешно, как это делается в функции **main** в **kapi-user.c**.

```
char out[MAX_SIZE];
int ret;

dev_fd = open("/dev/kapi", O_RDONLY, 0666);

if (dev_fd < 0){
  perror("open failed");
  return 0;
}

/* вызов заглушки */
ret = my_kernel_func(10, "Hello Kernel World", out);
```

```
printf("result = %d, out_str = %s\n", ret, out);
```

Распечатка 6.1 Заголовочный файл драйвера kapi

Распечатка 6.1

```
/* kapi.h */

#ifndef _KAPI_H
#define _KAPI_H

#define MAX_ARGS 7

typedef enum _dir_t {
  DIR_IN = 1,
  DIR_OUT,
}dir_t;

typedef struct _arg_struct {
  void *val;
  unsigned int size;
  dir_t dir;
}arg_t;

typedef struct _kfunc_struct {
  int num;
  arg_t arg[MAX_ARGS];
  arg_t ret;
}kfunc_t;

enum _function_id {
  MY_KERNEL_FUNC = 1,
  MAX_FUNC
};

#endif /* _KAPI_H */
```

Распечатка 6.2 Пример функции-заглушки пользовательского пространства

Распечатка 6.2

```
/* kapi-user.c */

#include <fcntl.h>
#include "kapi.h"

/* Дескриптор файла для “/dev/kapi” */
int dev_fd;

#define MAX_SIZE 50

int my_kernel_func(int val, char* in_str, char *out_str){
```

```
  kfunc_t data;
  int ret_val;

  /* Общее число аргументов - три */
  data.num = 3;

  /*
   * Аргумент 1.
   * Даже аргументы-не указатели должны быть переданы как указатели.
   * Направлением для таких аргументов является DIR_IN
   */
  data.arg[0].val = (void *)&val;
  data.arg[0].size = sizeof(int);
  data.arg[0].dir = DIR_IN;

  /* Аргумент 2 */
  data.arg[1].val = (void *)in_str;
  data.arg[1].size = strlen(in_str) + 1;
  data.arg[1].dir = DIR_IN;

  /*
   * Аргумент 3. Так как это аргумент для вывода данных, необходимо
   * указать размер приёмного буфера
   */
  data.arg[2].val = (void *)out_str;
  data.arg[2].size = MAX_SIZE;
  data.arg[2].dir = DIR_OUT;

  /*
   * Возвращаемое значение функции ядра. Установка поля направления
   * не требуется, так как оно всегда выходное
   */
  data.ret.val = (void *)&ret_val;
  data.ret.size = sizeof(int);

  /*
   * Наконец, вызываем ioctl для /dev/kapi. Затем драйвер kapi
   * вызывает функцию ядра my_kernel_func. Она также заполняет
   * data.ret.val возвращаемым значением функции ядра.
   */
  if (ioctl(dev_fd, MY_KERNEL_FUNC, (void *)&data) < 0){
    perror("ioctl failed");
    return -1;
  }

  /* Возвращаемое значение функции */
  return ret_val;
}
```

6.5.2 Реализация драйвера kapi

Драйвер kapi является символьным драйвером, реализованным в виде модуля ядра. В этом разделе обсуждаются детали реализации драйвера для ядра 2.6.

Существуют две основные структуры данных.

- **struct file_operations kapi_fops**: эта таблица содержит функции файловых операций, такие как **open**, **close**, **ioctl** и другие для данного драйвера.

```
static struct file_operations kapi_fops = {
  .owner   = THIS_MODULE,
  .llseek  = NULL,
  .read    = NULL,
  .write   = NULL,
  .ioctl   = kapi_ioctl,
  .open    = kapi_open,
  .release = kapi_release,
};
```

- Обратите внимание, что файловые операции **read**, **write** и **lseek** установлены в **NULL**. Эти операции недопустимы для драйвера kapi, так как все операции выполняются через интерфейс ioctl.
- **struct miscdevice kapi_dev**: драйвер kapi зарегистрирован как символьный драйвер, выполняющий разнообразные операции (miscellaneous character driver). Младшим номером является **KAPI_MINOR** (111). Старшим номером любого символьного драйвера, выполняющего разнообразные операции, является 10.

```
static struct miscdevice kapi_dev = {
  KAPI_MINOR,
  "kapi",
  &kapi_fops,
};
```

- Каждый модуль ядра имеет функцию инициализации модуля и функцию освобождения модуля. Драйвер kapi предоставляет **kapi_init** и **kapi_cleanup_module** в качестве своих функций инициализации и очистки, соответственно.

Функция **kapi_init** регистрирует драйвер kapi как символьный драйвер, выполняющий разнообразные операции, с младшим номером **KAPI_MINOR**. Функция вызывается при загрузке модуля.

```
static int __init
kapi_init(void)
{
  int ret;
  ret = misc_register(&kapi_dev);
  if (ret)
    printk(KERN_ERR "kapi: can't misc_register on
                     minor=%d\n", KAPI_MINOR);
  return ret;
}
```

kapi_cleanup_module вызывается, когда модуль выгружается. Эта функция отменяет регистрацию драйвера.

```
static void __exit
kapi_cleanup_module(void)
{
  misc_deregister(&kapi_dev);
}
```

Процедуры **open** и **close** просто отслеживают количество одновременно работающих пользователей данного драйвера. Они используются в основном для отладки.

```
static int
kapi_open(struct inode *inode, struct file *file)
{
  kapi_open_cnt++;
  return 0;
}

static int
kapi_release(struct inode *inode, struct file *file)
{
  kapi_open_cnt--;
  return 0;
}
```

Давайте теперь обсудим ядро драйвера kapi, функцию **kapi_ioctl**. **kapi_ioctl** зарегистрирована в таблице **fops** как операция ioctl для данного драйвера.

Функция **kapi_ioctl** выполняет следующие операции:

1. Копирование передаваемого пользователем объекта **kfunc_t** в объект ядра **kfunc_t**.
2. Выделение памяти для **DIR_IN** и **DIR_OUT** аргументов.
3. Копирование из пользовательских буферов всех аргументов, которые имеют флаг направления **DIR_IN**, в буферы ядра.
4. Вызов запрошенной функции ядра.
5. Копирование из буферов ядра всех аргументов с флагом направления **DIR_OUT** в пользовательские буферы.
6. Наконец, копирование возвращаемого значения функции ядра в пространство пользователя и освобождение всех выделенных буферов ядра.

Аргументом для **kapi_ioctl** является указатель на объект типа **kfunc_t**. Первый шаг заключается в копировании объекта **kfunc_t** в память ядра.

```
static int
kapi_ioctl(struct inode *inode, struct file *file,
           unsigned int cmd, unsigned long arg)
{
  int i,err;
  kfunc_t kdata,udata;

  if(copy_from_user(&udata, (kfunc_t *)arg,
                    sizeof(kfunc_t)))
```

```
return -EFAULT;
```

Выделяем буферы ядра для всех аргументов и возвращаемого значения. Выполняем копирование всех аргументов, имеющих направление **DIR_IN**.

```
for (i = 0 ; i < udata.num ; i++){
  kdata.arg[i].val = kmalloc(udata.arg[i].size,
                             GFP_KERNEL);
  if (udata.arg[i].dir == DIR_IN){
    if (copy_from_user(kdata.arg[i].val, udata.arg[i].val,
                       udata.arg[i].size))
      goto error;
  }
}
kdata.ret.val = kmalloc(udata.ret.size, GFP_KERNEL);
```

Вызываем запрошенную функцию ядра. В этом примере мы предоставили ioctl для функции ядра **my_kernel_func**. Вам необходимо таким же образом добавить свои функции в инструкцию switch. Идентификатор функции также должен быть добавлен в перечисление **function_id** в **kapi.h**. Возвращаемое значение вызываемой функции следует сохранить в **kdata.ret.val**.

```
switch (cmd) {

case MY_KERNEL_FUNC:
  *(int *)(kdata.ret.val) =
      my_kernel_func(*(int *)kdata.arg[0].val,
                     (char *)kdata.arg[1].val,
                     (char *)kdata.arg[2].val);
  break;

default:
  return -EINVAL;
}
```

Теперь пришло время отправить результат работы вызываемой функции ядра обратно в функцию-заглушку пространства пользователя. Копируем все аргументы, имеющие направление **DIR_OUT**, и возвращаемое значение функции в пользовательское пространство. Также освобождаем выделенные буферы ядра.

```
err = 0;
for (i = 0 ; i < udata.num ; i++){
  if (udata.arg[i].dir == DIR_OUT){
    if (copy_to_user(udata.arg[i].val, kdata.arg[i].val,
                     udata.arg[i].size))
      err = -EFAULT;
  }
  kfree(kdata.arg[i].val);
}

/* копирование возвращаемого значения */
if (copy_to_user(udata.ret.val, kdata.ret.val,
                 udata.ret.size))
```

```
  err = -EFAULT;

kfree(kdata.ret.val);
return err;
}
```

Наконец, **my_kernel_func** просто выводит пользовательский ввод и возвращает целое значение 2. Для простоты мы поместили эту функцию в **kapi-kernel.c**. Вы не должны добавлять свои функции в этот файл. Также не забудьте проэкспортировать функцию, используя **EXPORT_SYMBOL**, так как драйвер kapi загружается как модуль ядра.

```
int my_kernel_func(int val, char *in_str, char *out_str){
  printk(KERN_DEBUG"val = %d, str = %s\n", val, in_str);
  strcpy(out_str, "Hello User Space");
  return 2;
}

EXPORT_SYMBOL(my_kernel_func);
```

6.5.3 Использование драйвера kapi

- Собираем драйвер kapi как модуль ядра. Обратитесь для получения инструкций к Главе 8, Сборка и отладка 240.
- Компилируем **kapi-user.c**.

```
# gcc -o kapi-user kapi-user.c
```

- Загружаем модуль ядра.

```
# insmod kapi-kernel.ko
```

- Создаём символьное устройство **/dev/kapi**.

```
# mknod /dev/kapi c 10 111
```

- Наконец, запускаем приложение.

```
# ./kapi-user

result = 2, out_str = Hello User Space
```

- Смотрим вывод драйвера kapi.

```
# dmesg
  ...
val = 10, str = Hello Kernel World
```

Глава 7, Linux для систем реального времени

Системы реального времени это такие системы, в которых корректность системы зависит не только от её функциональной корректности, но и от времени, за которое получены результаты. Например, если MPEG декодер внутри вашего DVD плеера не способен декодировать кадры с заданной скоростью (скажем, 25 или 30 кадров в секунду), то вы получите дёргающееся видео. Таким образом, хотя MPEG декодер функционально корректен, поскольку он способен декодировать входной видеопоток, он не в состоянии произвести результат за необходимое время. В зависимости от степени важности требования по времени выполнения, системы реального времени могут быть классифицированы либо как системы жёсткого реального времени, либо как системы мягкого реального времени.

- ***Системы жёсткого реального времени***: система жёсткого реального времени должна гарантировать время отклика в худшем случае. Вся система целиком, включая ОС, приложения, оборудование, и так далее, должны быть разработаны, чтобы гарантировать требования к системе по отклику. Не имеет значения, какие требования по времени выполнения должны быть жёсткого привязаны к реальному времени (микросекунды, миллисекунды, и так далее), просто они должны быть выполнены всегда. Невыполнение этого требования может привести к тяжёлым последствиям, таким как потеря жизни. Примерами систем жёсткого реального времени являются оборонительные системы, системы управления полётом и транспортными средствами, спутниковые системы, системы сбора данных, медицинские приборы, управление космическими кораблями или ядерными реакторами, игровые системы, и так далее.
- ***Системы мягкого реального времени***: в системах мягкого реального времени для системы не является необходимым, чтобы ограничения успешно выполнялись каждый раз. В приведённом выше примере плеера DVD, если декодер не в состоянии удовлетворить временным требованиям один раз в час, это нормально. Но частые пропуски кадров декодером в короткий период времени могут оставить впечатление, что система неисправна. Примерами таких систем являются мультимедийные приложения, передача голоса поверх IP, устройства бытовой техники, передача потокового звука или видео, и так далее.

7.1 Операционная система реального времени

POSIX 1003.1b определяет режим реального времени для операционных систем как способность операционной системы обеспечить требуемый уровень сервиса за ограниченное время отклика.

RTOS (Real-Time Operating System, Операционной Системе Реального Времени) могут быть приписаны следующие наборы функций:

- ***Многозадачность/многопоточность***: RTOS должна поддерживать многозадачность и многопоточность.
- ***Приоритеты***: задачи должны иметь приоритеты. Критические и ограниченные по времени выполнения функции должны быть обработаны задачами, имеющими более высокий приоритет.
- ***Наследование приоритета***: RTOS должна иметь механизм для поддержки наследования приоритета.
- ***Вытеснение***: RTOS должна поддерживать вытеснение; то есть, если задача с более

высоким приоритетом готова к работе, она должна вытеснять задачу с более низким приоритетом.

- ***Задержка прерывания***: задержка прерывания это время, проходящее между поступлением аппаратного прерывания и вызовом обработчика прерывания. RTOS должна иметь предсказуемую задержку прерывания и желательно, чтобы она была как можно меньше.
- ***Задержка планировщика***: это разница во времени между моментом, когда задача становится работоспособной, и моментом, когда реально начинает работать. RTOS должна иметь детерминированные задержки планировщика.
- ***Межпроцессное взаимодействие и синхронизация***: самая популярная форма взаимодействия между задачами во встроенной системе это передача сообщений. RTOS должна предлагать механизм передачи сообщений за постоянное время. Также для целей синхронизации она должна обеспечивать семафоры и мьютексы.
- ***Динамическое распределение памяти***: RTOS должна обеспечивать для приложений процедуры выделения памяти за фиксированное время.

7.2 Linux и работа в реальном времени

Linux развивались как операционная система общего назначения. После того, как Linux начали устанавливать на встраиваемые устройства, стала ощущаться необходимость работы в режиме реального времени. Основные причинами, указывающими на то, что Linux по своей природе не был системой реального времени, были:

- Высокая задержка прерывания
- Высокая задержка планировщика из-за того, что ядро по своей природе не имело поддержки вытеснения
- Различные службы ОС, такие как механизмы межпроцессного взаимодействия, выделение памяти, и тому подобные, не имеют детерминированного времени работы.
- Другие функции, такие как виртуальная память и системные вызовы, также делают Linux недетерминированным в его откликах.

Ключевым различием между любой операционной системой общего назначения, подобной Linux, и ОС жёсткого реального времени является детерминированное поведение во времени в RTOS всех служб ОС. Под детерминированным временем мы имеем в виду, что любая предполагаемая задержка или время, затраченное любой службой ОС, также должно быть ограничено. Говоря в математических терминах, вы должны иметь возможность выразить эти временные параметры с использованием алгебраической формулы без какой-либо переменной составляющей. Переменная составляющая вводит неопределённость, сценарий, неприемлемый для систем жёсткого реального времени.

Так как Linux в своей основе ОС общего назначения, требуются серьёзные изменения, чтобы получить приемлемое время отклика для всех служб ОС. Поэтому было сделано разделение: чтобы использовать Linux в системах жёсткого реального времени, были сделаны варианты Linux жёсткого реального времени, RTLinux и RTAI. С другой стороны, в ядро была добавлена поддержка для снижения задержек и улучшения времени отклика различных служб ОС, чтобы сделать его пригодным для требований мягкого реального времени.

В этом разделе обсуждается конструкция ядра, которая поддерживает использование Linux в качестве ОС мягкого реального времени. Лучший способ понять это - проследить ход обработки прерываний в системе и обратить внимание на различные вводимые задержки. Давайте возьмём пример, когда задача ожидает завершения ввода/вывода с диска и

заканчивает ввод/вывод. Выполняются следующие шаги:

- Завершение I/O. Устройство вызывает прерывание. Это вызывает работу ISR (Interrupt Service Routine, Процедуру Обработки Прерываний) драйвера блочного устройства.
- ISR проверяет очередь ожидания драйвера и находит задачу, ожидающую ввод/вывод. Затем вызывается одна из функций, предназначенных для пробуждения. Эта функция удаляет задачу из очереди ожидания и добавляет её в очередь выполнения планировщика.
- Затем ядро, когда оно попадает в точку, где планирование разрешено, вызывает функцию **schedule**.
- Наконец, **schedule()** находит следующего подходящего кандидата для работы. Контекст ядра переключается на нашу задачу, если она имеет достаточно высокий приоритет, чтобы быть запланированной.

Таким образом, ***время отклика ядра*** представляет собой количество времени, которое проходит с момента установки сигнала прерывания, до момента, когда задача, которая ожидала ввод/вывод, завершает работу. Как можно увидеть из данного примера, на время отклика ядра влияют четыре составляющих:

- ***Задержка прерывания***: задержка прерывания представляет собой разницу во времени между моментом, когда устройство установило сигнал прерывания и моментом, когда вызывается соответствующий обработчик.
- ***Продолжительность работы ISR***: время, необходимое обработчику прерывания для выполнения.
- ***Задержка планировщика***: задержка планировщика это количество времени, которое проходит между завершением процедуры прерывания и запуском функции планирования.
- ***Продолжительность работы планировщика***: это время, затраченное функцией планировщика, чтобы выбрать для запуска следующую задачу и переключить на неё контекст.

Теперь обсудим различные причины описанных выше задержек и способы их снижения, которые включены в ядро.

7.2.1 Задержка реакции на прерывание

Как уже отмечалось, задержка прерывания является одним из основных факторов, способствующих недетерминированному времени отклика системы. В этом разделе мы обсудим некоторые распространённые причины большой задержки реакции на прерывание.

- ***Запрет всех прерываний в течение длительного времени***: всякий раз, когда драйверу или другому куску кода ядра необходимо защитить какие-то данные из обработчика прерывания, обычно запрещаются все прерывания с помощью макросов **local_irq_disable** или **local_irq_save**. Удерживание спин-блокировки с помощью функций **spin_lock_irqsave** или **spin_lock_irq** перед входом в критическую секцию также отключает все прерывания. Всё это увеличивает время реакции системы на прерывание.
- ***Регистрация быстрого обработчика прерываний неправильно написанными драйверами устройств***: драйвер устройства может зарегистрировать свой обработчик прерываний в ядре либо как быстрый обработчик прерываний, либо как

медленный обработчик прерываний. Все прерывания запрещены, когда выполняется быстрый обработчик прерывания, и разрешены, когда выполняются медленные обработчики прерываний. Задержка прерывания увеличивается, если устройство с низким приоритетом регистрирует свой обработчик прерывания, как быстрый обработчик прерываний, а высокоприоритетное устройство регистрирует своё прерывание как медленное прерывание.

Как программист ядра или автор драйвера, вы должны убедиться, что ваш модуль или драйвер не способствует увеличению времени реакции на прерывание. Задержка прерывания может быть измерена с помощью инструмента **intlat**, написанного Эндрю Мортоном. Он изменялся последний раз в течение выпуска ядер версий 2.3 и 2.4 и был также зависим от архитектуры x86. Возможно, вам придется переносить его на вашу архитектуру. Он может быть загружен с http://www.zipworld.com. Вы также можете написать специальный драйвер для измерения времени реакции на прерывание. Например, для ARM это может быть достигнуто вызовом прерывания от таймера в известный момент времени и последующим сравнением с фактическим временем выполнения вашего обработчика прерывания.

7.2.2 Продожительность работы ISR

Продолжительность работы ISR это время, необходимое обработчику прерывания для выполнения, и оно находится под контролем автора ISR. Однако, может возникнуть недетерминизм, если ISR имеет также компонент программного прерывания. Что же такое программное прерывание? Мы все знаем, что для того, чтобы иметь меньшее время реакции на прерывание, обработчик прерывания должен делать минимальную работу (например, копирование каких-либо буферов ввода-вывода в оперативную память), а остальная часть работы (такая как обработка данных ввода-вывода, задачи пробуждения) должна быть выполнена вне обработчика прерываний. Так что обработчик прерывания был разделён на две части: верхняя половина, которая делает минимальную работу, и программное прерывание, которое делает остальную часть обработки. Задержка, внесённая в обработку программного прерывания неограничена. В обработке программного прерывания участвуют следующие задержки:

- Программное прерывание работает с разрешёнными прерываниями и может прерываться аппаратными прерываниями (за исключением некоторых критических секций).
- Программное прерывание также может выполняться в контексте службы ядра **ksoftirqd**, которая не является потоком реального времени.

Таким образом, вы должны убедиться, что ISR вашего устройства реального времени не имеет какого-либо компонента программного прерывания и вся работа должна выполняться только верхней половиной.

7.2.3 Задержка планировщика

Среди всех обсуждённых задержек, задержка планировщика является основной причиной увеличенного времени отклика ядра. Некоторые причины для больших задержек планировщика в ранних ядрах Linux версии 2.4 заключаются в следующем:

- ***Ядро по своей натуре не вытесняющее***: решения о планировании принимаются

ядром в таких местах, как возвращение из прерывания или возвращение из системного вызова, и так далее. Однако, если текущий процесс выполняется в режиме ядра (то есть выполняется системный вызов), решение откладывается, пока процесс не возвращается в пользовательский режим. Это означает, что высокоприоритетный процесс не может вытеснить процесс с низким приоритетом, если последний выполняет системный вызов. Таким образом, из-за невытесняющей природы режима выполнения ядра, задержки планирования могут варьироваться от десятков до сотен миллисекунд, в зависимости от продолжительности системного вызова.

- ***Моменты запрета прерываний***: решение о планирования принимается уже при возвращении из следующего прерывания по таймеру. Если в течение длительного времени отключены все прерывания, прерывание от таймера задерживается, тем самым увеличивая задержку планирования.

Чтобы уменьшить задержку планирования в Linux, было приложено немало усилий. Усилия прилагаются в двух направлениях: вытеснение в ядре и патчи для снижения задержки.

Вытеснение в ядре

Так как поддержка симметричной многопроцессорной обработки в Linux возросла, его инфраструктура блокировки также начала улучшаться. Было идентифицировано много критических секций и они были защищены с помощью спин-блокировок. Было отмечено, что вытеснение процесса, выполняющегося в режиме ядра, безопасно, если он не находится в какой-либо критической секции, защищённой с помощью спин-блокировки. Это свойство было использовано поставщиком встраиваемого Linux MontaVista, и они представили патч для поддержки вытеснения в ядре. Патч был включён в основную линейку ядра во время разработки версии 2.5 и в настоящее время поддерживается Робертом Лавом.

Поддержка вытеснения в ядре ввела в структуру задачи процесса новый член **preempt_count**. Если **preemp_count** равен нулю, ядро может быть вытеснено безопасно. Вытеснение ядра запрещено при ненулевом **preempt_count**. Работа с **preemp_count** происходит с помощью следующих основными макросов:

- **preempt_disable**: запрет вытеснения путём увеличения на единицу **preemp_count**.
- **preempt_enable**: уменьшение **preemp_count** на единицу. Вытеснение разрешено только если счётчик достиг нуля.

Чтобы соответствующим образом вызывать макросы **preempt_disable** и **preempt_enable**, были модифицированы все процедуры спин-блокировки. Процедуры спин-блокировки вызывают **preempt_disable** на входе, а процедуры разблокировки вызывают **preempt_enable** при выходе. Архитектурно-зависимые файлы, содержащие ассемблерный код для возврата из прерываний, и системные вызовы также были изменены так, чтобы проверять **preempt_count** до принятия решений о планировании. Если этот счётчик равен нулю, планировщик вызывается независимо от того, находится ли процесс в режиме ядра, или в пользовательском режиме.

Для более подробной информации смотрите, пожалуйста файлы ***include/linux/preempt.h***, ***kernel/sched.c*** и ***arch/<your-arch>/entry.S*** в исходных текстах ядра. Как уменьшается задержка планировщика, когда в ядро вносится вытеснение, показывает Рисунок 7.1.

Рисунок 7.1 Задержка планировщика в ядрах с вытеснением и без вытеснения.

Патчи для уменьшения времени задержки

Патчи для уменьшения времени задержки от Инго Молнара и Эндрю Мортона направлены на снижение задержки планирования через добавление явных точек планирования в блоки кода ядра, которые выполняются за более длительный срок. Были идентифицированы такие участки кода (например, перебор длинного списка какой-либо структуры данных). Такие куски кода были переписаны, чтобы безопасно ввести точку планирования. Иногда это влекло освобождение спин-блокировки, выполнение перепланирования, а затем новый захват блокировки. Это называется прерыванием блокировок.

При использовании патчей для уменьшения времени задержки максимальная задержка планирования снижается до максимального времени между двумя точками перепланирования. Поскольку эти патчи настраивались довольно долгое время, они выполняются на удивление хорошо. Задержка планирования может быть измерена с помощью инструмента **Schedstat**. Вы можете загрузить патч с http://eaglet.rain.com/.

Измерения показывают, что наилучший результат даёт использование одновременно и вытеснения в ядре, и патчей для снижения времени задержки.

7.2.4 Время работы планировщика

Как уже обсуждалось ранее, продолжительность времени работы планировщика представляет собой время, затраченное планировщиком на выбор следующей задачей для выполнения и на переключение на неё контекста. Планировщик Linux, как и остальная часть системы, был изначально написан для настольного ПК и оставался почти неизменным, за исключением добавления возможностей реального времени POSIX. Основным недостатком планировщика было его недетерминированное поведение: длительность работы планировщика линейно возрастала с увеличением числа задач в системе по причине того, что все задачи, в том числе задачи реального времени, содержались в одной очереди выполнения и каждый раз, когда вызывался планировщик, он проходил через всю очередь выполнения, чтобы найти задачу с наивысшим приоритетом. Этот цикл называется ***полезным циклом***. Кроме того, когда квант времени всех исполняемых процессов истекает, он заново перерасчитывает их новые временные интервалы. Этот цикл известен как ***цикл перерасчёта***. Чем больше количество задач (независимо от того, являются ли они задачами реального времени или нет), тем больше было время, затрачиваемое планировщиком в обоих этих циклах.

Создание планировщика реального времени: планировщик O(1)

В ядро версии 2.4.20 был введён планировщик O(1), который внёс детерминизм. Планировщик O(1) от Инго Молнара является превосходным куском кода, который пытается решить проблемы задачи составления расписания работы от больших серверов, стараясь сбалансировать нагрузку, вплоть до встраиваемых систем, которым требуется детерминированное время планирования. Как следует из названия, для перерасчёта интервалов времени процессов и их перепланирования, планировщик выполняет расчёт O (1), вместо прежнего O(***n***) (где ***n*** обозначает количество процессов в очереди выполнения). Он делает это с помощью двух массивов: массива активных задача и массива закончивших работу задач. Оба массива упорядочены по приоритету и они поддерживают отдельные очереди выполнения для каждого приоритета. Индексы массивов хранятся в битовом виде, так что поиск наиболее приоритетной задачи становится операцией поиска O(1). Когда задача исчерпывает свой квант времени, она перемещается в массив закончивших работу задач и переопределяется её новый квант времени. Когда массив активных задач становится пустым, планировщик переключает оба массива, так что массив закончивших работу задач становится новым массивом активных задач и начинается планирование из нового массива. Обращение к очереди активных и закончившихся задач происходит с помощью указателей, так что переключение между двумя массивами означает просто переключение указателей.

Таким образом, наличие упорядоченных массивов решает проблему полезного цикла, а переключение между указателями решает проблему цикла перерасчёта. Наряду с этими, планировщик O(1) предлагает давать более высокий приоритет интерактивным задачам. Хотя это более полезно для среды настольного ПК, системы реального времени, выполняющие смесь из процессов реального времени и обычных процессов, тоже могут извлечь выгоду из этой возможности. Рисунок 7.2 показывает планировщик O(1) в упрощённом виде.

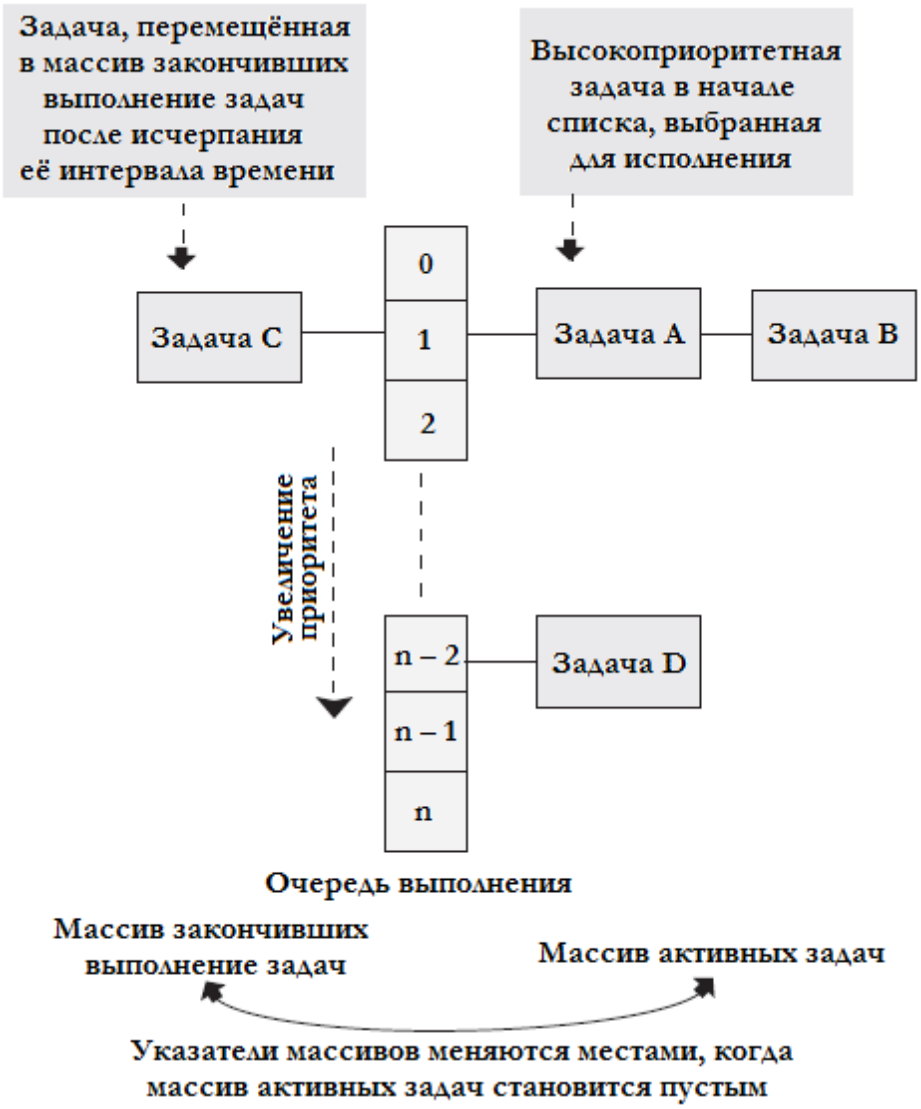

Рисунок 7.2 Упрощённый вид планировщика O(1).

Время переключения контекста

Измерение времени переключения контекста в Linux было любимым времяпрепровождением для энтузиастов режима реального времени Linux. Как сделать его в Linux сопоставимым с временем переключения контекста коммерческих RTOS? Поскольку переключение контекста осуществляется планировщиком, оно влияет на продолжительность работы планировщика и, следовательно, время отклика ядра. Объектами для планирования в Linux являются:

- ***Потоки ядра***: они проводят время своей жизни только в режиме ядра. Они не имеют памяти, отображённой в пространство пользователя.
- ***Пользовательские процессы и пользовательские потоки***: потоки пользовательского пространства совместно используют общее пространство для текста, данных и "кучи". Они имеют отдельные стеки. Другие ресурсы, такие как открытые файлы и обработчики сигналов, также являются общими для всех потоков.

При принятии решений о планировании планировщик не делает различия между любыми из этих объектов. Время переключения контекста меняется, когда планировщик пытается переключить процессы вместо потоков. Переключение контекста в основном заключается в следующем:

- ***Переключение на новый набор регистров и стек ядра***: это время переключения контекста является общим для потоков и процессов.
- ***Переключение с одной виртуальной области памяти для другую***: это необходимо для переключения контекста между процессами. Оно явно или неявно приводит к перезагрузке TLB (или таблиц страниц) новыми значениями, что является дорогостоящей операцией.

Значения времени переключения контекста варьируются в зависимости от архитектуры. Измерение времени переключения контекста осуществляется с помощью программы **lmbench**. Для получения более подробной информации о LMBench™ посетите, пожалуйста, www.bitmover.com/lmbench/.

7.2.5 Пользовательское пространство и режим реального времени

До сих пор мы обсуждали различные усовершенствования, внесённые в ядро для улучшения его времени отклика. Планировщик O(1) вместе с вытеснением в ядре и патчами уменьшения задержки делают Linux операционной системой мягкого реального времени. Теперь, что относительно приложений пользовательского пространства? Разве нельзя что-то сделать, чтобы убедиться, что у них тоже есть некоторые руководящие принципы, чтобы вести себя детерминированным образом?

Для поддержки приложений реального времени IEEE вышел со стандартом POSIX.1b. Стандарт IEEE 1003.1b (или POSIX.1b) определяет интерфейсы для поддержки переносимости приложений с требованиями реального времени. Кроме 1003.1b, POSIX определяет для систем реального времени также стандарты 1003.1d, .1j, .21 и .2h, но обычно реализуются расширения, определённые в .1b. Различными расширениями реального времени, определёнными в POSIX.1b, являются:

- Планирование с фиксированным приоритетом с классами планирования в реальном времени
- Блокировка памяти
- Очереди сообщений POSIX
- Разделяемая память POSIX
- Сигналы реального времени
- Семафоры POSIX
- Часы и таймеры POSIX
- Асинхронный ввод/вывод (Asynchronous I/O, AIO)

Классы планирования реального времени, блокировка памяти, разделяемая память и сигналы реального времени получили поддержку в Linux с самых первых дней. Очереди сообщений POSIX, часы и таймеры поддерживаются в ядре версии 2.6. Асинхронный ввод/вывод также поддерживается с самых первых дней, но эта реализация была полностью сделана в библиотеке языка Си пользовательского пространства. Linux версии 2.6 имеет поддержку AIO в ядре. Отметим, что наряду с ядром, библиотека языка Си GNU и **glibc** также претерпели изменения для поддержки этих расширений реального времени. Для обеспечения лучшей поддержке в Linux POSIX.1b ядро и **glibc** работают вместе.

В этом разделе мы обсуждали поддержку в Linux режима мягкого реального времени. Мы также кратко обсудили различные расширения реального времени POSIX.1b. Вы, как разработчик приложения, ответственны за то, чтобы писать приложения таким образом, чтобы не свести к нулю выгоды от режима мягкого реального времени, предоставляемого Linux. Конечный пользователь должен понимать каждый из этих методов, чтобы могли быть написаны приложения для поддержки ядра реального времени, предоставляемого в Linux. Далее в этой главе описывается каждый из этих методов с соответствующими примерами.

7.3 Программирование в режиме реального времени в Linux

В этом разделе мы обсуждаем различные расширения реального времени POSIX 1003.1b, поддерживаемые в Linux, и их эффективное использование. Мы подробно обсудим планирование, часы и таймеры, очереди сообщений реального времени, сигналы реального времени, блокировку памяти, асинхронный ввод/вывод, разделяемую память POSIX и семафоры POSIX. Большинство расширений реального времени реализуются и распространяются в пакете **glibc**, но находятся в отдельной библиотеке **librt**. Поэтому, чтобы скомпилировать программу, которая использует функции режима реального времени POSIX.1b в Linux, эта программа должна также компоноваться с **librt**, наряду с **glibc**. В этом разделе рассматриваются различные расширения реального времени POSIX.1b, поддерживаемые в ядре Linux версии 2.6.

7.3.1 Планирование процессов

В предыдущем разделе мы обсуждали детали планировщика Linux. Теперь мы понимаем, как планировщиком управляются задачи реального времени. В этом разделе мы обсудим планировщик по отношению к ядру 2.6. Для определения задачи реального времени в Linux есть три основных параметра:

- Класс планирования
- Приоритет процесса
- Интервал времени

Они объясняются ниже.

Класс планирования

Планировщик Linux предлагает три класса планирования, два для приложений реального времени и один для приложений не реального времени. Этими тремя классами являются:

- **SCHED_FIFO**: политика планирования реального времени первый вошёл, первый вышел (First-In First-Out). Алгоритм планирования не использует никаких интервалов времени. Процесс **SCHED_FIFO** выполняется до завершения, если он не заблокирован запросом ввода/вывода, вытеснен высокоприоритетным процессом, или он добровольно отказывается от процессора. Следует обратить внимание на следующие моменты:
  – Процесс **SCHED_FIFO**, который был вытеснен другим процессом более высокого приоритета, остаётся во главе списка с его приоритетом и возобновит выполнение, как только все процессы с более высоким приоритетом будут вновь заблокированы.
  – Когда процесс **SCHED_FIFO** готов к работе (например, после пробуждения от

операции блокировки), он будет вставлен в конец списка с его приоритетом.
– Вызов **sched_setscheduler** или **sched_setparam** поставит процесс **SCHED_FIFO** в начало списка. Как следствие, это может вытеснить работающий в данный момент процесс, если его приоритет такой же, как и у работающего процесса.

- **SCHED_RR**: циклическая (Round-Robin) политика планирования реального времени. Она похожа на **SCHED_FIFO** с той лишь разницей, что процессу **SCHED_RR** разрешено работать как максимум время кванта. Если процесс **SCHED_RR** исчерпывает свой квант времени, он помещается в конец списка с его приоритетом. Процесс **SCHED_RR**, который был вытеснен процессом с более высоким приоритетом, завершит оставшуюся часть своего кванта времени после возобновления выполнения.
- **SCHED_OTHER**: стандартный планировщик Linux с разделением времени для процессов, работающих не в реальном времени.

Для установки и получения политики планирования процесса используются функции **sched_setscheduler** и **sched_getscheduler**, соответственно.

Приоритет

Диапазоны приоритетов для различных политик планирования показаны в Таблице 7.1. Функции **sched_get_priority_max** и **sched_get_priority_min** возвращают максимальный и минимальный приоритет, разрешённый для политики планирования, соответственно. Чем выше число, тем выше приоритет. Таким образом, процесс **SCHED_FIFO** или **SCHED_RR** всегда имеет более высокий приоритет, чем процесс **SCHED_OTHER**. Для процессов **SCHED_FIFO** и **SCHED_RR** функции **sched_setparam** и **sched_getparam** используются для установки и получения приоритета, соответственно. Для изменения приоритета процессов **SCHED_OTHER** используется системный вызов **nice** (или команды).

Таблица 7.1 Диапазон приоритетов в пространстве пользователя

Класс планирования	*Диапазон приоритетов*
SCHED_OTHER	0
SCHED_FIFO	1 - 99
SCHED_RR	1 - 99

Ядро позволяет значению **nice** *быть установленным как для процесса* **SCHED_RR** *или* **SCHED_FIFO**, *но это не будет иметь никакого влияния на планирование, пока задача выполняется с классом* **SCHED_OTHER**.

Точка зрения ядра на приоритеты процессов отличается от точки зрения процессов. Соответствие между приоритетами пользовательского пространства и пространства ядра для задач реального времени в ядре версии 2.6.3 показывает Рисунок 7.3.

Рисунок 7.3 Связь приоритетов задач реального времени.

Для ядра низкое значение означает высокий приоритет. Приоритеты реального времени в ядро находятся в диапазоне от 0 до 98. Ядро связывает пользовательские приоритеты **SCHED_FIFO** и **SCHED_RR** с приоритетами ядра с помощью следующих макросов:

```
#define MAX_USER_RT_PRIO 100
kernel priority = MAX_USER_RT_PRIO -1 - (user priority);
```

Таким образом, пользовательский приоритет 1 связывается с приоритетом ядра 98, приоритет 2 с 97, и так далее.

Интервал времени

Как обсуждалось ранее, интервал времени действителен только для процессов **SCHED_RR**. Процессы **SCHED_FIFO** можно рассматривать как имеющие бесконечный интервал времени. Так что это обсуждение касается только процессов **SCHED_RR**.

Linux устанавливает минимальный интервал времени для процесса как 10 мс, интервал времени по умолчанию как 100 мс, а максимальный интервал времени как 200 мс. Интервалы времени заполняются вновь после их окончания. В версии 2.6.3 интервал времени процесса рассчитывается так:

```
#define MIN_TIMESLICE (10)
#define MAX_TIMESLICE (200)
#define MAX_PRIO (139)   // MAX внутренний приоритет ядра
#define MAX_USER_PRIO 39 // MAX nice при переводе к положительной шкале

/* 'p' это структура задач процесса */
#define BASE_TIMESLICE(p) \
   (MIN_TIMESLICE + ((MAX_TIMESLICE - MIN_TIMESLICE) *
   (MAX_PRIO-1 - (p)->static_prio) / (MAX_USER_PRIO-1)))
```

static_prio содержит значение **nice** для процесса. Ядро преобразует диапазон **nice** с -20 до +19 во внутренний диапазон **nice** в ядре от 100 до 139. **nice** процесса конвертируется в такой масштаб и сохраняется в **static_prio**. Таким образом, значение **nice** -20 соответствует **static_prio** 100, а +19 для **nice**, **static_prio** 139. Наконец, интервал времени процесса возвращает функция **task_timeslice**.

```
static inline unsigned int task_timeslice(task_t *p) {
  return BASE_TIMESLICE(p);
```

```
}
```

Обратите внимание, что **static_prio** является единственной переменной в расчёте интервала времени. Таким образом, можно сделать некоторые важные выводы:

- Все процессы **SCHED_RR** выполняются по умолчанию с интервалом времени в 100 мс, поскольку они обычно имеют значение **nice**, равное 0.
- При значении **nice** -20 процесс **SCHED_RR** получит интервал времени 200 мс, а при **nice** +19 процесс **SCHED_RR** получит интервал времени 10 мс. Таким образом, значение **nice** может быть использовано для управления выделением интервала времени для процессов **SCHED_RR**.
- Чем меньше значение **nice** (то есть, приоритет более высокий), тем больше интервал времени.

Функции планирования

Функции планирования, предоставляемые для поддержки приложений реального времени в Linux, перечислены в Таблице 7.2.

Таблица 7.2 Функции планирования POSIX.1b

Метод	*Описание*
sched_getscheduler	Получение класса планирования процесса.
sched_setscheduler	Установка класса планирования процесса.
sched_getparam	Получение приоритета процесса.
sched_setparam	Установка приоритета планирования.
sched_get_priority_max	Получение максимального разрешённого значения приоритета для класса планирования.
sched_get_priority_min	Получение минимального разрешённого значения приоритета для класса планирования.
sched_rr_get_interval	Получение текущего временного интервала для процесса **SCHED_RR**.
sched_yield	Передача выполнения другому процессу.

Функции **sched_setscheduler** *и* **sched_setparam** *следует вызывать с привилегиями суперпользователя.*

Использование этих функций иллюстрирует Распечатка 7.1 189. Данный пример создаёт процесс **SCHED_FIFO** с приоритетом, который имеет среднее значение между минимальным и максимальным приоритетом для класса планирования **SCHED_FIFO**. Он также динамически изменяет приоритет процесса **SCHED_FIFO**. Как значение **nice** может быть использовано для управления выделением интервала времени для класса **SCHED_RR** показывает Распечатка 7.2 190.

Влияние **nice** *на выделении интервала времени* **SCHED_RR** *не предусмотрено POSIX. Это делает возможным реализация планировщика в Linux. Вы не должны использовать эту функцию в переносимых программах. Такое влияние* **nice** *на* **SCHED_RR** *происходит от ядра версии 2.6.3 и может измениться в будущем.*

Распечатка 7.1 Операции планирования процесса

Распечатка 7.1.

```
/* sched.c */

#include <sched.h>
int main(){
  struct sched_param param, new_param;

  /*
   * Процесс запускается с политикой по умолчанию SCHED_OTHER,
   * если не порождён процессом SCHED_RR или SCHED_FIFO.
   */

  printf("start policy = %d\n", sched_getscheduler(0));
  /*
   * выводит -> start policy = 0 .
   * (Для политик SCHED_FIFO или SCHED_RR, sched_getscheduler
   * возвращает 1 и 2 соответственно)
   */

  /*
   * Создаём процесс SCHED_FIFO, работающий со средним приоритетом
   */
  param.sched_priority = (sched_get_priority_min(SCHED_FIFO) +
                          sched_get_priority_max(SCHED_FIFO))/2;
  printf("max priority = %d, min priority = %d,
          my priority = %d\n",sched_get_priority_max(SCHED_FIFO),
                              sched_get_priority_min(SCHED_FIFO),
                              param.sched_priority);
  /*
   * выводит -> max priority = 99, min priority = 1,
   * my priority = 50
   */

  /* Делаем процесс SCHED_FIFO */
  if (sched_setscheduler(0, SCHED_FIFO, &param) != 0){
    perror("sched_setscheduler failed\n");
    return;
  }

  /*
   * выполнение критичных ко времени операций
   */

  /*
   * Даём шанс поработать какому-либо другому потоку/процессу
```

```
   * реального времени. Обратите внимание, что вызов sched_yield
   * поместит текущий процесс в конец очереди с его приоритетом.
   * Если в этой очереди нет другого процесса, этот вызов
   * не будет иметь эффекта
   */
  sched_yield();

  /* Вы можете также изменять приоритет во время работы */
  param.sched_priority = sched_get_priority_max(SCHED_FIFO);
  if (sched_setparam(0, &param) != 0){
    perror("sched_setparam failed\n");
    return;
  }
  sched_getparam(0, &new_param);
  printf("I am running at priority %d\n",
                 new_param.sched_priority);
  /* выводит -> I am running at priority 99 */

  return ;
}
```

Распечатка 7.2 Управление интервалами времени процесса SCHED_RR

Распечатка 7.2.

```
/* sched_rr.c */

#include <sched.h>
int main(){
  struct sched_param param;
  struct timespec ts;
  param.sched_priority = sched_get_priority_max(SCHED_RR);

  /* Необходим максимальный интервал времени */
  nice(-20);
  sched_setscheduler(0, SCHED_RR, &param);
  sched_rr_get_interval(0, &ts);
  printf ("max timeslice = %d msec\n", ts.tv_nsec/1000000);
  /* выводит -> max timeslice = 199 msec */

  /* Необходим минимальный интервал времени. Обратите также
   * внимание, что аргументом для nice является "приращение",
   * а не абсолютное значение. Таким образом, выполнение
   * nice(39) приводит к работе с приоритетом nice +19
   */
  nice(39);
  sched_setscheduler(0, SCHED_RR, &param);
  sched_rr_get_interval(0, &ts);
  printf ("min timeslice = %d", ts.tv_nsec/1000000);
  /* output -> min timeslice = 9 msec */

  return ;
}
```

7.3.2 Блокировка памяти

Одна из задержек, с которой должны иметь дело приложения реального времени, это замещение страниц (подкачка) по требованию. Приложение реального времени требует детерминированного времени отклика и подкачка является одной из основных причин неожиданных задержек при исполнении программы. Задержки из-за подкачки можно было бы избежать с помощью блокировки памяти. Предоставляются функции либо для блокировки адресного пространства программы целиком, либо для выбранной области памяти.

Функции блокировки памяти

Функции блокировки памяти приведены в Таблице 7.3. **mlock** запрещает замещение страниц для указанного диапазона памяти, а **mlockall** запрещает замещение для всех страниц, которые связаны с адресным пространством процесса. Это включает в себя страницы кода, данных, стека, разделяемых библиотек, общей памяти и отображённых на память файлов. Использование этих функций иллюстрирует Распечатка 7.3 194. Эти функции должны вызываться с привилегией суперпользователя.

Таблица 7.3 Функции блокировки памяти POSIX.1b

Метод	*Описание*
mlock	Блокировка указанной области адресного пространства процесса
mlockall	Блокировка всего адресного пространства процесса
munlock	Разблокировка области, заблокированной с помощью **mlock**
munlockall	Разблокировка всего адресного пространства процесса

Приложение с требованиями реального времени как правило многопоточное, некоторые потоки которого работают режиме реального времени, а некоторые не в режиме реального времени. Для таких приложений **mlockall** использоваться не должна, поскольку это также блокирует память потоков, не работающих в режиме реального времени. В следующих двух разделах мы обсудим два подхода к компоновке для выполнения селективной блокировки памяти в таких приложениях.

Эффективная блокировка с помощью сценария компоновщика

Идея заключается в размещении объектных файлов, содержащих код и данные реального времени, в отдельном разделе компоновщика с помощью сценария компоновщика. Блокирование с помощью **mlock** такого раздела во время запуска программы привело бы к блокировка только кода и данных, использующихся в режиме реального времени. Для иллюстрации этого возьмём пример приложения. В Распечатке 7.4 195 мы предполагаем, что **hello_rt_world** - это функция реального времени, которая работает с **rt_data** в **rt_bss** как неинициализированными данными.

Для достижения селективного блокирования должны быть выполнены следующие шаги:

1. Разделить приложение на файловом уровне на файлы, содержащие функции реального времени и не содержащие функции реального времени. В файлы реального времени не включать функции, не работающие в режиме реального времени, и наоборот. В этом примере мы имеем:
   a. **hello_world.c**: содержит функции, не работающие в режиме реального времени
   b. **hello_rt_world.c**: содержит функции, работающие в режиме реального времени
   c. **hello_rt_data.c**: содержит данные реального времени
   d. **hello_rt_bss.c**: содержит bss реального времени
   e. **hello_main.c**: целевое приложение
2. Сгенерировать объектный код, но не делать связывание.

```
# gcc -c hello_world.c hello_rt_world.c hello_rt_data.c \
      hello_rt_bss.c hello_main.c
```

3. Получить сценарий компоновщика по умолчанию и сделать копию.

```
# ld -verbose > default
# cp default rt_script
```

4. Отредактировать **rt_script** и удалить команды компоновщика. (Удаляем всё перед командой **OUTPUT_FORMAT**, а также **==== ..** в конце файла.)
5. Найти в **rt_script** разделы **.text**, **.data** и **.bss** и добавить перед ними записи **rt_text**, **rt_data** и **rt_bss**, соответственно, как показано в Распечатке 7.5 197. Таким образом, все функции, определённые в **hello_rt_world.c**, войдут в раздел **rt_text**. Данные, определённые в **hello_rt_data.c**, войдут в раздел **rt_data**, а все неинициализированные данные в **hello_rt_bss.c** войдут в раздел **rt_bss**. Переменные **__start_rt_text**, **__start_rt_data** и **__start_rt_bss** отмечают начало разделов **rt_text**, **rt_data** и **rt_bss**, соответственно. Аналогично, **__end_rt_text**, **__end_rt_data** и **__end_rt_bss** отмечают адрес конца соответствующих разделов.
6. Наконец, компонуем приложение.

```
# gcc -o hello hello_main.o hello_rt_bss.o \
     hello_rt_data.o hello_rt_world.o hello_world.o \
     -T rt script
```

Вы можете проверить, что все функции и данные реального времени находятся в соответствующих разделах с помощью команды **objdump**, как показано ниже.

```
# objdump -t hello
  .....
08049720   g     .rt_bss   00000000   __start_rt_bss
08049760   g  O  .rt_bss   00000064   rt_bss
080497c4   g     .rt_bss   00000000   __end_rt_bss.....
080482f4   g     .rt_text  00000000   __start_rt_text
080482f4   g  F  .rt_text  0000001d   hello_rt_world
0804834a   g     .rt_text  00000000   __end_rt_text
                     ......
080496c0   g     .rt_data  00000000   __start_rt_data
080496c0   g  O  .rt_data  00000011   rt_data
08049707   g     .rt_data  00000000   __end_rt_data
```

Эффективная блокировка с помощью раздела атрибутов GCC

Этот подход может быть использован, если трудно поместить код режима реального времени и код, не работающий в реальном времени, в отдельные файлы. При таком подходе, чтобы разместить наш код и данные реального времени в соответствующих разделах, мы используем раздел атрибутов GCC. Наконец, наша цель достигается блокировкой только этих разделов. Этот подход является очень гибким и простым в использовании. Распечатка 7.6 показывает Распечатку 7.4 195, переписанную, чтобы попасть в эту категорию.

Вы можете проверить, что все функции и данные реального времени находятся в соответствующих разделах с помощью команды **objdump**, как показано ниже.

```
# gcc -o hello hello.c
# objdump -t hello
   .....
   .....
08049724 g     *ABS*      00000000   __stop_real_bss
08048560 g     *ABS*      00000000   __stop_real_text
080496a0 g     *ABS*      00000000   __start_real_data
080496b1 g     *ABS*      00000000   __stop_real_data
080496c0 g     *ABS*      00000000   __start_real_bss
0804852c g     *ABS*      00000000   __start_real_text
080496c0 g  O  real_bss   00000064   rt_bss
080496a0 g  O  real_data  00000011   rt_data
0804852c g  F  real_text  00000034   hello_rt_world
   ......
   ......
```

Обратите внимание на определённые компоновщиком символы **__start_real_text**, **__stop_real_text**, и так далее.

Что следует помнить

- Один вызов **munlock** или **munlockall** разблокирует область памяти, даже если она заблокирована процессом несколько раз.
- Страницы, отображённые в несколько областей памяти или используемые несколькими процессами, будут заблокированы в памяти до тех пор, пока они заблокированы по крайней мере одним процессом или в одной области памяти.
- Дочерние процессы не наследуют блокировки страниц через ветвление.
- Страницы, заблокированные **mlock** или **mlockall**, гарантированно остаются в памяти, пока страницы не разблокированы **munlock** или **munlockall**, не отключено их отображение через **munmap**, или пока процесс не завершится или не запустит другую программу с помощью **exec**.
- Лучше делать блокировку памяти при инициализации программы. Все динамические выделения памяти, создание совместно используемой памяти и отображение файлов должно быть сделано при инициализации с последующей блокировкой их **mlock**.
- В случае, если вы хотите быть уверенными, что выделения памяти для стека остаются детерминированными, вам также необходимо заблокировать некоторые страницы стека. Чтобы избежать замещения для сегмента стека, вы можете написать небольшую функцию **lock_stack** и вызвать её во время инициализации.

```
void lock_stack(void){
  char dummy[MAX_APPROX_STACK_SIZE];
  /* Это делает блокировку страниц стека */
  memset(dummy, 0, MAX_APPROX_STACK_SIZE);
  mlock(dummy, MAX_APPROX_STACK_SIZE);
  return;
}
```

MAX_APPROX_STACK_SIZE является оценкой размера стека, используемого вашим потоком реального времени. Как только это сделано, ядро гарантирует, что это пространство для стека всегда находится в памяти.

- Будьте великодушны к другим процессам, работающим в вашей системе. Агрессивная блокировка может забрать ресурсы других процессов.

Распечатка 7.3 Операции блокировки памяти

Распечатка 7.3.

```
/* mlock.c */

#include <sys/mman.h>
#include <unistd.h>

#define RT_BUFSIZE 1024
int main(){

  /* rt_buffer должен быть заблокирован в памяти */
  char *rt_buffer = (char *)malloc(RT_BUFSIZE);
  unsigned long pagesize, offset;

  /*
   * В Linux нет необходимости выравнивать адрес страницы перед
   * mlocking, это делает ядро. Но POSIX требует выровнять
   * адрес памяти по границе страницы перед вызовом mlock для
   * улучшения переносимости. Так что выравниваем rt_buffer
   * по границы страницы.
   */
  pagesize = sysconf(_SC_PAGESIZE);
  offset = (unsigned long) rt_buffer % pagesize;
  /* Блокируем rt_buffer в памяти */
  if (mlock(rt_buffer - offset, RT_BUFSIZE + offset) != 0){
    perror("cannot mlock");
    return 0;
  }

  /*
   * После успешного выполнения mlock страница, которая содержит
   * rt_buffer, находится в памяти и заблокирована. Она никогд
   * не будет замещена. Так что rt_buffer может использоваться
   * без беспокойства о задержке вследствие замещения страниц.
   */
```

```
  /* Разблокируем rt_buffer после использования */
  if (munlock(rt_buffer - offset, RT_BUFSIZE + offset) != 0){
    perror("cannot mulock");
    return 0;
  }

  /*
   * В зависимости от приложения мы можем решить заблокировать
   * в памяти всё адресное пространство процесса.
   */

  /* Блокируем текущую память процесса, а также все будущие
   * выделения памяти.
   * MCL_CURRENT - Блокировать все страницы, которые в настоящее
   * время включены в адресное пространство процесса
   * MCL_FUTURE - Блокировать также все страницы, которые будут
   * включены в адресное пространство процесса в будущем.
   */
  if (mlockall(MCL_CURRENT | MCL_FUTURE) != 0){
    perror("cannot mlockall");
    return 0;
  }

  /*
   * Если mlockall был успешным, все новые выделения памяти
   * будут блокироваться. Так что страницы, содержащие
   * rt_buffer, будут заблокированы в памяти.
   */
  rt_buffer = (char *)realloc(rt_buffer , 2*RT_BUFSIZE);

  /*
   * В конце разблокируем всю память, которая была заблокирована
   * либо mlock, либо mlockall, вызовом функции munlockall.
   */
  if (munlockall() != 0){
    perror("cannot munlock");
    return 0;
  }
  return 0;
}
```

Распечатка 7.4 Эффективная блокировка — 1

Распечатка 7.4.

```
/* hello_world.c */

#include <stdio.h>
/* Функция, не работающая в реальном времени */
void hello_world(void) {
  printf("hello world");
  return;
}
```

```
/* hello_rt_world.c */

#include <stdio.h>
/* Это функция реального времени */
void hello_rt_world(void){
  extern char rt_data[],rt_bss[];
  /* работаем с rt_data */
  printf("%s", rt_data);
  /* работаем с rt_bss */
  memset(rt_bss, 0xff, sizeof(rt_bss));
  return ;
}


/* hello_rt_data.c */

/* Данные реального времени */
char rt_data[] = "Hello Real-time World";


/* hello_rt_bss.c */

/* bss реального времени */
char rt_bss[100];


/* hello_main.c */

#include <stdio.h>
extern void hello_world(void);
extern void hello_rt_world(void);

/*
 * Мы определяем эти символы в скрипте компоновщика.
 * Это станет понятно дальше.
 */
extern unsigned long __start_rt_text, __end_rt_text;
extern unsigned long __start_rt_data, __end_rt_data;
extern unsigned long __start_rt_bss, __end_rt_bss;

/*
 * Эта функция блокирует в памяти все функции и данные
 * реального времени.
 */
void rt_lockall(void){
  /* блокировка сегмента текста реального времени */
  mlock(&__start_rt_text, &__end_rt_text - &__start_rt_text);
  /* блокировка данных реального времени */
  mlock(&__start_rt_data, &__end_rt_data - &__start_rt_data);
  /* блокировка bss реального времени */
  mlock(&__start_rt_bss, &__end_rt_bss - &__start_rt_bss);
}
```

```
int main(){
  /* Первым шагом является выполнение блокировки памяти */
  rt_lockall();
  hello_world();
  /* Это наша функция реального времени */
  hello_rt_world();
  return 0;
}
```

Распечатка 7.5 Модифицированный сценарий компоновщика

Распечатка 7.5.

```
   .....
   .....
.plt      : { *(.plt) }
.rt_text  :
{
  PROVIDE (__start_rt_text = .);
  hello_rt_world.o
  PROVIDE (__end_rt_text = .);
} =0x90909090
.text     :
   ....
   ....
.got.plt  : { . = DATA_SEGMENT_RELRO_END (. + 12); *(.got.plt) }
.rt_data  :
{
  PROVIDE (__start_rt_data = .);
  hello_rt_data.o
  PROVIDE (__end_rt_data = .);
}
.data     :
   ....
   ....
__bss_start = .;
.rt_bss   :
{
  PROVIDE (__start_rt_bss = .);
  hello_rt_bss.o
  . = ALIGN(32 / 8);
  PROVIDE (__end_rt_bss = .);
}
.bss     :
   ......
   ......
```

Распечатка 7.6 Эффективная блокировка — 2

Распечатка 7.6.

```
/* hello.c */
```

```
#include <stdio.h>

/*
 * Определяем макросы для использования раздела атрибутов GCC.
 * Для хранения нашего кода, данных и bss реального времени мы
 * определяем 3 раздела, real_text, read_data и real_bss
 */
#define __rt_text __attribute__ ((__section__ ("real_text")))
#define __rt_data __attribute__ ((__section__ ("real_data")))
#define __rt_bss __attribute__ ((__section__ ("real_bss")))

/*
 * Компоновщик очень легко поддаётся доработке. Он обычно
 * определяет символы, содержащие начальные и конечные адреса
 * разделов. Компоновщик определяет следующие символы
 */
extern unsigned long __start_real_text, __stop_real_text;
extern unsigned long __start_real_data, __stop_real_data;
extern unsigned long __start_real_bss, __stop_real_bss;

/* Инициализированные данные для раздела real_bss */
char rt_bss[100] __rt_bss;

/* Неинициализированные данные для раздела real_data */
char rt_data[] __rt_data = "Hello Real-time World";

/* Функция, которая пойдёт в раздел real_text */
void __rt_text hello_rt_world(void){
  printf("%s", rt_data);
  memset(rt_bss, 0xff, sizeof(rt_bss));
  return ;
}

/* Наконец, блокируем в памяти наши разделы
 * 'реального времени'
 */
void rt_lockall(void){
  mlock(&__start_real_text,
        &__stop_real_text - &__start_real_text);
  mlock(&__start_real_data,
        &__stop_real_data - &__start_real_data);
  mlock(&__start_real_bss,
        &__stop_real_bss - &__start_real_bss);
}

/* Функция, не работающая в реальном времени */
void hello_world(void) {
  printf("hello world");
  return;
}

int main(){
  rt_lockall();
  hello_world();
```

```
  hello_rt_world();
  return 0;
}
```

7.3.3 Совместно используемая память POSIX

Приложения реального времени часто требуют быстрых, с высокой пропускной способностью механизмов межпроцессного взаимодействия (interprocess communication, IPC). В этом разделе мы обсудим разделяемую память POSIX, которая является самым быстрым и самым простым механизмом IPC. Общая память является самым быстрым механизмом IPC по двум причинам:

- Отсутствуют накладные расходы системного вызова при чтении или записи данных.
- Данные копируются прямо в область общей памяти. Без участия буферов ядра или других промежуточных буферов.

Функции, используемые для создания и удаления общей памяти, приведены в Таблице 7.4.

Таблица 7.4 Функции для работы с общей памятью POSIX.1b

Метод	*Описание*
shm_open	Открывает объект в общей памяти
shm_unlink	Удаляет объект из общей памяти

shm_open создаёт новый POSIX объект в разделяемой памяти или открывает существующий. Функция возвращает дескриптор, который может быть использован другими функциями, такими как **ftruncate** и **mmap**. **shm_open** создаёт сегмент разделяемой памяти размером 0. **ftruncate** устанавливает желаемый размер сегмента общей памяти, а **mmap** затем связывает этот сегмент с адресным пространством процесса. Сегмент разделяемой памяти удаляется с помощью **shm_unlink**. Их использование иллюстрирует Распечатка 7.7 200.

Реализация в Linux

Поддержка общей памяти POSIX в Linux использует файловую систему **tmpfs**, смонтированную в **/dev/shm**.

```
# cat /etc/fstab
none    /dev/shm    tmpfs    defaults    0 0
```

Объект разделяемой памяти, созданный с помощью **shm_open**, представлен в **tmpfs** в виде файла. Удалите в Распечатке 7.7 200 вызов **shm_unlink** и запустите программу снова. Вы должны увидеть в **/dev/shm** файл **my_shm**.

```
# ls -l /dev/shm
-rw-r--r--  1 root root 1024 Aug 19 18:57 my_shm
```

Это показывает файл **my_shm** размером 1024 байт, который является размером нашей общей памяти. Таким образом, мы можем использовать для работы с общей памятью все файловые операции. Например, мы можем получить содержимое общей памяти, выполнив для этого файла команду **cat**. Также мы можем использовать для удаления общей памяти команду **rm** прямо из оболочки.

Что следует помнить

- Помните о блокировании области разделяемой памяти с помощью **mlock**.
- Для синхронизации доступа к совместно используемой области памяти используйте семафоры POSIX.
- Размер области общей памяти может быть получен с помощью функции **fstat**.
- Если несколько процессов открывают один и тот же регион разделяемой памяти, этот регион будет удалён только после заключительного вызова **shm_unlink**.
- Не вызывайте **shm_unlink**, если вы хотите сохранить общую область памяти даже после завершения процесса.

Распечатка 7.7 Операции с общей памятью POSIX

Распечатка 7.7.

```
/* shm.c */

#include <sys/types.h>
#include <sys/mman.h>
#include <fcntl.h>

/* Размер сегмента нашей общей памяти */
#define SHM_SIZE 1024

int main(){
  int shm_fd;
  void *vaddr;

  /* Получаем дескриптор общей памяти */
  if ((shm_fd = shm_open("my_shm", O_CREAT | O_RDWR, 0666)) ==
                                                        -1){
    perror("cannot open");
    return -1;
  }

  /* Устанавливаем размер общей памяти равным SHM_SIZE */
  if (ftruncate(shm_fd, SHM_SIZE) != 0){
    perror("cannot set size");
    return -1;
  }

  /*
   * Подключаем общую память в адресное пространство. Флаг
   * MAP_SHARED говорит, что это подключение общей памяти.
   */
```

```
  if ((vaddr = mmap(0, SHM_SIZE, PROT_WRITE, MAP_SHARED,
                shm_fd, 0)) == MAP_FAILED){
    perror("cannot mmap");
    return -1;
  }

  /* Блокируем общую память. Не забываем про этот шаг */
  if (mlock(vaddr, SHM_SIZE) != 0){
    perror("cannot mlock");
    return -1;
  }

  /*
   * Общая память готова к использованию
   */

  /*
   * В конце отделяем сегмент общей памяти от адресного
   * пространства. Это также приведёт к разблокировке этого
   * сегмента.
   */
  munmap(vaddr, SHM_SIZE);
  close(shm_fd);
  /* Удаляем сегмент общей памяти */
  shm_unlink("my_shm");
  return 0;
}
```

7.3.4 Очереди сообщений POSIX

Очередь сообщений POSIX 1003.1b обеспечивает детерминированные и эффективные средства IPC. Она предлагает следующие преимущества для приложений реального времени:

- Буферы сообщений в очереди сообщений являются предварительно выделенными, обеспечивая доступность ресурсов, когда они необходимы.
- Сообщениям могут быть назначены приоритеты. Высокоприоритетные сообщения всегда принимаются первыми, независимо от количества сообщений в очереди.
- Она предлагает асинхронное уведомление при поступлении сообщения, если приёмник не хочет ожидать получения сообщения.
- Функции отправки и получения сообщений по умолчанию являются блокирующими вызовами. Приложения могут указать время ожидания при отправке или получении сообщений, чтобы избежать недетерминированной блокировки.

Интерфейсы перечислены в Таблице 7.5. Использование некоторых основных функций очереди сообщений иллюстрирует Распечатка 7.8 205. В этом примере создаются два процесса: один отправляет сообщение в очередь сообщений, а другой получает сообщение из очереди.

Таблица 7.5 Функции для работы с очередью сообщений POSIX.1b

Метод	*Описание*
mq_open	Открытие/создание очереди сообщений.
mq_close	Закрытие очереди сообщений.
mq_getattr	Получение атрибутов очереди сообщений.
mq_setattr	Установка атрибутов очереди сообщений.
mq_send	Отправка сообщения в очередь.
mq_receive	Приём сообщения из очереди.
mq_timedsend	Отправка сообщения в очередь. Блокируется в течение заданного времени.
mq_timedreceive	Приём сообщения из очереди. Блокируется в течение заданного времени.
mq_notify	Регистрация для получения уведомления всякий раз, когда получено сообщение в пустой очереди сообщений.
mq_unlink	Удаление очереди сообщений.

Компиляция и запуск приведённых выше двух программ даёт следующий результат:

```
# gcc -o mqueue-1 mqueue-1.c -lrt
# gcc -o mqueue-2 mqueue-2.c -lrt
# ./mqueue-1
# ./mqueue-2
O_NONBLOCK not set
Message: Hello Posix World, prio = 1
```

Временем блокировки приложения для отправки или получения сообщений можно управлять с помощью функций **mq_timedsend** и **mq_timedreceive**. Если очередь сообщений заполнена и флаг **O_NONBLOCK** не установлен, функция **mq_timedsend** завершается за указанное время (это может произойти, если очередь заполнена и функция отправки блокируется, пока не получит свободный буфер). Точно так же **mq_timedreceive** завершается за указанное время, если в очереди нет сообщений. Использование функций **mq_timedsend** и **mq_timedreceive** иллюстрирует следующий фрагмент кода. Обе ожидают не более 10-ти секунд для отправки или получения сообщений.

```
/* Отправка сообщения */
struct timespec ts;

/* С этого момента указывает время ожидания как 10 с. */
ts.tv_sec = time(NULL) + 10;
ts.tv_nsec = 0;
if (mq_timedsend(ds, text, SIZE,PRIOTITY, &ts) == -1){
  if (errno == ETIMEDOUT){
    printf("Timeout when waiting for message.");
    return 0;
  }
  return -1;
}
```

```
/* Приём сообщения */
if (mq_timedreceive(ds, new_text, SIZE, &prio, &ts) == -1){
  if (errno == ETIMEDOUT){
    printf("Timeout when waiting for message.");
    return 0;
  }
  return -1;
}
```

Асинхронные уведомления

Асинхронный механизм для процессов для получения уведомления, что в очереди сообщений есть сообщения, вместо синхронной блокировки в **mq_receive** или **mq_timedreceive**, обеспечивает функция **mq_notify**. Этот интерфейс очень удобен для приложений реального времени. Процесс может вызвать функцию **mq_notify**, чтобы зарегистрироваться для асинхронных уведомлений, а затем может выполнять другую работу. Когда приходит сообщение, в очередь процессу направляется уведомление. После уведомления, для получения сообщения процесс может вызвать **mq_receive**. Прототип **mq_notify**:

```
int mq_notify(mqd_t mqdes,
              const struct sigevent *notification);
```

Приложение может зарегистрироваться на два типа уведомлений:

- **SIGEV_SIGNAL**: отправить процессу сигнал, указанный в **notification->sigev_signo**, когда в очередь приходит сообщение. Использование иллюстрирует Распечатка 7.9 207.
- **SIGEV_THREAD**: вызвать при поступлении сообщения в очередь в отдельном потоке **notification->sigev_notify_function**. Использование иллюстрирует Распечатка 7.10 208.

Реализация в Linux

Подобно реализации в Linux совместно используемой памяти POSIX, Linux реализует очереди сообщений POSIX как файловую систему **mqueue**. Файловая система **mqueue** обеспечивает необходимую поддержку ядра для библиотеки пользовательского пространства, которая реализует интерфейсы очереди сообщений POSIX. По умолчанию ядро монтирует файловую систему внутренне и её не видно в пространстве пользователя. Тем не менее, вы можете смонтировать файловую систему **mqueue**.

```
# mkdir /dev/mqueue
# mount -t mqueue none /dev/mqueue
```

Эта команда монтирует файловую систему **mqueue** в **/dev/mqueue**. Очередь сообщений представлена в виде файла в **/dev/mqueue**. Но вы не можете отправить или получить сообщение из очереди "записывая" или "читая" из "файла" очереди сообщений. Чтение файла даёт размер очереди и уведомительную информацию, которая не доступна через стандартные процедуры. Удалите из Распечатки 7.8 205 **mq_unlink**, а затем скомпилируйте и запустите.

```
# gcc mqueue-1.c -lrt
# ./a.out
# cat /dev/mqueue/my_queue
QSIZE:17        NOTIFY:0    SIGNO:0    NOTIFY_PID:0
```

В выводе, показанном выше:

- **QSIZE**: размер очереди сообщений
- **NOTIFY**: либо 0, либо **SIGEV_SIGNAL** или **SIGEV_THREAD**
- **SIGNAL**: номер сигнала, используемого для уведомления
- **NOTIFY_PID**: PID процесса, ожидающего уведомление

Для регулировки количества ресурсов, используемых файловой системой, файловая система **mqueue** также предоставляет управление параметрами (sysctls) в каталоге **/proc/sys/fs/mqueue**. Этими параметрами являются:

- **queues_max**: чтение/запись этого файла позволяет получить/установить максимальное количество очередей сообщений, разрешённых в системе. Например, **echo 128 > queues_max** позволяет создание максимум 128 очередей сообщений в системе.
- **msg_max**: чтение/запись этого файла позволяет получить/установить максимальное количество сообщений в очереди.Максимальное количество сообщений, указанное при вызове **mq_open**, должно быть меньше или равно **msg_max**.
- **msgsize_max**: чтение/запись файла позволяет получить/установить максимальное значение размера сообщения. Оно является значением по умолчанию, если во время вызова **mq_open** максимальный размер сообщения не задан.

Что следует помнить

- Когда процесс получает из очереди сообщение, это сообщение удаляется из очереди.
- Если флаг **O_NONBLOCK** не указан, **mq_send** блокируется, пока в очереди не появится свободное место для добавления сообщения в очередь. Если, чтобы отправить сообщение, ждёт более чем один поток или процесс, и в очереди появляется свободное место, то для отправки сообщения разблокируется поток/процесс самого высокого приоритета, который ждал дольше всех. То же самое касается **mq_receive**.
- В любой момент времени только один процесс может быть зарегистрирован для получения уведомления от очереди сообщений. Если вызывающий процесс или любой другой процесс уже зарегистрирован для уведомления о прибытии сообщения, последующие попытки зарегистрироваться в очереди сообщений тем же или другим процессом окончатся неудачей.
- Для отмены существующей регистрации вызовите **mq_notify** с параметром "уведомление", равным NULL.
- После отправки уведомления зарегистрированному процессу его регистрация удаляется и очередь сообщений доступна для дальнейшей регистрации.
- Если какой-нибудь поток процесса заблокирован в **mq_receive** и этот процесс также зарегистрировал уведомление, прибывающее сообщение поступает в **mq_receive** и уведомление не посылается.

Распечатка 7.8 Операции с очередью сообщений POSIX

Распечатка 7.8.

```
/* mqueue-1.c */

/* Эта программа посылает сообщение в очередь */
#include <stdio.h>
#include <string.h>
#include <mqueue.h>

#define QUEUE_NAME "/my_queue"
#define PRIORITY   1
#define SIZE       256

int main(){

  mqd_t ds;
  char text[] = "Hello Posix World";
  struct mq_attr queue_attr;

  /*
   * Атрибуты нашей очереди. Они могут быть установлены только
   * при создании.
   */
  queue_attr.mq_maxmsg = 32; /* максимальное число сообщений
                                в очереди в один момент времени */
  queue_attr.mq_msgsize = SIZE; /* максимальный размер очереди */

  /*
   * Создаём новую очередь с именем "/my_queue" и открываем её
   * для отправки и приёма. Разрешения для файла очереди
   * устанавливаем как rw для владельца и не разрешаем ничего
   * для группы/других. Ограничения очереди заданы указанными
   * выше величинами.
   */
  if ((ds = mq_open(QUEUE_NAME, O_CREAT | O_RDWR , 0600,
                    &queue_attr)) == (mqd_t)-1){
    perror("Creating queue error");
    return -1;
  }

  /*
   * Посылаем сообщение в очередь с приоритетом 1. Чем больше,
   * число, тем выше приоритет. Сообщение с высоким приоритетом
   * вставляется перед сообщением с низким приоритетом. Для
   * сообщений с одинаковым приоритетом работает принцип
   * первый вошёл, первый вышел.
   */
  if (mq_send(ds, text, strlen(text), PRIORITY) == -1){
    perror("Sending message error");
    return -1;
  }
  /* Закрываем очередь... */
```

```
  if (mq_close(ds) == -1)
    perror("Closing queue error");
  return 0;
}
```

```
/* mqueue-2.c */

/* Эта программа принимает сообщение из очереди */
#include <stdio.h>
#include <mqueue.h>
#define QUEUE_NAME "/my_queue"
#define PRIORITY   1
#define SIZE       256

int main(){

  mqd_t ds;
  char new_text[SIZE];
  struct mq_attr attr, old_attr;
  int prio;

  /*
   * Открываем "/my_queue" для отправки и приёма. При приёме
   * сообщения не блокируемся (O_NONBLOCK). Разрешения для файла
   * очереди устанавливаем как rw для владельца и не разрешаем
   * ничего для группы/других.
   */
  if ((ds = mq_open(QUEUE_NAME, O_RDWR | O_NONBLOCK, 0600,
                    NULL)) == (mqd_t)-1){
    perror("Creating queue error");
    return -1;
  }

  /*
   * Меняем приём на блокирующий. (Это сделано, чтобы
   * продемонстрировать использование функций mq_setattr и
   * mq_getattr. Чтобы перевести очередь в блокирующим режим,
   * вы также можете сделать показанный выше вызов mq_open
   * без O_NONBLOCK). Помните, что mq_setattr не может
   * использоваться для изменения значений параметров очереди
   * сообщений mq_maxmsg, mq_msgsize и других. Она может
   * использоваться только для изменения поля mq_flags в
   * структуре mq_attr. mq_flags один из O_NONBLOCK, O_RDWR и т.д.
   */
  attr.mq_flags = 0; /* set !O_NONBLOCK */
  if (mq_setattr(ds, &attr, NULL)){
    perror("mq_setattr");
    return -1;
  }

  /*
   * Теперь убедимся, O_NONBLOCK не установлен. Фактически,
   * эта функция также заполняет параметры очереди сообщений в
   * структуре old_addr.
```

```
   */
  if (mq_getattr(ds, &old_attr)) {
    perror("mq_getattr");
    return -1;
  }
  if (!(old_attr.mq_flags & O_NONBLOCK))
    printf("O_NONBLOCK not set\n");

  /*
   * Теперь принимаем сообщение из очереди. Это блокирующий вызов.
   * Приоритет принятого сообщения сохраняется в prio. Функция
   * принимает самое старое из сообщений с наивысшим приоритетом
   * из очереди сообщений. Если размер буфера, указанный аргументом
   * msg_len, меньше, чем атрибут mq_msgsize очереди сообщений,
   * вызов функции не будет успешным и вернёт ошибку.
   */
  if (mq_receive(ds, new_text, SIZE, &prio) == -1){
    perror("cannot receive");
    return -1;
  }

  printf("Message: %s, prio = %d\n", new_text, prio);

  /* Закрываем очередь... */
  if (mq_close(ds) == -1)
    perror("Closing queue error");

  /*
   * ...и наконец отсоединяем её. После отсоединения
   * очередь сообщений удаляется из системы.
   */
  if (mq_unlink(QUEUE_NAME) == -1)
    perror("Removing queue error");
  return 0;
}
```

Распечатка 7.9 Асинхронное уведомление с помощью SIGEV_SIGNAL

Распечатка 7.9.

```
struct sigevent notif;
sigset_t sig_set;
siginfo_t info;
      ....

/* Сигналом уведомления является SIGUSR1. */
sigemptyset(&sig_set);
sigaddset(&sig_set, SIGUSR1);

/*
 * Блокируем SIGUSR1, так как мы будем ждать его
 * в вызове sigwaitinfo
 */
```

```
sigprocmask(SIG_BLOCK, &sig_set, NULL);

/* Теперь настраиваем уведомление */
notif.sigev_notify = SIGEV_SIGNAL;
notif.sigev_signo = SIGUSR1;

if (mq_notify(ds, &notif)){
  perror("mq_notify");
  return -1;
}

/*
 * Если в очередь поступит сообщение, будет
 * доставлен SIGUSR1
 */
do {
  sigwaitinfo(&sig_set, &info);
} while(info.si_signo != SIGUSR1);

/* Теперь можно принять сообщение. */
if (mq_receive(ds, new_text, SIZE, &prio) == -1)
  perror("Receiving message error");

      ....
```

Распечатка 7.10 Асинхронное уведомление с помощью SIGEV_THREAD

Распечатка 7.10.

```
struct sigevent notif;
sigset_t sig_set;
siginfo_t info;
      ....

/*
 * Указываем уведомление как SIGEV_THREAD. Обратите внимание,
 * что когда вызывается функция уведомления, она работает в
 * отдельном потоке
 */
notif.sigev_notify = SIGEV_THREAD;
/* Процедура уведомления, которая будет вызвана */
notif.sigev_notify_function = notify_routine;

/*
 * Передаём в функцию уведомления, когда она вызывается,
 * id очереди сообщений как аргумент
 */
notif.sigev_value.sival_int = ds;
/* Поток уведомления должен быть в состоянии DETACHED */
notif.sigev_notify_attributes = NULL;

/* Наконец, настраиваем уведомление */
if (mq_notify(ds, &notif)){
```

```
  perror("mq_notify");
  return -1;
}
           ....

/*
 * .. а это процедура уведомления. Она будет вызываться
 * каждый раз при поступлении в очередь сообщения.
 */
void notify_routine(sigval_t value){
           ...
  /* Теперь, конечно, можно получить это сообщение. */
  if ((len = mq_receive(value.sival_int, new_text, SIZE,
                        &prio)) == -1)
    perror("Receiving message error");
           ...
}
```

7.3.5 Семафоры POSIX

Семафоры являются счётчиками, совместно используемыми потоками или процессами ресурсов. Основными операциями с семафорами являются:

- Атомарное увеличение счетчика.
- Ожидание, пока счётчик не станет равен нулю, а затем его атомарное уменьшение.

Для межпроцессной или межпотоковой синхронизации также могут быть использованы двоичные семафоры. В основном они используются для синхронизации доступа к общим ресурсам, таким как общая память, глобальные структуры данных, и так далее. Есть два типа семафоров POSIX:

- ***Именованные семафоры***: они могут быть использованы для синхронизации между несколькими несвязанными процессами.
- ***Безымянные семафоры***: они могут быть использованы потоками внутри процесса или для синхронизации между связанными процессами (например, родительским и дочерним процессом).

Семафоры POSIX 1003.1b в Linux реализует библиотека **pthread**, которая является частью пакета **glibc**. **Glibc** 2.3 с NPTL имеет полную поддержку для семафоров, в том числе именованных и используемых разными процессами для доступа к общим ресурсам семафоров. Более ранние версии **glibc** поддерживали только безымянные семафоры. Операции с семафорами перечислены в Таблице 7.6. Использование именованного семафора иллюстрирует Распечатка 7.11 210.

Таблица 7.6 Функции для работы с семафорами POSIX.1b

Метод	*Описание*
sem_open	Открытие/создание именованного семафора.
sem_close	Закрытие именованного семафора.

Метод	*Описание*
sem_unlink	Удаление именованного семафора.
sem_init	Инициализация неименованного семафора.
sem_destroy	Удаление неименованного семафора.
sem_getvalue	Получение текущего значения счётчика семафора.
sem_wait	Выполнение операции блокировки семафора.
sem_trywait	Выполнение операции попытки блокировки семафора.
sem_post	Освобождение семафора.

Что следует помнить

- Семафорная защита работает только между взаимодействующими процессами; то есть процесс должен ждать семафор, если он не доступен, и должен освободить семафор после использования.
- Дескриптор семафора передаётся по наследству через ветвление. Дочерним процессам нет необходимости заново открывать семафор. После использования они могут вызвать **sem_close**.
- Функция **sem_post** является безопасной для передачи асинхронных сигналов и может быть вызвана из обработчиков сигналов.
- Может произойти инверсия приоритета, если процесс с низким приоритетом блокирует семафор, необходимый процессу с высоким приоритетом.

Распечатка 7.11 Операции с семафорами POSIX

Распечатка 7.11.

```
/* sem.c */

#include <stdio.h>
#include <semaphore.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <errno.h>

#define SEM_NAME "/mysem"
/*
 * Интерфейсы именованных семафоров вызываются в следующем порядке
 *    sem_open()
 *     ...
 *    sem_close()
 *    sem_unlink()
 * Интерфейсы неименованных семафоров вызываются в следующем порядке
 *    sem_init()
 *     ...
 *    sem_destroy()
```

```
 */

int main(){
  /* Наш именованный семафор */
  sem_t *sema_n;
  int ret,val;

  /*
   * Создаём именованный семафор (O_CREAT) с начальным значением 1
   * (то есть разблокирован)(Если вы хотите создать неименованный
   * семафор, то замените вызов sem_open на sem_init)
   */
  if ((sema_n = sem_open(SEM_NAME, O_CREAT, 0600, 1)) ==
                                              SEM_FAILED){
    perror("sem_open");
    return -1;
  }

  /* Получаем текущее значение семафора */
  sem_getvalue(sema_n, &val);
  printf("semaphore value = %d\n", val);

  /*
   * Пробуем получить семафор. Если не получается, используем
   * блокирующуюся версию. Это сделано только для того, чтобы
   * показать семантику и sem_trywait, и sem_wait. В реальном
   * коде не надо делать, как здесь.
   */
  if ((ret = sem_trywait(sema_n)) != 0 && errno == EAGAIN)
    /* permanent wait */
    sem_wait(sema_n);
  else if (ret != 0){
    perror("sem_trywait");
    return -1;
  }

  /*
   * Семафор захвачен. Работаем с общими данными.
   */

  /*
   * Освобождаем семафор после манипулирования общими данными
   */
  if (sem_post(sema_n) != 0)
    perror("post error");

  /*
   * Закрываем и удаляем семафор. (Для неименованных семафоров
   * замените следующие два вызова на sem_destroy. Для
   * неименованных семафоров sem_unlink не пригоден)
   */
  sem_close(sema_n);
  sem_unlink(SEM_NAME);
  return 0;
```

```
}
```

7.3.6 Сигналы реального времени

Сигналы расширения POSIX 1003.1b играют очень важную роль в приложениях реального времени. Они используются для уведомления процессов о возникновении асинхронных событий, таких как завершение работы таймера высокого разрешения, завершение асинхронного ввода/вывода, приём сообщения в пустой очереди сообщений POSIX, и так далее. Некоторые преимущества, которыми обладают сигналы реального времени по отношению к нативным сигналам, перечислены в Таблице 7.7. Эти преимущества делают их пригодными для приложений реального времени. Сигнальные интерфейсы реального времени POSIX.1b перечислены в Таблице 7.8.

Таблица 7.7 Сравнение сигналов реального времени и нативных сигналов

Сигналы реального времени	*Нативные сигналы*
Диапазон предназначенных для приложения сигналов от SIGRTMIN до SIGRTMAX. Все сигналы реального времени определяются в этом диапазоне, например, SIGRTMIN + 1, SIGRTMIN + 2, SIGRTMAX – 2, и так далее.	Только два предназначенных для приложения сигнала, **SIGUSR1** и **SIGUSR2**.
Доставка сигналов может иметь приоритет. Чем меньше номер сигнала, тем выше приоритет. Например, если ждут обработки сигналы SIGRTMIN и SIGRTMIN + 1, то первым будет доставлен SIGRTMIN.	Нет приоритета для доставки сигнала.
Отправитель может передать принимающему процессу вместе с сигналом реального времени дополнительную информацию.	Вместе с сигналом нельзя послать дополнительную информацию.
Сигналы поступают в очередь (то есть, если сигнал доставлен процессу несколько раз, получатель будет обрабатывать все экземпляры сигнала). Сигналы реального времени не теряются.	Сигналы могут потеряться. Если сигнал доставлен процессу несколько раз, получатель обработает только один экземпляр.

Таблица 7.8 Функции сигналов реального времени POSIX.1b

Метод	*Описание*
sigaction	Регистрация дескриптора сигнала и механизма уведомления.
sigqueue	Отправка процессу сигнала и дополнительной информации.
sigwaitinfo	Ожидание доставки сигнала.

Метод	*Описание*
sigtimedwait	Ожидание доставки сигнала и завершение по истечении времени ожидания, если сигнал не прибыл.
sigsuspend	Приостановка процесса, пока не получен сигнал.
sigprocmask	Изменение текущей маски блокировки процесса.
sigaddset	Добавление сигнала к набору сигналов.
sigdelset	Удаление сигнала из набора сигналов.
sigemptyset	Очистка набора сигналов от всех сигналов.
sigfillset	Установка всех сигналов в наборе сигналов.
sigismember	Проверка, является ли сигнал членом набора сигналов.

Объясним вышеописанные интерфейсы на примере. В этом примере родительский процесс посылает сигналы реального времени дочернему процессу, а затем они обрабатываются. Пример разделён на две части, как показано на Рисунке 7.4.

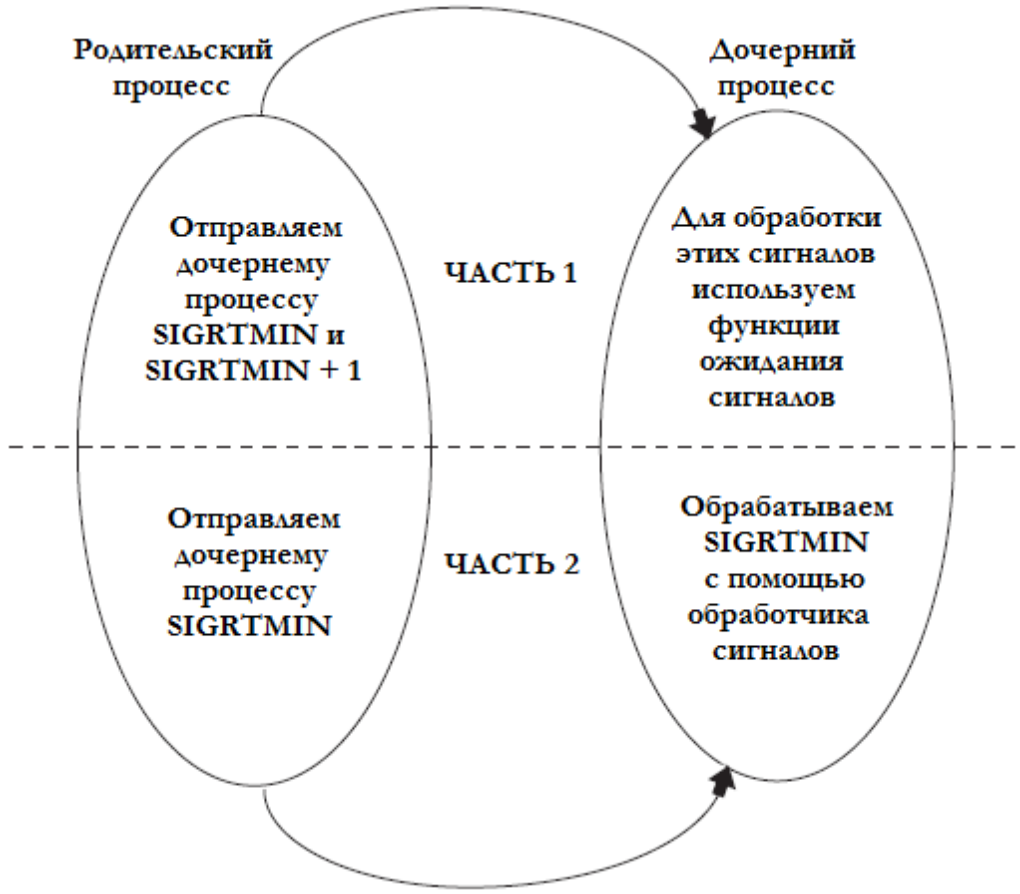

Рисунок 7.4 Сигналы реального времени: пример приложения.

Основное приложение

```
#include <signal.h>
```

```
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>

int child(void);
int parent(pid_t);
/* Обработчик сигнала */
void rt_handler(int signum, siginfo_t *siginfo,
                void * extra);
int main(){
  pid_t cpid;
  if ((cpid = fork()) == 0)
    child();
  else
    parent(cpid);
}
```

Родительский процесс

Родительский процесс использует для отправки дочернему процессу **SIGRTMIN** и **SIGRTMIN + 1** функцию **sigqueue**. Последний аргумент функции используется, чтобы передавать вместе с сигналом дополнительную информацию.

```
int parent(pid_t cpid){
  union sigval value;

  /* -------- НАЧАЛО 1-ой ЧАСТИ  ----------- */

  /* Спим, пока запускается дочерний процесс */
  sleep(3);
  /* Дополнительная информация для SIGRTMIN */
  value.sival_int = 1;
  /* Отправка дочернему процессу SIGRTMIN */
  sigqueue(cpid, SIGRTMIN, value);
  /* Отправка дочернему процессу SIGRTMIN+1 */
  sleep(3);
  value.sival_int = 2;
  sigqueue(cpid, SIGRTMIN+1, value);

  /* -------- НАЧАЛО 2-ой ЧАСТИ  ----------- */

  /* Наконец, посылаем SIGRTMIN ещё раз */
  sleep(3);
  value.sival_int = 3;
  sigqueue(cpid, SIGRTMIN, value);

  /* --------- КОНЕЦ 2-ой ЧАСТИ  ----------- */
}
```

Дочерний процесс

```
int child(void){
  sigset_t mask, oldmask;
```

```
  siginfo_t siginfo;
  struct sigaction action;
  struct timespec tv;
  int count = 0, recv_sig;
```

Для хранения сигналов, которые необходимо блокировать, мы определяем маску типа **sigset_t**. Прежде чем продолжить, очищаем маску, вызывая **sigemptyset**. Эта функция инициализирует маску, чтобы исключить все сигналы. Затем вызывается **sigaddset**, чтобы добавить к заблокированному набору сигналы **SIGRTMIN** и **SIGRTMIN + 1**. Наконец, для блокировки доставки сигнала вызывается **sigprocmask**. (Мы блокируем эти сигналы в процессе их доставки с помощью функции ожидания сигнала, вместо того, чтобы использовать обработчик сигнала.)

```
/* -------- НАЧАЛО 1-ой ЧАСТИ  ----------- */

/* Очищаем маску */
sigemptyset(&mask);
/* Добавляем к маске SIGRTMIN */
sigaddset(&mask, SIGRTMIN);
/* Добавляем к маске SIGRTMIN+1 */
sigaddset(&mask, SIGRTMIN+1);

/*
 * Блокируем доставку сигналов SIGRTMIN, SIGRTMIN+1.
 * После возвращения предыдущее значение
 * заблокированной сигнальной маски хранится в oldmask
 */
sigprocmask(SIG_BLOCK, &mask, &oldmask);
```

Теперь дочерний процесс ждёт доставки **SIGRTMIN** и **SIGRTMIN + 1**. Для ожидания блокированных сигналов используются функция **sigwaitinfo** или **sigtimedwait**. Набором сигналов для ожидания является первый аргумент. Второй аргумент заполняется любой дополнительной информацией, полученной вместе с сигналом (подробнее об **siginfo_t** позже). Последним аргументом для **sigtimedwait** является время ожидания.

```
/* Задаём время ожидания 1 с */
tv.tv_sec = 1;
tv.tv_nsec = 0;

/*
 * Ждём доставки сигнала. Мы ожидаем 2 сигнала,
 * SIGRTMIN и SIGRTMIN+1. Цикл завершится, когда
 * будут приняты оба сигнала
 */
while(count < 2){
  if ((recv_sig = sigtimedwait(&mask, &siginfo, &tv)) == -1){
    if (errno == EAGAIN){
      printf("Timed out\n");
      continue;
    }else{
      perror("sigtimedwait");
      return -1;
    }
```

```
  }else{
    printf("signal %d received\n", recv_sig);
    count++;
  }
}
/* --------- КОНЕЦ 1-ой ЧАСТИ  ----------- */
```

Другим методом для обработки сигналов является регистрация обработчика сигналов. В этом случае процесс не должен блокировать сигнал. Зарегистрированный обработчик сигнала вызывается, когда сигнал доставляется процессу. Обработчик сигнала регистрирует функция **sigaction**. Это второй аргумент структуры **sigaction**, определённой как

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
}
```

Полями этой структуры являются:

- **sa_handler**: функция регистрации обработчика сигналов для сигналов не реального времени.
- **sa_sigaction**: функция регистрации обработчика сигналов для сигналов реального времени.
- **sa_mask**: маска сигналов, которые должны быть заблокированы, когда выполняется обработчик сигналов.
- **sa_flags**: для сигналов реального времени используется **SA_SIGINFO**. При получении сигнала, который имеет установленный **SA_SIGINFO**, вместо **sa_handler** вызывается функция **sa_sigaction**.

Теперь для обработки сигнала реального времени **SIGRTMIN** дочерний процесс регистрирует обработчик сигналов **rt_handler**. Эта функция будет вызываться, когда дочернему процессу доставляется **SIGRTMIN**.

```
/* -------- НАЧАЛО 2-ой ЧАСТИ  ----------- */

/* Устанавливаем SA_SIGINFO */
action.sa_flags = SA_SIGINFO;
/* Очищаем маску */
sigemptyset(&action.sa_mask);

/*
 * Регистрируем обработчик сигнала для SIGRTMIN.
 * Обратите внимание, что для регистрации
 * обработчика мы используем интерфейс
 * action.sa_sigaction
 */
action.sa_sigaction = rt_handler;
if (sigaction(SIGRTMIN, &action, NULL) == -1){
  perror("sigaction");
```

```
  return 0;
}
```

Затем дочерний процесс ждёт доставки сигнала. **sigsuspend** временно заменяет текущую маску сигнала маской, указанной в её аргументе. Затем она ожидает доставки разблокированного сигнала. После доставки сигнала **sigsuspend** восстанавливает исходную маску и возвращается после выполнения обработчика сигнала. Если обработчик сигнала вызывает прекращение процесса, **sigsuspend** не возвращается.

```
  /* Wait from SIGRTMIN */
  sigsuspend(&oldmask);

  /* --------- КОНЕЦ 2-ой ЧАСТИ  ----------- */
}
```

Таким образом, дочерний процесс успешно обработал сигналы, используя и функции ожидания сигнала, и обработчик сигналов.

Наконец, обработчик сигнала:

```
/* Обработчик сигнала для SIGRTMIN */
void rt_handler(int signum, siginfo_t *siginfo,
                void * extra){
  printf("signal %d received. code = %d, value = %d\n",
           siginfo->si_signo, siginfo->si_code,
           siginfo->si_int);
}
```

Второй аргумент обработчика сигнала имеет тип **siginfo_t** и содержит всю информацию о полученном сигнале. Эта структура определяется так:

```
siginfo_t {
  int si_signo; // Номер сигнала
  int si_errno; // Ошибка сигнала
  int si_code;  // Код сигнала
  union {
      ...
    // Сигналы POSIX .1b
    struct {
      pid_t pid;
      uid_t uid;
      sigval_t sigval;
    }_rt;
      ...
  }_sifields
}
```

Чтобы узнать о всех полях этой структуры, обратитесь, пожалуйста, к **/usr/include/bits/siginfo.h**. Кроме **si_signo**, **si_errno** и **si_code** все остальные поля входят в объединение. Так что следует читать только те поля, которые имеют смысл в данном контексте. Например, поля, приведённые выше, действительны только для сигналов POSIX.1b.

- **si_signo** является номером сигнала принятого сигнала. Это то же самое, что и первый аргумент обработчика сигналов.
- **si_code** передаёт источник сигнала. Важные значения **si_code** для сигналов реального времени перечислены в Таблице 7.9.
- **si_value** это дополнительная информация, передаваемая отправителем.
- **pid** и **uid** являются идентификатором процесса и идентификатором пользователя отправителя, соответственно.

Таблица 7.9 Коды сигналов

Код сигнала	*Значение*	*Источник*
SI_USER	0	**kill**, **sigsend**, или **raise**
SI_KERNEL	0x80	Ядро
SI_QUEUE	-1	Функция **sigqueue**
SI_TIMER	-2	Окончание работы таймера POSIX
SI_MESGQ	-3	Изменение состояния очереди сообщений POSIX с не-пусто на пусто
SI_ASYNCIO	-4	Завершение асинхронного ввода-вывода

Обращения к вышеописанным поля должны осуществляться с помощью макросов, которые определены в **siginfo.h**:

```
#define si_value    _sifields._rt._sigval
#define si_int      _sifields._rt._sigval.sival_int
#define si_ptr      _sifields._rt._sigval.sival_ptr
#define si_pid      _sifields._kill._pid
#define si_uid      _sifields._kill._uid
```

Таким образом, для доступа к дополнительной информации, передаваемой отправителем, можно использовать **siginfo->si_int** и **siginfo->si_ptr**.

7.3.7 Время и таймеры POSIX.1b

Традиционные обычные таймеры UNIX в Linux, функции **setitimer** и **getitimer**, не отвечают требованиям большинства приложений реального времени. Таймеры POSIX.1b имеют следующие преимущества по сравнению с обычными таймерами UNIX.

- Процесс может иметь несколько таймеров.
- Лучшая точность таймеров. Таймеры могут использовать значения с точностью до наносекунд.
- Уведомление о завершении работы таймера может быть сделано либо с помощью любого произвольного сигнала (реального времени) или с помощью потоков. Для уведомление о завершении работы таймера в обычных таймерах UNIX есть лишь ограниченный набор сигналов.
- Таймеры POSIX.1b предоставляют поддержку различных часов, таких как **CLOCK_REALTIME**, **CLOCK_MONOTONIC**, и так далее, которые могут иметь

различные источники с различным разрешением. Обычный таймер UNIX, с другой стороны, связан с системными часами.

Ядро таймеров POSIX.1b представляет собой набор часов, которые используются как привязка ко времени. Linux обеспечивает поддержку следующих часов:

- **CLOCK_REALTIME**: общесистемные часы реального времени, видимые для всех процессов, работающих в системе. Часы измеряют количество времени в секундах и наносекундах с начала эпохи (то есть 00:00:00 1 января 1970 по Гринвичу).Разрешение часов равно 1/**HZ** секунд. Таким образом, если **HZ** равно 100, то разрешение часов составляет 10 мс. Если **HZ** равно 1000, то разрешение часов составляет 1 мс. Чтобы узнать значение HZ в вашей системе, посмотрите, пожалуйста, файл **<kernel-source>/include/asm/param.h**. Так как это время базируются на времени настенных часов, оно может быть изменено.
- **CLOCK_MONOTONIC**: время непрерывной работы системы, видимое всем процессам в системе. В Linux оно измеряется как количество времени в секундах и наносекундах после загрузки системы. Его разрешение равно 1/**HZ** с. Его поддержка доступна начиная с ядра версии 2.5 и glibc 2.3.3. Это время не может быть изменено каким-либо процессом.
- **CLOCK_PROCESS_CPUTIME_ID**: часы, измеряющие время работы процесса. Время текущего процесса, потраченное на выполнение в системе, измеряется в секундах и наносекундах. Разрешение равно 1/**HZ**. Это время может быть изменено.
- **CLOCK_THREAD_CPUTIME_ID**: То же, что и **CLOCK_PROCESS_CPUTIME_ID**, но для текущего потока.

Обычно **CLOCK_REALTIME** используется для указания абсолютного времени ожидания. **CLOCK_MONOTONIC** используется для относительного времени ожидания и периодических задач. Поскольку время этих часов не может быть изменено, периодическим задачам не нужно беспокоиться о преждевременном или задержанном пробуждении, которое могло бы произойти с **CLOCK_REALTIME**. Двое других часов могут использоваться для целей учёта. Интерфейсы времени и таймеров POSIX.1b перечислены в Таблице 7.10.

Таблица 7.10 Функции времени и таймеров POSIX.1b

Метод	*Описание*
clock_settime	Установка значения указанным часам.
clock_gettime	Получение времени.
clock_getres	Получение разрешения (точности) часов.
clock_nanosleep	Приостановка выполнения вызывающего приложения на указанное время.
timer_create	Создание таймера на основе указанного времени.
timer_delete	Удаление таймера.
timer_settime	Установка времени работы таймера.
timer_gettime	Получение текущего значения таймера.
timer_getoverrun	Возвращает число, показывающее сколько раз таймер

Метод	*Описание*
	закончил работу между моментом генерации сигнала и его доставкой

Объясним использование описанных выше интерфейсов на примере. В этом примере мы создаём таймер POSIX, основанный на **CLOCK_MONOTONIC**. Это периодический таймер с периодом четыре секунды. По окончании работы таймера происходит уведомление процесса с помощью сигнала реального времени **SIGRTMIN**. Процесс зарегистрировал обработчик сигнала для **SIGRTMIN**, который хранит счётчик числа раз завершения работы таймера. Когда счётчик достигает заданного значения, названного в примере как **MAX_EXPIRE**, таймер останваливается и процесс завершается.

```
#include <unistd.h>
#include <time.h>
#include <signal.h>

#define MAX_EXPIRE 10
int expire;

void timer_handler(int signo, siginfo_t *info,
                   void *context);

int main(){
  struct timespec ts, tm, sleep;
  sigset_t mask;
  siginfo_t info;
  struct sigevent sigev;
  struct sigaction sa;
  struct itimerspec ival;
  timer_t tid;
```

Сначала распечатаем некоторые статистические данные о **CLOCK_MONOTONIC**. **clock_getres** даёт разрешение часов, а **clock_gettime** даёт время работы системы. Обратите внимание, что разрешение **CLOCK_MONOTONIC** равно 1/**HZ**.

```
clock_getres(CLOCK_MONOTONIC, &ts);
clock_gettime(CLOCK_MONOTONIC, &tm);
printf("CLOCK_MONOTONIC res: [%d]sec [%d]nsec/n",
                          ts.tv_sec, ts.tv_nsec);
printf("system up time: [%d]sec [%d]nsec\n",
                          tm.tv_sec, tm.tv_nsec);
```

Настраиваем обработчик сигнала для **SIGRTMIN**. Как упоминалось ранее, по истечении времени работы таймера процесс получит сигнал реального времени **SIGRTMIN**.

```
/* Мы не хотим никаких блокированных сигналов */
sigemptyset(&mask);
sigprocmask(SIG_SETMASK, &mask, NULL);

/* Регистрируем обработчик для SIGRTMIN */
sa.sa_flags = SA_SIGINFO;
```

```
sigemptyset(&sa.sa_mask);
sa.sa_sigaction = timer_handler;
if (sigaction(SIGRTMIN, &sa, NULL) == -1) {
  perror("sigaction failed");
  return -1;
}
```

Создаём таймер. Второй аргумент **timer_create** представляет тип желаемого уведомления об окончании работы таймера. Вспомните, пожалуйста, из нашего рассказа об очереди сообщений POSIX, что уведомления бывают двух типов, **SIGEV_SIGNAL** и **SIGEV_THREAD**. В случае таймеров POSIX, в качестве механизма уведомления может быть использован любой из них. В этом примере мы используем механизм уведомления **SIGEV_SIGNAL**.

```
/*
 * По завершении работы таймера должен быть послан
 * сигнал SIGRTMIN с произвольным значением 1
 */
sigev.sigev_notify = SIGEV_SIGNAL;
sigev.sigev_signo = SIGRTMIN;
sigev.sigev_value.sival_int = 1;

/*
 * Создаём таймер. Обратите внимание, что если вызов
 * успешен, идентификатор таймера возвращается в
 * третьем аргументе.
 */
if (timer_create(CLOCK_MONOTONIC, &sigev, &tid) == -1){
  perror("timer_create");
  return -1;
}
printf("timer-id = %d\n", tid);
```

Взводим таймер. Время истечёт через пять секунд и после каждых четырёх секунд впоследствии.

```
ival.it_value.tv_sec = 5;
ival.it_value.tv_nsec = 0;
ival.it_interval.tv_sec = 4;
ival.it_interval.tv_nsec = 0;
if (timer_settime(tid, 0, &ival, NULL) == -1){
  perror("timer_settime");
  return -1;
}
```

Наконец, ждём истечения времени работы таймера. Если счётчик числа раз окончания работы таймера достигает **MAX_EXPIRE**, выключаем таймер и выходим.

```
  /* Спим и ждём сигнал */
  for(;;){
    sleep.tv_sec = 3;
    sleep.tv_nsec = 0;
    clock_nanosleep(CLOCK_MONOTONIC, 0, &sleep, NULL);
```

```
    printf("woken up\n");
    if (expire >= MAX_EXPIRE){
      printf("Program quitting.\n");
      /*
       * Если it_value == 0, вызываем timer_settime,
       * чтобы отключить таймер
       */
      memset(&ival, 0, sizeof (ival));
      timer_settime(tid, 0, &ival, NULL);
      return 0;
    }
  }
  return 0;
}
```

Наконец, мы имеем **timer_handler**: вспомните наше обсуждение обработчиков сигналов из Раздела 7.3.6 212. Второй аргумент обработчика имеет тип **siginfo_t**, который содержит информацию относительно принимаемого сигнала. В этом случае значением **info->si_code** является **SI_TIMER**.

```
void timer_handler(int signo, siginfo_t *info,
                   void *context)
{
  int overrun;
  printf("signal details: signal (%d), code (%d)\n",
                      info->si_signo, info->si_code);
  if (info->si_code == SI_TIMER){
    printf("timer-id = %d \n", info->si_timerid);
    expire++;

    /*
     * Спецификация говорит, что в один момент времени
     * для данного таймера только один экземпляр сигнала
     * помещается в очередь процесса. Если таймер, сигнал
     * которого всё еще ожидает доставки, заканчивает
     * работу, сигнал не помещается в очередь и происходит
     * ситуация переполнения таймера. timer_getoverrun
     * возвращает дополнительное число окончаний работы
     * таймера, которые произошли между моментом, когда
     * был сгенерирован сигнал (помещён в очередь) и
     * моментом, когда он был доставлен или принят
     */
    if ((overrun = timer_getoverrun(info->si_timerid)) !=
                                  -1 && overrun != 0){
      printf("timer overrun %d\n", overrun);
      expire += overrun;
    }
  }
}
```

Таймеры высокого разрешения

Для часов, о которых говорилось выше, Linux обеспечивает наилучшее разрешение в 1 мс

(HZ = 1000). Этого разрешения не достаточно для большинства приложений реального времени, так как они требуют точности порядка микросекунд и наносекунд. Для поддержки таких приложений инженерами MontaVista был начат проект High-Resolution Timers (HRT), Таймеры Высокого Разрешения. HRT являются таймерами POSIX с разрешением до микросекунды. Введены двое дополнительных часов POSIX, **CLOCK_REALTIME_HR** и **CLOCK_MONOTONIC_HR**. Они такие же, как их коллеги, не обладающие высокой точностью; отличие в том, что разрешение времени у них имеет порядок микросекунд или наносекунд, в зависимости от генератора частоты для аппаратных часов. На момент написания, поддержка HRT не включена в основное дерево исходных текстов и доступна в виде патча. Более подробно о проекте можно узнать на www.sourceforge.net/projects/high-res-timers.

Что следует помнить

- Точность часов является фиксированной и не может быть изменена во время выполнения приложения.
- Чтобы деактивировать таймер, вызовите **timer_settime** со значением члена **it_value** структуры **itimespec**, равным нулю.
- Таймер может быть периодическим или однократным. Если член **it_interval** структуры **itimerspec** во время вызова **timer_setime** равен нулю, то таймер однократный; в противном случае он является периодическим.
- POSIX.1b также обеспечивает функцию **nanosleep**. Она такая же, как и **clock_nanosleep**, с **CLOCK_REALTIME** в качестве первого аргумента.
- Таймеры, предназначенные для каждого процесса, не наследуются дочерним процессом.

7.3.8 Асинхронный ввод-вывод

Традиционные системные вызовы чтения и записи являются блокирующими вызовами. Большинство приложений реального времени может нуждаться в прерывании их работы и процесса ввода/вывода с целью улучшения детерминизма. Например, приложение может предпочесть асинхронный ввод/вывод (Asynchronous I/O, AIO), если требуется сбор большого объёма данных от какого-то источника и если обработка данных - интенсивное вычисление. POSIX.1b определяет асинхронные интерфейсы ввода/вывода для выполнения требований таких приложений.

Механизм очень прост. Приложение может поместить в очередь запрос на AIO, а затем продолжить нормальную обработку. По завершении ввода/вывода приложение уведомляется. Затем оно может запросить состояние ввода/вывода, чтобы узнать был ли он успешным или закончился неудачей. Используя интерфейсы AIO, приложение может выполнять следующие операции:

- Формировать нескольких неблокирующих запросов ввода/вывода для различных источников с помощью единственного вызова. (Таким образом, приложение может иметь много операций ввода/вывода в процессе выполнения, в то время как оно выполняет другой код.)
- Отменять любые не выполненные запросы ввода/вывода.
- Ожидать завершения ввода/вывода.
- Отслеживать состояние ввода/вывода: в процессе работы, ошибка, или завершён.

Блок управления асинхронным вводом-выводом

Блок управления AIO, структура **aiocb**, является ядром AIO POSIX.1b. Эта структура содержит сведения, которые необходимы для представления AIO. Структура определяется следующим образом:

```
struct aiocb
{
  int aio_fildes;               /* Файловый дескриптор. */
  int aio_lio_opcode;           /* Операция для выполнения,
                                   чтение или запись. Используется
                                   при нескольких запросах AIO
                                   в одном запросе */
  int aio_reqprio;               /* Изменение приоритета запроса. */
  volatile void *aio_buf;        /* Расположение буфера для чтения
                                    или записи */
  size_t aio_nbytes;             /* Размер передачи. */
  struct sigevent aio_sigevent; /* Информация уведомления. */
  off_t aio_offset;              /* Смещение в файле для начала
                                    чтения или записи в него */
}
```

Обратите внимание, что в отличие от традиционных операций чтения или записи, вам необходимо указать смещение в файле, откуда должен начаться AIO. После выполнения ввода/вывода ядро не будет увеличивать значение поля, содержащее смещение в файле, в файловом дескрипторе. Вы должны следить за смещениями в файле вручную.

Функции асинхронного ввода-вывода

Функции AIO приведены в Таблице 7.11. Использование интерфейсов AIO POSIX.1b иллюстрирует Распечатка 7.12 226. Этот пример просто копирует один файл в другой с помощью AIO. Для простоты будем считать, что функции AIO не возвращают ошибку.

Таблица 7.11 Функции асинхронного ввода-вывода

Метод	*Описание*
aio_read	Старт асинхронного чтения.
aio_write	Старт асинхронной записи.
aio_error	Возвращает статус завершения последней **aio_read** или **aio_write**.
aio_return	Возвращает число байт, переданных в **aio_read** или **aio_write**.
aio_cancel	Отменяет любые ожидающие операции AIO.
aio_suspend	Вызывает процесс, если завершается любой из указанных запросов.
lio_listio	Запрос нескольких операций чтения или записи.

Ввод/вывод, направляемый в список

Для передачи произвольного числа запросов чтения или записи в одном вызове может быть использована функция **lio_listio**:

```
int lio_listio(int mode, struct aiocb *list[], int nent,
               struct sigevent *sig);
```

- **mode**: этот аргумент может быть **LIO_WAIT** или **LIO_NOWAIT**. Если этот аргумент имеет значение **LIO_WAIT**, функция ждёт, пока все вводы-выводы завершатся и **sig** игнорируется. Если аргументом является **LIO_NOWAIT**, функция возвращается немедленно и после завершения ввода/вывода будет происходить асинхронное уведомление, как указано в **sig**.
- Список **aiocb**: этот аргумент содержит список **aiocb**.
- **nent**: количество **aiocb** во втором аргументе.
- **sig**: желаемый механизм уведомления. Уведомление не генерируется, если этот аргумент NULL.

Распечатка 7.12 может быть изменена для использования функции **lio_listio**.

```
while(1){
  memcpy(write_buf, read_buf, read_n);
  a_write.aio_nbytes = read_n;
  a_read.aio_nbytes = XFER_SIZE;

  /* Готовим список aiocb для lio_listio */
  cblist_lio[0] = &a_read;
  cblist_lio[1] = &a_write;

  /*
   * Вызываем lio_listio, чтобы отправить запросы
   * асинхронного чтения и записи
   */
  lio_listio(LIO_NOWAIT, cblist_lio, 2, NULL);
       ..............
}
```

Реализация в Linux

Первоначально AIO в Linux был полностью реализован в пользовательском пространстве с помощью потоков. Был один пользовательский поток, создаваемый для каждого запроса. Это привело к плохой масштабируемости и низкой производительности. Начиная с версии 2.5, в ядро была добавлена поддержка AIO. Тем не менее, интерфейсы ядра, предоставляющие AIO, отличаются от интерфейсов POSIX. Интерфейсы базируются на новом наборе системных вызовов. Они перечислены в Таблице 7.12. Эти интерфейсы предоставляются в пользовательском пространстве библиотекой **libaio**.

Таблица 7.12 Интерфейсы асинхронного ввода-вывода в ядре

Метод	*Описание*
io_setup	Создание нового контекста запросов для AIO.
io_submit	Отправка запроса AIO (также известна как **aio_read**, **aio_write**, **lio_listio**).
io_getevents	Узнать о завершённых операциях ввода/вывода (также известна как **aio_error**, **aio_return**).
io_wait	Ожидание завершения ввода/вывода (также известна как **aio_suspend**).
io_cancel	Отмена I/O (также известна как **aio_cancel**).
io_destroy	Уничтожение контекста AIO. Происходит по умолчанию при завершении процесса.

Что следует помнить

- Блок управления не должен изменяться во время выполнения операции чтения или записи. Также, указатель буфера в **aiocb** должен быть действителен, пока запрос не завершен.
- Возвращаемое значение **lio_listio** не указывает состояние отдельных запросов ввода/вывода. Неудачное завершение запроса не препятствует завершению других запросов.

Распечатка 7.12 Копирование файла с помощью асинхронного ввода-вывода

Распечатка 7.12.

```
/* aio_cp.c */

#include <unistd.h>
#include <aio.h>
#include <sys/types.h>
#include <errno.h>

#define INPUT_FILE "./input"
#define OUTPUT_FILE "./output"
/* Размер для передачи в одной операции чтения или записи */
#define XFER_SIZE 1024
#define MAX 3

/* Функция для заполнения aiocb значениями */
void populate_aiocb(struct aiocb *aio, int fd, off_t offset,
                    int bytes, char *buf){
  aio->aio_fildes = fd;
  aio->aio_offset = offset;

  /*
   * Мы не используем здесь механизм уведомления, чтобы уделить
   * больше внимания интерфейсам AIO. Для получения уведомления
   * после завершения AIO мы можем использовать механизмы
   * уведомления либо SIGEV_SIGNAL, либо SIGEV_THREAD
```

```
   */
  aio->aio_sigevent.sigev_notify = SIGEV_NONE;
  aio->aio_nbytes = bytes;
  aio->aio_buf = buf;
}

/*
 * Это приложение копирует один файл в другой
 */
int main(){

  /* Файловые дескрипторы чтения/записи */
  int fd_r , fd_w;
  /* Блоки управления AIO для чтения и записи */
  struct aiocb a_write, a_read;

  /*
   * Этот список используется для хранения блоков
   * управления исходящих запросов чтения или записи
   */
  struct aiocb *cblist[MAX];
  /* Статус операции чтения или записи */
  int err_r, err_w;
  /* число на самом деле считанных байтов */
  int read_n = 0;
  /* Метка завершения потока для файла-источника */
  int quit = 0;
  /* Используются для передачи данных между источником
   * и файлом назначения
   */
  char read_buf[XFER_SIZE];
  char write_buf[XFER_SIZE];

  /*
   * Открываем файлы источника и места назначения. Вызываем
   * функцию populate_aiocb, чтобы проинициализировать блоки
   * управления AIO для операции чтения и записи. Хорошей
   * практикой является очистка блоков aiocb перед
   * их использованием
   */
  fd_r = open(INPUT_FILE, O_RDONLY, 0444);
  fd_w = open(OUTPUT_FILE, O_CREAT | O_WRONLY, 0644);

  memset(&a_write, 0 , sizeof(struct aiocb));
  memset(&a_read, 0 , sizeof(struct aiocb));

  /* Заполняем блоки aiocb значениями по умолчанию */
  populate_aiocb(&a_read, fd_r, 0, XFER_SIZE, read_buf);
  populate_aiocb(&a_write, fd_w, 0, XFER_SIZE, write_buf);

  /*
   * Запускаем асинхронное чтение из файла-источника с помощью
   * функции aio_read. Эта функция читает a_read.aio_nbytes
   * байтов из файла a_read.aio_fildes, начиная со смещения
```

```
 * a_read.aio_offset в буфер a_read.aio_buf. В случае успеха
 * возвращается 0. Эта функция возвращается сразу после
 * помещения запроса в очередь.
 */
aio_read(&a_read);

/*
 * Ожидаем завершения чтения. После старта любой асинхронной
 * операции (чтения или записи), можно получить её статус с
 * помощью функции aio_error. Если запрос не завершён, эта
 * функция возвращает EINPROGRESS, если запрос завершён
 * успешно, она возвращает 0, в противном случае возвращается
 * значение ошибки. Если aio_read возвращает EINPROGRESS, вызов
 * aio_suspend ожидает завершения операции.
 */
while((err_r = aio_error(&a_read)) == EINPROGRESS){
  /*
   * Функция aio_suspend приостанавливает вызывающий процесс,
   * пока не завершится по крайней мере один из асинхронных
   * запросов ввода/вывода в списке cblist, или не будет
   * доставлен сигнал. Здесь мы ждём завершения aio_read
   * для блока a_read.
   */
  cblist[0] = &a_read;
  aio_suspend(cblist, 1, NULL);
}

/*
 * Если возвращаемое значение функции aio_error равно 0, то
 * операция чтения была успешной. Вызываем aio_return, чтобы
 * узнать количество прочитанных байтов. Функция должна быть
 * вызвана только один раз после того, как aio_error
 * возвращает нечто, отличное от EINPROGRESS.
 */
if (err_r == 0){
  read_n = aio_return(&a_read);
  if (read_n == XFER_SIZE)
    /* Смещениями должны управлять мы */
    a_read.aio_offset += XFER_SIZE;
  else {
    /*
     * Для простоты предположим, что размер файла-источника
     * больше, чем XFER_SIZE
     */
    printf("Source file size < %d\n", XFER_SIZE);
    exit(1);
  }
}

/*
 * В этом цикле мы копируем данные, считанные выше, в буфер
 * записи и запускаем операцию асинхронной записи. Также,
 * думаем вперёд и помещаем в очередь запрос на чтение для
 * следующего цикла.
```

```
   */
  while(1){
    memcpy(write_buf, read_buf, read_n);

    /*
     * Настраиваем блок управления записи. Чтобы поместить в
     * очередь запрос на запись вызываем aio_write. Эта функция
     * запишет a_write.aio_nbytes байтов из буфера
     * a_write.aio_buf в файл a_write.aio_fildes со смещением
     * a_write.aio_offset. В случае успеха возвращает 0.
     */
    a_write.aio_nbytes = read_n;
    aio_write(&a_write);

    /* Помещаем в очередь следующий запрос на чтение */
    a_read.aio_nbytes = XFER_SIZE;
    aio_read(&a_read);

    /*
     * Перед началом обработки ожидаем завершения и чтения,
     * и записи
     */
    while((err_r = aio_error(&a_read)) == EINPROGRESS ||
          (err_w = aio_error(&a_write)) == EINPROGRESS){
      cblist[0] = &a_read;
      cblist[1] = &a_write;
      aio_suspend(cblist, 2, NULL);
    }

    /* Это конец? */
    if (quit)
      break;
    /* Увеличиваем на единицу указатель записи */
    a_write.aio_offset += aio_return(&a_write);
    /* Увеличиваем на единицу указатель чтения */
    read_n = aio_return(&a_read);
    if (read_n == XFER_SIZE)
      a_read.aio_offset += XFER_SIZE;
    else
      /* Это последний блок */
      quit = 1;
    }
  }

  /* Очистка */
  close(fd_r);
  close(fd_w);
}
```

7.4 Linux и режим жёсткого реального времени

Напомним, что стандартный Linux не обеспечивает гарантий предельного срока жёсткого реального времени. Для поддержки работы приложений жёсткого реального времени под

Linux добавлены некоторые расширения. Наиболее популярным является подход двойного ядра, в котором Linux рассматривается как низкоприоритетная задача реального времени. Базовую архитектуру подхода двойного ядра показывает Рисунок 7.5.

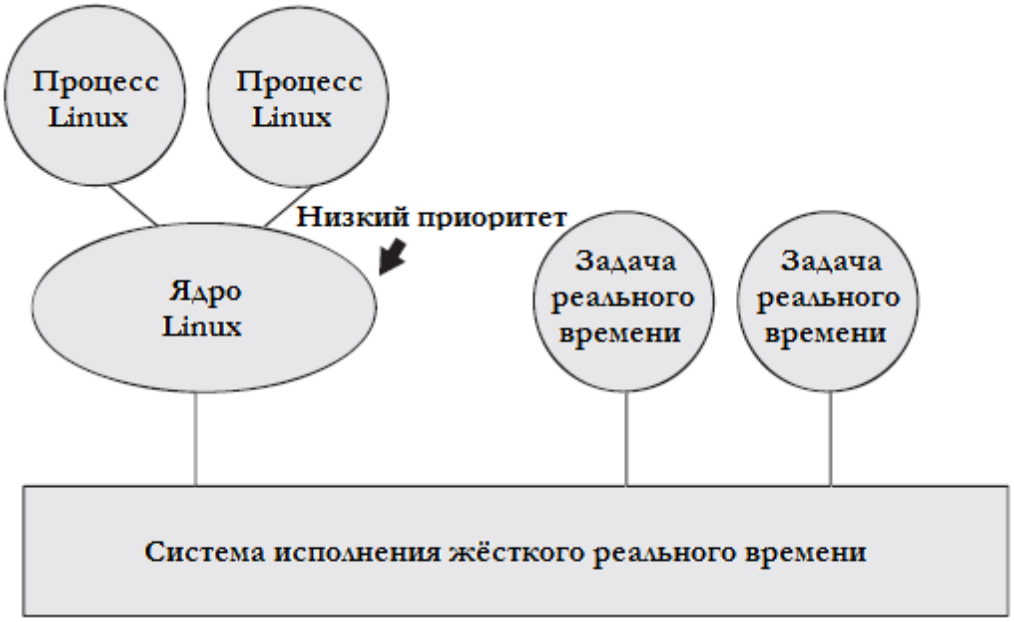

Рисунок 7.5 Архитектура двойного ядра.

В подходе двойного ядра Linux выполняется, только когда не работают задачи реального времени. Он никогда не сможет запретить прерывания или защитить себя от вытеснения. Аппаратные прерывания находится под контролем системы исполнения реального времени и прерывания отправляются в Linux только если они не забираются системой исполнения реального времени. Даже если Linux запрещает прерывание (с помощью **cli**), аппаратное прерывание не отключается. Система исполнения реального времени просто не будет посылать прерывания в Linux, если последний отключил его. Таким образом, система исполнения реального времени ведёт себя как "контроллер прерываний" для Linux. Linux никогда не добавит какой-либо задержки к времени отклика на прерывание системы исполнения реального времени. В этой конструкции Linux управляет всем, что исполняется не в режиме реального времени, например, ведением журнала, инициализацией системы, управлением оборудованием, не вовлечённым в обработки в реальном времени, и так далее.

Есть два основных варианта Linux жёсткого реального времени: RTLinux и RTAI. RTLinux была разработан в Технологическом Институте Нью-Мексико Михаилом Барабановым под руководством профессора Виктора Йодайкина. RTAI был разработан в Миланском Политехническом Департаменте Аэрокосмической Техники профессором Паоло Мантегацца. Оба этих варианта реализованы в виде модулей ядра Linux. Они похожи по своей природе тем, что если нет активных задач реального времени, все прерывания первоначально обрабатываются ядром реального времени, а затем передаются в Linux.

В этом разделе мы обсудим решение RTAI для предоставления поддержки в Linux жёсткого реального времени. В конце мы также очень кратко обсудим ADEOS, являющуюся основой для поддержки ядра реального времени и ОС общего назначения (например, Linux) на той же платформе.

7.4.1 Интерфейс приложения реального времени (RTAI)

RTAI (Real-Time Application Interface, Интерфейс Приложения Реального Времени) является расширением реального времени для Linux с открытым исходным кодом. Ядро RTAI представляет собой уровень абстрагирования оборудования (hardware abstraction layer, HAL) поверх которого может работать Linux и ядро жёсткого реального времени. HAL предоставляет механизм для улавливания прерываний от периферии и направляет их в Linux только если они не нужны для какой-либо обработки в режиме жёсткого реального времени. В рамках RTAI задача жёсткого реального времени может быть создана и запланирована с использованием интерфейсов RTAI. Для планирования задач реального времени RTAI использует свой собственный планировщик. Этот планировщик отличается от встроенного планировщика Linux. Для взаимодействия с другими задачами реального времени или обычными процессами Linux задача реального времени может использовать механизмы IPC, предоставляемые RTAI.

Для задач, запланированных в рамках RTAI, мы используем термин "задача реального времени", если не указано иное.

Архитектуру системы Linux на основе RTAI показывает Рисунок 7.6. (Для простоты мы не показали на схеме задачи LXRT.) Детали этой диаграммы станут понятны в следующий параграфах.

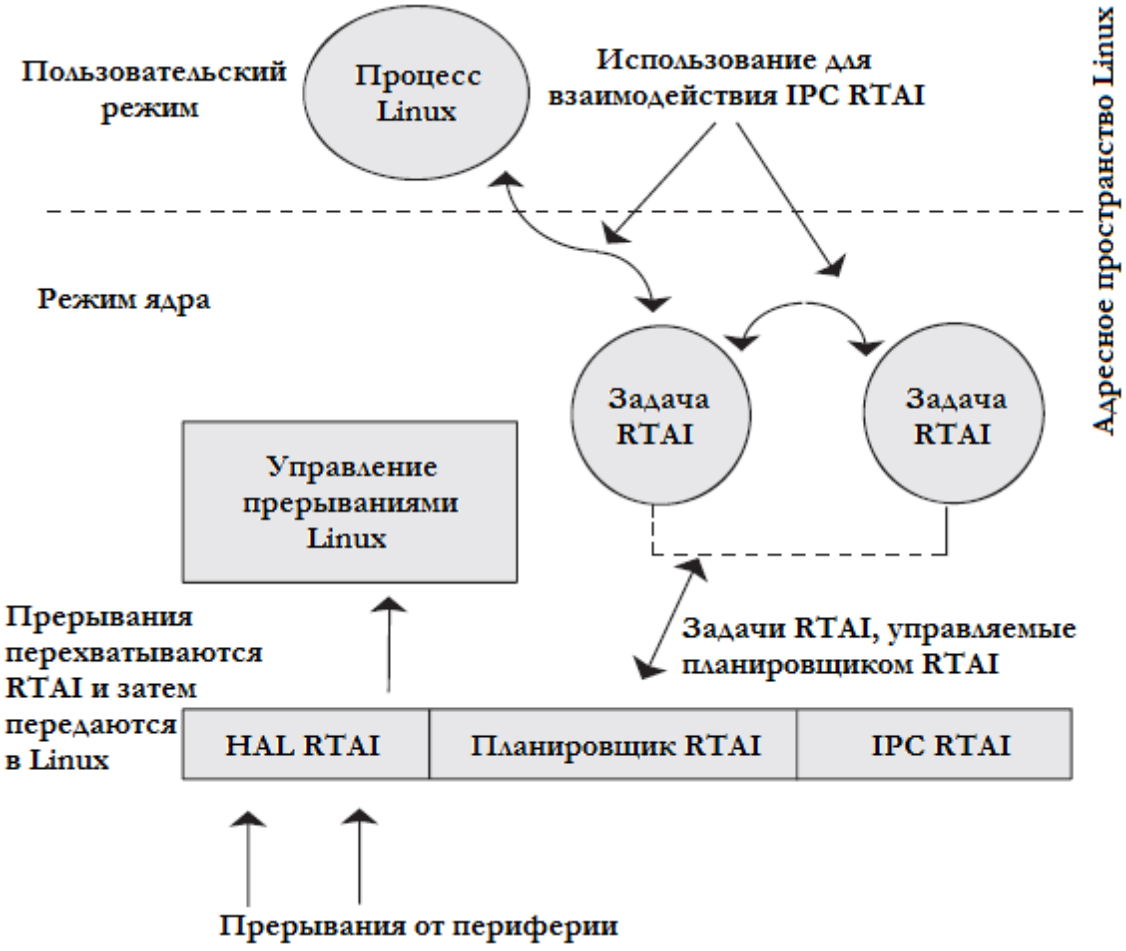

Рисунок 7.6 Архитектура RTAI.

RTAI поддерживает несколько архитектур, такие как x86, ARM, MIPS, PowerPC, CRIS, и другие. Модули RTAI, такие как планировщик, IPC, и так далее, не выполняются в отдельном адресном пространстве. Они реализованы в виде модулей ядра Linux; таким образом, они выполняются в адресном пространстве Linux. Не все модули ядра должны присутствовать всегда; основным модулем является **rtai** и он должен присутствовать всегда, а другие модули могут быть загружены динамически, когда это требуется. Например, **rtai_fifo** реализует функциональность FIFO RTAI и должен быть загружен только если требуется такая функциональность.

HAL

HAL перехватывает все аппаратные прерывания и направляет их либо стандартному Linux, либо задачам реального времени, в зависимости от требований планировщиков RTAI. Прерывания, предназначенные для запланированных в реальном времени задач, направляются непосредственно в такую задачу, в то время как прерывания, не требуемые какими-либо запланированными задачами реального времени, отправляются Linux. Таким образом, HAL предоставляет основу, на которую смонтирован RTAI, с возможностью полностью контролировать прерывания от периферии и вытеснять Linux. В RTAI существуют две реализации HAL:

- ***RTHAL (Real-Time Hardware Abstraction Layer, Уровень Абстрагирования Оборудования Реального Времени)***: он заменяет таблицы дескрипторов прерываний Linux своей таблицей для перехвата прерываний периферии.
- ***ADEOS***: В этой модели RTAI является более высокоприоритетным доменом, чем Linux.

Планировщики

Дистрибутив RTAI включает в себя четыре различных вытесняющих планировщика реального времени, работающих на основе приоритетов.

- ***UP***: планировщик, предназначенный для однопроцессорных платформ.
- ***SMP***: предназначен для машин с симметричной многопроцессорной обработкой.
- ***MUP***: предназначен для многопроцессорных машин без симметричной многопроцессорной обработки.
- ***NEWLXRT***: он объединяет три вышеописанных планировщика и может планировать задачи Linux и потоки ядра наряду с задачами RTAI ядра, расширяя таким образом интерфейс приложения жёсткого реального времени для задач Linux и потоков ядра.

Планировщики UP, MUP и SMP могут планировать только задачи RTAI ядра.

LXRT

Все задачи реального времени, запланированные под RTAI, работают в адресном пространстве ядра. Приложения RTAI загружаются как модули ядра. Первой функцией, которую они вызывают, является **rt_task_init**. **rt_task_init** создаёт задачу реального времени RTAI, которая становится запланированной планировщиком RTAI. Эта задача теперь может использовать все службы реального времени RTAI.

LXRT предоставляет службы RTAI реального времени в пространство пользователя. Таким образом, интерфейс RTAI может вызвать любой обычный процесс Linux. Это очень

мощный механизм, который устраняет разрыв между задачами реального времени RTAI и обычными процессами пользовательского пространства Linux. Когда процесс Linux вызывает **rt_task_init**, LXRT создаёт задачу ядра RTAI, которая действует как представитель процесса Linux. Этот представитель запускает службы RTAI от имени процесса Linux. Например, если процесс Linux вызывает **rt_sleep**, этот представитель выполнит эту функцию под управлением планировщика RTAI. Когда этот представитель возвращается из **rt_sleep**, управление возвращается процессу Linux.

IPC

RTAI поддерживает несколько механизмов IPC, которые могут использоваться для взаимодействия между задачами RTAI–RTAI и процессами RTAI–Linux. Они перечислены в Таблице 7.13.

Таблица 7.13 Механизмы IPC RTAI

IPC	*Описание*	*Модуль ядра*
Флаги событий	Используются для синхронизации задач при появления нескольких событий. Их использование аналогично использованию семафоров, за исключением того, что сигнал и операция ожидания зависят не от счётчика, а от набора событий, которые могут произойти.	**rtai_bits.o**
FIFO	Обеспечивают взаимодействие между приложениями пользовательского пространства Linux и задачами RTAI пространства ядра.	**rtai_fifos.o**
Совместно используемая память	Модуль позволяет совместно использовать память между задачами RTAI и процессами Linux.	**rtai_shm.o**
Семафоры	Семафоры RTAI поддерживают наследование приоритетов.	**rtai_sem.o**
Почтовый ящик (Mailbox)	Очень гибкий механизм IPC. Он может быть использован для отправки сообщений переменной длины между Linux и RTAI. RTAI также поддерживает типизированные почтовые ящики для широковещательной передачи сообщений и срочной доставки сообщений.	**rtai_mbx.o**, **rtai_tmbx.o** (типизированный mailbox)
NetRPC	Механизм передачи сообщений между задачами, который расширяет RTAI для работы в распределённом окружении.	**rtai_netrpc.o**
Pqueue	Pqueue RTAI реализует интерфейсы	**rtai_mq.o**

IPC	Описание	Модуль ядра
	очереди сообщений POSIX 1003.1b.	

Модули разного назначения

RTAI также предоставляет следующее:

- ***malloc реального времени (*rtai_malloc*)***: RTAI обеспечивает реализацию **malloc()** для режима реального времени. Задача RTAI можно смело выделять или освобождать память без какой-либо блокировки. Это достигается путём выделения памяти в режиме реального времени из заранее выделенной памяти в глобальной "куче".
- ***Микрозадачи или тасклеты (*rtai_tasklets*)***: иногда приложения должны выполнять какую-либо функцию периодически или когда происходят некоторые события. Использование для этой цели служб планировщика RTAI могло бы быть дорогостоящим с точки зрения используемых ресурсов. Для удовлетворения таких потребностей приложения, RTAI предоставляет микрозадачи. Они бывают двух типов:
  – ***Обычные микрозадачи***: это простые функции, выполняемые в пространстве ядра. Они вызываются либо из задачи реального времени, либо из обработчика прерываний.
  – ***Микрозадачи, привязанные ко времени***: это простые функции таймера, которые выполняются задачей сервера RTAI однократно или в периодическом режиме.
- ***Pthread (*rtai_pthread*)***: поддерживают потоки POSIX.1c (в том числе мьютексы и условные переменные).
- ***Сторожевой таймер (*rtai_wd*)***: этот модуль защищает RTAI и Linux от ошибок программирования в приложениях RTAI.
- ***Xenomai***: данная подсистема обеспечивает плавный переход от патентованной операционной системы (например, VxWorks, pSOS и VRTX™) к RTAI.
- ***Драйверы режима реального времени***: RTAI предоставляет драйверы реального времени последовательной линии и параллельного порта. Также поддерживается интерфейс драйвера устройства **comedi**. Проект **comedi** разрабатывает драйверы, наборы программ и библиотеки для карт сбора данных с открытым исходным кодом (http://www.comedi.org/).

Написание приложений RTAI

Задача реального времени под RTAI может быть либо модулем ядра, либо задачей LXRT пользовательского пространства. В этом разделе мы, используя простые примеры, объясняем оба подхода.

Задача RTAI, работающая как модуль ядра, состоит из трёх частей:

- ***Функция* module_init**: эта функция вызывается при загрузке модуля (с использованием **insmod** или **modprobe**). Следует выделить необходимые ресурсы, создать задачу реального времени и запланировать её выполнение.
- ***Код, относящийся к задаче реального времени***: он состоит из различных процедур, которые реализуют функциональность задачи реального времени.
- ***Функция* module_cleanup**: эта функция вызывается всякий раз, когда модуль ядра выгружается. Она должна уничтожить все выделенные ресурсы, остановить и удалить задачу реального времени, и тому подобное.

Структура задачи RTAI в виде модуля ядра показана в Распечатке 7.13 235. В этом

примере мы создаём периодическую задачу, которая печатает "Hello Real-Time World".

Как уже упоминалось, LXRT экспортирует интерфейсы RTAI в пространство пользователя. Для поддержки этого LXRT требует от задачи пользовательского пространства использовать политику планирования **SCHED_FIFO** с блокировкой всей их памяти (с помощью системного вызова **mlockall**). LXRT предлагает следующие преимущества по сравнению с задачей RTAI, работающей как модуль ядра:

- Для отладки LXRT задачи реального времени можно использовать средства отладки пользовательского пространства.
- Задачи LXRT реального времени подчиняются механизму защиты памяти Linux. Таким образом, ошибка в задаче не вызовет падение всей системы.
- Поскольку нет зависимости от ядра, задача может поставляться только в двоичном виде, без предоставления исходного кода.
- Для запуска задачи вам не нужно иметь права суперпользователя (эта поддержка осуществляется через интерфейс **rt_allow_nonroot_hrt**).

Чтобы проиллюстрировать структуру задачи реального времени LXRT пользовательского пространства, мы ещё раз написали наш пример "Hello Real-Time World". Пожалуйста, обратитесь к Распечатке 7.14 237.

Распечатка 7.13 Задача RTAI в виде модуля ядра

Распечатка 7.13.

```
/* rtai-kernel.c */

/* Заголовочный файл модуля ядра */
#include <linux/module.h>

/* Используемые интерфейсы RTAI находятся в
 * этих заголовочных файлах
 */
#include <rtai.h>
#include <rtai_sched.h>

/* Частота тиков таймера в наносекундах */
#define TIMER_TICK500000 /* 0.5 мс */

/* Структура задачи для нашей задачи реального времени */
static RT_TASK hello_task;

/* Будет вызываться при загрузке модуля */
int init_module(void)
{
  /* 'период' нашей периодической задачи */
  RTIME period;
  RTIME curr; /* текущее время */

  /*
   * rt_task_init создаёт задачу реального времени
   * в состоянии ожидания
   */
```

```
  rt_task_init(&hello_task,   /* структура задачи */
               hello_thread, /* функция задачи */
               0,            /* аргумент для функции задачи */
               1024,         /* размер стека */
               0,            /* Приоритет.
                                Наивысший приоритет ->0,
                                Нижайший ->RT_LOWEST_PRIORITY
                             */
               0,            /* Задача не использует FPU */
               0             /* Обработчика сигналов нет */
              );
  /*
   * Следующие две функции таймера предназначены для вызова
   * только один раз в начале всей работы в реальном времени.
   * Таймер фактически запустил "системные часы реального
   * времени". Таймер используется планировщиком как датчик
   * времени.
   */

  /*
   * Таймеры могут быть настроены в периодический режим или
   * режим однократного запуска
   */
  rt_set_oneshot_mode();

  /*
   * Тик таймера в наносекундах конвертируется во внутренние
   * единицы времени.
   */
  period = start_rt_timer(nano2count(TICKS));

  /* Получаем текущее время */
  curr = rt_get_time();

  /* Наконец, делаем эту задачу периодической */
  rt_task_make_periodic(&hello_task, /* Указатель на структуру
                                        задачи */
                        curr + 5*period, /* время старта
                                            задачи */
                        period // период задачи
                        );
  return 0;
}

void cleanup_module(void)
{
  /*
   * Останавливаем таймер, ждём какое-то время и, наконец,
   * удаляем задачу
   */
  stop_rt_timer();
  rt_busy_sleep(10000000);
  rt_task_delete(&hello_task);
}
```

```
/* Наш основной поток реального времени */
static void hello_thread(int dummy)
{
  while(1){
    rt_printk("Hello Real-time world\n");
    /* Ждём следующего периода */
    rt_task_wait_period();
  }
}
```

Распечатка 7.14 Процесс LXRT

Распечатка 7.14.

```
/* lxrt.c */

/*
 * Заголовки, которые определяют функции планирования
 * блокировки памяти
 */
#include <sys/mman.h>
#include <sched.h>

/* Заголовки RTAI */
#include <rtai.h>
#include <rtai_sched.h>
#include "rtai_lxrt.h"

#define TICK_TIME 500000

int main(){

  RT_TASK *task;
  RTIME period;
  struct sched_param sched;

  /* Создаём задачу SCHED_FIFO с максимальным приоритетом */
  sched.sched_priority = sched_get_priority_max(SCHED_FIFO);
  if (sched_setscheduler(0, SCHED_FIFO, &sched) != 0){
    perror("sched_setscheduler failed\n");
    exit(1);
  }

  /* Блокируем все текущие и будущие выделения памяти */
  mlockall(MCL_CURRENT | MCL_FUTURE);

  /* ---- module_init ---- */

  /*
   * rt_task_init создаёт для этой задачи представителя
   * реального времени ядре. Все интерфейсы RTAI будут
   * выполняться этим представителем под планировщиком RTAI от
```

```
   * имени этой задачи. Обратите внимание, что первый аргумент
   * имеет тип unsigned long. Строка может быть преобразована
   * в unsigned long с помощью функции nam2num.
   */
  if (!(task = rt_task_init(nam2num("hello"), 0, 0, 0))) {
    printf("LXRT task creation failed\n");
    exit(2);
  }

  rt_set_oneshot_mode();
  period = start_rt_timer(nano2count(TICK_TIME));
  /* Наконец, делаем нашу задачу периодической */
  rt_task_make_periodic(task, rt_get_time() + 5*period, period);

  /* --- Основная работа нашей задачи реального времени --- */

  count = 100;
  while(count--) {
    rt_printk("Hello Real-time World\n");
    rt_task_wait_period();
  }

  /* ---- cleanup_module ---- */

  stop_rt_timer();
  rt_busy_sleep(10000000);
  rt_task_delete(task);
}
```

7.4.2 ADEOS

ADEOS (Adaptive Domain Environment for Operating Systems, Адаптивная Доменная Среда для Операционных Систем) обеспечивает среду, которая даёт возможность совместного использования аппаратных ресурсов между несколькими операционными системами или между несколькими экземплярами одной операционной системе. Каждая ОС в ADEOS представлена в виде домена. Обработка прерываний является ключевым действием в среде ADEOS. Для обработки прерываний используется конвейер прерываний (interrupt pipeline). Каждый домен представлен в виде стадии конвейера. Прерывания распространяются по конвейеру от более высокоприоритетного домена к домену с более низким приоритетом. Домен может сделать выбор, чтобы принять, отказаться или прекратить обработку прерывания. Если домен принимает прерывание, ADEOS вызывает свой обработчик прерывания, а затем передаёт это прерывание следующему домену с более низким приоритетом в конвейере. Если домен отказывается от прерывания, прерывание просто передаётся конвейеру следующей стадии. Если домен прекращает обработку прерывания, прерывание далее по конвейеру не передаётся.

ADEOS и Linux

ADEOS может предоставить поддержку жёсткого реального времени в Linux. Под ADEOS могли бы быть реализованы два домена: один, охватывающий обычный Linux, и второй системы исполнения реального времени, которая обеспечивает гарантии жёсткого реального времени. RTAI уже использует ADEOS качестве своего HAL. Рисунок 7.7 показывает

конвейер прерываний ADEOS в RTAI.

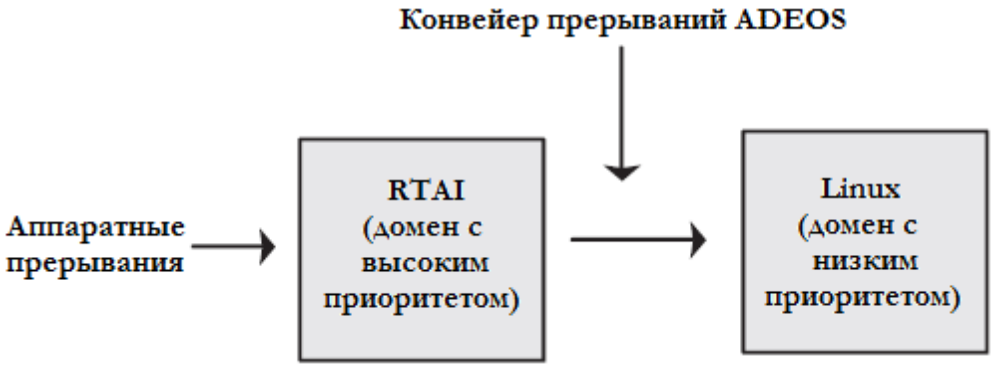

Рисунок 7.7 Конвейер прерываний ADEOS.

ADEOS также предоставляет среду для реализации в Linux отладчиков ядра и профайлеров. В рамках ADEOS отладчики ядра и профайлеры могут быть представлены как высокоприоритетный домен, а Linux как низкоприоритетный домен. Затем они могут легко контролировать поведение Linux захватом различных прерываний.

Глава 8, Сборка и отладка

Эта глава разделена на две части. Первая половина посвящена среде сборки Linux. Это включает в себя:

- Сборку ядра Linux
- Сборку приложений пользовательского пространства
- Сборку корневой файловой системы
- Обсуждение популярных интегрированных сред разработки (Integrated Development Environment, IDE)

Вторая половина посвящена техникам отладки и профилирования для встраиваемого Linux. Это включает в себя:

- Профилирование памяти
- Отладка ядра и приложения
- Профилирование приложения и ядра

Обычно традиционная RTOS собирает ядро и приложения в единый образ. В ней нет разграничения между ядром и приложениями. Linux предлагает совершенно иную парадигму сборки. Напомним, что в Linux каждое приложение имеет своё собственное адресное пространство, которое никак не связано с адресным пространством ядра. Пока используются подходящие файлы заголовков и библиотека языка Си, любое приложение может быть собрано независимо от ядра. В результате сборка ядра и сборка приложений совершенно не пересекаются.

Сборка ядра и приложения по отдельности имеет свои преимущества и недостатки. Основным преимуществом является то, что она проста в использовании. Если вы хотите добавить новое приложение, надо просто собрать это приложение и загрузить его на плату. Эта процедура проста и быстра. В этом отличие от большинства систем реального времени, где должен быть пересобран весь образ и система должна быть перезагружена. Тем не менее, основной недостаток процедуры раздельной сборки в том, что не существует автоматической взаимосвязи между функциями ядра и приложениями. Большинство разработчиков встраиваемых систем хотели бы иметь механизм сборки системы, когда конфигурация, выбранная для системы, отдельные компоненты (ядро, приложения и корневая файловая система), получается автоматически собранной вместе со всеми зависимостями. Однако, в Linux это не так. Сложность сборке добавляет необходимость сборки загрузчика и процесс упаковки корневой файловой системы в единый загружаемый образ.

Чтобы детально разобраться в этой проблеме, рассмотрим случай OEM поставщика, который поставляет два продукта: мост Ethernet и маршрутизатор, использующие одно и то же оборудование. Хотя большая часть программного обеспечения остаётся одной и той же (например, загрузчик, BSP, и тому подобное), различия в основных возможностях двух продуктов заключаются в программном обеспечении. В результате, OEM поставщик хотел бы сохранить единый базовый код для обоих продуктов, но чтобы программное обеспечение для системы собиралось в зависимости от выбранной системы (мост или маршрутизатор). Это, по сути, сводится к чему-то такому: по команде **make bridge** из каталога верхнего уровня должно быть выбрано программное обеспечение, необходимое для продукта "мост", и, аналогично, по команде **make router** должно быть собрано программное обеспечение для маршрутизатора. Для достижения этой цели придётся проделать много работы:

- Соответствующим образом должно быть сконфигурировано ядро и должны быть выбраны соответствующие протоколы (такие, как spanning bridge для моста или IP-forwarding для маршрутизатора), драйверы, и так далее.
- Соответствующим образом должны быть построены приложения пользовательского пространства (например, должна быть собрана служба маршрутизации).
- Должным образом должны быть настроены соответствующие файлы запуска (например, инициализации сетевого интерфейса).
- Должны быть выбраны и упакованы в корневую файловую систему соответствующие файлы конфигурации (например, HTML файлы и CGI скрипты).

Напрашивается вопрос: почему бы не положить программное обеспечение, необходимое и для моста, и для маршрутизатора в корневую файловую систему, а затем использовать драйверы и приложения в зависимости от того, как используется изделие? К сожалению, такой вариант требует лишнего расхода пространства хранения, которое не является роскошью для встраиваемых систем; поэтому целесообразным является выбор компонентов во время сборки. Настольные компьютеры и серверы могут себе это позволить; поэтому дистрибьюторы настольных и серверных систем редко заботятся об этом.

При выборе компонентов в процессе сборки необходима некоторая информация, так что может быть разработана платформа для сборки всей системы. Это может быть сделано путём разработки собственных скриптов и интеграции различных процедур сборки. В качестве альтернативы пользователь может оценить, подходят ли под его или её требования некоторые интегрированные среды разработки, доступные на рынке. Рынок IDE для Linux всё ещё находится в младенческой фазе и больше сконцентрирован на механизмах сборки ядра, просто потому, что сборка приложения варьируется в зависимости от приложения (нет стандартов, которым следуют при сборке приложений). Добавление ваших собственных приложений или экспорт зависимостей между приложениями просто не могут быть предложены многими средами разработки; даже если они предлагают это сделать, это может потребовать некоторых навыков. Интегрированные среды разработки обсуждаются в отдельном разделе. Если вы решили использовать IDE, пропустите раздел сборки и сразу переходите к разделу отладки. Но в случае, если вы планируете настраивать процедуры сборки, оставайтесь и читайте дальше.

8.1 Сборка ядра

Система сборки ядра (более популярный термин для неё это **kbuild**) поставляется вместе с исходниками ядра. Система **kbuild** основана на **make** от GNU; следовательно, все передаваемые команды предназначены для **make**. Механизм **kbuild** даёт весьма простую процедуру создания сборки ядра; сконфигурировать и собрать ядро и модули можно всего за несколько шагов. Также, она очень расширяема, в том смысле, что добавление своих собственных команд в процедуру сборки или настройка под себя процесса конфигурации является очень простым делом.

В процедуре **kbuild** ядра версии 2.6 произошли некоторые существенные изменения. Так что эта глава объясняет процедуры сборки ядра как версии 2.4, так и версии 2.6. Сборка ядра состоит из четырёх шагов:

1. ***Настройка среды кросс-разработки***: поскольку Linux имеет поддержку многих архитектур, процедура **kbuild** должна быть настроена на архитектуру, для которой собираются образ ядра и модули. По умолчанию среда сборки ядра собирает образы для

машины, на которой происходит сборка.

2. ***Процесс конфигурации***: это процедура выбора компонентов. С помощью этого шага может быть указан список какое программное обеспечение должно войти в ядро, а какое может быть скомпилировано в виде модулей. В конце этого шага **kbuild** записывает эту информацию в набор известных файлов, так что остальные части **kbuild** знают о выбранных компонентах. Объектами при выборе компонентов обычно являются:
   a. Выбор процессора
   b. Выбор платы
   c. Выбор драйверов
   d. Некоторые общие параметры ядра
   Для процедуры конфигурации существует много внешних интерфейсов; ниже перечислены те, которые могут быть использованы для ядер версии 2.4 и 2.6.
   a. **make config**: это сложный способ настройки, потому что это заставило бы выбирать компоненты на вашем терминале.
   b. **make menuconfig**: это внешний интерфейс на базе curses для процедуры **kbuild**, показанный на Рисунке 8.1. Его удобно использовать на машинах, которые не имеют доступа к графическому дисплею; однако, для его работы вы должны установить библиотеку разработки **ncurses**.
   c. **make xconfig**: это графический интерфейс для процесса конфигурации, показанный на Рисунке 8.2. Версия 2.4 использовала X, в то время как версия 2.6 использует QT. Есть ещё одна версия для 2.6, которая использует GTK и вызывается запуском **make gconfig**.
   d. **make oldconfig**: часто вы хотели бы сделать минимальные изменения в существующей конфигурации. Эта опция позволяет сборке сохранить параметры по умолчанию из существующей конфигурации и запрашивать только новые изменения. Эта опция очень полезна, когда вы хотите автоматизировать процедуру сборки с помощью сценариев.
3. ***Сборка объектных файлов и их компоновка, чтобы создать образ ядра***: после завершения выбора компонентов для сборки ядра необходимы следующие шаги:
   a. Для ядра версии 2.4 с помощью команды **make dep** должна быть создана информация о зависимостях из заголовочных файлов (какой файл **.c** зависит от какого файла **.h**). В ядре версии 2.6 в этом необходимости нет.
   b. Тем не менее, шаг очистки является общим для ядер обоих версий, 2.4 и 2.6; команда **make clean** очищает сборку от всех объектных файлов, образа ядра и всех промежуточных файлов, но сохраняет информацию о конфигурации. Существует ещё одна команда, которая делает всю это очистку вместе с очисткой конфигурации: это команда **make mrpoper**.
   c. Последним шагом является создание образа ядра. Если вы просто наберёте команду **make**, то на выходе получите образ ядра с названием **vmlinux**. Тем не менее, сборка ядра на этом не останавливается; как правило, выполняются некоторые постобработки, которые необходимо сделать, чтобы, например, сжать его, добавить код начальной загрузки, и так далее. Именно постобработка создаёт образ, который может быть использован на целевой платформе (постобработка не стандартизирована, поскольку она варьируется в зависимости от платформы и используемых загрузчиков).
4. ***Сборка динамически загружаемых модулей***: работу по созданию модулей выполнит команда **make modules**.

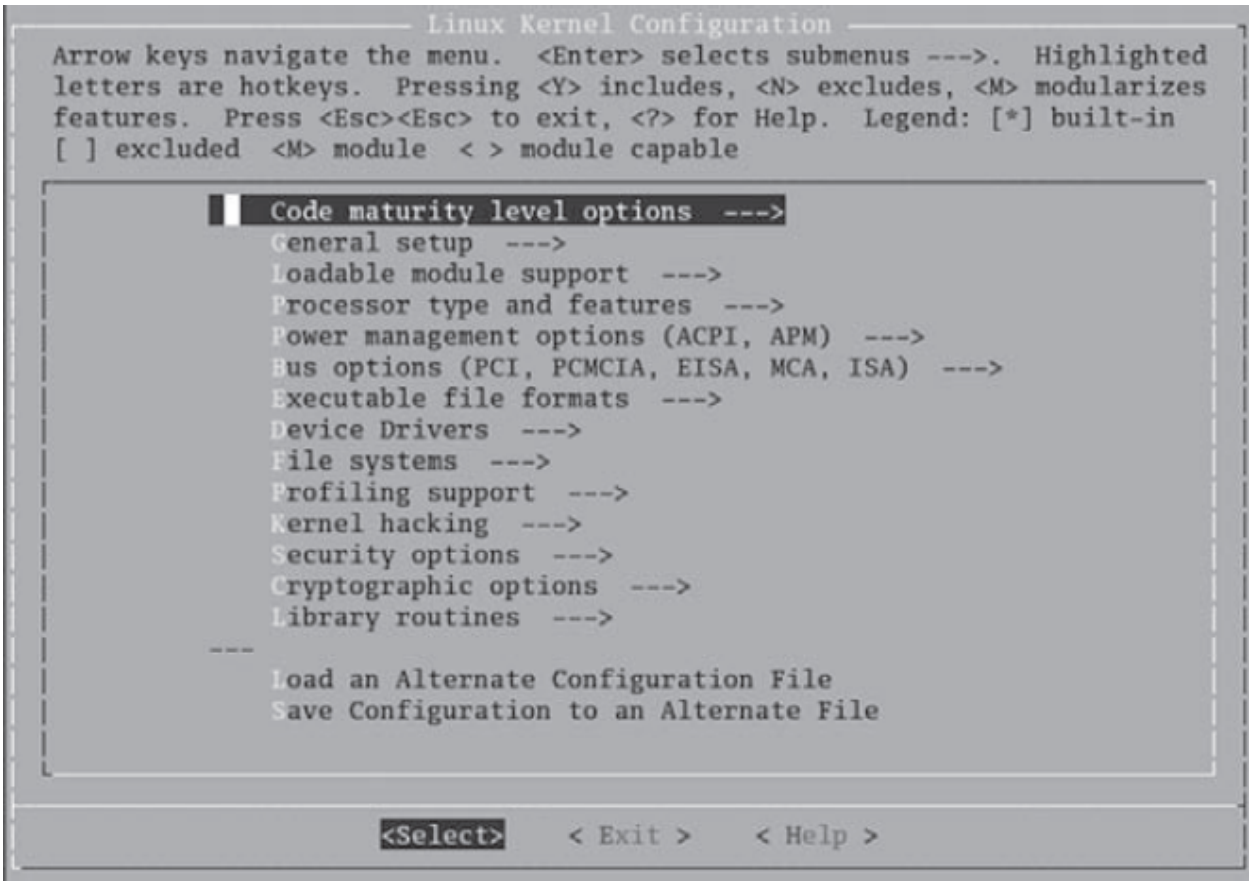

Рисунок 8.1 Конфигурация ядра на базе curses.

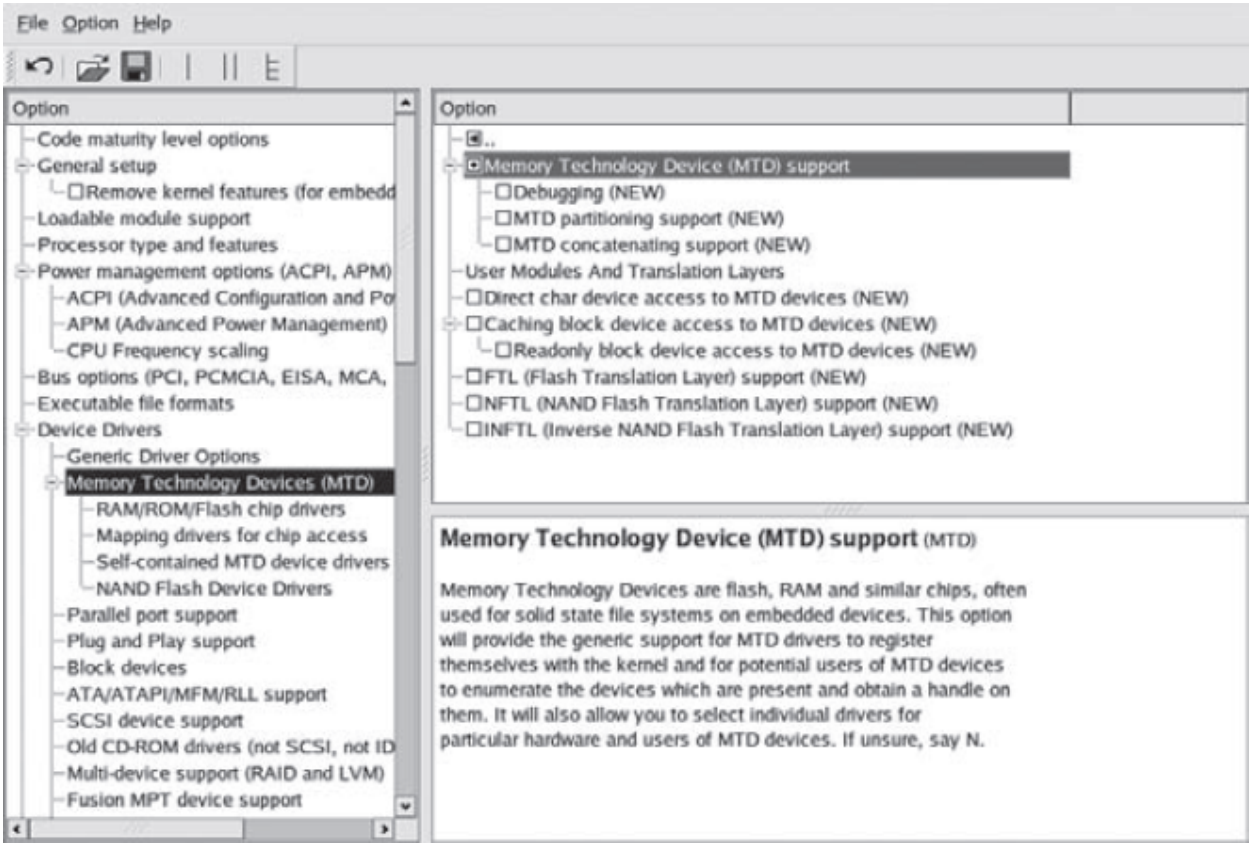

Рисунок 8.2 Конфигурация ядра на базе X.

Чтобы использовать **kbuild** для сборки ядра, конечному пользователя достаточно перечисленных выше команд. Однако, в случае встраиваемых системах вы бы захотели более глубокой настройки процесса сборки для себя; некоторые причины указаны ниже.

- Вы можете захотеть добавить свой BSP в отдельный каталог и изменить конфигурацию так, чтобы **kbuild** собирал компоненты программного обеспечения, необходимые для вашей платы.
- Вы можете захотеть добавить к процессу сборки свой собственный компоновщик, компилятор и флаги ассемблера.
- Вы можете захотеть настроить постобработку образа ядра, после того, как оно будет собрано.
- Вы можете захотеть добавить в **kbuild** дополнительные возможности выбора, чтобы выполнять сборку всей системы.

По этим причинам в следующем разделе будем вдаваться в тонкости процесса сборки.

8.1.1 Как работает процедура сборки

Ниже описаны характерные черты процедуры **kbuild** для ядер версии 2.4 и 2.6.

- За сборку образа ядра и модулей отвечает высокоуровневый **Makefile** в исходных текстах ядра. Это достигается рекурсивным спуском в подкаталоги дерева исходных текстов ядра. Список подкаталогов, в которые необходимо зайти, зависит от выбора компонентов, то есть от процедуры конфигурации ядра. Как именно это делается,

объясняется позже.**Makefile**-ы, находящиеся в подкаталогах, наследуют правила сборки объектов; в версии 2.4 они делают это подключая файл правил под названием **Rules.make**, который должен быть явно включён в каждый **Makefile**, находящийся в подкаталоге. Однако, это требование было исключено в процедуре **kbuild** версии 2.6.

- Каждая архитектура (вариант для какого-либо процессора) должна экспортировать список компонентов для выбора в процессе конфигурирования; это включает в себя:
  – Выбор разновидности процессора. Например, если ваша архитектура определена как ARM, то вам будет предложено выбрать вариант ARM.
  – Оборудование на плате.
  – Конфигурацию зависимого от платы оборудования.
  – Компоненты подсистем ядра, которые остаются более или менее одинаковыми для всех архитектур, например, сетевой стек.
  Каждая архитектура поддерживает базу данных компонентов в виде файлов; он может быть найден в каталоге **arch/$ARCH**. В ядре версии 2.4 этот файл называется **config.in**, тогда как в ядре версии 2.6 этот файл называется **Kconfig**. Во время конфигурации ядра этот файл анализируется и пользователю предлагается для выбора список компонентов. Возможно, вам придётся добавить в этот файл конфигурацию, относящуюся к вашему оборудованию.
- Каждая архитектура должна экспортировать архитектурно-зависимый **Makefile**; следующий список сборочной информации является уникальным для каждой архитектуры.
  – Флаги, которые должны быть переданы различным инструментам
  – Подкаталоги, которые необходимо посетить для сборки ядра
  – Шаги постобработки, выполняемые после того, как образ собран
  Они поставляются в архитектурно-зависимом **Makefile** в подкаталоге **arch/$(ARCH)**. **Makefile** верхнего уровня импортирует архитектурно-зависимый **Makefile**. Чтобы понять архитектурно-зависимые определения для сборки, рекомендуем читателю заглянуть в какой-нибудь архитектурно-зависимый файл в дереве исходных текстов ядра (например, **arch/mips/Makefile**).

Ниже приводятся некоторые основные различия между процедурами сборки ядра версий 2.4 и 2.6.

- Механизм конфигурации и сборки версии 2.6 имеет другую основу. **kbuild** версии 2.6 гораздо проще. Например, в ядре версии 2.4 архитектурно-зависимый **Makefile** не следует никакому стандарту; так что он отличается для различных архитектур. Платформа версии 2.6 была исправлена, чтобы поддерживать однородность.
- В версии 2.4 простой набор **make** привёл бы в конечном итоге к разным результатам, в зависимости от состояния процедуры сборки. Например, если пользователь не выполнил конфигурирование и ввёл **make**, **kbuild** бы вызвал **make config**, выдавая вопросы на терминал и приводя пользователя в замешательство. В версии 2.6, однако, это приведёт к ошибке с надлежащей помощью для инструктирования пользователя.
- В версии 2.4 объектные файлы создаются в том же каталоге, где находятся и исходные файлы. Однако, версия 2.6 позволяет иметь разные деревья исходных текстов и выходных объектных файлов (включая вывод конфигурации); это делается командой **make O=dir**, где **dir** - дерево объектных файлов.
- В версии 2.4 при выполнении **make dep** оказывается воздействие на исходные файлы (то есть изменяются их временные метки). Это вызывает проблемы с некоторыми системами управления исходным кодом. С другой стороны, в ядре версии 2.6 во время сборки ядра исходные файлы остаются нетронутыми. Это гарантирует, что вы можете

иметь дерево исходных текстов, доступное только для чтения. Это экономит дисковое пространство, если несколько пользователей хотят совместно использовать одно дерево исходных текстов, но иметь свои отдельные деревья объектных файлов.

8.1.2 Процесс конфигурирования

Хотя процесс настройки вызывается с помощью команды **make**, была определена отдельная грамматика конфигурации. Она также различается в версиях 2.4 и 2.6. Обратите внимание, что эта грамматика проста и близка к разговорному английскому языку; так что понять её вам может помочь простой взгляд на конфигурационные файлы (**Kconfig** для ядра версии 2.6 и файлы **Config.in** для ядра версии 2.4). Этот раздел не вдаётся в подробности грамматики; скорее, он нацелен на используемые методы. (* В версиях 2.4 и 2.6 методы остались почти теми же.)

- Каждый подраздел ядра определяет правила для конфигурации в отдельном файле. Например, конфигурация сети хранится в файле **Config.in** (для ядра версии 2.4) или **Kconfig** (для ядра версии 2.6) в каталоге исходных текстов ядра **net/**. Этот файл импортируется архитектурно-зависимым файлом конфигурации. Например, в версии 2.4 конфигурационный файл для архитектуры MIPS **arch/mips/config-shared.in** имеет строку для импорта правил конфигурации для исходного файла VFS (**fs/config.in**).
- Элемент конфигурации хранится в виде пары **имя=значение**. Имя элемента конфигурации начинается с префикса **CONFIG_**. Остальная часть имени является именем компонента, определённым в файле конфигурации. Ниже приведены значения, которые может иметь переменная конфигурации:
  – **bool**: переменная конфигурации может иметь значение **y** или **n**.
  – **tristate**: здесь переменная может иметь значения **y**, **n** или **m** (для модуля).
  – **string**: здесь может быть задана любая строка ASCII символов. Например, в случае, если вы должны передать адрес сервера NFS, с которого хотите смонтировать начальную корневую файловую систему, он может быть задан во время сборки с помощью переменной, которая содержит строковое значение.
  – **integer**: переменной может быть присвоено любое десятичное число.
  – **hexadecimal**: переменной может быть присвоено любое шестнадцатеричное число.
- При определении переменной конфигурации может быть указано, должен ли пользователь получить запрос для присвоения значения этой переменной. Если нет, этой переменной присваивается значение по умолчанию.
- При определении переменной могут быть созданы зависимости. Зависимости используются для определения видимости записи.
- Каждая переменная конфигурации может иметь связанный с ней текст помощи. Он отображается во время конфигурации. В ядре версии 2.4 весь текст помощи хранится в одном файле **Documentation/Configure.help**; текст помощи, связанный с определённой переменной, хранится после имени переменной. Тем не менее, в ядре версии 2.6 он хранится в отдельных файлах **Kconfig**.

Теперь мы подошли к последней, но самой важной части. Это понимание того, как процесс конфигурации экспортирует список выбранных компонентов для остальной части **kbuild**. Для достижения этого создаётся файл **.config**, который содержит список выбранных компонентов в формате **имя = значение**. Файл **.config** хранится в основном каталоге ядра и включён в **Makefile** верхнего уровня. Поле значения компонента используется, чтобы оценить, является ли исходный файл кандидатом для сборки и узнать, должен ли быть

собран компонент (как модуль или напрямую скомпонован с ядром). Для этого **kbuild** использует умный метод. Давайте предположим, что в каталоге **drivers/net** существует драйвер **sample.c**, который экспортируется в процесс конфигурации под именем **CONFIG_SAMPLE**. Во время настройки с помощью команды **make config** пользователю будет предложено:

```
Build sample network driver (CONFIG_SAMPLE) [y/N]?
```

Если выбрать **y**, то в файл **.config** будет добавлено **CONFIG_SAMPLE=y**. В **drivers/net/Makefile** будет строка

```
obj-$(CONFIG_SAMPLE)+= sample.o
```

Когда при рекурсии в подкаталоге **drivers/net** встретится этот **Makefile**, **kbuild** преобразует эту строку к виду

```
obj-y+= sample.o
```

Так произойдёт потому, что включённый файл **.config** определил **CONFIG_SAMPLE=y**. Сборка ядра имеет правило для сборки **obj-y**; следовательно, этот исходный файл выбран для сборки. Аналогичным образом, если эта переменная выбрана в качестве модуля, то во время сборки модулей эта строка будет выглядеть следующим образом:

```
obj-m+= sample.o
```

Снова правило сборки **obj-m** определяет **kbuild**. Исходный код ядра тоже должен быть осведомлён о списке выбранных компонентов. Например, в коде **init/main.c** ядра версии 2.4 есть следующие строки:

```
#ifdef CONFIG_PCI
  pci_init();
#endif
```

Если пользователь выбрал во время конфигурации PCI, должен быть определён макрос **CONFIG_PCI**. Для того, чтобы сделать это, **kbuild** преобразует пару **имя=значение** в макроопределение в файле **include/linux/autoconf.h**. Этот файл разбивается на набор файлов заголовков в каталоге **include/config**. Например, в приведенном выше примере был бы файл **include/config/pci.h**, имеющий строку:

```
#define CONFIG_PCI
```

Таким образом, механизм **kbuild** гарантирует, что исходные файлы тоже могут быть осведомлены о компонентах.

8.1.3 Платформа Makefile в ядре

Чтобы понять основы **Makefile** ядра, рассмотрим пример **Makefile** драйвера. Для этого возьмём **drivers/net/Makefile**. Сначала рассмотрим **Makefile** версии 2.4, а затем версии 2.6.

Распечатка 8.1 248 показывает в упрощённом виде **drivers/net/Makefile** в Linux 2.4.

Первые четыре переменные имеют особое значение. В **obj-y** находится список объектов, которые непосредственно встраиваются в ядро. В **obj-m** находится список объектных файлов, которые собираются в виде модулей. Две другие переменные просто игнорируются в процессе сборки.

O_TARGET является целью (то есть результатом работы) для этого **Makefile**; финальный образ ядра создаётся при сборке всех файлов **O_TARGET** из с различных подкаталогов. Правила для упаковки всех объектных файлов в этот файл, указанный **O_TARGET**, определяются **$TOPDIR/Rules.make** (* TOPDIR является переменной сборки и используется для получения основного каталога исходных кодов ядра, чтобы сборка не зависела от расположения базового каталога.), который явно включается в **Makefile**. Файл **net.o** добавляется в окончательный образ ядра Makefile-ом верхнего уровня.

Специальному объектному файлу, называемому многоэлементным объектом (multipart object), процессом **make** задано специальное правило. Многоэлементный объект создан с помощью нескольких объектных файлов. Одноэлементый объект не требует специального правила; механизм сборки выбирает исходный файл для сборки, заменяя часть **.o** целевого объекта на **.c**. С другой стороны, при сборке многоэлементного объекта должен быть указан список объектов, которые составляют многоэлементный объект. Этот список многоэлементных объектов определяется в переменной **list-multi**. Для каждого имени, которое появляется в этом списке, эта переменная получается путём добавления строки **-objs** к имени получаемого списка объектов, необходимого для создания многоэлементного модуля.

Наряду с **obj-$(...)**, ядро версии 2.4 должно с помощью **subdir-$(...)** указать список подкаталогов для обхода. Снова, то же самое правило, которое применяется для **obj-***, справедливо также для подкаталоги (то есть **subdir-y** используется для просмотра списка каталогов при сборке образа ядра, в то время как **subdir-m** используется для обхода при сборке модулей). Наконец, мы подошли к переменной **export-objs**. Это список файлов, который может экспортировать символы.

Makefile ядра версии 2.6, показанный в Распечатке 8.2 249, гораздо проще.

Основными отличиями в процедуре сборки версии 2.6 по сравнению с процедурой сборки версии 2.4 являются:

- Не нужно привлекать **Rules.make**; правила для сборки экспортируются в неявном виде.
- **Makefile** не указывает имя цели, потому что есть установленный для сборки целевой **built-in.o**. Чтобы собрать образ ядра, компонуются **built-in.o** из различных подкаталогов.
- Список подкаталогов, которые должны быть посещены, использует ту же переменную **obj-*** (в отличие от версии 2.4, где используется переменная **subdirs-***).
- Объекты, которые экспортируют символы, не должны быть упомянуты специальным образом (чтобы узнать эту информацию, процесс сборки использует макрос **EXPORT_SYMBOL**, встречаемый им в исходном коде).

Распечатка 8.1 Пример Makefile в ядре версии 2.4

Распечатка 8.1.

```
obj-y :=
obj-m :=
obj-n :=
obj-  :=
```

```
mod-subdirs   := appletalk arcnet fc irda … wan

O_TARGET      := net.o

export-objs   := 8390.o arlan.o … mii.o

list-multi    := rcpci.o
rcpci-objs    := rcpci45.o rclanmtl.o

ifeq ($(CONFIG_TULIP),y)
  obj-y+= tulip/tulip.o
endif

subdir-$(CONFIG_NET_PCMCIA)+= pcmcia
…
subdir-$(CONFIG_E1000)     += e1000
obj-$(CONFIG_PLIP)         += plip.o
…
obj-$(CONFIG_NETCONSOLE)   += netconsole.o

include $(TOPDIR)/Rules.make
clean:
 rm -f core *.o *.a *.s

rcpci.o : $(rcpci-objs)
 $(LD) -r -o $@ $(rcpci-objs)
```

Распечатка 8.2 Пример Makefile в ядре версии 2.6

Распечатка 8.2.

```
rcpci-objs:= rcpci45.o rclanmtl.o

ifeq ($(CONFIG_ISDN_PPP),y)
  obj-$(CONFIG_ISDN) += slhc.o
endif

obj-$(CONFIG_E100) += e100/
…
obj-$(CONFIG_PLIP) += plip.o
…
obj-$(CONFIG_IRDA) +=irda
```

8.2 Сборка приложений

Теперь, поняв процедуру сборки ядра, мы переходим к сборке программ пользовательского пространства. Эта область очень разнообразна; здесь может быть бесчисленное множество механизмов сборки, используемых отдельными пакетами. Однако, большая часть программ с открытым кодом для конфигурации и сборки следует общепринятому методу. Учитывая большое количество программ с открытым исходным кодом, которые могут быть развёрнуты на встраиваемых системах, понимание этой темы может облегчить перенос

общедоступных программ с открытым кодом на вашу целевую плату. Также вы захотели бы настроить процедуру сборки, чтобы убедиться, что для сборки программы не выбраны нежелательные компоненты; это гарантирует, что ваше драгоценное дисковое пространство не будет потрачено на хранение ненужных программ.

Как и ядро, приложения также должны быть собраны с использованием инструментов кросс-разработки. Большинство программ с открытым кодом следуют стандарту сборки GNU. Система сборки GNU решает следующие вопросы переносимости:

- Аппаратные различия, такие как порядок байтов, размеры типов данных и так далее
- Различия ОС, такие как соглашения об именовании файлов устройств и так далее
- Различия в библиотеках, такие как номер версии, аргументы API, и так далее
- Различия в компиляторах, такие как имя компилятора, аргументы и так далее

Инструменты GNU для сборки представляют собой набор из нескольких инструментов, наиболее важные из которых перечислены ниже.

- **autoconf**: служит общей основой переносимости, основываясь на тестировании функций базовой системы на этапе сборки. Это делается проведением тестов для обнаружения особенностей базовой машины.
- **automake**: это система для описания того, как собрать программу, что позволяет разработчику писать упрощённый **Makefile**.
- **libtool**: это стандартизированный подход к построению общих библиотек.

Следует отметить, что понимание этих инструментов является главной задачей только если разработчик приложения намерен создать приложение, которое будет использоваться на различных платформах, включая различные аппаратные архитектуры, а также различные платформы UNIX, такие как Linux, FreeBSD и Solaris. С другой стороны, если читатель заинтересован только в кросс-компиляции приложения, то всё, что нужно сделать, это ввести в командной строке следующую команду:

```
# ./configure
# make
```

В этой главе мы вкратце обсудим различные части, которые помогают скрипту конфигурации сгенерировать файлы **Makefile**, необходимые для компиляции программы. Также мы предоставляем рекомендации по устранению неполадок и работе с некоторыми общими проблемами, которые возникают при использовании **configure** для кросс-компиляции. Однако, как написать сценарий конфигурирования для программы не обсуждается и не входит в рамки этой книги. Если читатель заинтересован в написании скрипта конфигурирования, обратитесь, пожалуйста, к системе конфигурирования GNU на www.gnu.org.

Все программы, которые используют систему конфигурирования сборки от GNU, поставляют скрипт оболочки, называемый **configure**, и несколько файлов поддержки, а также исходные тексты программ. Любой проект для Linux, который использует систему конфигурирования сборки от GNU, требует для процесса сборки этот набор вспомогательных файлов. Наряду с набором файлов, который сопровождает данный дистрибутив статически, есть файлы, создаваемые динамически во время процесса сборки. Ниже описаны оба эти набора файлов.

Файлы, которые входят в состав дистрибутива, включают **configure**, **Makefile.in** и **config.in**. **configure** это сценарий оболочки. Чтобы увидеть различные параметры, которые

он принимает, используйте **./configure --help**. Сценарий **configure** по сути содержит ряд программ и тестов, которые будут выполнены на базовой системе, на основе которых изменяются входные данные для сборки. Чтобы читатель понял типы тестов, выполняемые **configure**, ниже перечислены некоторые обычно выполняемые проверки.

- Проверка на существование заголовочных файлов, таких как **stdlib.h**, **unistd.h**, и так далее
- Проверка на наличие интерфейсов библиотеки, таких как **strcpy**, **memcpy**, и так далее
- Получение размера типа данных, такого как **sizeof(int)**, **sizeof(float)**, и так далее
- Проверка/определение размещения наличия других внешних библиотек, необходимых для программы. Например, **libjpeg** для поддержки JPEG, или **libpng** для поддержки PNG
- Проверка, соответствуют ли номера версий библиотек

Обычно есть зависимости, которые делают программу зависящей от системы. Создание скрипта **configure**, осведомлённого об этих зависимостях, будет гарантировать, что программа станет переносимой между платформами UNIX. Для выполнения вышеперечисленных задач **configure** использует два основных метода:

- ***Пробная сборка тестовой программы***: это используется там, где **configure** должен найти наличие заголовка или API, или библиотеки. Чтобы обнаружить присутствие **stdlib.h**, **configure** просто использует простую программу, похожую на показанную ниже.

  ```
  #include <stdlib.h>
  main() {
    return 0;
  }
  ```

  Если вышеописанная программа компилируется успешно, это свидетельствует о наличии годного к употреблению **stdlib.h**. Подобные тесты проводятся и для обнаружения присутствия API и библиотек.
- ***Выполнение тестовой программы и получение выходного значения***: в ситуациях, когда **configure** должен получить размер типа данных, единственным доступным методом является компиляция, выполнение и получение результата работы программы. Например, чтобы найти размер целого числа на платформе, выполняется приведённая ниже программа

  ```
  main() {
    return sizeof(int)
  };
  ```

Результат работы тестов/программ, выполняемых **configure**, обычно хранится в виде конфигурационного (препроцессорного) макроса **config.h**, и, если его создание было успешно завершено, **configure** начинает создание Makefile-ов. Эти конфигурационные макросы могут быть использованы в коде для выбора части кода, необходимого для конкретной платформы UNIX. Скрипт конфигурации принимает много входных аргументов; их можно узнать, запустив **configure** с параметром **-help**.

Для создания во время сборки **Makefile**, скрипт **configure** работает с **Makefile.in**. В каждом подкаталоге программы будет один такой файл. Конфигурационный скрипт также

преобразует **config.in** в **config.h**, который изменяет **CFLAGS**, определённые для компиляции. Определение **CFLAGS** зависит от базовой системы, на которой выполняется процесс сборки. Большинство вопросов переносимости решается с помощью макросов препроцессора, которые получают определения в этом файле.

Файлы, создаваемые во время сборки приложения:

- **Makefile**: это файл, который будет использовать **make** для сборки программы. **Makefile.in** в **Makefile** преобразует конфигурационный скрипт.
- **config.status**: скрипт **configure** создаёт файл **config.status**, который является сценарием оболочки. Он содержит правила для повторной генерации генерируемых файлов и запускается автоматически, когда изменяется любой из входных файлов. Например, возьмём случай, когда у вас уже есть предварительно сконфигурированная директория для сборки (то есть такая, в которой хотя бы один раз был выполнен скрипт конфигурации). Теперь, если вы измените **Makefile.in**, то когда вы просто выполните команду **make**, получите созданные автоматически Makefile-ы. Повторная генерация происходит с помощью этого скрипта, без обращения к скрипту **configure**.
- **config.h**: этот файл определяет конфигурацию макросов препроцессора, которую код языка Си может использовать для изменения своего поведения на различных системах.
- **config.cache**: **configure** кэширует в этом файле результаты работы между запусками скрипта. В этом файле сохраняются выходные результаты различных этапов настройки. Каждая строка имеет вид присвоения **переменная = значение**.Переменная представляет собой имя, созданное сценарием, которое используется **configure** во время сборки. Прежде чем приступить к фактической проверке базовой машины, конфигурационный скрипт зачитывает значения переменных в этом файле в память.
- **config.log**: он хранит выходные данные работы сценария конфигурирования. Опытные пользователи **configure** могут использовать этот скрипт, чтобы обнаружить проблемы с процессом конфигурации.

8.2.1 Кросс-компиляция с помощью configure

Наиболее общая форма использования **configure** для кросс-компиляции такая:

```
# export CC=<target>-linux-gcc
# export NM=<target>-linux-nm
# export AR=<target>-linux-ar
# ./configure --host=<target> --build=<build_system>
```

<build_system> это система, на которой выполняется сборка, чтобы создать программы, которые работают на **<target>**. Например, для настольного компьютера Linux/i686 и целевой платформы ARM, **<build_system>** это **i686-linux**, а **<target>** это **arm-linux**.

```
# export CC=arm-linux-gcc
# export NM=arm-linux-nm
# export AR=arm-linux-ar
# ./configure --host=arm-linux --build=i686-linux
```

Флаг **--build** должен указываться не всегда. В большинстве случаев скрипт **configure** сделает подходящее предположение о системе, на которой происходит сборка.

Обратите внимание, что не всегда необходимо, чтобы работа **configure** для кросс-компиляции была успешной при первой попытке. Самой распространённой ошибкой во время кросс-компиляции является

```
configure: error: cannot run test program while
                  cross compiling
```

Эта ошибка возникает из-за того, что **configure** пытается запустить какую-то тестовую программу и получить результат её работы. Если это кросс-компиляция, то в этом случае тестовая программа скомпилирована в исполняемый файл для целевой платформы и не может работать на системе, на которой происходит сборка.

Чтобы решить эту проблему, изучите вывод скрипта конфигурации, чтобы определить тест, который даёт ошибку. Чтобы получить более подробную информацию об ошибке, откройте файл **config.log**. Например, предположим, что вы запустили настройку и получили ошибку:

```
# export CC=arm-linux-gcc
# ./configure --host=arm-linux

     …
checking for fcntl.h... yes
checking for unistd.h... yes
checking for working const... yes
checking size of int...
configure: error: cannot run test program while
                  cross compiling
```

В приведенном выше примере **configure** пытается найти размер **int**. Для того, чтобы найти размер целого числа на целевой системе, он компилирует программу вида **main() { return (sizeof(int))}**. Выполнение программы будет неудачным, так как система, на которой происходит сборка, не соответствует целевой системе.

Для решения таких проблем вам необходимо отредактировать файл **config.cache**. Напомним, что перед началом проверки **configure** считывает значения из файла **config.cache**. Всё, что вам необходимо сделать, это найти тестовую переменную в скрипт **configure** и добавить эту запись в желаемом виде в файл **config.cache**. В приведённом выше примере предположим, что размер целого числа в скрипте **configure** определяет переменная **ac_sizeof_int_set**. Затем добавляем в **config.cache** следующую строку:

```
ac_sizeof_int_set=4
```

После этого изменения вывод **configure** теперь такой:

```
    …
checking for fcntl.h... yes
checking for unistd.h... yes
checking for working const... yes
checking size of int...(cached) yes
    …
```

8.2.2 Поиск и устранение проблем в конфигурационном скрипте

Теперь, когда у нас есть представление о том, что делает конфигурационный скрипт, попробуем посмотреть, что может пойти не так. Есть два сигнала о проблеме. Один из них, когда конфигурационный скрипт корректен, а ваша система действительно не соответствует условию. Чаще всего это будет правильно диагностировано конфигурационным скриптом. Более тревожный случай, когда скрипт конфигурации является некорректным. Это может привести либо к неспособности создавать конфигурацию, или к созданию неправильной конфигурации. В первом случае, когда конфигурационный скрипт обнаруживает, что есть несоответствие условию, обычно большинство конфигурационных скриптов достаточно хороши, чтобы вывести подходящее сообщение об ошибке, описывающее требуемую версию отсутствующего компонента. Всё, что мы должны сделать, это установить этот необходимый отсутствующий компонент и перезапустить скрипт конфигурации. Ниже приведены некоторые советы по устранению проблем, связанных с конфигурационным скриптом.

- ***Прочитайте* README *и посмотрите параметры настройки в* ./configure --help**: там может быть какие-нибудь специальный параметр для указания пути к какой-то зависимой библиотеке, который если не указан, может по умолчанию содержать неправильную информацию о пути.
- ***Изобразите графически дерево зависимостей***: будьте внимательны при чтении проектной документации и запишите зависимые библиотеки и требования к номерам версий. Это позволит вам сэкономить много времени. Например, библиотека GTK зависит от библиотеки GLIB, которая зависит от библиотек ATK и PANGO. Библиотека PANGO, в свою очередь, зависит от библиотеки FREETYPE. Лучше иметь удобную диаграмму зависимостей, чтобы собрать и установить независимые узлы (библиотеки) в дерево, а затем скомпилировать родителя (библиотеку).
- ***Пробный запуск на ix86***: иногда перед кросс-компиляцией для понимания работы скрипта и его зависимостей может быть полезным выполнение скрипта конфигурации на x86.
- ***Учитесь читать config.log***: когда запускается конфигурационный скрипт, он создаёт файл под названием **config.log**. Этот файл хранит полный протокол пути выполнения этого скрипта. Каждая строка хранит точную команду оболочки, которая выполняется. Тщательное чтение файла журнала покажет, какой тест выполняется, и поможет вам понять причину неудачи.
- ***Исправление плохих скриптов конфигурирования***: плохо написанные конфигурационные скрипты всегда кошмарны при работе с ними. Они могут выполнять неправильные тестовые программы или иметь жёстко закодированные пути к библиотекам и тому подобное. Все, что вам нужно, это немного терпения и время, чтобы исправить такой сценарий.

8.3 Сборка корневой файловой системы

Теперь, когда мы ознакомились с процессом сборки ядра и приложений, следующим логическим шагом будет понять процесс создания корневой файловой системы. Как объяснялось в Главах 2 1 и 4 59, существуют три метода, которые можно использовать для этой цели:

- ***Использование initrd/initramfs***: **initrd** был подробно рассмотрен в Главах 2 1 и 4 59. В этом разделе мы обсудим **initramfs**. Для создания таких образов могут быть

использованы скрипты, имеющиеся в конце этого раздела.

- ***Монтаж корневой файловой системы по сети с использованием NFS***: это имеет смысл на этапах разработки; все изменения могут делаться на базовой (хост) машине, на которой выполняется разработка, и корневая файловая система с базовой машины может быть смонтирована по сети. Подробности о монтировании корневой файловой системы с помощью NFS можно получить из документации, которая является частью дерева исходных кодов ядра и находится в каталоге **Documentation/nfsroot**.
- ***Программирование корневой файловой системы во флеш-память***: это делается на стадии производства. На базовой машине создаётся образ корневой файловой системы для запуска на целевой платформе (например, JFFS2 или CRAMFS), а затем он записывается во флеш-память. Различные инструменты, которые доступны для создания образов, описаны в Главе 4 59.

Универсальный сценарий **initrd** показывает Распечатка 8.3 256. Он используется так:

```
mkinitrd <rfs-folder> <ramdisk-size>
```

где

- **<rfs-folder>** это абсолютный путь к родительскому каталогу, содержащему корневую файловую систему.
- **<ramdisk-size>** является размером **initrd**.

Скрипт создаёт образ **initrd /tmp/ramdisk.img**, который может быть смонтирован на целевой платформе как файловая система **ext2**. Чтобы скопировать файлы из каталога с корневой системой **<rfs-folder>** в целевой образ **/tmp/ramdisk.img**, он использует петлевое устройство (loopback device) **/dev/loop0**.

Initramfs была введена в ядро версии 2.6, чтобы обеспечить раннее пространство пользователя. Идея заключалась в том, чтобы переместить много всего, выполняемого при инициализации, из ядра в пространство пользователя. Было решено, что такие части инициализации, как поиск корневого устройства, монтаж корневой файловой системы локально или по сети, и так далее, которые были частью последовательности загрузки ядра, могут быть легко выполнены в пространстве пользователя. Это упрощает ядро. Таким образом, для достижения этой цели была разработана **initramfs**.

Так же как вы можете смонтировать в качестве корневой файловой системы образ **initrd**, вы можете также аналогичным образом смонтировать в качестве корневой файловой системы образ **initramfs**. **Initramfs** основана на файловой системе **RAMFS** и **initrd**, базирующемся на электронном диске. Различия между **RAMFS** и виртуальным диском (ramdisk) показаны в Таблице 8.1.

Таблица 8.1 Сравнение RAMFS и RAMDISK

Виртуальный диск (RAMDISK)	*RAMFS*
Виртуальный диск реализован в виде блочного устройства в оперативной памяти и чтобы использовать его, необходимо создать на нём файловую систему.	RAMFS, с другой стороны, является файловой системой, реализованной непосредственно в оперативной памяти. Для каждого файла, созданного в RAMFS, ядро хранит данные файла и метаданные в кэшах ядра.

Виртуальный диск (RAMDISK)	*RAMFS*
Перед использованием виртуального диска необходимо выделить место в оперативной памяти.	Нет необходимости предварительного выделения памяти, динамичный рост осуществляется по мере необходимости.
При запуске программ с электронного диска поддерживаются два экземпляра страниц программы: один на виртуальном диске, а другой в кэше страниц ядра.	При запуске программ из RAMFS всегда используется только одна копия, которая находится в кэше ядра. Дублирования нет.
Виртуальный диск медленнее, потому что при любом обращении к данным необходимо пройти через файловую систему и драйвер блочного устройства.	RAMFS относительно быстрее, так как фактические данные и метаданные файлов находятся в кэше ядра, и нет накладных расходов, связанных с использованием файловой системы и драйвером блочного устройства.

Образ **initramfs** можно создать с помощью сценария **mkinitramfs**. Он используется так:

```
mkinitramfs <rfs-folder>
```

где **<rfs-folder>** это абсолютный путь к родительскому каталогу, содержащему корневую файловую систему. Для создания образа **initramfs** необходимо создать архив **cpio** из **<rfs-folder>** с последующим архивированием с помощью gzip.

```
#!/bin/sh

#mkinitramfs

(cd $1 ; find . | cpio --quiet -o -H newc | gzip -9
                                  >/tmp/img.cpio.gz)
```

Распечатка 8.3 mkinitrd

Распечатка 8.3.

```
#!/bin/sh

# Создаём образ файла виртуального диска
/bin/rm -f /tmp/ramdisk.img
dd if=/dev/zero of=/tmp/ramdisk.img bs=1k count=$2

# Настраиваем петлевое устройство
/sbin/losetup -d /dev/loop0 > /dev/null 2>&1
/sbin/losetup /dev/loop0 /tmp/ramdisk.img || exit $!

# Сначала, если /tmp/ramdisk0 смонтирован, размонтируем его
if [ -e /tmp/ramdisk0 ]; then
  umount /tmp/ramdisk0 > /dev/null 2>&1
fi
```

```
# Создаём файловую систему
/sbin/mkfs -t ext2 /dev/loop0 || exit $!

# Создаём точку монтирования
if [ -e /tmp/ramdisk0 ]; then
  rm -rf /tmp/ramdisk0
fi
mkdir /tmp/ramdisk0

# Монтируем файловую систему
mount /dev/loop0 /tmp/ramdisk0 || exit $!

# Копируем данные файловой системы
echo "Copying files and directories from $1"
(cd $1; tar -cf - * ) | (cd /tmp/ramdisk0; tar xf -)
chown -R root /tmp/ramdisk0/*
chgrp -R root /tmp/ramdisk0/*

ls -lR /tmp/ramdisk0

# размонтируем
umount /tmp/ramdisk0
rm -rf /tmp/ramdisk0

# отключаем петлевое устройство
/sbin/losetup -d /dev/loop0
```

8.4 Интегрированная среда разработки

По мере увеличения программного проекта становится необходимым сборка и управление им. Компонентами, которые участвуют в процессе разработки программы, являются:

- ***Текстовый редактор***: необходимо писать файлы исходного кода. Удобнее работать с текстовыми редакторами, которые понимают ваш язык программирования. Подсветка синтаксиса, автозавершение ввода символов и навигация по коду - желательные для редактора функции.
- ***Компилятор***: для генерации объектного кода.
- ***Библиотеки***: для локализации повторно используемого кода.
- ***Компоновщик***: для компоновки объектного кода и создания конечного двоичного файла.
- ***Отладчик***: отладчик на уровне исходных текстов для поиска программных ошибок.
- ***Система сборки на базе make***: для эффективного управления процессом сборки.

Можно сэкономить много времени, если инструменты, необходимые для выполнения вышеуказанных задач, работают вместе в единой среде разработки, то есть, под IDE (Integrated Development Environment, Интегрированной Средой Разработки). IDE объединяет все инструменты, которые необходимы в процессе разработки, в единую среду.

IDE, используемая для разработки встраиваемых систем на базе Linux, должна иметь следующие возможности:

- ***Сборка приложений***: некоторыми из желаемых функций являются создание Makefile-ов

для импортированного исходного кода, импорт существующих Makefile-ов и проверка зависимостей исходных кодов.
- ***Управление приложениями***: она должна интегрироваться с инструментами управления исходным кодом, такими как CVS, ClearCase®, Perforce®, и так далее.
- ***Настройка и сборка ядра***: она должна предоставить интерфейс для настройки и сборки ядра.
- ***Сборка корневой файловой системы***: корневая файловая система в зависимости от системы может находиться во флеш-памяти, в оперативной памяти, или в сети. IDE должна обеспечить механизм для добавления или удаления в корневую файловую систему приложений, утилит, и так далее.
- ***Отладка приложений***: она должна предоставлять отладку приложений, работающих на целевой платформе, на уровне исходного кода.
- ***Отладка ядра***: если IDE обеспечивает поддержку для отладки ядра и его модулей, это является дополнительным преимуществом.

В этом разделе мы рассмотрим интегрированные среды разработки как с открытым исходным кодом, так и коммерческие, которые могут быть использованы в качестве среды разработки.

8.4.1 Eclipse

Eclipse является проектом для разработки с открытым исходным кодом (www.eclipse.org) предназначенным для обеспечения надёжной, полнофункциональной платформы для разработки IDE. Eclipse предоставляет основную платформу и различные функции интегрированных сред разработки, реализованные в виде отдельных модулей, называемых плагинами. На самом деле эта плагинная платформа и делает Eclipse очень мощным. При запуске Eclipse пользователю предоставляется IDE, состоящая из набора доступных плагинов. Большинство коммерческих интегрированных сред разработки, таких как TimeStorm, строятся с использованием платформы Eclipse.

8.4.2 KDevelop

KDevelop является IDE с открытым исходным кодом для KDE™ (www.kdevelop.org). Некоторые возможности KDevelop:

- Он управляет в одной среде всеми средствами разработки, такими как компилятор, компоновщик и отладчик.
- Он предоставляет простой в использовании пользовательский интерфейс для большинства необходимых функций систем управления исходным кодом, таких как CVS.
- Он поддерживает ***проекты Automake*** для автоматической генерации Makefile и управления процессом сборки. Он также поддерживает ***пользовательские проекты***, чтобы пользователь мог управлять Makefile-ми и процессами сборки.
- Поддержка кросс-компиляции.
- Встроенный текстовый редактор на основе Kwrite от KDE, Qeditor от Trolltec, и так далее с такими функциями, как подсветка синтаксиса, автоматическое завершение ввода символов, и так далее.
- Интеграция с Doxygen для создания документации об интерфейсах.
- Мастер создания приложений для создания приложений-примеров.
- Поддержка проектов Qt/embedded.

- Графический пользовательский интерфейс для GDB.

8.4.3 TimeStorm

TimeStorm Linux Development Suite (LDS) является коммерческой средой разработки для встраиваемого Linux, предоставляемой TimeSys (www.timesys.com). Она основана на платформе IDE от Eclipse. Некоторые из её возможностей:

- Работает на системах Linux и Windows.
- Интегрирована с инструментов управления исходным кодом, такими как CVS, ClearCase, Perforce и так далее.
- Инструменты для разработки и отладки приложений для встраиваемой системы.
- Работа с дистрибутивами Linux не от TimeSys.
- Интерфейс для настройки и компиляции ядра Linux для выбранной целевой платформы.
- Графический интерфейс для создания корневой файловой системы для целевой платформы.
- Предоставляет возможность загружать и исполнять программы на целевой платформе.
- Графический пользовательский интерфейс для удалённой отладки приложений с помощью GDB.

8.4.4 CodeWarrior

Metrowerks CodeWarrior Development Studio является полноценной коммерческой IDE, которая облегчает разработку от запуска оборудования до отладки приложений (www.metrowerks.com). Некоторые из её возможностей:

- Встроенный текстовый редактор с такими функциями, как подсветка синтаксиса, авто-отступы, и так далее.
- Включает в себя поисковую систему для быстрой навигации по исходному коду.
- Интегрированный эмулятор набора инструкций для быстрого старта разработки приложений.
- Она предоставляет высокопроизводительный оконный отладчик на уровне исходных текстов. Отладчик включает в себя программатор флеш-памяти и инструменты для диагностики оборудования.
- Встроенная система управления версиями, такими как CVS, Perforce, и так далее.

8.5 Устранение проблем, связанных с виртуальной памятью

При запуске приложения на Linux пользователь часто сталкивается с проблемами управления памятью. Они могут быть условно разделены на три категории:

- ***Утечки памяти***: утечки памяти возникают, когда кусок памяти, который был выделен, не освобождается. Повторяющиеся утечки памяти могут оказаться фатальными для встраиваемых систем, потому что система не сможет работать при нехватке памяти.
- ***Переполнение***: переполнение является состоянием, при котором происходит обращение к адресам за пределами выделенной области памяти. Переполнение представляет собой очень серьёзную угрозу для безопасности и используется злоумышленниками для взлома системы.
- ***Повреждение***: повреждение памяти происходит, когда указатели памяти содержат неправильные или недостоверные значения. Использование таких указателей может

привести к нарушению работы программы и обычно ведёт к завершению программы.

Проблемы управления памятью очень тяжелы для поиска, в том смысле, что их очень трудно найти с помощью проверки кода, и они могут происходить или непостоянно, или после многих часов использования системы. К счастью, для отслеживания проблем, связанных с управлением памятью, есть ряд инструментов с открытым исходным кодом. В следующих подразделах о них рассказывается подробно с соответствующими примерами. Выделение динамической памяти в Linux рассматривается в Главе 10 322.

Утечки памяти связаны прежде всего с двумя причинами:

- ***Небрежность автора кода***: разработчик программы не обращает должного внимания на освобождение выделенной памяти, когда она больше не используется.
- ***Повреждение указателя***: это происходит, когда указатель, содержащий ссылку на фрагмент памяти, повреждается, следовательно, теряется ссылка на кусок памяти.

Повторные утечки памяти на встроенной системе, не имеющей подкачки, заставляют систему испытывать недостаток памяти. Как в таком случае ведёт себя система? Когда системе начинает не хватать памяти, она переходит в режим урезания (prune mode) и пытается сжать системные кэши (например, кэш страниц, кэш буферов и кэши файловой системы, а также кэши кусков памяти) и в этом процессе освобождается от файлов образа процесса. Если даже по завершении этой работы системе не хватает памяти, вызывается печально известное сообщении о ***недостатке памяти (out of memory)*** или убийца OOM. Когда вызывается OOM, вы видите в консоли следующее сообщение:

```
Out of Memory: Killed process 10(iisd)
```

В этом случае убийца OOM убил процесс **iisd** (с pid 10). Убийцу OOM в ядро версии 2.2.15 добавил Рик Ван Рейл. В основе философии убийцы OMM лежит то, что когда в системе мало памяти, вместо того, чтобы позволить ядру паниковать или вызвать блокировку системы, лучше убить процесс или набор процессов, чтобы снова вернуть системе память. Поэтому вместо того, чтобы дать системе упасть, дайте ей работать с одним или несколькими убитыми приложениями. Очевидно, что ключом к реализации OOM является выбор процесса, который будет убит; убийство важных общесистемных процессов может быть столь же вредным, как и падение системы. Следовательно убийца OOM был очень обсуждаемой темой, в особенности потому что очень трудно найти универсальное решение, так как Linux работает на очень разных системах. Проект OOM развивался в этом направлении. В ядре версии 2.6 убийца OOM проходит через список всех процессов и вычисляет предполагаемое значение испорченной памяти. Процесс, который имеет максимальное значение испорченной памяти, убивается.

OOM это последняя попытка системы, чтобы оправиться от проблемы с недостатком памяти. Разработчик в первую очередь обязан убедиться, что этого не происходит. Ниже приведены два метода, которые могут сделать систему более устойчивой к утечкам памяти:

- ***Настройка водяных знаков памяти для каждого процесса***: первым шагом в этом направлении является выявление плохих программ, которые вызывают утечки памяти. Это можно сделать, установив ограничении RSS для каждого процесса, работающего в системе, с помощью системного вызова **setrlimit()**. Есть два системных вызова, предоставляемых для этого ядром Linux: **setrlimit()** и **getrlimit()** для установки и получения ограничения ресурсов, соответственно. Каждый ресурс имеет жёсткое и мягкое ограничение, определяемое структурой **rlimit** (смотрите заголовочный файл **sys/resource.h**). Мягкое ограничение является значением, которое ядро обеспечивает

для соответствующего ресурса. Жёсткое ограничение действует в качестве потолка для мягкого ограничения. Могут иметь место различные виды ресурсных ограничений; наиболее значимое, связанное с памятью, это **RLIMIT_RSS**, которое ограничивает количество страниц, принадлежащих резиденту процесса в оперативной памяти. (Для его использования обратитесь к главной странице **setrlimit**.)

- ***Отключение чрезмерного выделения памяти (over-commit) в системе***: чрезмерное выделение памяти представляет собой ипотечную схему выделения памяти, при которой ядро отводит приложению большое количество динамической памяти, даже если оно не располагает достаточными ресурсами памяти. За превышением ресурсов стоит идея, заключающаяся в том, что обычно приложения для настольных компьютеров выделяют много памяти, но редко используют большую её часть. Следовательно, ядро выполняет выделение памяти, не заботясь о проверке, имеет ли оно такие ресурсы. (Всегда, поскольку из-за подкачки по запросу фактическая память не выделяется, если она не используется.) На встроенных системах рекомендуется выключить эту функцию по двум причинам:
  – Вы не должны иметь никаких приложений, желающих выделять огромное количество памяти, а затем использовать её только частично. Такие приложения небрежно обращаются с памятью и не оптимизированы для встроенных систем. (Если приложение небрежно относится к выделению памяти, оно также может быть небрежным и при освобождении памяти.)
  – Лучше потерпеть неудачу, когда приложение запрашивает память и памяти не хватает, чем позволить произойти выделению памяти, а затем позже вызвать состояние нехватки памяти при обращении к памяти. Первое проще для отладки и может быть исправлено более легко. Linux предлагает пользователю отключать чрезмерное выделение памяти с помощью **proc** файла **/proc/sys/vm/overcommit**. Запись 0 в этот файл отключает чрезмерное выделение памяти.

Тем не менее, в случае если вы попали в состояние OOM, и вы уверены, что какое-то приложение вызывает утечку памяти, лучшим решением является использование отладчиков памяти, которые направлены на обнаружение утечек.

8.5.1 Устранение утечек памяти

В этом разделе мы обсудим инструменты для устранения утечек памяти **mtrace** и **dmalloc**.

mtrace

mtrace является инструментом **glibc** для борьбы с утечками памяти. Как следует из названия, он используется для отслеживания выделения и освобождения памяти. Для этой цели предназначены два вызова **glibc**:

- **mtrace(void)**: запускает трассировку. Когда вызывается функция **mtrace**, она ищет переменную окружения с именем **MALLOC_TRACE**. Предполагается, что эта переменная содержит достоверное имя файла, для которого пользователь должен иметь доступ на запись. Ничего не делается, если переменная окружения не установлена, или если файл не может быть открыт для записи. Однако, если указанный файл успешно открыт, **mtrace** устанавливает специальные обработчики для функций выделения памяти, которые записывают в файл журналы трассировки.
- **muntrace(void)**: останавливает трассировку, убирая обработчики трассировки.

Распечатка 8.4 показывает простую программу, которая вызывает утечку памяти. Мы покажем, как эта утечка может быть обнаружена с помощью **mtrace**. Скомпилируйте программу и выполните следующие шаги:

```
# gcc -g leak.c -o leak
# export MALLOC_TRACE=./log
# ./a.out
# cat log
= Start
@ ./a.out:(mtrace+0xf5)[0x8048445] + 0x8049a40 0x13
@ ./a.out:(mtrace+0x105)[0x8048455] + 0x8049a58 0x11
@ ./a.out:(mtrace+0x162)[0x80484b2] - 0x8049a58
= End
```

Как видите, файл журнала, который был сформирован **mtrace**, довольно загадочный. Glibc предоставляет другую программу с таким же именем, **mtrace** (которая является производным от Perl скрипта **mtrace.pl**). Эта программа анализирует содержимое файла журнала и показывает фактические утечки в дружественном человеку виде.

```
# mtrace ./a.out log
Memory not freed:
-----------------
Address     Size      Caller
0x8048445   0x13  at  ./leak.c:6
```

Таким образом, пользователю сообщается, что когда была включена трассировка, была обнаружена утечка памяти. Кусок памяти размером 19 байт (0x13), не была освобождена память, выделенная в файле **leak.c** в строке номер 6.

dmalloc

dmalloc это более совершенный инструмент, который обеспечивает обнаружение утечек памяти, а также множество других функций, таких как проверка смещения (fencepost error, ошибка на единицу при вычислении смещения или числе циклов итерации) и проверка "кучи". Данный раздел посвящён использованию **dmalloc** в первую очередь для обнаружения утечек памяти. Официальным веб-сайтом для **dmalloc** является http://dmalloc.com.

dmalloc реализован с помощью библиотеки, которая предоставляет обёртку вокруг интерфейсов выделения памяти, таких как **malloc**, **free**, и так далее. Поэтому, чтобы пользоваться **dmalloc**, приложение должно быть скомпоновано с этой библиотекой. Проиллюстрируем это далее с помощью примера, показанного в Распечатке 8.5. Скомпилируем и скомпонуем **dmalloc_test.c** с **libdmalloc.a**. (* для многопоточных программ компонуем с libdmallocth.a. Для программ C++ компонуем с libdmallocxx.a и для многопоточных программ C++ компонуем с libdmallocthcxx.a.)

```
# ls -l libdmalloc.a
-rw-rw-r-- 1 raghav raghav 255408 Sep 4 10:48 libdmalloc.a
# gcc dmalloc_test.c -o dmalloc_test ./libdmalloc.a
```

Теперь, когда мы скомпоновали наше приложение, пришло время для его запуска. Но прежде, чем запустить программу, мы должны установить переменную окружения, которая,

среди множества других вещей, будет информировать библиотеку, что необходимо включить при выполнении отладки, и куда должен записываться журнал. Подробно эта переменная окружения обсуждается позже.

```
# export DMALLOC_OPTIONS=debug=0x3,log=dlog
# ./dmalloc_test
```

Вывод показан в Распечатке 8.6[265]. Строки, выделенные жирным шрифтом, показывают количество утечек памяти. Обратите внимание, что отладочная информация, такая, как имя файла и номер строки, отсутствует. Мы можем получить эту информацию, используя такие инструменты, как **gdb** или **addr2line**. Тем не менее, **dmalloc** предоставляет механизм включения более подробной отладочной информации в файле журнала с помощью файла **dmalloc.h**. Этот файл поставляется вместе с пакетом **dmalloc**. Во все файлы языка Си, которые скомпонованы для создания приложения, которое будет отлаживаться, необходимо включить этот заголовочный файл. Этот заголовочный файл определяет такие функции выделения памяти, как **malloc()** и **free()**, с такими макросами препроцессора, как **__FILE__** и **__LINE__**. Например, определение **malloc** в **dmalloc.h** сделано следующим образом:

```
#undef malloc
#define malloc(size) \
dmalloc_malloc(__FILE__, __LINE__, (size), DMALLOC_FUNC_MALLOC, 0, 0)
```

Первая строка отменяет все определения **malloc**, которые поступают из ранее подключённых заголовочных файлы (**stdlib.h**). Разумеется, **dmalloc.h** должен быть последним подключённым файлом.

DMALLOC_OPTIONS, как показано в приведённом выше примере, управляет отладкой во время выполнения. Эта переменная окружения может быть установлена вручную или с помощью программы под названием **dmalloc**. Для встроенных сред вы можете предпочесть установить этот параметр вручную. Чтобы получить более подробную информацию об утилите **dmalloc**, запустите её из командной строки с аргументом **--usage**. **DMALLOC_OPTIONS** представляет собой список из следующих (важных) параметров, разделённых запятыми:

- **debug**: этот параметр имеет шестнадцатеричное значение, получающееся путём сложения всех функциональных параметров. Параметр функциональности включает отладку во время выполнения. Например, параметр функциональности 0x1 включает ведение журнала общей статистики, а 0x2 включает ведение журнала утечек памяти. Поэтому добавление значения 0x3 включает и регистрацию общей статистики, и утечек памяти. Список всех функциональных параметров можно получить, запустив программу **dmalloc** с аргументом **--DV**.
- **log**: используется, чтобы установить имя файла для журналирования статистики, утечек и другие ошибки.
- **addr**: когда он установлен, при обращении к этому адресу выполнение **dmalloc** прервётся.
- **inter**: используется, если включена функциональность проверки "кучи". Например, если это значение равно 10, то "куча" проверяется на наличие ошибок после каждого 10-го вызова любой функции работы с памятью.

Распечатка 8.4 Использование mtrace

Распечатка 8.4.

```
/* leak.c */

#include <mcheck.h>
func()
{
  char *str[2];
  mtrace();
  str[0] = (char *)malloc(sizeof("memory leak start\n"));
  str[1] = (char *)malloc(sizeof("memory leak end\n"));
  strcpy(str[0], "memory leak start\n");
  strcpy(str[1], "memory leak end\n");
  printf("%s",str[0]);
  printf("%s",str[1]);
  free(str[1]);
  muntrace();
  return;
}

main()
{
  func();
}
```

Распечатка 8.5 Использование dmalloc

Распечатка 8.5.

```
/* dmalloc_test.c */

#include <stdio.h>
#include <stdlib.h>

#ifdef USE_DMALLOC
#include <dmalloc.h>
#endif

int main()
{
  char *test[5];
  unsigned int i;

  for (i=0; i < 5; i++)
  {
    unsigned int size = rand()%1024;
    test[i] = (char *)malloc(size);
    printf ("Allocated memory of size %d\n",size);
  }
  for (i=0; i<2; i++)
    free(test[i*2]);
```

```
}
```

Распечатка 8.6 Вывод dmalloc

Распечатка 8.6.

```
calling dmalloc malloc
Allocated memory of size 359
calling dmalloc malloc
Allocated memory of size 966
calling dmalloc malloc
Allocated memory of size 105
calling dmalloc malloc
Allocated memory of size 115
calling dmalloc malloc
Allocated memory of size 81
bash>cat dlog
1094293908: 8: Dmalloc version '5.3.0' from 'http://dmalloc.com/'
1094293908: 8: flags = 0x3, logfile 'dlog'
1094293908: 8: interval = 0, addr = 0, seen # = 0, limit = 0
1094293908: 8: starting time = 1094293908
1094293908: 8: process pid = 4709
1094293908: 8: Dumping Chunk Statistics:
1094293908: 8: basic-block 4096 bytes, alignment 8 bytes, heap
               grows up
1094293908: 8: heap address range: 0x80c3000 to 0x80ca000, 28672
               bytes
1094293908: 8:     user blocks: 3 blocks, 12217 bytes (42%)
1094293908: 8:    admin blocks: 4 blocks, 16384 bytes (57%)
1094293908: 8: external blocks: 0 blocks, 0 bytes (0%)
1094293908: 8:    total blocks: 7 blocks, 28672 bytes
1094293908: 8: heap checked 0
1094293908: 8: alloc calls: malloc 5, calloc 0, realloc 0, free 3
1094293908: 8: alloc calls: recalloc 0, memalign 0, valloc 0
1094293908: 8: alloc calls: new 0, delete 0
1094293908: 8:   current memory in use: 1081 bytes (2 pnts)
1094293908: 8:  total memory allocated: 1626 bytes (5 pnts)
1094293908: 8:  max in use at one time: 1626 bytes (5 pnts)
1094293908: 8: max alloced with 1 call: 966 bytes
1094293908: 8: max unused memory space: 294 bytes (15%)
1094293908: 8: top 10 allocations:
1094293908: 8:  total-size  count in-use-size  count  source
1094293908: 8:        1626      5        1081  2  ra=0x8048a46
1094293908: 8:        1626      5        1081  2  Total of 1
1094293908: 8: Dumping Not-Freed Pointers Changed Since Start:
1094293908: 8:  not freed: '0x80c6c00|s1' (966 bytes) from
                'ra=0x8048a46'
1094293908: 8:  not freed: '0x80c8f00|s1' (115 bytes) from
                'ra=0x8048a46'
1094293908: 8: total-size  count  source
1094293908: 8:       1081      2  ra=0x8048a46
1094293908: 8:       1081      2  Total of 1
1094293908: 8: ending time = 1094293908, elapsed since start =
```

```
                0:00:00
```

8.5.2 Устранение переполнений памяти

Несмотря на простоту в использовании, язык Си является небезопасным языком, потому что он не предотвращает переполнения буфера. Многие функции в библиотеке Си описаны как небезопасные для использования (например, печально известная функция **gets()**), но это не мешает тому, чтобы небрежный программисту использовал такие функции и вызывал нарушения безопасности в системе. Ниже приведены некоторые примеры этого, которые возникают из-за такого небрежного программирования.

- ***Изменение хода выполнения кода***: переполнение буфера может изменить адрес возврата в стеке или произвольного указателя функции в памяти.
- ***Перезапись переменной с данными***: переменная, которая содержит какие-то секретные сведения, например, строку подключения к базе данных.

Если вы загрузили исходный код приложения из Сети, и хотели бы проверить его на ошибки переполнения, есть готовые инструменты, доступные для помощи вам. Инструментом, который был написан в основном для поиска переполнений буфера, является ***Electric Fence***. Обратите внимание, что **dmalloc** также способен делать проверки на ошибки при вычислении смещения (обычно на плюс/минус единицу). Но метод, предоставляемый **dmalloc**, не совсем понятный, потому что это полностью реализован в программном обеспечении. Есть два недостатка схемы **dmalloc**:

- **dmalloc** реализует проверку на ошибку вычисления смещения путём добавления в выделенную область магического числа и проверкой, что магическое число не перезаписано. Частота проверок контролируется с помощью параметра отладки **inter**. Эта схема может быть эффективной при поиске переполнения, но она может быть неэффективной для указывания на ошибочную инструкцию.
- **dmalloc** может обнаружить только записи за границы буфера; однако, чтения за границами буфера по-прежнему остаются незамеченными.

Electric Fence позволяет использовать оборудование, чтобы точно выявить неправильную инструкцию, которая пытается читать и писать за границами буфера. Кроме того, он может быть использован для обнаружения любых ситуаций, когда программное обеспечение обращается к памяти, которая уже была освобождена. Electric Fence управляет этим путём выделения страницы (или набора страниц) для каждого запроса памяти, а затем делает страницы за пределами буфера недоступными для чтения или записи. Таким образом, если программное обеспечение пытается получить доступ к памяти за пределами этих границ, это приводит к ошибке сегментации. Аналогично, если память освобождается вызовом **free()**, она делается недоступной с помощью защиты виртуальной памяти и любой код, который затрагивает освобождённую память, получит ошибку сегментации.

Electric Fence доступен как библиотека **libefence.a**; она должна быть скомпонована с приложением, которое должно быть отлажено на предмет переполнения буфера. Использование Electric Fence иллюстрирует Распечатка 8.7 267. Как видно в Распечатке 8.7 267, рекомендуемым способом запуска Electric Fence является запуск его из **gdb**, чтобы неправильная инструкция могла быть поймана с полной отладочной информацией. Этот пример показывает случай переполнения буфера. Electric Fence также может поймать обращения за нижней границей выделенной памяти, но для того, чтобы сделать это, до запуска приложения должна быть экспортирована переменная окружения

EF_PROTECT_BELOW. Это необходимо, потому что по умолчанию Electric Fence ловит только переполнения буфера путем размещения памяти, недоступной для обращения, после конца каждого выделенного куска. Установка этой переменной гарантирует, что недоступная для обращения память размещается до начала выделенной памяти. Таким образом, чтобы убедиться, что приложение не имеет ни переполнений, ни обращений за нижнюю границу, это приложение должно быть вызвано дважды: один раз без установки переменной **EF_PROTECT_BELOW** и второй раз после её установки.

Поскольку Electric Fence требует, чтобы при каждом выделении памяти выделялась по крайней мере одна страница памяти, это может потребовать весьма много памяти. Рекомендуется использовать Electric Fence только для отладки систем; при поставке систем он должен быть выключен.

Распечатка 8.7 Использование Electric Fence

Распечатка 8.7.

```
/* efence-test.c */

#include <stdio.h>

main()
{
  int i,j;
  char * c = (char *)malloc(20);
  printf("start of efence test\n");
  for(i=0; i < 24; i++)
    c[i] = 'c';
  free(c);
  printf("end of efence test\n");
}
```

```
# ls -l libefence.a
-rw-rw-r-- 1 raghav   raghav  76650 Sep  4 20:38 libefence.a

# gcc -g efence-test.c  -L. -lefence -lpthread -o efence-test
# gdb ./efence-test

...
(gdb) run
Starting program: /home/raghav/BK/tmp/memory-debugging/src/efence/
efence-test
[New Thread 1073838752 (LWP 6413)]

Electric Fence 2.4.10
Copyright (C) 1987-1999 Bruce Perens <bruce@perens.com>
Copyright (C) 2002-
2004 Hayati Ayguen <hayati.ayguen@epost.de>, Procitec GmbH
start of efence test:4004dfec

Program received signal SIGSEGV, Segmentation fault.
[Switching to Thread 1073838752 (LWP 6413)]
0x08048a20 in main () at efence-test.c:9
```

```
9                         c[i] = 'c';
(gdb) print i
$1 = 20
(gdb)
```

8.5.3 Устранение повреждений памяти

Повреждение памяти происходит, когда в ячейку памяти записывается неправильное значение, что приводит к неправильной работе программы. Переполнение памяти, рассмотренное в предыдущем разделе, это один из видов повреждения памяти. Некоторые из обычных примеров повреждения памяти:

- Разыменование указателей памяти, содержащих неинициализированные значения.
- Перезапись указателей памяти неправильными значениями с последующим разыменовыванием.
- Разыменование указателей памяти после того, как эта память была освобождена.

Такие ошибки очень трудно поймать. Поиск ошибку вручную может оказаться сложнейшей задача, так как это будет означать сканирование всего кода. Изучим программный инструмент для поиска повреждений памяти, Valgrind. Valgrind может быть загружен с веб-сайта http://valgrind.kde.org.

Ниже приводятся некоторые важные особенности Valgrind.

- Valgrind работает только под Linux на платформе x86. Это может выглядеть как основной сдерживающий фактор от его использования, учитывая, что многие встраиваемые платформы базируются не на архитектуре x86. Однако, многие проекты для встраиваемых систем перед переносом их на целевую платформу первоначально запускают свои приложения на платформе на базе x86; так происходит потому, что целевая платформа может быть недоступна, когда приложения находятся в разработке. В таких случаях разработчики могут тестировать свои приложения на наличие повреждений памяти до их перевода в целевую платформу полностью на платформе x86.
- Valgrind очень сильно привязан к ОС и её библиотекам; на момент написания, дистрибутив Valgrind версии 2.2.0 предполагается запускать на ядре версии 2.4 или 2.6 и **glibc** версии 2.2.x или 2.3.x.
- В случае с Valgrind приложение может работать без какой-либо пересборки. Например, если бы вы захотели запустить Valgrind с исполняемым файлом **ps**, вы бы запустили его так, как показано в Распечатке 8.8 270.
- Valgrind работает имитируя для выполнения вашей программы CPU x86. Побочным эффектом этого является то, что программа работает медленнее, так как для учёта использования системных ресурсов Valgrind отлавливает соответствующие вызовы (системные вызовы, вызовы работы с памятью). Кроме того, при использовании с Valgrind приложение имеет тенденцию потреблять больше памяти. Valgrind это отличный инструмент, который может сделать гораздо больше, чем просто бороться с повреждением памяти, например, профилирование кэша и обнаружение состояния гонок в многопоточных программах. Тем не менее, настоящий раздел изучает использование Valgrind для борьбы с повреждением памяти. Ниже приводится список проверок памяти, которые можно выполнить с помощью Valgrind:
  – Использование не проинициализированной памяти
  – Утечки памяти
  – Переполнения памяти

– Повреждение стек
– Использование указателей памяти после того, как соответствующая память была освобождена
– Несоответствующие указатели в **malloc**/**free**

Архитектура Valgrind может быть разложена на два уровня: ядро и внешняя облочка. Ядро представляет собой эмулятор x86, который переводит весь исполняемый код в свой собственный код операций. Затем переведённый код оценивается и выполняется на реальном CPU. Оценка зависит от выбранного вида внешней оболочки. Архитектура Valgrind является модульной, позволяя легко подключить к ядру новую внешнюю оболочку.

Мы сосредоточим наше внимание на оболочку проверки памяти, **memcheck**. Это используемая по умолчанию оболочка Valgrind (любая другая оболочка должна вызываться специально с помощью аргумента командной строки **--skin**). Memcheck работает, связывая каждый байт оперативной памяти с двумя значениями: битом V (допустимое значение) и битом A (достоверный адрес). Бит V определяет, определено ли в программе значение этому байту. Например, проинициализированный байт памяти имеет установленный бит V; соответственно, не проинициализированные переменные можно отслеживать с помощью бита A. Бит A отслеживает, может ли ячейка памяти быть доступной. Аналогично, когда выполняется вызов выделения памяти с помощью **malloc()**, все байты выделенной памяти имеют установленный бит V. Другая оболочка, **addrcheck**, предоставляет все возможности **memcheck**, кроме проверки на неопределённое значение. Она делает это используя бит V. Оболочка **addrcheck** может использоваться в качестве легковесного контролера памяти; она быстрее и легче, чем **memcheck**.

Теперь давайте рассмотрим некоторые практические демонстрации работы Valgrind. Первый пример показывает, как Valgrind обнаруживает использование не проинициализированной переменной. Valgrind обнаруживает неправильное использование не проинициализированной переменной в условном переходе или когда она используется для формирования адреса памяти, как показано ниже.

```
#include <stdlib.h>
main()
{
  int *p;
  int c = *p;
  if(c == 0)
    ...
  return;
}
```

При запуске этой программы с Valgrind он будет генерировать следующий вывод:

```
==4409== Use of uninitialized value of size 4
==4409==    at 0x804833B: main (in /tmp/x)
==4409==    by 0x40258A46: __libc_start_main
            (in /lib/libc-2.3.2.so)
==4409== by 0x8048298: ??? (start.S:81)
==4409==
==4409== ERROR SUMMARY: 1 errors from 1 contexts
          (suppressed: 0 from 0)
==4409== malloc/free: in use at exit: 0 bytes in 0 blocks.
==4409== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
```

Второй пример показывает, как Valgrind может быть использован для обнаружения разыменования указателя памяти после освобождения памяти.

```
#include <stdlib.h>
main()
{
  int *i = (int *)malloc(sizeof(int));
  *i = 10;
  free(i);
  printf("%d\n",*i);
}
```

При запуске этой программы с Valgrind он будет генерировать следующий вывод:

```
==4437== 1 errors in context 1 of 1:
==4437== Invalid read of size 4
==4437==    at 0x80483CD: main (x.c:6)
==4437==    by 0x40258A46: __libc_start_main
                          (in /lib/libc-2.3.2.so)
==4437==   by 0x8048300: ??? (start.S:81)
==4437==   Address 0x411C7024 is 0 bytes inside a block of
           size 4 free'd
```

Распечатка 8.8 Пример работы Valgrind

Распечатка 8.8.

```
shell> valgrind ps
==4187== Memcheck, a.k.a. Valgrind, a memory error detector for
         x86-linux.
==4187== Copyright (C) 2002-2003, and GNU GPL'd, by Julian
         Seward.
==4187== Using valgrind-2.0.0, a program supervision framework
         for x86-linux.
==4187== Copyright (C) 2000-2003, and GNU GPL'd, by Julian
Seward.
==4187== Estimated CPU clock rate is 2194 MHz
==4187== For more details, rerun with: -v
==4187==
  PID TTY          TIME CMD
 3753 pts/3    00:00:00 bash
 4187 pts/3    00:00:00 ps
==4187== discard syms in /lib/libnss_files-2.3.2.so due to
         munmap()
==4187==
==4187== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2
         from 1)
==4187== malloc/free: in use at exit: 125277 bytes in 117 blocks.
==4187== malloc/free: 353 allocs, 236 frees, 252784 bytes
         allocated.
==4187== For a detailed leak analysis,  rerun with: --leakcheck=
         yes
```

```
==4187== For counts of detected errors, rerun with: -v
```

8.6 Отладчики ядра

В отличие от обычных RTOS, где для отладки программного обеспечения используется один отладчик, система на базе Linux требует двух отладчиков: отладчик ядра и отладчик приложений. Так происходит потому, что ядро и приложения используют разные адресные пространства. Этот раздел рассказывает об использовании двух популярных отладчиков ядра: KDB и KGDB.

Ядро Linux не имеет встроенного отладчика; отладчики ядра сохраняются в виде отдельных проектов. (* Хотя это больное место многих разработчиков ядра, Линус считает, что отладчики исправляют симптомы, а не предлагают лечение.) KDB и KGDB имеют разные среды работы и предоставляют разные функциональные возможности. В то время, как KDB является частью ядра Linux и обеспечивает механизм выполнения для просмотра различных компонентов, таких как память и структуры данных ядра, KGDB работает в тандеме с GDB и требует для взаимодействия с заглушкой KGDB, работающей на целевой платформе, отдельную базовую машину. Таблица 8.2 сравнивает KGDB и KDB.

Таблица 8.2 Сравнение KDB и KGDB

	KDB	*KGDB*
Среда отладчика	Это отладчик, который должен быть встроен в ядро. Всё, что требуется, это консоль, с помощью которой могут быть введены команды, вывод также отображается в консоли.	Требует отдельной машины разработчика, чтобы запустить отладчик как обычный процесс, который взаимодействует с целевой платформой с использованием протокола GDB через последовательный кабель. Последние версии KGDB поддерживают интерфейс Ethernet.
Требования к поддержке ядром/ патчам	KDB требует два патча: универсальный патч для ядра, который реализует архитектурно-независимые функциональные возможности, и архитектурно-зависимый патч.	KGDB использует один патч, который состоит из трёх компонентов: ▪ Заглушка GDB, которая реализует протокол GDB на целевой стороне, ▪ Изменения в последовательном (или Ethernet) драйвере для отправки и получения сообщений между целевой платформой и машиной разработчика, ▪ Изменения в обработчиках исключений, чтобы дать управление отладчику, когда происходят исключения.
Поддержка отладки на	Отладка на уровне исходных текстов не поддерживается	Поддержка отладки на уровне исходных текстов

	KDB	KGDB
уровне исходных текстов		предоставляется при условии, что ядро на машине разработчика скомпилировано с флагом **-g** и доступно дерево исходных кодов ядра. На машине разработчика, на которой запускается приложение отладчика, опция **-g** указывает **gcc** создать во время компиляции отладочную информацию, которая в сочетании с исходными файлами обеспечивает отладку на уровне исходных текстов.
Предлагаемые возможности для отладки	Наиболее часто используемыми функциями отладки KDB являются: ▪ отображение и изменение содержимого памяти и регистров ▪ применение точек останова ▪ трассировка стека Наряду с установленными пользователем точками останова, KDB вызывается, когда ядро попадает в состояние неисправимой ошибки, такой как паника или OOPS. Для выявления проблемы пользователь может использовать вывод KDB.	Поддержка команд контроля исполнения GDB, трассировки стека и установленных KGDB точек наблюдения, наряду с множеством других функций, таких как анализ потоков.
Отладка модулей ядра	KDB обеспечивает поддержку для отладки модулей ядра.	Отладка модулей с использованием KGDB сложна, потому что модуль загружен на целевой машине, а отладчик (GDB) работает на другой машине; так что отладчик KGDB должен быть проинформирован об адресе загрузки модуля. KGDB версии 1.9 сопровождается специальным GDB, который может автоматически обнаружить загрузку и выгрузку модуля. Для KGDB версий равной или меньшей 1.8, разработчик для загрузки модуля объекта в память GDB и указания адреса загрузки модуля должен использовать явную команду GDB **addsymbol-file**.
Веб-сайты для загрузки	http://oss.sgi.com/projects/kdb/	http://kgdb.linsyssoft.com/

Использование KDB может быть разделено на два этапа:

- ***Сборка ядра с патчем KDB***: загрузите патчи с веб-сайта, указанного в Таблице 8.2, и примените его к дереву исходных текстов ядра. После настройки ядра с включённой опцией KDB соберите ядро.
- ***Запуск ядра с включённым KDB***: включен или выключен KDB при загрузке ядра определяет параметр конфигурации сборки **KDB_OFF**; при выборе этой опции KDB не активирован. В таком случае KDB должен быть активирован явно; это делается двумя способами. Один из способов состоит в передаче ядру аргумента строки загрузки **kdb=on**. Другим способом активации является использование файла **/proc/sys/kernel/kdb** с помощью следующей команды:

```
echo "1" > /proc/sys/kernel/kdb
```

Список команд KDB может быть найден в файлах каталога **Documentation/kdb** после того, как дерево исходных текстов ядра было пропатчено.

Использование KGDB можно разделить на три этапа:

- ***Сборка ядра с патчем KGDB***: это требует получения патча ядра KGDB, применение патча к ядру, сборку ядра с включённой поддержкой KGDB.
- ***Установление соединения между целевой машиной и базовой машиной с помощью протокола GDB***: когда ядро загружается, оно ожидает установления соединения от базовой машины с помощью протокола KGDB и сообщает это пользователю, выбрасывая следующее сообщение:

```
Waiting for connection from remote gdb...
```

  Пользователь должен запустить отладчик на базовой машине и подключиться к целевой машине, что укажет ядру продолжить загрузку.
- ***Использование отладчика для отладки ядра***: после этого можно удалённо отлаживать ядро при помощи стандартных команд GDB.

8.7 Профилирование

Профилирование представляет собой способ поиска узких мест в процессе выполнения программы, так что его результаты могут быть использованы для увеличения производительности. Есть несколько важных вопросов, которые требуют ответа:

- ***Зачем нужно профилирование?*** Большинство встраиваемых систем имеют весьма ограниченные ресурсы с точки зрения общей памяти и частоты процессора. Поэтому очень важно обеспечить, чтобы эти ресурсы использовались оптимально. С помощью профилирования определяются различные узкие места в программе. Устранение этих узких мест ведёт к увеличению производительности системы и оптимальному использованию ресурсов.
- ***Что измеряется в процессе профилирования?*** Это включает в себя такие величины, как процент времени выполнения в каждой части программы и использование памяти различными модулями программы. Для драйверов это может быть общее время запрета прерываний, и так далее.
- ***Как используется результаты профилирования?*** Результаты профилирования

используются для оптимизации программ; проблемные части кода могут быть переписаны с использованием лучшего алгоритма.

- ***Что представляют собой инструменты профилирования?*** Роль инструмента профилирования заключается в том, чтобы связать узкие места, выявленные в ходе исполнения, с лежащим в основе исходным кодом. Инструменты профилирования также предоставляют пользователю данные профилирования в виде графика, гистограммы или другого удобного для человека формата.
- ***Какой должна быть среда профилирования?*** Во время профилирования программе должны даваться реалистичные входные сигналы. Для получения более точных результатов профилирования вся отладочная информация должна быть отключена. Наконец, воздействие самого инструмента профилирования на результаты должно быть известно заранее.

В этом разделе мы обсудим три профилировщика: eProf, OProfile и Kernel Function Instrumentation (KFI). Сначала мы обсудим ***eProf***, встраиваемый профилировщик, который можно использовать во время разработки программы. Далее мы обсудим ***OProfile***, который представляет собой очень мощный инструмент профилирования. Наконец, мы обсудим ***KFI*** для профилирования функций ядра. Эти профилировщики в основном сконцентрированы на времени выполнения различных частей программы.

8.7.1 eProf — встраиваемый профилировщик

Во время разработки программы вы часто хотите узнать время, которое требуется функции для выполнения, или время, необходимое, чтобы добраться из одной точки программы в другую. На этом этапе вам не нужен полномасштабный профилировщик. Вы можете написать свой собственный профилировщик (который авторы называют eProf), чтобы помочь себе в случае потребности малого профилирования. eProf представляет собой профилировщик на базе функций. Предоставляемые интерфейсы могут быть встроены в программу для измерения задержки исполнения между любыми двумя точками в программе. eProf предоставляет несколько экземпляров профилировщика, каждый экземпляр которого способен измерять время в различных программных областях в одно и то же время. Предоставляемые интерфейсы перечислены в Таблице 8.3.

Таблица 8.3 Функции eProf

Интерфейс	*Описание*
eprof_init	Инициализация подсистемы профилировщика.
eprof_alloc	Создание экземпляра профилировщика.
eprof_start	Запуск профилирования.
eprof_stop	Остановка профилирования.
eprof_free	Удаление экземпляра профилировщика.
eprof_print	Печать результатов работы профилировщика.

Давайте сначала рассмотрим использование этих интерфейсов, а потом их реализацию. В Распечатке 8.9 278 мы профилируем две функции, которые выполняются параллельно. Запустите программу из Распечатки 8.9 278, чтобы получить результаты профилирования.

```
# gcc -o prof prof.c eprof.c -lpthread
# ./prof

eProf Result:
ID name           calls     usec/call
-------------------------------------------------
0: func1              1       10018200
1: func2             10         501894
```

eprof_print печатает данные профилирования всех экземпляров. Выходные данные показывают:

- Идентификатор экземпляра профилировщика
- Имя экземпляра профилировщика
- Количество раз вызова экземпляра (то есть, сколько раз была вызвана пара **eprof_start** и **eprof_stop**)
- Среднее время выполнения кода в микросекундах от вызова **eprof_start** до **eprof_stop**

Реализация eProf

В этом разделе мы рассмотрим реализацию eProf.

```
/*** eprof.c ***/
#include <stdio.h>
#include <sys/time.h>
```

MAX_INSTANCE определяет общее количество экземпляров профилировщика, поддерживаемое eProf. Вы можете изменить это значение для поддержки большего числа экземпляров профилировщика.

```
#define MAX_INSTANCE 5
```

Структуры данных eProf включают поле **eprof_curr**, которое содержит текущую метку времени и заполняется функцией **eprof_start**. **eprof_diff** хранит разницу в метках времени, записанных **eprof_stop** и **eprof_start**. **eprof_calls** хранит число, сколько раз для данного экземпляра была вызвана пара **eprof_start** - **eprof_stop**. Наконец, **eprof_label** хранит метку экземпляра. Оно заполняется функцией **eprof_alloc**.

```
/* Хранит текущую метку времени. Заполняется в eprof_start */
long long int eprof_curr[MAX_INSTANCE] ;
/*
 * timestamp(eprof_stop) - timestamp(eprof_start).
 * Заполняется в eprof_stop
 */
long long int eprof_diff[MAX_INSTANCE] ;
/*
 * Сколько раз была вызвана пара {eprof_start,eprof_stop}
 */
long eprof_calls[MAX_INSTANCE] ;
```

```
/* Метка экземпляра */
char * eprof_label[MAX_INSTANCE] ;
```

get_curr_time является ядром eProf. Она записывает текущее время. Для достижения высокой точности результатов профилирования необходимо, что эта функция была реализована с использованием источника времени высокого разрешения. Некоторые из вариантов:

- Использование аппаратных счётчиков, таких как Time Stamp Counter (TSC) на Pentium. **get_curr_time** может быть реализована с использованием TSC, как показано ниже.

```
static unsigned long long get_curr_time()
{
  unsigned long long int x;
  __asm__ volatile("rdtsc" : "=A"(x));

  /* convert x to microseconds */

  return x;
}
```

- Использование таймеров высокого разрешения POSIX (High-Resolution POSIX Timers, HRT), если они доступны. Для получения информации об их использовании обратитесь к Главе 7 176.
- Если ваша целевая платформа обеспечивает поддержку аппаратных таймеров высокого разрешения, то они могли бы быть использованы. Обычно интерфейсы для доступа к аппаратным таймерам недоступны в пространстве пользователя. Для экспорта таких интерфейсов в пространство пользователя вы можете использовать драйвер **kapi**, описанный в Главе 6 140.
- Наконец, если нет аппаратного источника времени, для получения текущего времени можно использовать **gettimeofday**.Точность **gettimeofday** определяется системными часами. Так, если значение **HZ** равно 100, точность составляет 10 мс. Если значение **HZ** равно 1000, точность составляет 1 мс.

```
/* Возвращаем текущее время */
static unsigned long long get_curr_time(){
  struct timeval tv;
  gettimeofday(&tv,NULL);
  return tv.tv_sec*1000000 + tv.tv_usec;
}
```

Различные структуры данных инициализирует **eprof_init**.

```
/* Инициализация */
void eprof_init () {
  int i;
  for (i=0; i<MAX_INSTANCE; i++) {
    eprof_diff[i] = 0;
    eprof_curr[i] = 0;
    eprof_calls[i] = 0;
    eprof_label[i] = NULL;
  }
```

```
}
```

eprof_alloc создаёт экземпляр профилировщика. Она проверяет число ненулевых записей в **eprof_label** и возвращает соответствующий индекс.

```
int eprof_alloc(char *label) {
  int id;
  for (id = 0; id < MAX_INSTANCE &&
                    eprof_label[id] != NULL; id++)
    ;
  if (id == MAX_INSTANCE)
    return -1;
  /* Store the label and return index */
  eprof_label[id] = label;
  return id;
}
```

eprof_start записывает в **eprof_curr** текущее время.

```
void eprof_start(int id) {
  if (id >= 0 && id < MAX_INSTANCE)
    eprof_curr[id] = get_curr_time();
}
```

eprof_stop сохраняет в **eprof_diff** разницу времени между моментом вызова себя и временем в **eprof_curr**. Напомним, что **eprof_curr** заполняется в **eprof_start**. Она также отслеживает в **eprof_calls**, сколько раз был вызван данный экземпляр профилировщика.

```
void eprof_stop(int id) {
  if (id >= 0 && id < MAX_INSTANCE) {
    eprof_diff[id] += get_curr_time() - eprof_curr[id];
    eprof_calls[id]++;
  }
}
```

eprof_free освобождает запись в **eprof_label**.

```
void eprof_free(char *label){
  int id;
  for (id = 0; id < MAX_INSTANCE &&
                  strcmp(label,eprof_label[id]) != 0; id++)
    ;
  if (id < MAX_INSTANCE)
    eprof_label[id] = NULL;
}
```

eprof_print в большей степени является функцией форматирования. Она печатает данные профилирования всех экземпляров профилировщика.

```
void eprof_print () {
  int i;
  printf ("\neProf Result:\n\n"
```

```
          "%s %.15s %20s %10s\n"
  "-------------------------------------------------\n",
          "ID", "name", "calls", "usec/call");
  for (i=0; i<MAX_INSTANCE; i++) {
    if (eprof_label[i]) {
      printf ("%d: %.15s %20d", i, eprof_label[i],
                                  eprof_calls[i]);
      if (eprof_calls[i])
        printf(" %15lld", eprof_diff[i] / eprof_calls[i]);
      printf ("\n");
    }
  }
}
```

Распечатка 8.9 Использование eProf

Распечатка 8.9.

```
/* prof.c */

#include <pthread.h>

/*
 * Заголовочный файл eProf. Исходный код разных интерфейсов eProf
 * находится в eprof.c
 */
#include "eprof.h"
#define MAX_LOOP 10 // счётчик циклов

/*
 * Эта функция работает в контексте разных потоков. Здесь мы
 * профилируем время, затраченное на выполнение основного цикла
 * функции
 */
void func_1(void * dummy){
  int i;
  /*
   * Создаём экземпляр профилировщика. Назначаем ему имя -
   * 'func1'
   */
  int id = eprof_alloc("func1");

  /*
   * Запускаем профилировщик. Аргументом является идентификатор
   * экземпляра, возвращаемый eprof_alloc
   */
  eprof_start(id);

  /* Мы профилируем этот цикл */
  for (i = 0; i < MAX_LOOP; i++){
    usleep(1000*1000);
  }
```

```
  /*
   * Останавливаем профилировщик. Мы печатаем результаты
   * профилирования в конце программы
   */
  eprof_stop(id);
}

/*
 * В этот примере мы профилируем каждую итерацию цикла.
 * Эта функция вызывается в main.
 */
void func_2(void){
  int i;
  /* Создаём экземпляр профилировщика */
  int id = eprof_alloc("func2");

  /*
   * Как видите, мы вызываем пару eprof_start и eprof_stop
   * много раз. Профилировщик записывает общее число раз
   * вызова этой пары, а затем показывает результаты,
   * усредняя их
   */
  for (i = 0 ; i < MAX_LOOP; i++){
    /* Запускаем профилировщик */
    eprof_start(id);
    usleep(500*1000);
    /* Останавливаем профилировщик */
    eprof_stop(id);
  }
}

/*
 * Основное приложение. Оно создаёт поток, запускающий функцию
 * func_1. Затем оно вызывает func_2 и ждёт завершения потока.
 * В конце оно печатает результаты профилирования.
 */
int main(){
  pthread_t thread_id;

  /*
   * Инициализируем eProf. Это должно быть сделано во время
   * запуска программы.
   */
  eprof_init();
  /* Создаём поток, который запускает функцию func_1 */
  pthread_create(&thread_id, NULL, func_1, NULL);
  /* Запускаем функцию func_2 */
  func_2();
  /* Ждём завершения потока */
  pthread_join(thread_id, NULL);
  /* Печатаем результаты */
  eprof_print();
  return 0;
}
```

8.7.2 OProfile

OProfile представляет собой инструмент профилирования для Linux. Он имеет возможность оценки производительности всей системы, включая ядро, совместно используемые библиотеки и приложения. Он также может быть использован в профилирования модулей ядра и обработчиках прерывания. Он незаметно работает в фоновом режиме собирая информацию при низких накладных расходов.

В этом разделе мы обсудим использование OProfile в ядре Linux версии 2.6. Для этого вам необходимо пересобрать ядро с установленной опцией **CONFIG_OPROFILE**. Кроме того, необходима кросс-компиляция OProfile для вашей целевой платформы. Для настройки OProfile для вашей целевой платформы, пожалуйста, следуйте инструкциям кросс-компиляции в Разделе 8.2 249 и инструкциям по установке OProfile, доступным по адресу http://oprofile.sourceforge.net.

Для записи различных событий, таких как циклы ЦП, промахи в кэше, сбросы TLB, и так далее, OProfile использует различные аппаратные счетчики производительности. В архитектурах, которые не имеют счётчиков производительности, для сбора данных OProfile использует прерывание часов реального времени. Если часы реального времени тоже не доступны, OProfile возвращается к прерыванию таймера. Вы также можете принудительно включить режим прерывания по таймеру, передав в командной строке загрузки ядра **oprofile.timer = 1**. Обратите также внимание, что различные события, такие как сброс TLB, промахи в кэше и тому подобное, обычно связаны со счётчиками производительности, не доступными в режиме работы по часам реального времени/прерыванию по таймеру.

В этом разделе мы профилируем с помощью OProfile на ПК приложение видео-плеера, FFmpeg™ (http://ffmpeg.sourceforget.net). Идея этого примера в том, чтобы быстро ознакомить вас с OProfile. Для полноценного использования OProfile обратитесь к http://oprofile.sourceforge.net. Для профилирования FFmpeg выполняем описанные ниже шаги. Обратите внимание, что для выполнения команд OProfile вам необходимо иметь права суперпользователя.

1. ***Настройка OProfile***: мы не хотим профилировать ядро.

```
# opcontrol --no-vmlinux
```

2. ***Запуск профилировщика***:

```
# opcontrol -start
Using default event: GLOBAL_POWER_EVENTS:100000:1:1:1
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
```

3. ***Запуск приложения***:

```
# cd /usr/local/bin
# ./ffplay_g /data/movies/matrix.mpeg &
```

4. ***Сбор данных***: команду для сбора данных, а также вывод информации показывает Распечатка 8.10 281. Первый столбец вывода это общее количество измерений, сделанных в функции, а второй столбец показывает относительный процент от общего

числа измерений для функции. Так как **GLOBAL_POWER_EVENTS** представляет процессорное время, мы можем говорить, что во время воспроизведения видео функция **synth_filter** использовала 28.8 процента от общего процессорного времени, в то время как функция **mpeg_decode_mb** использовала 16.5 процентов от общего процессорного времени.

5. ***Получение аннотированных исходных текстов***: приложение должно быть скомпилировано с включённой отладкой. В этом примере аннотированные исходные файлы для всех измерений создаются в катаклоге **/usr/local/bin/ann**. Аннотированные исходники для символов **synth_filter** и **mpeg_decode_mb** содержат **mpegaudiodec.c** и **mpeg12.c**, соответственно. Пример их содержимого показан в Распечатке 8.11 282.
6. Получение полного отчёта о производительности системы:

```
# opreport --long-filenames
CPU: P4 / Xeon with 2 hyper-threads, speed 2993.82 MHz
(estimated)
Counted GLOBAL_POWER_EVENTS events (time during which
processor is not stopped) with a unit mask of 0x01
(count cycles when processor is active) count 100000
GLOBAL_POWER_E...|
 samples|       %|
------------------
223651 73.9875 /no-vmlinux
22727   7.5185 /lib/tls/libc.so.6
15134   5.0066 /usr/bin/local/ffplay_g
7329    2.4246 /usr/bin/nmblookup
       ...
       ...
```

7. ***Выключение профилировщика***:

```
# opcontrol --shutdown
```

Распечатка 8.10 Вывод OProfile

Распечатка 8.10.

```
# opreport -l ./ffplay_g
CPU: P4 / Xeon with 2 hyper-threads, speed 2993.82 MHz
(estimated)
Counted GLOBAL_POWER_EVENTS events (time during which
processor is not stopped) with a unit mask of 0x01 (count
cycles when processor is active) count 100000
samples  % symbol name
120522   28.8691  synth_filter
68783    16.4758  mpeg_decode_mb
36292     8.6932  ff_simple_idct_add_mmx
24678     5.9112  decode_frame
15623     3.7422  MPV_decode_mb
15514     3.7161  put_pixels8_mmx
14861     3.5597  clear_blocks_mmx
        ...
```

```
        ...
```

Распечатка 8.11 Вывод OProfile с ассоциированными исходными файлами

Распечатка 8.11.

```
# opannotate --source --output-dir=/usr/local/bin/ann ./ffplay_g
# vim /usr/local/bin/ann/data/ffmpeg/libavcodec/mpegaudiodec.c
               :static void synth_filter(MPADecodeContext *s1,
               :           int ch, int16_t *samples, int incr,
               :            int32_t sb_samples[SBLIMIT])
    179 0.0407 :{ /* synth_filter total: 126484 28.7945 */
        ...
     75 0.0171 :  offset = s1->synth_buf_offset[ch];
     89 0.0203 :  synth_buf = s1->synth_buf[ch] + offset;
        ...
   1097 0.2497 :   p = synth_buf + 16 + j;
  38956 8.8685 :   SUM8P2(sum, +=, sum2, -=, w, w2, p);
   1677 0.3818 :   p = synth_buf + 48 - j;
  41582 9.4663 :   SUM8P2(sum, -=, sum2, -=, w + 32, w2 + 32, p);
        ...
```

```
# vim /usr/local/bin/ann/data/ffmpeg/libavcodec/mpeg12.c
               :static int mpeg_decode_mb(MpegEncContext *s,
               :                          DCTELEM block[12][64])
    296 0.0674 :{ /* mpeg_decode_mb total: 72484 16.5012 */
               :    int i, j, k, cbp, val, mb_type, motion_type;
    155 0.0353 :    const int mb_block_count = 4 + (1<< s-
                                               >chroma_format)
        ...
    257 0.0585 :    if (s->mb_skip_run-- != 0) {
     74 0.0168 :        if(s->pict_type == I_TYPE){
        ...
               :        /* skip mb */
      2 4.6e-04 :        s->mb_intra = 0;
    114 0.0260 :        for(i=0;i<12;i++)
     54 0.0123 :            s->block_last_index[i] = -1;
     51 0.0116 :        if(s->picture_structure == PICT_FRAME)
        ...
```

8.7.3 Kernel Function Instrumentation

В последних двух разделах мы обсудили методы профилирования приложений пользовательского пространства. В этом разделе мы рассмотрим Kernel Function Instrumentation (KFI), инструмент для профилирования функций ядра. (Для профилирования ядра вы также можете использовать OProfile.)

KFI может быть использован для измерения времени, затраченного любой функцией ядра. Он основан на функциях контроля и возможностях профилирования GCC, используя флаг GCC **-finstrument-functions**. (* GCC ожидает, что пользователь определит две функции: __cyg_profile_func_enter и __cyg_profile_func_exit. Когда используется флаг -finstrument-functions, эти функции добавляются GCC на входе и выходе из функции, соответственно. KFI определяет эти две функции, в которых он собирает данные профилирования.) KFI

доступен в виде патча ядра и набора утилит для ядер версии 2.4 и 2.6. Патч ядра добавляет поддержку для генерации данных профилировщика и утилиты, используемые для последующего анализа данных профилирования. Загрузите их с www.celinuxforum.org. Примените патч ядра и включите при сборке ядра опцию **CONFIG_KFI**. Во время конфигурации ядра вы также можете решить сделать KFI ***работающим статически***, если вы хотите измерять время загрузки ядра и время, затраченное различными функциями ядра во время загрузки. Это достигается включением флага конфигурации **KFI_STATIC_RUN**. В этом разделе мы обсудим ***динамически работающий*** KFI; то есть мы собираем данные профилирования, когда ядро запущено и работает. Для шагов, которые необходимо сделать для работы статически, пожалуйста, следуйте инструкциям в файле **README.kfi**, который является частью пакета KFI.

Набор инструментальных средств KFI

Прежде чем перейти к использованию KFI, мы должны узнать о различных инструментах, которые являются частью набора инструментов KFI. Набор инструментов KFI состоит из трёх утилит: **kfi**, **kfiresolve** и **kd**. Триггеры **kfi** профилируют сбор данных в ядре. **kfiresolve** и **kd** изучают собранные данные и представляют их пользователю в удобном для чтения формате.

Вам необходима кросс-компиляция программы **kfi** для вашей целевой платформы. Командами **kfi** являются:

```
# ./kfi
Usage: ./kfi <cmds>
commands: new, start, stop, read [id], status [id], reset
```

- **new**: начало новой сессии профилирования. Каждый сеанс опознаётся по идентификатору выполнения.
- **start**: запуск профилирования.
- **stop**: остановка профилирования.
- **read**: чтение данных профилировщика из ядра.
- **status**: проверка состояние вашего сеанса профилирования.
- **reset**: сброс KFI.

Большую часть времени вы не хотите профилировать всё ядро целиком. Вы можете просто хотеть профилировать обработчики прерываний или какие-то специфические функции ядра. KFI обеспечивает фильтры для выборочного профилирования. Прежде чем начать сессию профилировщика, необходимо установить соответствующие фильтры. Чтобы настроить фильтры, необходимо модифицировать программу **kfi**. Исходный код программы **kfi** находится в файле **kfi.c**, а определения структур находится в **kfi.h**.

Каждый сеанс профилирования связан со **struct kfi_run**, которая содержит все необходимые детали сессии.

```
typedef struct kfi_run {
    ...
  struct kfi_filters filters;
    ...
} kfi_run_t;
```

Фильтры могут быть определены путём установки соответствующим образом полей структуры **kfi_filters_t**.

```
typedef struct kfi_filters {
  unsigned long min_delta;
  unsigned long max_delta;
  int no_ints;
  int only_ints;

  void** func_list;
  int func_list_size;
    ...
} kfi_filters_t;
```

Полями **kfi_filters_t** являются:

- **min_delta**: не профилировать функции с временем выполнения меньшим, чем **min_delta**.
- **max_delta**: не профилировать функции с временем выполнения большим, чем **max_delta**.
- **no_ints**: не профилировать функции, которые выполняются в контексте прерываний.
- **only_ints**: профилировать функции, которые выполняются в контексте прерываний.
- **func_list**: профилировать функции, указанные в списке.
- **func_list_size**: если задан **func_list**, то **func_list_size** содержит число записей в **func_list**.

myrun в **kfi.c** представляет собой структуру типа **kfi_run_t**.

```
struct kfi_run myrun = {
  0, 0,
  { 0 },
  { 0 },
  { 0 }, /* фильтры struct kfi_filters_t */
  myrunlog, MAX_RUN_LOG_ENTRIES, 0,
  0,
  NULL,
};
```

Как видите, все поля членов этого фильтра установлены в ноль. Вы должны изменить члены фильтров в соответствии с вашими потребностями. Например, предположим, что необходимо профилировать функцию ядра **do_fork**. Чтобы создать соответствующий фильтр, должны быть выполнены следующие шаги:

1. Откройте файл **System.map** вашего ядра и найдите **do_fork**.

```
....
c0119751 T do_fork
....
```

2. Определите **func_list**.

```
void *func_list[] = {0xc0119751};
```

3. Модифицируйте члены фильтра **myrun**.

```
struct kfi_run myrun = {
  ...
{ 0,0,0,0,func_list,1 },
  ...
};
```

4. Перекомпилируйте **kfi.c** и сгенерируйте **kfi**.

Аналогичным образом могут быть установлены и другие фильтры. Сгенерированные профилировщиком данные могут быть исследованы с помощью двух сценариев на языке Phython, **kfiresolve** и **kd**. Мы обсуждаем их использование в следующем разделе.

Использование KFI

В этом разделе мы обсудим с помощью примера динамическую работу KFI. Мы профилируем функцию ядра **do_fork**. Чтобы создать фильтры на базе функций, сделаны все изменения, упомянутые в предыдущем разделе. Работа происходит с ядром версии 2.6.8.1 с включённым KFI на машине P4 2.8 ГГц. На пример программы:

```
/* sample.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[]){
  int i;
  printf("Pid = %d\n", getpid());
  for (i = 0; i < 2; i++){
    if (fork() == 0)
      exit(0);
  }
}
```

Шаги следующие:

1. Создаём узел устройства:

```
mknod /dev/kfi c 10 51
```

2. Чтобы по-новый запустить KFI, сбрасываем **kfi**.

```
./kfi reset
```

3. Создаём новую сессию KFI.

```
# ./kfi new
new run created, id = 0
```

Заметим, что если при выполнении этой команды вы получаете ошибку выделения памяти, укажите для **MAX_RUN_LOG_ENTRIES** в **kfi.h** несколько меньшее число.

4. Запускаем профилирование.

```
# ./kfi start
runid 0 started
```

5. Запускаем приложение.

```
# ./sample
Pid = 4050
```

6. Останавливаем профилирование.

```
./kfi stop
runid 0 stopped
```

7. Читаем данные профилировщика из ядра.

```
./kfi read 0 > sample.log
```

8. Выполняем заключительную обработку данных.

```
./kfiresolve.py sample.log /boot/System.map > sample.out
```

Результат работы профилировщика содержит файл **sample.out**. Все временные интервалы измеряются с точностью до микросекунды. В выводе, показанном в Распечатке 8.12 288, **Entry** это время входа в функцию, а **Delta** - время, затраченное на выполнение функции. В выходных данных записи, соответствующие PID 4095, получены от нашего приложения. Две другие записи, соответствующие PID 3982, получены от оболочки.

Для анализа данных профилировщика вы также можете использовать **kd**.

```
# ./kd sample.out
Function                  Count Time     Average  Local
------------------------ ----- -------- -------- --------
do_fork                    4      195       48      195
```

Перенос KFI на другую платформу

KFI не зависит от архитектуры. Ядром KFI является функция **kfi_readclock**, которая возвращает текущее время в микросекундах.

```
static inline unsigned long __noinstrument kfi_readclock(void)
{
  unsigned long long t;
  t = sched_clock();
  /* переводим в микросекунды */
  do_div(t,1000);
  return (unsigned long)t;
}
```

kfi_readclock вызывает **sched_clock**, которая обычно определена в **arch/<your-arch>/kernel/time.c**.

```
/*
 * Часы планировщика – возвращают текущее время в наносекундах.
 */
unsigned long long sched_clock(void)
{
  return (unsigned long long)jiffies * (1000000000 / HZ);
}
```

Вышеописанная реализация **sched_clock** не зависит от архитектуры и возвращает значение, основанное на тиках. На многих встраиваемых платформах точность тиков составляет 10 мс. Таким образом, чтобы иметь лучшую точность результатов профилирования, вы должны предоставить лучшую **sched_clock** на основе какого-либо источника аппаратного времени, с точностью по крайней мере до микросекунды. Например, для процессоров x86 **sched_clock** реализуется с помощью счётчика времени с поддержкой TSC.

```
unsigned long long sched_clock(void)
{
  unsigned long long this_offset;

  .....

  /* Читаем счётчик времени */
  rdtscll(this_offset);

  /* Возвращаем значение в нс */
  return cycles_2_ns(this_offset);
}
```

Если вы не хотите использовать **sched_clock** как источник синхронизации, в качестве альтернативы вы можете написать свой собственный вариант **kfi_readclock**. Одна такая реализация включена в патч для PPC.

```
static inline unsigned long kfi_readclock(void)
{
  unsigned long lo, hi, hi2;
  unsigned long long ticks;
  do {
    hi = get_tbu();
    lo = get_tbl();
    hi2 = get_tbu();
  } while (hi2 != hi);
  ticks = ((unsigned long long) hi << 32) | lo;
  return (unsigned long)((ticks>>CLOCK_SHIFT) &
                                      0xffffffff);
}
```

Распечатка 8.12 Пример работы KFI

Распечатка 8.12.

```
Kernel Instrumentation Run ID 0

Logging started at 2506287415 usec by system call
Logging stopped at 2506609002 usec by system call

Filters:
    1-entry function list
    no functions in interrupt context
    function list

Filter Counters:
No Interrupt functions filter count = 0
Function List filter count = 57054552
Total entries filtered = 57054552
Entries not found = 0

Number of entries after filters = 4


 Entry    Delta     PID       Function      Called At
-------- --------  -----      --------   ---------------
  137565    82      3982       do_fork   sys_clone+0x4a
  138566    22      4050       do_fork   sys_clone+0x4a
  138661    21      4050       do_fork   sys_clone+0x4a
  320729    70      3982       do_fork   sys_clone+0x4a
```

Глава 9, Встроенная графика

В попытке обеспечить лучшее взаимодействие с пользователем, современные электронных продукты предоставляют графический пользовательский интерфейс. Сложность интерфейса зависит от продукта и сценария его использования. Для примера рассмотрим такие устройства: DVD/MP3 плеер, мобильный телефон и КПК. DVD/MP3 плеер требует наличия примитивного интерфейса, способного пролистывать содержимое CD/DVD. Создание плейлистов и поиска треков будут наиболее сложными операциями, которые можно сделать на DVD/MP3 плеере. Очевидно, что это не является достаточным для мобильного телефона. Мобильный телефон имеет более широкую функциональность. Наиболее сложные требования в КПК. На КПК должна быть возможность запустить почти все приложения, такие как текстовые редакторы, электронные таблицы, планировщики и так далее.

Появляются различные вопросы, касающиеся поддержки графики на встраиваемом Linux.

- Что входит в состав графической системы? Как это работает?
- Могу ли я использовать графику рабочего стола Linux на встраиваемых системах, как она есть?
- Какие возможности выбора для графики на встраиваемых системах Linux?
- Есть ли общее решение, которые может решать весь спектр задач встраиваемых устройств, требующих графику (например, от мобильного телефона до DVD проигрывателя)?

В этой главе изложены ответы на эти вопросы.

9.1 Графическая система

Графическая система несёт ответственность за

- Управление устройством отображения
- Управление в случае необходимости одним или более интерфейсом ввода для человека
- Предоставление абстракции для базового устройства отображения (для использования приложениями)
- Управление разными приложениями, так что они сосуществуют и совместно эффективно используют дисплей и оборудование ввода

Независимо от операционных систем и платформ, обычная графическая система может быть представлена в виде разных модульных уровней, как показано на Рисунке 9.1.

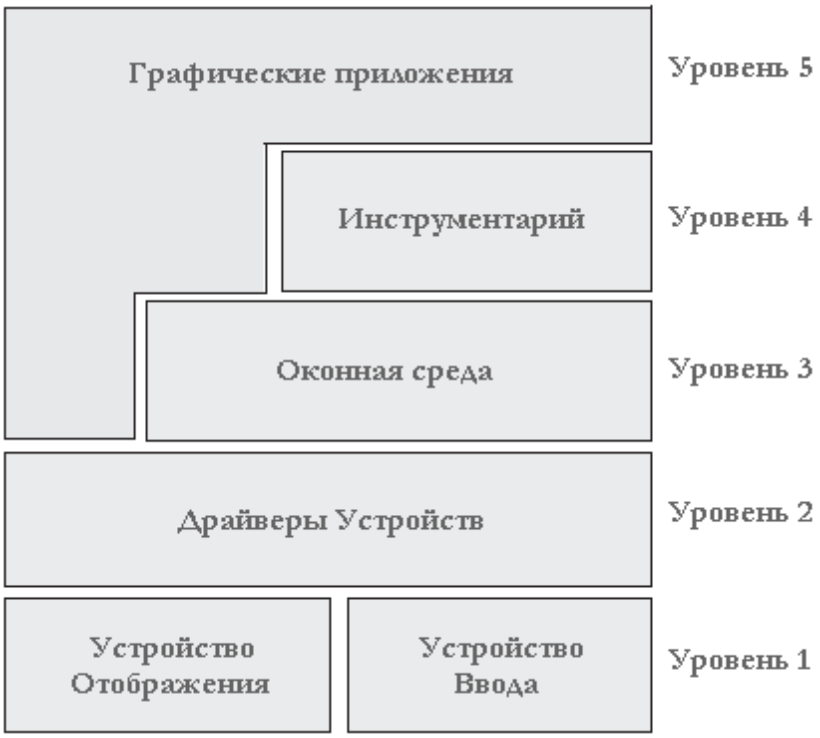

Рисунок 9.1 Архитектура графической системы.

Разные уровни это:

- Уровень 1 является графическим оборудованием и оборудованием ввода, основными аппаратными компонентами в любой системе графики. Например, банкомат имеет сенсорный дисплей, что является сразу и его интерфейсом ввода и оборудованием отображения, DVD плеер имеет видеовыход на телевизор, а передняя панель ЖК дисплея имеет свое оборудование отображения и дистанционное управление в качестве входного интерфейса.
- Уровень 2 представляет собой уровень драйверов, который обеспечивает взаимодействие с операционной системой. Каждая операционная система имеет свой собственный механизм взаимодействия и производители устройств стараются убедиться, что они обеспечивают драйверы для всех популярных операционных систем. Например, карты с чипом NVIDIA® поставляются с драйверами для Linux и Windows.
- Уровень 3 состоит из оконной среды, которая представляет собой механизм отрисовки, отвечающий за создание графики и механизм шрифтов, ответственный за отрисовку шрифтов. Например, механизм отрисовки обеспечивает линии, прямоугольники и функциональные возможности для отрисовки других геометрических форм.
- Уровень 4 является инструментальным уровнем. Набор инструментальных средств строится поверх определённой оконной среды и предоставляет API для использования приложением. Некоторые инструменты доступны поверх нескольких оконных сред и тем самым обеспечивают переносимость приложений. Инструментарии обеспечивают функции для рисования сложных элементов управления, таких как кнопки, поля ввода, списки, и так далее.
- Самый верхний уровень - это графическое приложение. Приложение не всегда использует инструментарий и оконную среду. С помощью некоторой минимальной абстракции или промежуточного уровня можно было бы написать приложение, которое

непосредственно взаимодействует с оборудованием через интерфейс драйвера. Кроме того, некоторые приложения, такие как видео-плеер, требуют интерфейс ускорителя для обхода графического уровня и прямое взаимодействие с драйвером. Для таких случаев графическая система предусматривает специальное взаимодействие, такое как знаменитый Direct-X в Windows. Рисунок 9.2 сравнивает уровни в различных операционных системах.

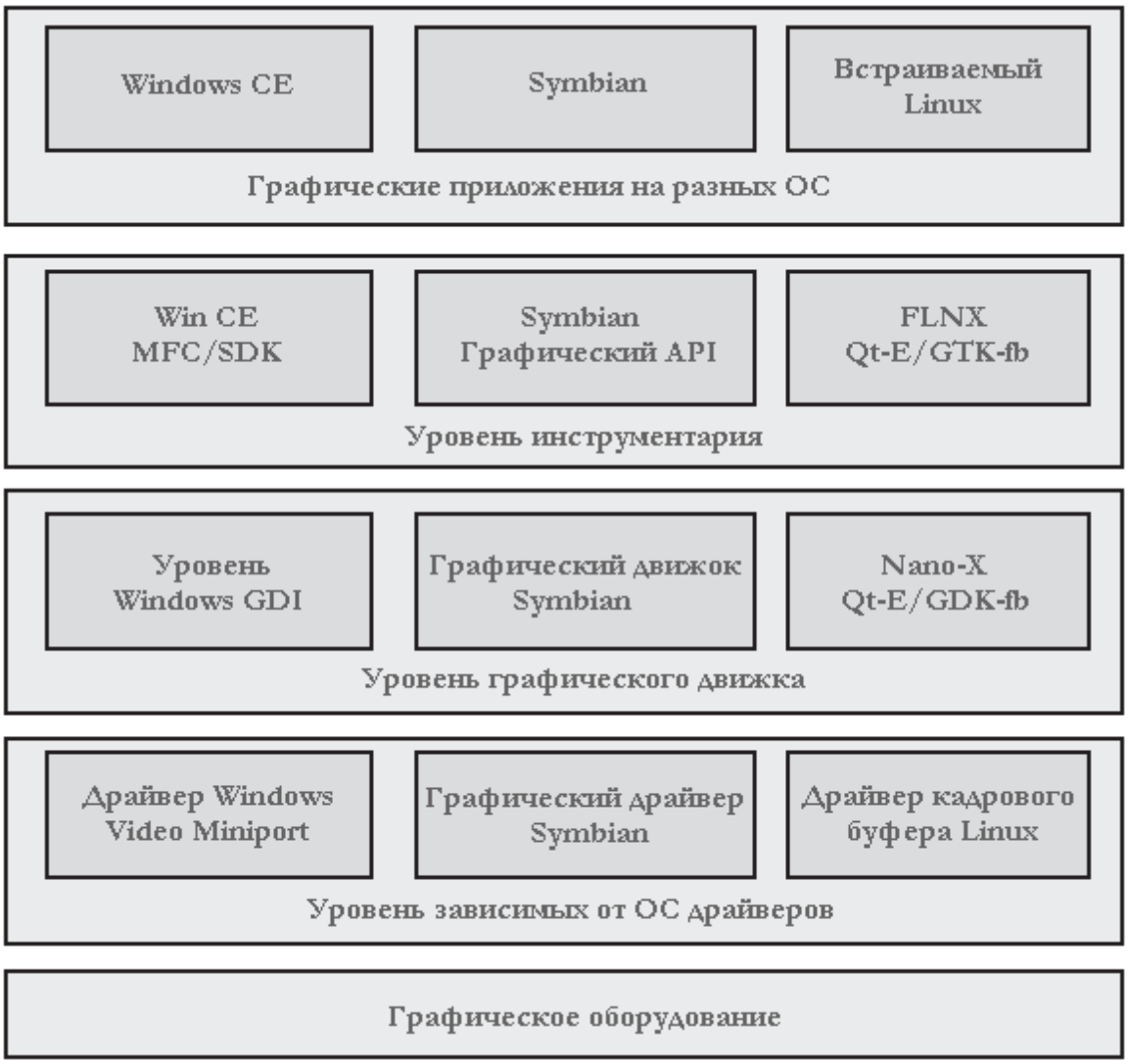

Рисунок 9.2 Слои графики в различных операционных системах

В этой главе постепенно подробно обсуждается каждый уровень по отношению к встраиваемому Linux.

9.2 Графика настольного Linux - графическая система X

Графику рабочего стола Linux предоставляет X Windowing System (оконная система X). Мы используем её как пример, чтобы понять многоуровневую архитектуру полноценного графического решения.

Система X первоначально написана для настольного компьютера. Графические карты

для настольных компьютеров следуют определённым стандартам, VGA/SVGA. Устройства ввода, такие как драйверы ввода с мыши и клавиатуры, также имеют стандарты. Поэтому обычный драйвер поддерживает оборудование для отображение и ввода. Система X реализует интерфейс драйвера, необходимый для взаимодействия с оборудованием отображения ПК. Интерфейс драйвера изолирует остальную часть системы X от деталей, связанных со специфическим оборудованием.

Оконная среда в X имеет модель клиент/сервер. X приложения являются клиентами; они взаимодействуют с сервером и выдают запросы, а также получают информацию от сервера. X сервер управляет дисплеем и обслуживает запросы от клиентов. Приложениям (клиентам) необходимо только знать, как общаться с сервером, и нет необходимости заботится о деталях отрисовки графики на устройстве отображения. Этот механизм коммутации (протокол) может работать поверх любого механизма межпроцессного взаимодействия, которые обеспечивает надёжный поток байтов. X использует для этого сокеты: как результат, X Протокол. Поскольку X основана на сокетах, она может работать по сети и может быть использована также для дистанционного отображения. Для отрисовки объектов на экране, клиенты X используют API, предоставляемый оконной системой X. Эти интерфейсы являются частью библиотеки X-lib, которая связана с клиентским приложением.

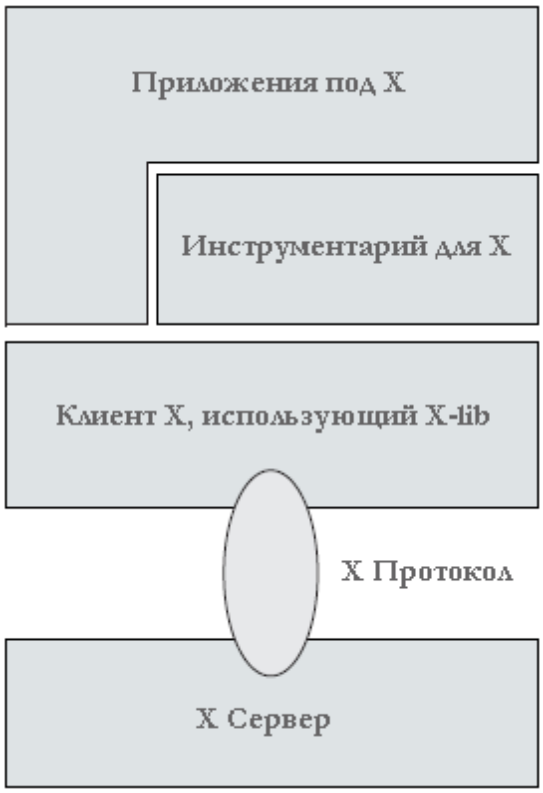

Рисунок 9.3 Арихитектура инструментария X.

Архитектура ***набора инструментов системы X*** показана на Рисунке 9.3. Она включает API, которые предоставляют возможности создания окон. Элементы управления, такие как списки, кнопки, флажки, поля ввода, и так далее, а также окна, построенные поверх примитивов X-lib и коллекцию таких библиотек, называемых ***виджетами/ инструментариями*** (widget/toolkit). Набор инструментальных средств делает жизнь программиста приложения легкой, предоставляя простые функции для рисования элементов управления. Так как к серверу подключено множество клиентов, возникает необходимость

управления различными окнами клиентов. X сервер предоставляет ***оконный менеджер***, другой клиент X, но единственный привилегированный. Архитектура X предоставляет специальные функции для выполнения оконным менеджером, такие действия, как перемещение, изменение размеров, сворачивание и разворачивание окна, и так далее. Для получения дополнительной информации вы можете посетить официальный сайт X11, http://www.x.org.

9.2.1 Встраиваемые системы и X

X сильно ориентирована на сеть и на встроенных системах напрямую не применяется. Причины, по которым X не может быть использована во встраиваемой системе, перечислены далее.

- X имеет механизмы для экспорта изображения через сеть. Во встраиваемых системах это не требуется. Клиент/серверная модель не направлена на однопользовательские среды, использующиеся на встраиваемых системах.
- X имеет много зависимостей, таких как X сервер шрифтов, X менеджер ресурсов, X оконный менеджер и список можно продолжать. Всё вышеперечисленное увеличивает размер X и её требования к памяти.
- X была написана, чтобы работать на полноресурсных гигантских машинах с процессором Pentium. Поэтому работа X системы "как есть" на небольших по мощности и скорости встраиваемых микропроцессорах не представляется возможным.

Требованиями для графической структуры на встраиваемой системе являются:

- Быстрый/близкий к реальному масштабу времени отклик
- Использование малого объёма памяти
- Небольшой размер библиотеки инструментария (дискового пространства)

Для X было сделано много изменений и для работы на встраиваемых платформах стали доступны микроверсии. Tiny-X и Nano-X - примеры популярных и успешных встраиваемых версий на базе графической системы X. Мы обсуждаем Nano-X в Разделе 9.6 317.

9.3 Введение в оборудование для отображения

В этом разделе мы рассмотрим различную терминологию графики, а также общие графические аппаратные функции.

9.3.1 Система отображения

Каждая графическая система отображения имеет контроллер видео/дисплея, который является основным графическим оборудованием. Видео контроллер имеет область памяти, известную как ***кадровый буфер***. На экране отображается содержание кадрового буфера.

Любое изображение на экране состоит из горизонтальных строк развертки, отображаемых оборудованием дисплея. После каждой горизонтальной строки развертки луч съезжает в вертикальном направлении и прочерчивает следующую горизонтальную строку развертки. Таким образом, всё изображение состоит из горизонтальных линий, отсканированных сверху внизу, а каждый цикл сканирования называется ***обновлением***. Количество обновлений экрана происходящих в секунду называется ***частотой обновления***.

Изображение, до его представления на экране, становится доступным в памяти кадрового буфера контроллера. Это цифровое изображение делится на отдельные области памяти, называемые ***пикселями*** (точками, сокращение от pictorial elements, элементы отображения). Количество пикселей в горизонтальном и вертикальном направлении называется

разрешением экрана. Например, разрешение экрана 1024 × 768 - это матрица пикселей из 1024 столбцов и 768 строк. Эти 1024 × 768 пикселей передаются на экран за один цикл обновления.

Каждый пиксель - это, по существу, информация о цвете по определённому (строка, столбец) индексу отображаемой матрицы. Цветовая информация, представленная в пикселях, обозначается с использованием стандартного представления цвета. Цвет либо представлен в области RGB, используя Красные, Зелёные и Синие биты, или в области YUV (* более часто YUV используется в системах кодирования/декодирования видео), используя значения яркости (Y) и цветности (U и V). RGB является общим представлением в большинстве графических систем. Расположение бит и число битов, занимаемых каждым цветом, приводят к различным форматам, перечисленным в Таблице 9.1.

Таблица 9.1 Форматы цвета RGB

Формат	*Биты*	*Красный*	*Зелёный*	*Синий*	*Цветов*
Монохромный	1	—	—	—	2
Индексный - 4 бита	4	—	—	—	2^4 = 16
Индексный - 8 бит	8	—	—	—	2^8 = 256
RGB444	12	4	4	4	2^12
RGB565	16	5	6	5	2^16
RGB888	24	8	8	8	2^24

Число или диапазон значений цвета, которое будет отображаться, определяет количество байт, занимаемых одним пикселем, выражаемое в термине ***размер пикселя***. Рассмотрим, например, мобильное устройство отображения с разрешением 160 × 120 с 16 цветами. Размер пикселя здесь 4 бита на пиксель (16 уникальных значений наилучшим образом представляются с помощью 4-х битов), другими словами, ½ байта на пиксель. Требуемый размер памяти кадрового буфера рассчитывается по формуле

Память кадрового буфера = Ширина дисплея * Высота дисплея * Байты-на-Пиксель

В приведённом выше примере требуемая область памяти - это 160 × 120 × (½) байт. Большинство реализаций кадрового буфера являются линейными в том смысле, что они представляют собой непрерывный место в памяти, как и массив. Начальный байт каждой строки отделён на постоянное число байт, называемое шириной строки или шаг по индексу (stride). В нашем примере шириной строки является 160 * (½) байт. Таким образом, расположением любого пикселя (x, y) = (line_width * y) + x. Рисунок 9.4 иллюстрирует расположение пикселя (40, 4), выделенного заливкой.

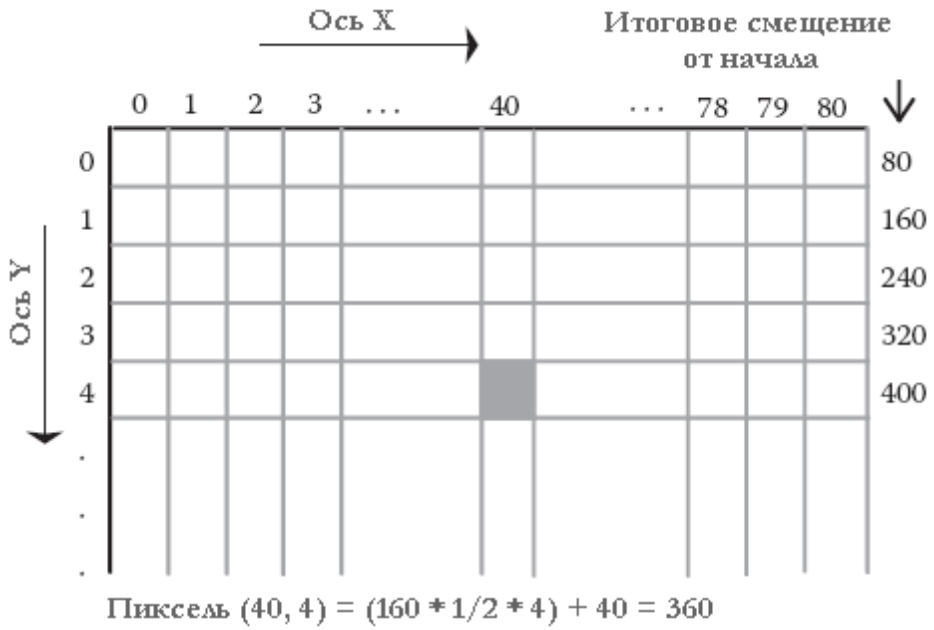

Рисунок 9.4 Расположение пикселей в линейном кадровом буфере.

Теперь посмотрим на первые три записи в Таблице 9.1. Они отличаются от остальных записей, в смысле, что они являются ***индексными цветовыми форматами***. Индексные форматы присваивают индексы, которые соответствуют определённому цветовому оттенку. Например, в системе с монохромным дисплеем, только с одним битом, с двумя значениями (0 или 1), можно привязать 0 к красному и 1 к чёрному. По сути то, что мы имеем теперь, представляет собой таблицу со значениями цвета, зависящими от их индекса. Эта таблица называется ***справочной таблицей цветов (Color LookUp Table, CLUT)***. Их также называют ***цветовыми палитрами***. Каждая запись в палитре связывает значение пикселя с заданным пользователем уровнем интенсивности красного, зелёного и синего. Теперь, после ознакомления с CLUT, обратите внимание, как содержимое кадрового буфера получает перевод в различные цветовые оттенки на основе значений в CLUT. Например, CLUT можно разумно использовать на мобильном телефоне для изменения цветовых тем, как показано на Рисунке 9.5.

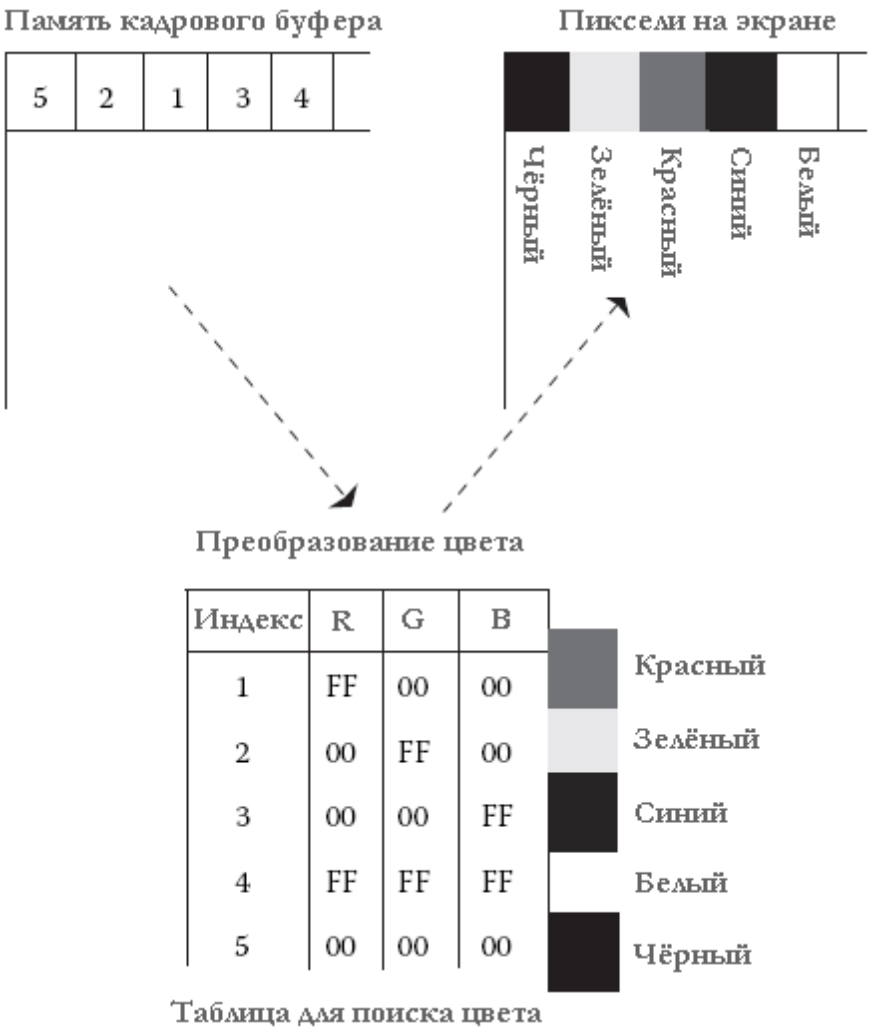

Рисунок 9.5 Работа с таблицей цветов.

9.3.2 Интерфейс для ввода

Оборудование ввода встраиваемых систем обычно использует кнопки, дистанционное управление по ИК, сенсорный экран, и так далее. Для нормальных интерфейсов ввода, таких как клавиатуры и мыши, в ядре Linux доступны стандартные интерфейсы. Устройства дистанционного управления по ИК могут быть подключены через последовательный интерфейс. Взаимодействием ИК приемников и передатчиков с приложениями Linux занимается проект LIRC. Ядро версии 2.6 имеет чётко определённый уровень устройств ввода, которое охватывает все классы устройств ввода. HID (Human Interface Device, устройство для взаимодействия с человеком) представляет собой огромную тему и его обсуждение выходит за рамки данной главы.

9.4 Графика встраиваемого Linux

В предыдущем разделе мы обсуждали оборудование, относящееся к встраиваемым системам. В следующих разделах мы подробно рассмотрим различные графические компоненты встраиваемого Linux. Рисунок 9.6 даёт краткий обзор различных вовлечённых уровней.

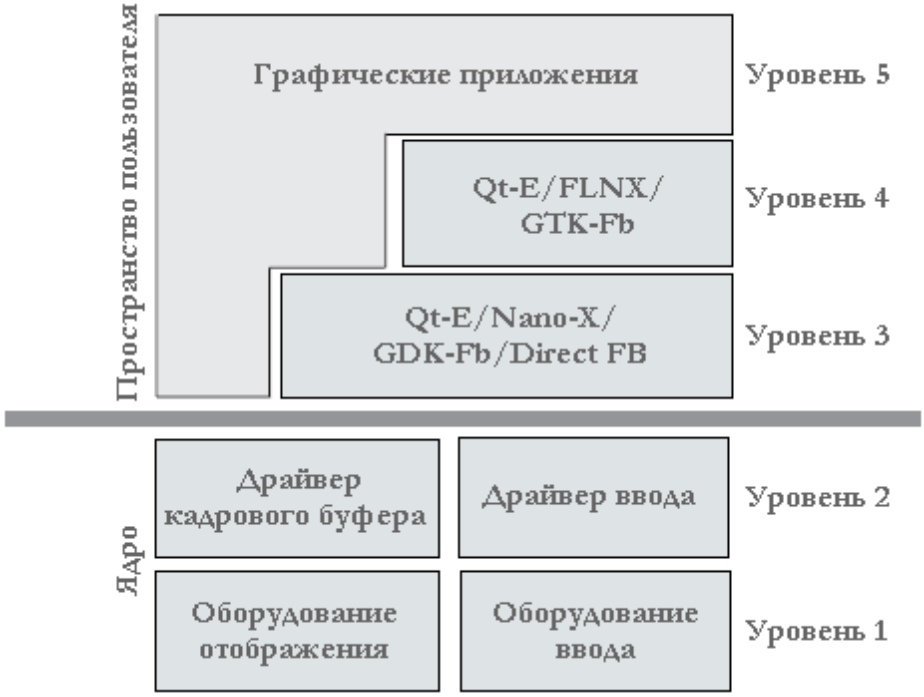

Рисунок 9.6 Графическая система встраиваемого Linux.

9.5 Графический драйвер встраиваемого Linux

Первый драйвер кадрового буфера был введён в ядро версии 2.1. Оригинальные драйвер кадрового буфера был разработан, чтобы просто обеспечить консоль системам, которым недостаёт видеоадаптеров со встроенными текстовыми режимами (таким, как m68k). Драйвер обеспечил средства для эмуляции текстового режима консоли поверх обычных базирующихся на пикселях системах отображения. Из-за упрощённого дизайна и простого в использовании интерфейса, драйвер кадрового буфера находил применение в графических приложениях на всех типах видеокарт. Многие инструментарии, которые были в основном написаны для традиционных оконных систем X, были портированы для работы с интерфейсом кадрового буфера. Вскоре для этого нового графического интерфейса Linux были написаны "с нуля" новые оконные среды. Сегодня драйвер кадрового буфера в ядре - это больше уровень абстракции видеокарты, который обеспечивает для графических приложений общий интерфейс устройства. Сегодня практически все графические приложения на встраиваемых системах Linux используют для отображения графики поддержку ядром кадрового буфера. Некоторыми из причин широкого использования интерфейса кадрового буфера являются:

- Удобство в использовании и простой интерфейс, который зависит от основного принципа графических устройств, линейного кадрового буфера
- Предоставляет приложениям пользовательского пространства прямой доступ к видеопамяти, огромная свобода программирования
- Устраняет зависимость от наследия архитектуры отображения, нет сети, нет модели клиент-сервер; простые однопользовательские приложения с прямым отображением
- Обеспечивает на Linux графику без излишнего потребления памяти и системных ресурсов

9.5.1 Интерфейс кадрового буфера Linux

Кадровый буфер на Linux реализован как интерфейс символьного устройства. Это означает, что приложения делают стандартные системные вызовы, такие как **open()**, **read()**, **write()** и **ioctl()** на указанном имени устройства. Устройство с кадровым буфером доступно в пространстве пользователя как **/dev/fb[0-31]**. В Таблице 9.2 перечислены интерфейсы и их операции. Первые две операции в списке являются общими для любого другого устройства. Третья, **mmap**, это то, что делает интерфейс кадрового буфера уникальным. Сейчас немного отойдём от основой темы и обсудим возможности системного вызова **mmap ()**.

Таблица 9.2 Интерфейс кадрового буфера

Интерфейс	*Операция*
Обычный ввод/вывод	open, read, write с /dev/fb
ioctl	Команды для настройки видеорежима, запроса информации о чипсете, и так далее
mmap	Отображение области памяти видеобуфера в память программ

Мощь mmap

Драйверы являются частью ядра и, следовательно, работают в памяти ядра, в то время как приложения находятся на стороне пользователя и работают в пользовательской памяти. Единственным интерфейсом, доступным для взаимодействия между драйверами и приложениями являются файловые операции (**fops**), такие как **open**, **read**, **write** и **ioctl**. Рассмотрим простую операцию записи. Вызов **write** происходит от пользовательского процесса, с данными, размещёнными в пользовательском буфере (выделенном в памяти пользовательского пространства), и переданными драйверу. Драйвер выделяет буфер в пространстве ядра и копирует пользовательский буфер в буфер ядра с помощью функции ядра **copy_from_user** и выполняет с буфером необходимые действия. В случае драйверов с кадровым буфером необходимо скопировать/DMA его для вывода в память кадрового буфера. Если приложение вынуждено писать с определённым смещением, становится необходимо делать вызов **seek()**, а за ним **write()** Рисунок 9.7 подробно показывает разные этапы в ходе операции записи.

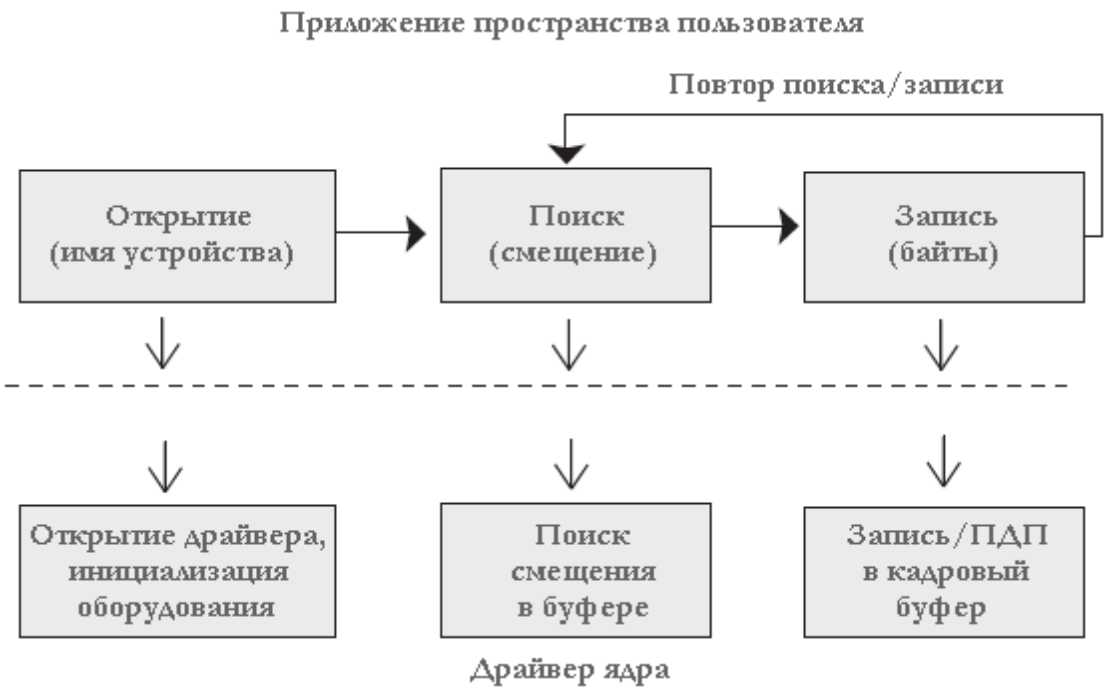

Рисунок 9.7 Поиск/запись с повтором.

Теперь рассмотрим графическое приложение. Оно должно писать данные по всей области экрана. Возможно, придётся обновить какой-то один прямоугольник или даже весь экран, или иногда просто помигать курсором. Каждое выполнение **seek()**, а затем **write()** стоит дорого и отнимает много времени. Для использования в таких приложениях интерфейс **fops** предлагает API **mmap()**. Если драйвер в своей структуре **fops** реализует **mmap()**, пользовательское приложение может напрямую получить в пользовательском пространстве отображённый на память эквивалент аппаратного адреса кадрового буфера. Для драйверов класса кадрового буфера реализация **mmap()** является обязательной (* Обратите внимание, что это не требуется в операционных системах без MMU, таких как VxWorks или uClinux, потому что всё адресное пространство памяти - это единая область. В модели flataddresing любой процесс может обратиться к любой области памяти, независимо ядро это, или пользователя.). Рисунок 9.8 показывает различные шаги при использовании **mmap**.

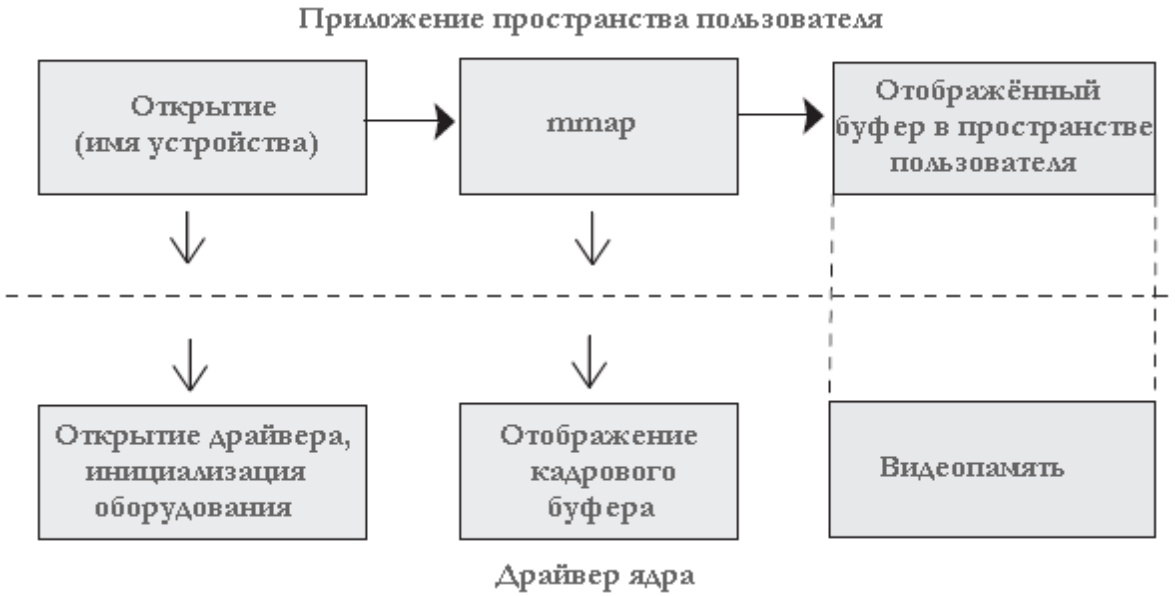

Рисунок 9.8 Запись через mmap.

Все приложения, работающие с кадровым буфером, просто вызывают **open()** для **/dev/fb**, делая необходимой **ioctl()**, чтобы установить разрешение экрана, размерность пикселя, частоту обновления и так далее, и, наконец, вызывают **mmap()**. Реализация **mmap** в драйвере просто отображает полный буфер кадра видеооборудования. В результате приложение получает указатель на память кадрового буфера. Любые изменения, сделанные в этой области памяти, прямо отражаются на экране.

Чтобы понять, чем же интерфейс кадрового буфера лучше, мы настроим драйвер кадрового буфера на рабочем столе Linux и напишем простое приложение, использующее кадровый буфер.

Установка драйвера кадрового буфера

Ядро Linux должно быть загружено с опцией кадрового буфера, чтобы использовать кадровый буфер консоли и проинициализировать драйвер кадрового буфера. Передайте при запуске ядра параметр командной строки **vga=0x761** (* Для достижения того же результата можно отредактировать конфигурационные файлы загрузчика /etc/lilo.conf или grub.conf). Число представляет собой режим разрешения экрана карты; для получения полной информации о том, что означает это число, обратитесь к **Documentation/fb.txt** в каталоге исходных текстов ядра Linux. Если ядро грузится с активированным режимом кадрового буфера, вы сразу же заметите при загрузке в верхней левой части экрана известный образ пингвина. Кроме того, распечатки ядра будут отображаться шрифтами с высоким разрешением. Для дальнейшего подтверждения просто сделайте для любого файла **cat**, направив вывод в **/dev/fb0**. Вы увидите на экране мусор. Если это не сработало, вам, возможно, придётся скомпилировать поддержку кадрового буфера в ядре, а затем повторить попытку. Для получения дополнительной помощи по настройке устройства с кадровым буфером смотрите раздел **frame buffer HOWTO**, доступный на http://www.tldp.org/.

Теперь мы обсудим структуры данных и команды **ioctl**, которые обеспечивают интерфейс кадрового буфера для пространства пользователя. Обычные графические

устройства имеют поддержку нескольких разрешений экрана и режимов. Например, одно устройство может иметь следующие настраиваемые режимы.

- ***Режим 1***: 640 × 480, цвет 24 бита, RGB 888
- ***Режим 2***: 320 × 480, цвет 16 бит, RGB 565
- ***Режим 3***: 640 × 480, монохромный, индексный

Драйвер устройства сообщает об установленных параметрах в определённом режиме с помощью структуры **fb_fix_screeninfo**. Иными словами, структура **fb_fix_screeninfo** определяет зафиксированные или неизменные свойства видеокарты, когда началась работа в том или ином разрешении экрана/режиме.

```
struct fb_fix_screeninfo {
    char id[16];              /*Идентификационная строка, например "ATI Radeon
360"*/
    unsigned long smem_start; /*Начало памяти кадрового буфера*/
    __u32 smem_len;           /*Размер памяти кадрового буфера*/
    __u32 type;               /*один из многих FB_TYPE_XXX*/
    __u32 visual;             /*один из FB_VISUAL_XXX*/
    …
    …
    __u32 line_length;        /*длина строки в байтах*/
    …
    …
};
```

smem_start - это физический начальный адрес памяти кадрового буфера размером **smem_len**. Поля **type** и **visual** указывают формат пикселя и режим цвета. Для чтения структуры **fb_fix_screeninfo** используется **ioctl FBIOGET_FSCREENINFO**.

```
fb_fix_screeninfo fix;
ioctl(FrameBufferFD, FBIOGET_FSCREENINFO, &fix)
```

Структура **fb_var_screeninfo** содержит изменяемые параметры графического режима. Можно использовать эту структуру и установить необходимый графический режим. Важными членами структуры являются:

```
struct fb_var_screeninfo {
    __u32 xres;         /*видимое разрешение по X, Y*/
    __u32 yres;
    __u32 xres_virtual; /*виртуальное разрешение*/
    __u32 yres_virtual;
    __u32 xoffset;      /*смещение от виртуального к видимому разрешению*/
    __u32 yoffset;
    __u32 bits_per_pixel; /*Число битов в пикселе*/
    __u32 grayscale;    /*!= 0 Уровни серого вместо цветов*/
    …
    …
    struct fb_bitfield red;    /*битовое поле в памяти к/б в режиме true
color*/
    struct fb_bitfield green;
    struct fb_bitfield blue;
```

```
    struct fb_bitfield transp; /*уровень прозрачности*/
    __u32 nonstd;              /*!= 0 Нестандартный формат пикселя*/
    …
    …
};
```

Вышеупомянутая структура **fb_bitfield** детализирует размер и смещение битов для каждого цвета пикселя.

```
struct fb_bitfield {
    __u32 offset;    /*начало битового поля*/
    __u32 length;    /*размер битового поля*/
    __u32 msb_right; /*!= 0 : Старший бит справа*/
};
```

Напомним, что разные цветовые режимы, которые были обсуждены, это RGB565, RGB888, и так далее, и элементы **fb_bitfield** представляют собой то же самое. Например, один пиксель RGB 565 имеет размер 2 байта и имеет формат, показанный на Рисунке 9.9.

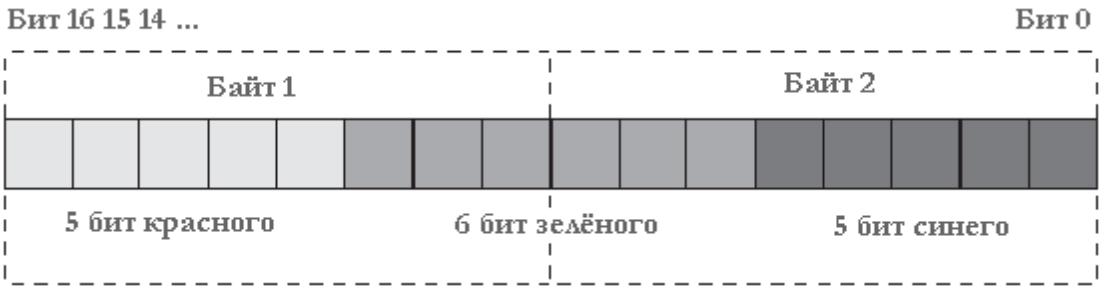

Рисунок 9.9 Формат пикселя RGB565.

- **red.length = 5**, **red.offset = 16** → 32 оттенка чистого красного
- **green.length = 6** и **green.offset = 11** → 64 оттенка чистого зелёного
- **blue.length = 5** и **blue.offset = 5** → 32 оттенка чистого синего
- размер пикселя 16 бит или 2 байта → 65536 всего оттенков.

Поля **xres** и **yres** представляют собой разрешение видимого экрана по X и Y. Поля **xres_virtual** и **yres_virtual** указывают виртуальное разрешение видеокарты, которые могут быть больше, чем видимые координаты. Например, рассмотрим случай, когда видимое разрешение по Y равно 480, а виртуальное разрешение по Y равно 960. Поскольку видны только 480 линий, необходимо указать, какие 480 линий из 960 будут видимы на экране. Для этих целей используются поля **xoffset** и **yoffset**. В нашем примере, если **yoffset** равно 0, то первые 480 строк будут видны или, если **yoffset** равно 480, видны последние 480 линий. Изменяя только значение смещения программисты могут осуществлять двойную буферизацию и переключение страниц в их графических приложениях (* Двойная буферизация и переключение страниц - это техники графики, в которых изображение отрисовывается в закадровый (невидимый) буфер. Отрисовка изображения на экране выполняется простым переключением указателя).

ioctl с **FBIOGET_VSCREENINFO** возвращает структуру **fb_var_screenifno**, а **ioctl** с **FBIOSET_VSCREENINFO** передаёт настроенную структуру **fb_var_screeninfo**.

```
fb_var_screeninfo var;
/* Читаем изменяемую информацию */
ioctl(FrameBufferFD, FBIOGET_VCREENINFO, &var);

/* Устанавливаем разрешение экрана 1024x768 */
var.xres = 1024; var.yres = 768;
ioctl(FrameBufferFD, FBIOSET_VCREENINFO, &var);
```

Следующей важной структурой в программировании кадрового буфера является структура **fb_cmap**.

```
struct fb_cmap {
    __u32 start;   /* Первая запись */
    __u32 len;     /* Число записей */
    __u16 *red;    /* Значения цветов */
    __u16 *green;
    __u16 *blue;
    __u16 *transp; /* прозрачность, может быть NULL */
};
```

Каждое поле цвета **red**, **green** и **blue** является массивом из значений цвета длиной **len**. Таким образом, эта структура представляет собой таблицу карты цветов или **CLUT**, где значение цвета для любого индекса **n** получается поиском в таблице по **red[n]**, **green[n]**, **blue[n]**, где **start <= n < len**. Поле **transp** используется для указания уровня прозрачности, если это необходимо и не является обязательным (* Поле прозрачности также называют альфа-каналом. Мы добавляем к списку известных режимов новый режим, 32-х битный режим RGBA8888, где A означает альфа-канал (8 бит)).

ioctl FBIOGETCMAP используется для чтения из существующей таблицы карты цветов, а **ioctl FBIOPUTCMAP** программирует/загружает новую таблицу карты цветов/CLUT. (* Загрузка новой CLUT или таблицы цветов упоминается как программирование палитры.)

```
/* Читаем текущую таблицу цветов */
fb_cmap cmap;
/* Инициализируем структура данных cmap */
allocate_cmap(&cmap, 256);
ioctl(FrameBufferFD, FBIOGETCMAP, &cmap);

/* Изменяем записи cmap и загружаем 8-ми битовую индексную таблицу цветов */
#define RGB(r, g, b) ((r<<red_offset)|
                      (g << green_offset)|
                      (b << blue_offset))

/* Устанавливаем смещение для режима RGB332 */
red_offset = 5; green_offset = 2; blue_offset = 0;
for(r=0;r<3;r++) {
    for(g=0;j<3;g++) {
        for(b=0;b<2;b++) {
            q=RGB(r, g, b);
            cmap.red[q]=r;
            cmap.green[q]=g;
            cmap.blue[q]=b;
        }
```

```
    }
}

/* Наконец, загружаем нашу CLUT */
ioctl(FrameBufferFD, FBIOPUTCMAP, &cmap);
```

Теперь мы готовы написать нашу программу ***Hello world*** для работы с кадровым буфером. Распечатка 9.1 304 ставит в центре экрана один белый пиксель.

Первый шаг достаточно очевиден: открыть устройство с кадровым буфером **/dev/fb0** с **O_RDWR** (чтение/запись). Вторым шагом является чтение структур с информацией об устройстве. Устройство с кадровым буфером имеет две важные структуры: постоянную и переменную. Постоянная информация предназначена только для чтения: читается с помощью **ioctl FBIOGET_FSCREENINFO**. Структура **fb_fix_screeninfo** содержит идентификационную информацию об устройстве с кадровым буфером, информацию о пиксельном формате, поддерживаемом этим устройством и адреса для отображения памяти кадрового буфера. Переменная информация экрана читается с использованием **ioctl FBIOGET_VSCREENINFO** и записывается с помощью **ioctl FBIOSET_VSCREENINFO**. Структура **fb_var_screeninfo** описывает геометрию и временные параметры текущего видеорежима. Следующий шаг - отобразить аппаратный буфер в пространство процесса нашего приложения используя системный вызов **mmap()**.

Теперь мы готовы установить наш пиксель. Позаботимся о различных битовых полях и схемах упаковки пикселей. Число бит и смещения для отдельных цветовых каналов предоставляет структура **fb_var_screeninfo**. Наконец, используя свойство линейного кадрового буфера (пиксель (x, y) = (line_width * y) + x), мы ставим единственный пиксель. Так много, чтобы установить всего 1 пиксель!

Распечатка 9.1 Пример работы с кадровым буфером

Распечатка 9.1

```
/* File: fbs.c */

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>
#include <asm/page.h>
#include <sys/mman.h>
#include <sys/ioctl.h>
#include <asm/page.h>
#include <linux/fb.h>

/* имя устройства, подобное /dev/fb */
char *fbname;
/* дескриптор устройства с к/б */
int FrameBufferFD;
/* неизменяемая информация об экране */
struct fb_fix_screeninfo fix_info;
```

```
/* изменяемая информация экрана */
struct fb_var_screeninfo var_info;
/* указатель на память кадрового буфера */
void *framebuffer;
/* функция для отрисовки пикселя в позиции (x,y) */
void draw_pixel(int x,int y, u_int32_t pixel);

int main(int argc, char *argv[])
{
    int size;
    u_int8_t red, green, blue;
    int x, y;
    u_int32_t pixel;

    fbname = "/dev/fb0";

    /* Открываем устройство с кадровым буфером в режиме чтения и записи */
    FrameBufferFD = open(fbname, O_RDWR);
    if (FrameBufferFD < 0) {
        printf("Unable to open %s.\n", fbname);
        return 1;
    }

    /* Выполняем Ioctl. Запрашиваем неизменяемую информацию об экране. */
    if (ioctl(FrameBufferFD, FBIOGET_FSCREENINFO, &fix_info) < 0) {
        printf("get fixed screen info failed: %s\n",
                strerror(errno));
        close(FrameBufferFD);
        return 1;
    }

    /* Выполняем Ioctl. Получаем изменяемую информацию экрана. */
    if (ioctl(FrameBufferFD, FBIOGET_VSCREENINFO, &var_info) < 0) {
        printf("Unable to retrieve variable screen info: %s\n",
                strerror(errno));
        close(FrameBufferFD);
        return 1;
    }

    /* Печатаем доступную в настоящее время некоторую информацию об экране */
    printf("Screen resolution: (%dx%d)\n",
            var_info.xres,var_info.yres);
    printf("Line width in bytes %d\n", fix_info.line_length);
    printf("bits per pixel : %d\n", var_info.bits_per_pixel);
    printf("Red: length %d bits, offset %d\n",
            var_info.red.length,var_info.red.offset);
    printf("Green: length %d bits, offset %d\n",
            var_info.red.length, var_info.green.offset);
    printf("Blue: length %d bits, offset %d\n",
            var_info.red.length,var_info.blue.offset);

    /* Рассчитываем размер для mmap */
    size=fix_info.line_length * var_info.yres;
```

```
    /* Теперь для кадрового буфера выполняем mmap. */
    framebuffer = mmap(NULL, size, PROT_READ | PROT_WRITE,
                       MAP_SHARED, FrameBufferFD,0);
    if (framebuffer == NULL) {
        printf("mmap failed:\n");
        close(FrameBufferFD);
        return 1;
    }

    printf("framebuffer mmap address=%p\n", framebuffer);
    printf("framebuffer size=%d bytes\n", size);

    /* Программа будет работать только в TRUECOLOR */
    if (fix_info.visual == FB_VISUAL_TRUECOLOR) {
        /* Белый пиксель ? максимальные значения красного, зелёного и синего
*/
        /* Максимальное 8-ми битовое значение = 0xFF */
        red = 0xFF;
        green = 0xFF;
        blue = 0xFF;

        /*
         * Теперь пакуем пиксель на основе смещения битов rgb.
         * Вычисляем значение каждого цвета на основе числа битов
         * и сдвигаем его в пикселе на соответствующее значение.
         *
         * Например: При работе с RGB565, формула будет
         * выглядеть так:
         * Красный len=5, off=11 : Зелёный len=6, off=6 : Синий len=5, off=0
         * pixel_value = ((0xFF >> (8 - 5) << 11)|
         * ((0xFF >> (8 - 6) << 6) |
         * ((0xFF >> (8 - 5) << 0) = 0xFFFF // Белый
         */
        pixel = ((red >> (8-var_info.red.length)) <<
                var_info.red.offset) |
                ((green >> (8-var_info.green.length)) <<
                var_info.green.offset) |
                ((blue >>(8-var_info.blue.length)) <<
                var_info.blue.offset);
    }else {
        printf("Unsupported Mode.\n");
        return 1;
    }

    /* Вычисляем центр экрана */
    x = var_info.xres / 2 + var_info.xoffset;
    y = var_info.yres / 2 + var_info.yoffset;

    /* Рисуем пиксель по координатам x,y */
    draw_pixel(x,y, pixel);

    /* Освобождаем mmap. */
    munmap(framebuffer,0);
    close(FrameBufferFD);
```

```
    return 0;
}

void draw_pixel(int x, int y, u_int32_t pixel)
{
    /*
     * Базируясь на числе бит на пиксель, присваиваем значение pixel_value
     * адресу, на который указывает framebuffer. Вспомните метод матричной
     * индексации, описанный для линейного кадрового буфера.
     * pixel(x,y)=(line_width * y) + x.
     */
    switch (var_info.bits_per_pixel) {
    case 8:
        *((u_int8_t *)framebuffer + fix_info.line_length * y +x) =
            (u_int8_t)pixel;
        break;

    case 16:
        *((u_int16_t *)framebuffer + fix_info.line_length / 2 * y + x) =
            (u_int16_t)pixel;
        break;

    case 32:
        *((u_int32_t *)framebuffer + fix_info.line_length / 4 * y + x) =
            (u_int32_t)pixel;
        break;

    default:
        printf("Unknown depth.\n");
    }
}
```

9.5.2 Внутреннее строение кадрового буфера

Ядро обеспечивает основу драйвера кадрового буфера (реализованную в **drivers/video/fbmem.c** и **drivers/video/fbgen.c**). Эта основа обеспечивает лёгкую интеграцию в ядро настоящего аппаратного драйвера с кадровым буфером. Все зависимые от оборудования драйверы с кадровым буфером регистрируют этот интерфейс в ядре. Основа обеспечивает API и определяет структуры данных для подключения аппаратно-зависимого кода. Скелет любого драйвера, который использует эту основу, выглядит следующим образом.

- Заполнение структуры операций драйвера, **struct fb_ops**.
- Заполнение фиксированной информации кадрового буфера, **struct fb_fix_screeninfo**.
- Заполнение структуры информации драйвера, **struct fb_info**.
- Инициализация аппаратных регистров и области видеопамяти.
- Выделение памяти и инициализация карты цветов, **struct fb_cmap**, если необходимо. (* Не каждое оборудование обеспечивает хранение таблиц карт цветов.)
- Регистрация структуры **fb_info** с использованием драйверного ядра с помощью **register_framebuffer**.

Мы уже обсуждали структуры **fb_fix_screeninfo**, **fb_var_screeninfo** и **fb_cmap**. Теперь рассмотрим две другие структуры драйвера, **fb_ops** и **fb_info**. Важными полями структуры **fb_ops** являются указатели на функции таких операций, как **open**, **close**, **read**, **write** и

ioctl. Большинство из них имеют универсальную обработку в ядре. Так что если нет необходимости сделать нечто особенное для вашего оборудования, нет необходимости определять большинство из этих полей. Например, отключение экрана, **fb_blank**, и установка регистра цвета, **fb_setcolreg**, - это аппаратно-зависимые процедуры. Они должны быть определены, если ваше оборудование поддерживает их и соответственно обрабатывает. Описания различных полей структуры **struct fb_info** можно найти в **incude/linux/fb.h**.

Структура **fb_info** является наиболее важной структурой, так как это единая точка объединения для всех других структур данных. Драйвер регистрируется в ядре с указателем на зависимую от драйвера структуру **fb_info**. Важными полями в этой структуры являются:

```
struct fb_info {
    …
    …
    struct fb_var_screeninfo var; /*Текущая переменная информация экрана*/
    struct fb_fix_screeninfo fix; /*Постоянная информация экрана*/
    …
    …
    struct fb_cmap cmap;  /*Текущая таблица Цветов*/
    struct fb_ops *fbops; /*Указатель на структуру fb_ops*/
    char *screen_base;    /*Базовый адрес видеопамяти (виртуальный)*/
    …
    …
};
```

Поле **screen_base** - это базовый адрес видеопамяти, указатель на фактический кадровый буфер в аппаратной памяти. Но следует отметить, что аппаратный адрес должен быть переотображён из пространства ввода-вывода, прежде чем предоставлять этот адрес ядру. После того, как структуры данных подготовлены, драйвер должен зарегистрироваться в ядре, вызвав **register_framebuffer**.

```
int register_framebuffer(struct fb_info *fb_info);
```

В итоге, драйверу необходимо заполнить

- **fb_info.fix**: Постоянная информация об области экрана и типе.
- **fb_info.var**: Переменная информация о разрешении экрана и глубине пикселя для текущего режима.
- **fb_info.fb_ops**: Указатели на функции для операций кадрового буфера; используется только когда требуется зависимая от оборудования обработка.
- **fb_info.screen_base**: Базовый (виртуальный) адрес видеопамяти, передаваемый пользовательским приложениям через **mmap**.
- **fb_info.fb_cmap**: Настройка записей карты цветов, если это необходимо.
- **fb_ops.fb_blank**, **fb_ops.fb_setcolreg**: Настройка аппаратно-зависимых записей **fb_ops**, если это необходимо.
- И, наконец, вызов **register_framebuffer(&fb_info)**.

Идею написания простого драйвера кадрового буфера можно увидеть в **drivers/video/vfb.c**, примере виртуального кадрового буфера в ядре. Чтобы получить представление о

деталях написания кода драйвера, можно также посмотреть на исходные коды других драйверов. Сейчас мы обсуждаем пример драйвера с кадровым буфером для Linux 2.6. (* В драйверах кадрового буфера для ядер версий 2.4 и 2.6 есть небольшое отличие. Структура info кадрового буфера 2.4 хранит прямые указатели на данные консольного драйвера; 2.6 удалило эту зависимость, полностью отделив консоль от графического интерфейса.) Таблицы 9.3 и 9.4 перечисляют детали спецификаций и данных нашего гипотетического графического оборудования: устройства Простого Кадрового Буфера (Simple Frame Buffer, SFB).

Давайте сначала для этого драйвера заполним зависимые от оборудования макросы. Остальная часть кода носит общий характер и не зависит от оборудования. Вся подробная информация, связанная с аппаратным обеспечением, содержится в Распечатке 9.2 310. Распечатка 9.3 312 представляет собой простой скелет драйвера с кадровым буфером, который будет работать с любым оборудованием, при условии обновления соответствующим образом Распечатки 9.2 310.

Таблица 9.3 Детали оборудования SFB

Параметр	*Значение*
Начальный адрес видеопамяти	0xA00000
Размер видеопамяти	0x12C000
Конечный адрес видеопамяти	0xB2C000
Максимальное разрешение по X	640
Максимальное разрешение по Y	480
Минимальное разрешение по X	320
Минимальное разрешение по Y	240
Форматы цветов	32 бита, true color, RGBX888, Старшие 8 бит не используются 16 бит, high color, RGB565 8 бит, индексный цвет, требуется программирование палитры
Наличие палитры	Да, 256 аппаратных индексных регистров цвета
Первый регистр палитры	0xB2C100
Регистр режима	0xB2C004
Регистр разрешения	0xB2C008. Старшие 2 байта это разрешение по Y, а младшие 2 байта - это разрешение по X

Таблица 9.4 Регистр режима

Значение в регистре	*Режим карты*
0x100	RGB X888
0x010	RGB 565

Значение в регистре	*Режим карты*
0x001	8 бит, индексный режим

Распечатка 9.2 Зависимые от оборудования определения драйвера кадрового буфера

Распечатка 9.2

```
/* sfb.h */

#define SFB_VIDEOMEMSTART 0xA00000
#define SFB_VIDEOMEMSIZE 0x12C000
#define SFB_MAX_X 640
#define SFB_MAX_Y 480
#define SFB_MIN_X 320
#define SFB_MIN_Y 240

/* Наше оборудование не поддерживает прозрачность */
#define TRANSP_OFFSET 0
#define TRANSP_LENGTH 0

/*
 * Оборудование имеет от 0 до 255 (256) программируемых регистров
 * цветовой палитры
 */
#define SFB_MAX_PALETTE_REG256
#define SFB_PALETTE_START 0xB2C100

/* Регистр режимов и режимы */
#define SFB_MODE_REG 0xB2C004
#define SFB_8BPP 0x1
#define SFB_16BPP 0x10
#define SFB_32BPP 0x100

/* Регистр разрешения экрана */
#define SFB_RESOLUTION_REG 0xB2C008

/*
 * Другой режим bits_per_pixel (8/16/24/32) является аппаратно
 * зависимым. Таким образом, этот режим должен быть обработан соответствующим
образом.
 * Оборудование SFB поддерживает только режимы 8, 16 и 32 бита, Проверьте
 * соответствие режимов и внесите соответствующие коррективы
 */

static inline int sfb_check_bpp(struct fb_var_screeninfo *var)
{
    if (var->bits_per_pixel <= 8)
        var->bits_per_pixel = 8;
    else if (var->bits_per_pixel <= 16)
        var->bits_per_pixel = 16;
    else if (var->bits_per_pixel <= 32)
```

```
        var->bits_per_pixel = 32;
    else
        return -EINVAL;
    return 0;
}

static inline void sfb_fixup_var_modes(struct fb_var_screeninfo *var)
{
    switch (var->bits_per_pixel) {
    case 8:
        var->red.offset = 0;
        var->red.length = 3;
        var->green.offset = 3;
        var->green.length = 3;
        var->blue.offset = 6;
        var->blue.length = 2;
        var->transp.offset = TRANSP_OFFSET;
        var->transp.length = TRANSP_LENGTH;
        break;

    case 16: /*RGB565*/
        var->red.offset = 0;
        var->red.length = 5;
        var->green.offset = 5;
        var->green.length = 6;
        var->blue.offset = 11;
        var->blue.length = 5;
        var->transp.offset = TRANSP_OFFSET;
        var->transp.length = TRANSP_LENGTH;
        break;

    case 24:
    case 32: /* RGBX 888 */
        var->red.offset = 0;
        var->red.length = 8;
        var->green.offset = 8;
        var->green.length = 8;
        var->blue.offset = 16;
        var->blue.length = 8;
        var->transp.offset = TRANSP_OFFSET;
        var->transp.length = TRANSP_LENGTH;
        break;
    }
    var->red.msb_right = 0;
    var->green.msb_right = 0;
    var->blue.msb_right = 0;
    var->transp.msb_right = 0;
}

/* Программируем оборудование, базируясь на настройках пользователя */
static inline sfb_program_hardware(struct fb_info *info)
{
    *((unsigned int*)(SFB_RESOLUTION_REG)) =
        ((info->var.yres_virtual & 0xFFFF) << 0xFFFF) |
```

```
        (info->var.xres_virtual & 0xFFFF)

    switch(info->var.bits_per_pixel) {
    case 8:
        *((unsigned int*)(SFB_MODE_REG)) = SFB_8BPP;
        break;

    case 16:
        *((unsigned int*)(SFB_MODE_REG)) = SFB_16BPP;
        break;

    case 32:
        *((unsigned int*)(SFB_MODE_REG)) = SFB_32BPP;
        break;
    }
}
```

Распечатка 9.3 Обычный драйвер кадрового буфера

Распечатка 9.3

```
/* sfb.c */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/mm.h>
#include <linux/tty.h>
#include <linux/slab.h>
#include <linux/vmalloc.h>
#include <linux/delay.h>
#include <linux/interrupt.h>
#include <asm/uaccess.h>
#include <linux/fb.h>
#include <linux/init.h>

static char *videomemory = VIDEOMEMSTART;
static u_long videomemorysize = VIDEOMEMSIZE;
static struct fb_info fb_info;

/* Устанавливаем начальную информацию экрана */
static struct fb_var_screeninfo sfb_default __initdata = {
    .xres           = SFB_MAX_X,
    .yres           = SFB_MAX_Y,
    .xres_virtual   = SFB_MAX_X,
    .yres_virtual   = SFB_MAX_Y,
    .bits_per_pixel = 8,
    .red            = { 0, 8, 0 },
    .green          = { 0, 8, 0 },
    .blue           = { 0, 8, 0 },
    .activate       = FB_ACTIVATE_TEST,
    .height         = -1,
```

```
    .width          = -1,
    .left_margin    = 0,
    .right_margin   = 0,
    .upper_margin   = 0,
    .lower_margin   = 0,
    .vmode          = FB_VMODE_NONINTERLACED,
};
static struct fb_fix_screeninfo sfb_fix __initdata = {
    .id        = "SimpleFB",
    .type      = FB_TYPE_PACKED_PIXELS,
    .visual    = FB_VISUAL_PSEUDOCOLOR,
    .xpanstep  = 1,
    .ypanstep  = 1,
    .ywrapstep = 1,
    .accel     = FB_ACCEL_NONE,
};

/* Прототипы функций */
int sfb_init(void);
int sfb_setup(char *);
static int sfb_check_var(struct fb_var_screeninfo *var,
                         struct fb_info *info);
static int sfb_set_par(struct fb_info *info);
static int sfb_setcolreg(u_int regno, u_int red, u_int green,
                         u_int blue, u_int transp,
                         struct fb_info *info);
static int sfb_pan_display(struct fb_var_screeninfo *var,
                           struct fb_info *info);
static int sfb_mmap(struct fb_info *info, struct file *file,
                    struct vm_area_struct *vma);

/*
 * Определяем структуру fb_ops, которая регистрируется в ядре.
 * Замечание: процедуры cfb_xxx - это универсальные процедуры,
 * реализованные в ядре. Можно их переопределить в случае, если
 * ваше оборудование предоставляет функции ускорения графики
 */
static struct fb_ops sfb_ops = {
    .fb_check_var = sfb_check_var,
    .fb_set_par   = sfb_set_par,
    .fb_setcolreg = sfb_setcolreg,
    .fb_fillrect  = cfb_fillrect,
    .fb_copyarea  = cfb_copyarea,
    .fb_imageblit = cfb_imageblit,
};

/*
 * Вспомните обсуждение о line_length в разделе о графическом
 * оборудовании. Длина строки выражается в байтах, обозначая
 * число байтов в каждой строке.
 */
static u_long get_line_length(int xres_virtual, int bpp)
{
    u_long line_length;
```

```
    line_length = xres_virtual * bpp;
    line_length = (line_length + 31) & ~31;
    line_length >>= 3;
    return (line_length);
}

/*
 * xxxfb_check_var, не пишет что-то в оборудование, а только
 * проверяет пригодность, базируясь на данных оборудования
 */
static int sfb_check_var(struct fb_var_screeninfo *var,
                         struct fb_info *info)
{
    u_long line_length;
    /* Проверяем достоверность разрешения экрана */
    if (!var->xres)
        var->xres = SFB_MIN_XRES;
    if (!var->yres)
        var->yres = SFB_MIN_YRES;
    if (var->xres > var->xres_virtual)
        var->xres_virtual = var->xres;
    if (var->yres > var->yres_virtual)
        var->yres_virtual = var->yres;
    if (sfb_check_bpp(var)) return -EINVAL;
    if (var->xres_virtual < var->xoffset + var->xres)
        var->xres_virtual = var->xoffset + var->xres;
    if (var->yres_virtual < var->yoffset + var->yres)
        var->yres_virtual = var->yoffset + var->yres;

    /*
     * Убедимся, что карта имеет достаточно видеопамяти в этом режиме
     * Вызываем формулу
     * Память к/б = Ширина Дисплея * Высота Дисплея * Байтов На Пиксель
     */
    line_length = get_line_length(var->xres_virtual,
                            var->bits_per_pixel);
    if (line_length * var->yres_virtual > videomemorysize)
        return -ENOMEM;
    sfb_fixup_var_modes(var);
    return 0;
}

/*
 * Эта процедура реально устанавливает видеорежим. Все проверки уже
 * были сделаны
 */
static int sfb_set_par(struct fb_info *info)
{
    sfb_program_hardware(info);
    info->fix.line_length = get_line_length(
                    info->var.xres_virtual,
                    info->var.bits_per_pixel);
    return 0;
}
```

```
/*
 * Устанавливаем один регистр цвета. Полученные значения уже
 * подогнаны к возможностям оборудования (соответствуют записям
 * в структуре var). Возвращаем != 0 для неверного regno.
 */
static int sfb_setcolreg(u_int regno, u_int red, u_int green,
                         u_int blue, u_int transp,
                         struct fb_info *info)
{
unsigned int *palette = SFB_PALETTE_START;

    if (regno >= SFB_MAX_PALLETE_REG) //номера аппаратных регистров
        return 1;
    unsigned int v = (red << info->var.red.offset) |
                     (green << info->var.green.offset) |
                     (blue << info->var.blue.offset) |
                     (transp << info->var.transp.offset);

    /* Программируем оборудование */
    *(palette+regno) = v;
    return 0;
}

/* Точка входа в драйвер */
int __init sfb_init(void)
{
    /* Заполняем структуру fb_info */
    fb_info.screen_base = io_remap(videomemory, vidememory);
    fb_info.fbops = &sfb_ops;
    fb_info.var = sfb_default;
    fb_info.fix = sfb_fix;
    fb_info.flags = FBINFO_FLAG_DEFAULT;

    fb_alloc_cmap(&fb_info.cmap, 256, 0);

    /* Регистрируем драйвер */
    if (register_framebuffer(&fb_info) < 0) {
        return -EINVAL;
    }

    printk(KERN_INFO "fb%d: Sample frame buffer device
                     initialized \n", fb_info.node);

    return 0;
}

static void __exit sfb_cleanup(void)
{
    unregister_framebuffer(&fb_info);
}

module_init(sfb_init);
module_exit(sfb_cleanup);
```

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

9.6 Оконные среды, инструментарии и приложения

Приложения, написанные для работы непосредственно через интерфейс кадрового буфера, действительно существуют, но только простые. Так как GUI работает с большим набором форм и элементов управления, существует необходимость в абстракции. Библиотеки/уровни API, которые делают программирование GUI простым и удобным, уже в течение многих лет существуют на настольных платформах. Эти библиотеки покрывают интерфейс драйвера более простыми API, которые имеют смысл для программиста графического приложения. Эти библиотеки важны во всех оконных средах. Универсальная оконная среда состоит из:

- Интерфейсный уровень для низкоуровневых драйверов, таких как драйверы экрана и ввода данных
- Графический движок для рисования объектов на экране
- Движок шрифтов, который способен декодировать один или несколько файловых форматов шрифтов и отрисовать их
- API, которые предоставляют доступ к различным функциям, экспортируемым движками графики и шрифтов

Напомним, мы обсуждали X как оконную среду, используемую на настольных компьютерах с Linux. X-lib является уровнем API, предоставляемым оконной средой X. Инструментарии GUI также представляют собой библиотеки, построенные над оконными средами для преодоления некоторых недостатков библиотеки нижнего уровня.

- Библиотеки оконной среды зависят от платформы. Например, код приложения, написанный поверх X-lib, практически невозможно портировать на Windows. Большинство инструментов доступны на нескольких платформах, и, следовательно, делают перенос возможным. Кросс-платформенным инструментарием, доступным на различных платформах, является Qt, такой как Qt/Windows (Windows XP, 2000, NT 4, Me/98/95), Qt/X11 (X windows), Qt/Mac (Mac OS X) и Qt/Embedded (встраиваемый Linux).
- API, экспортируемые библиотеками оконной среды, выполняют простые задачи. Инструментарии реализуют многие компоненты GUI/объекты и обеспечивают для них интерфейсы. Так, например, наборы инструментальных средств обеспечивают интерфейсы для часто используемых диалоговых окон, таких как "Открыть файл", "Распечатать файл", "Выбрать цвет", и так далее.
- Родные виджеты слишком просты и приложения не могут изменить то, как выглядит виджет, когда они используют оконные библиотеки. Инструментарии предоставляют поддержку тем и виджетов, которые часто загружаются с такими функциями, как поддержка 3-х мерного вида, анимация и так далее.
- Наиболее важным из всех являются инструментарии, предоставляющие дизайнерский GUI или инструменты Быстрой Разработки приложений (Rapid Application Development, RAD). Инструменты RAD являются основами GUI, которые предлагают интерфейс укажи-и-нажми для выполнения таких задач, как размещение виджета или определения обратного вызова. Qt предоставляет Qt Designer. Для Gtk используется Glade.

Наборы инструментальных средств не всегда выгодны. Некоторые из них занимают много места в вашем коде в обмен на огромный набор функций, которые они несут с собой. Существует много вариантов, когда речь идет о комбинациях оконные среды/набор инструментария для встраиваемого Linux. Наиболее популярные перечислены в Таблице 9.5.

Одним из главных преимуществ инструментария Linux является то, что он предоставляет среду моделирования на ПК. Приложения могут быть разработаны и прототипизированы на рабочем столе, уменьшая таким образом время разработки и упрощая отладку. В этом разделе мы рассмотрим оконную среду Nano-X.

Таблица 9.5 Популярные оконные среды

Название	*Лицензия*	*Комментарий*
Nano-X www.microwindows.org	GPL/MPL	Оконная среда, предоставляющая интерфейс, похожий на Win32 и X11, нацеленный на встраиваемые системы.
FLNX www.fltk.org	LGPL	Набор инструментария FLTK, портированный поверх Microwindows
MiniGUI www.minigui.com	LGPL	Компактная система графического пользовательского интерфейса для Linux. MiniGUI определяет для приложений некоторые похожие на Win32 интерфейсы; предоставляет небольшую библиотеку, поддерживающую оконную систему
DirectFB www.directfb.org	LGPL	Маленькая библиотека, которая содержит аппаратное ускорение графики, обработку устройства ввода и абстракцию, интегрирующую оконную систему с поддержкой полупрозрачных окон и нескольких слоёв отображения
PicoGUI www.picogui.org	LGPL	Новая архитектура графического пользовательского интерфейса, созданная с учётом особенностей встраиваемых систем; включает низкоуровневую графику и ввод, виджеты, темы, уровни, отрисовку шрифтов, схему прозрачности
Qt/Embedded www.trolltech.com/products/embedded/index.html	QPL GPL	Оконная система для встраиваемых устройств на основе C++, предоставляющая большинство из интерфейсов Qt
GTK+/FB www.gtk.org	LGPL	Вариант популярной оконной среды GTK+ с поддержкой кадрового буфера.

9.6.1 Nano-X

Проект Nano-X - это проект с открытым кодом, доступный в соответствии с лицензиями MPL/GPL. Он предоставляет простой, но мощный встраиваемый графический программный интерфейс. Основные функции Nano-X, которые делают его подходящим в качестве встраиваемой оконной среды, перечислены ниже.

- Разрабатывался с нуля с ориентацией на встраиваемые устройства, с учётом

различных ограничений, таких как память и дисковое пространство. Вся библиотека занимает менее 100 Кб и использует только от 50 до 250 Кб рабочей памяти.
- Архитектура Nano-X позволяет добавлять различные типы устройств отображения, мышей, сенсорных экранов и клавиатур. Это сделано путём предоставления отдельного уровня интерфейса драйвера устройств. Это делает перенос на любую аппаратную платформу простым.
- Nano-X реализует два популярных API: интерфейс на основе Microsoft Windows и X-lib, известный как API Nano-X. Это сокращает время изучения API.
- Уровень драйвера экрана поддерживает все возможные форматы пикселей, такие как 1, 2, 4, 16, 24 и 32 бита на пиксель, и, следовательно, прямо переносим на любое устройство от монохромных ЖК до полноцветных экранных систем.
- Обеспечивает настраиваемую архитектуру и выбор компонентов. Например, можно почти мгновенно добавить или убрать поддержку любого изображения или библиотеки шрифтов.

Архитектура Nano-X

Архитектура Nano-X сильно модулизирована и имеет три основных уровня. К ним относятся:

- Уровень драйвера устройства
- Независимый от оборудования графический движок
- Уровень API (Nano-X и Microwindows)

Архитектуру Nano-X объясняет Рисунок 9.10.

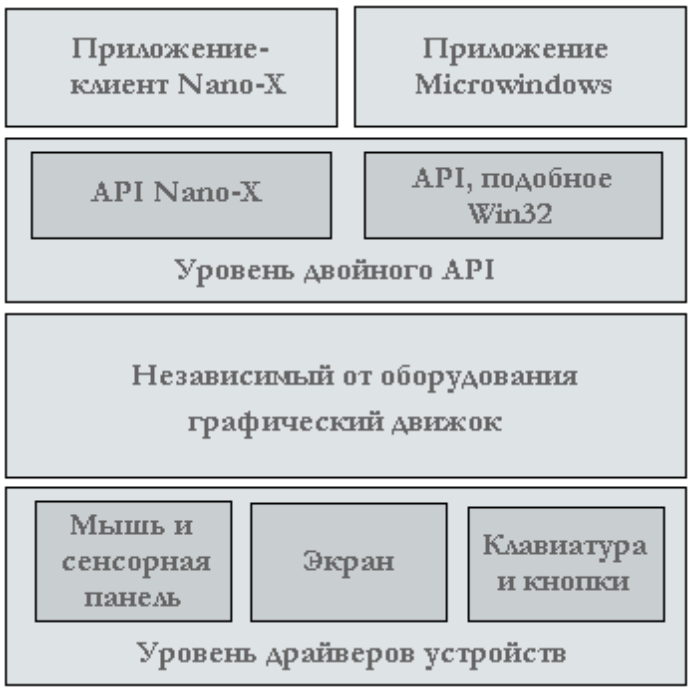

Рисунок 9.10 Архитектура оконной системы Nano-X.

Низкоуровневый драйвер устройства имеет интерфейс для различных устройств, таких как экран поверх кадрового буфера, сенсорная панель, мышь и клавиатура. Этот уровень

позволяет добавление для Nano-X нового устройства, не затрагивая остальные уровни.

Ядром Nano-X является графический движок, который реализует графические процедуры для отрисовки линии, окружности и полигона. Он также поддерживает отрисовку изображений различных форматов, таких как JPEG, BMP и PNG. Этот уровень также включает в себя движок шрифтов, ответственный за отрисовку на экране шрифтов и текста. Движок шрифтов поддерживает шрифты типа true-type и растровые.

Nano-X поддерживает два различных уровня API. Оба уровня API работают поверх ядра графического движка и драйвера устройств. Один набор очень похож на API X Lib, и, следовательно, называется Nano-X. Другой уровень API похож на API Microsoft Win32 и WinCE и, следовательно, называется Microwindows. В связи с такой поддержкой двух API, программистам обоих миров SDK, X и WIN32, предоставляется большое преимущество и им требуется небольшое обучение, или вообще его не требуется. Уровень API Nano-X базируется на X и по сути имеет клиент-серверную модель. ***Nano-X сервер*** является отдельным процессом, в котором работает серверный код. Приложение, ***клиент Nano-X***, работает как отдельный процесс и связан со специальной библиотекой, которая предоставляет связь с сервером на сокетах UNIX/IPC. Существует также возможность связать клиент и сервер вместе в форме одного приложения. Это осуществляется удалением IPC. API Microwindows, с другой стороны, базируется на модели SDK Win32. Microwindows в основном имеет механизм передачи сообщений, очень похожий на своего коллегу в Windows. Большинство оконных объекты и методов имеют те же структуры, как в SDK Windows. Обратитесь к документу об архитектуре, доступном на http://www.microwindows.org/.

Приступаем к работе с Nano-X

1. ***Получите последнюю версию исходного кода***: на время написания, последней версией является Nano-X v0.91, она доступна на ftp://microwindows.censoft.com/pub/.
2. ***Скомпилируйте***: разархивируйте исходники (скажем, в **/usr/local/microwin**). Смените каталог **cd src/Configs/**. Этот каталог содержит заранее созданные файлы конфигураций для различных платформ. Выберите подходящий для вашей платформы и скопируйте его в ***microwin/src/config***. Каждая запись в файле конфигурации имеет хорошее количество комментариев. Это даст вам понимание гибкости конфигурации, которую предоставляет эта миниатюрная оконная среда. Для начала работы мы скомпилируем её для i386. Разрешите опцию X11, установив **X11=Y**. Это гарантирует, что мы сможем запускать Nano-X просто как одно из приложений X11. Движок шрифтов Nano-X поддерживает широкий набор форматов шрифтов, включая true-type (контурные) шрифты и растровые шрифты. Чтобы предоставить поддержку файлов TTF, движок шрифтов использует внешние библиотеки, такие как Freetype-1, Freetype-2, и другие. Выберите систему отрисовки шрифтов, доступную на вашем ПК, наиболее вероятно - Freetype-2. Наконец, компилируйте, используя **make** в каталоге **microwin/src**.
3. ***Запустите демо***: после того, как компиляция закончилась, запустите демонстрационную программу в каталоге **microwin/src**, запустив **./demo2.sh**. Этот скрипт запускает сервер Nano-X и некоторые примеры клиентских программ.

Каталог **src/demos** содержит много примеров программ с разными уровнями сложности. Необходимо просмотреть по крайней мере несколько из них, чтобы познакомиться с программированием Nano-X.

Создание типового приложения Nano-X

Распечатка 9.4 320 - это простое приложение Nano-X, которое просто рисует прямоугольник. Скомпилируйте программу используя команду

```
#gcc simple.c -o simple -I$(MICROWINDIR)/src/include -lnano-x
```

Перед стартом приложения необходимо запустить сервер Nano-X.

```
#nano-X &; simple
```

Любое приложение-клиент Nano-X должно сначала вызвать **GrOpen()**, чтобы соединиться с сервером Nano-X. После того, как подключение успешно установлено, приложение создаёт окно размером 200 на 100 в координате (100 100), используя функцию **GrNewWindow()**. Граница, определённая для окна, имеет ширину в 5 пикселей с красным цветом бордера и белым цветом заливки окна. Затем окно отображается на экране с использованием **GrMapWindow()**. Отображение окна на экран сопровождается наиважнейшим “циклом сообщений” программы. Вызов **GrGetNextEvent()** проверяет, есть ли доступное для чтения сообщение от мыши, клавиатуры или графического интерфейса пользователя. Хотя этот пример приложения не обрабатывает какие-либо события, приложения должны будут соответственно реагировать в ответ на эти различные события. Для получения дополнительной информации о программировании Microwindows обращайтесь к различным справочным материалам на http://www.microwindows.org.

Инструментарий Nano-X

Простые приложения с графическим интерфейсом не требуют никакого инструментария и не должно быть проблем написать их в Nano-X. В этом разделе мы обсуждаем дополнительные инструментарии, доступные поверх Nano-X.

У FLTK (Fast Light Tool Kit, Быстрый Лёгкий Набор Инструментов) есть порт для Nano-X под названием FLNX. FLTK обеспечивает абстракцию C++ поверх API Nano-X. FLTK также предоставляет дизайнер GUI под названием Fluid, который может использоваться для проектирования форм графического интерфейса. Комбинация Nano-X/FLTK рекомендована и проверена; много встраиваемых приложений успешно использовали эту комбинацию. Например, браузер ViewML - реализация HTML браузера с использованием этой самой комбинации.

NXLIB (Nano-X/X-lib Compatibility Library, Библиотека для Совместимости Nano-X/X-lib) позволяет бинарным программам X11 работать с использованием сервера Nano-X без модификаций. Это означает, что многие из полнофункциональных приложений, которые работают на X-сервере, заработали бы на сервере Nano-X с небольшими изменениями или вообще без них. NXLIB не полная замена X-lib, потому что она обеспечивает только подмножество API X-lib. NXLIB поможет уменьшить время портирования огромных приложений, которые были написаны используя другие инструментарии, построенные поверх X-lib. Например, программа, написанная с использованием Gtk-X, будет работать на Nano-X без больших изменений кода.

Распечатка 9.4 Пример приложения Nano-X

Распечатка 9.4

```
/* nano_simple.c */

#define MWINCLUDECOLORS
```

```
#include <stdio.h>
#include "nano-X.h"

int main(int ac,char **av)
{
    GR_WINDOW_ID w;
    GR_EVENT event;

    if (GrOpen() < 0) {
        printf("Can't open graphics\n");
        exit(1);
    }

    /*
     * GrNewWindow(GR_ROOT_WINDOW_ID, X, Y, Width, Height,
     * Border, Windowcolor, Bordercolor);
     */

    w = GrNewWindow(GR_ROOT_WINDOW_ID, 100, 100, 200, 100, 5, WHITE, RED);

    GrMapWindow(w);

    /* Вход в цикл сообщений */
    for (;;) {
        GrGetNextEvent(&event);
    }

    GrClose();
    return 0;
}
```

9.7 Заключение

В этой главе рассматривались различные вопросы о графических системах и их архитектуре и вариантов, имеющихся на системах со встраиваемым Linux. Один вопрос остаётся без ответа: есть ли общие решения, которые могут покрыть весь спектр встраиваемых устройств, требующих графику, то есть от мобильных телефонов до DVD плееров? Кадровый буфер Linux предоставляет решение для всех типов устройств. Если программа достаточно проста, как в случае DVD плеера, программисты могут использовать интерфейс кадрового буфера вместе с крошечным оконным движком, таким как Nano-X, и получить готовую и работающую систему. Решение для мобильного телефона требует много программ, таких как календарь, телефонная книга, камера и способ навигации по удобным меню управления графического интерфейса, которые могут быть реализованы с помощью набора инструментальных средств, таких как Qt/Embedded или FLNX. На момент написания этой статьи смартфоны с использованием Qt уже доступны на рынке.

Глава 10, uClinux

Новая версия Linux, портированная на процессор M68k, была выпущена в январе 1998 года. Основное отличие этого выпуска от стандартного Linux в том, что этот вариант Linux впервые работал на процессоре без MMU. Этот вариант Linux широко известен как uClinux. (* uClinux произносится как "ю-си Линукс" и означает Linux для Микро(µ)Контроллеров.) До этого процессоры без MMU использовались для запуска только коммерческих или заказных ОС реального времени. Возможность работы Linux на таких процессорах даёт большое преимущество при проектировании и разработке. Эта глава пытается ответить на некоторые вопросы, связанные с Linux, работающим на процессорах без MMU.

- Зачем использовать отдельный Linux для систем без MMU?
- Какие инструменты необходимы для сборки и запуска uClinux на встраиваемой системе?
- Какие изменения затронули различные уровни, такие как ядро, пользовательское пространство, файловая система, и так далее?
- Будут ли приложения, скомпилированные для стандартного Linux, работать на uClinux?
- Какие существуют принципы для переноса приложений с Linux на uClinux

10.1 Linux на системах без MMU

Стандартный Linux в основном работает на процессорах общего назначения, которые имеют встроенную аппаратную поддержку для управления памятью в виде MMU (Memory Management Unit, блок управления памятью). MMU, по существу, обеспечивает следующие основные функции:

- Трансляцию адреса с помощью TLB (* TLB означает Translation LookUp Buffer, буфер быстрого преобразования адреса, аппаратную таблицу, используемую для перевода физического адреса в виртуальный и наоборот.)
- Вызов страниц по требованию с использованием ошибок из-за отсутствия страниц в основной памяти
- Защиту адресов с помощью режимов защиты (* Процессоры обеспечивают различные виды защиты, например у процессоров Intel это Real mode, реальный режим, Protected mode, защищённый режим, и так далее. Для обращения к разным диапазонам адресов памяти могут быть запрограммированы разные режимы.)

Linux тесно интегрирован с реализацией виртуальной памяти с использованием MMU и, следовательно, никогда не был предназначен для процессоров без MMU. Поскольку Linux имел такую зависимость от MMU, необходимо было перестроить некоторые части кода управления виртуальной памятью в ядре. Идея запуска Linux на процессорах без MMU и влияние этих изменений на в то время стабильное ядро Linux 2.0 не были ясными для всех. Поэтому для поддержки Linux на процессорах без MMU был начат новый проект, проект uClinux (http://www.uclinux.org).

Проект uClinux предоставлял Linux, который был способен работать на процессорах без MMU. Хотя uClinux был всегда сосредоточен на Linux для систем без MMU, поддержка имеющих MMU процессоров всегда остаётся. uClinux добавляет в ядро поддержку процессоров без MMU, но ничего не удаляет в этом процессе, небольшой, но важный момент, который часто упускается из виду при обсуждении uClinux. Следует отметить, что uClinux способен также работать на системах с MMU, но эта глава посвящена только uClinux,

работающему на системах без MMU.

10.1.1 Отличия Linux и uClinux

На обычной системе Linux пользовательский процесс может быть определён как исполняющаяся программа. Каждая программа представляет собой исполняемый файл, хранящийся в файловой системе; он или автономен (слинкован статически), или динамически слинкован с общими библиотеками. Виртуальная память позволяет выделять для каждого процесса отдельное пространство памяти. Однако на uClinux, поскольку MMU нет, будут иметься сомнения, есть ли в uClinux отдельные программы, или это будет похоже на традиционную ОС с линейной памятью, где ОС и приложения формируют один единственный образ. В этом разделе мы подробно рассмотрим различия между uClinux и Linux.

Обработка адресного пространства и защита памяти

В стандартном Linux приложения представляют собой пользовательские процессы, которые выполняются в собственном адресном пространстве, называемом адресным пространством процесса. MMU обеспечивает защиту памяти и, следовательно, ни одно из приложений не может повредить адресное пространство любого другого приложения. MMU также обеспечивает виртуальную память (VM) и каждое приложение способно адресовать максимально доступный предел виртуальной памяти (* Ограничением виртуальной памяти на 32-х разрядных (x86) машинах является 4 Гб.) независимо от ограничения, налагаемого размером системной физической памяти.

С другой стороны, в uClinux, не имеющем VM, приложения не могут работать в индивидуальном виртуальном адресном пространстве процесса. Поэтому распределители адресных пространств приложений совместно используют доступную память. Отсутствие MMU означает, что нет способа реализации защиты памяти и, следовательно, все процессы вместе используют единое глобальное пространство памяти. Поскольку все процессы используют совместно одно пространство памяти, любой процесс может повредить данные другого процесса. Это главный недостаток с которым придётся жить разработчикам uClinux.

Пользовательский режим и режим ядра

Защита памяти в пользовательском режиме/режиме ядра вместе с рабочими режимами процессора в системе Linux осуществляется с помощью эффективного использования MMU. Из-за отсутствия MMU uClinux не имеет защиты памяти и разделения ядра и приложений пользовательского пространства. Это означает, что любое приложение пользовательского пространства может повредить память ядра или даже вызвать поломку системы.

Вызов страниц по требованию и подкачка

В имеющих MMU процессорах можно запрограммировать обработчики ошибок отсутствия страниц в памяти. Linux использует обработчик ошибки отсутствия страницы в памяти (* На самом деле большинство особенностей виртуальной памяти используемого в Linux страничного кэша, COW (копирование при записи) и своп имеют части, реализованные в обработчике ошибки отсутствия страницы в памяти.) для реализации вызова страниц по требованию и свопинга. Поскольку обработчики ошибки отсутствия страницы в памяти в не имеющих MMU процессорах не могут быть установлены, это означает, что не может быть

вызова страниц по требованию и подкачки. Хотя uClinux имеет общее пространство памяти для ядра и приложений, он позволяет образу приложения отличаться от ядра. Таким образом, приложение и ядро могут быть сохранены как отдельные программы в файловой системе и могут быть загружены отдельно. uClinux достигает этого искусной настройкой набора инструментов для сборки программ (компиляторов, компоновщиков и так далее) и загрузчиков, которые необходимы для выполнения программы. uClinux имеет свой собственный набор программ для разработки ПО. На самом деле, большая часть работы по портированию uClinux на новую платформу состоит в том, чтобы получить рабочий набор инструментов для сборки программ. После того, как набор инструментов стал доступен, получение работающих на uClinux приложений становится удивительно простым. Обсуждение изменений, необходимых для портирования набора инструментов, выходит за рамки данной главы; вместо этого данная глава посвящена тому, как uClinux стремится поддерживать для разработчиков то же окружение, что и обычный Linux.

Проблемы разработки, связанные с работой платформы Linux на процессоре без MMU и поддержанием платформы как можно ближе к существующей системе, многочисленны и сложны. В этой главе мы попытаемся указать на некоторые из этих проблем разработки и как инженеры uClinux решили их, объясняя основы, используемые в каждом подходе.

Оставшаяся часть этой главе разделена на две части.

- В первой части (Разделы с 10.2 по 10.6) подробно обсуждаются концепции, лежащие в основе uClinux, очертания строительных блоков, а также описываются изменения, внесённые в Linux, чтобы иметь возможность работать на системе без MMU.
- Вторая часть является подробным руководством по портированию приложений на uClinux. Если вы заинтересованы только в портировании приложений и не интересуетесь внутренностями системы, то можете пропустить эти разделы и перейти к Разделу 10.7 353.

10.2 Загрузка и выполнение программ

Стандартные приложения Linux связаны (слинкованы) с абсолютными адресами. Другими словами, компилятор и компоновщик собирают приложения с предположением, что каждому приложению доступен весь диапазон виртуальных адресов памяти.

Мы напишем небольшую программу для x86, чтобы получить представление о том, как организованы различные сегменты в доступном адресном пространстве. (* Каждое приложение на x86 способно адресовать диапазон из 4 Гб виртуальных адресов. 1 Гб этого пространства отведено для пространства ядра, а остальные 3 Гб доступны для приложений пользовательского пространства.)

```
int data1=1; // будет располагаться в секции .data
int data2=2; // также будет располагаться в секции .data
int main(int argc, char *argv[]) // .text
{
  int stack1=1; // программный стек
  int stack2=2; // программный стек
  char *heap1 = malloc(0x100); // выделение памяти в "куче"
  char *heap2 = malloc(0x100); // также выделение памяти в "куче"

  printf(" text %p\n", main);
  printf(" data %p %p\n", &data1, &data2);
  printf(" heap %p %p\n", heap1, heap2);
```

```
  printf(" stack %p %p\n", &stack1, &stack2);
}
```

Вывод программы:

```
text    0x804835c
data    0x80494f0 0x80494f4
heap    0x8049648 0x8049750
stack   0xbfffe514 0xbfffe510
```

Вывод программы делает ясным, где находится каждый раздел, и в каком направлении растёт каждый сегмент. Стек расположен около верхней части **PAGE_OFFSET** (**0xC000_0000**) и растёт вниз. Текст находится ближе к нижней части памяти (библиотеки имеют некоторые области, зарезервированные ещё ниже), следом идёт раздел данных. После окончания данных (то есть инициализированные данные + bss) начинается распределение "кучи", растущее вверх, в направлении растущего стека. Что происходит, когда "куча" и стек встречаются? В этом случае обработчик ошибки страницы посылает в программу сигнал **SIGSEGV**. Рисунок 10.1 показывает стандартную карту памяти приложения Linux.

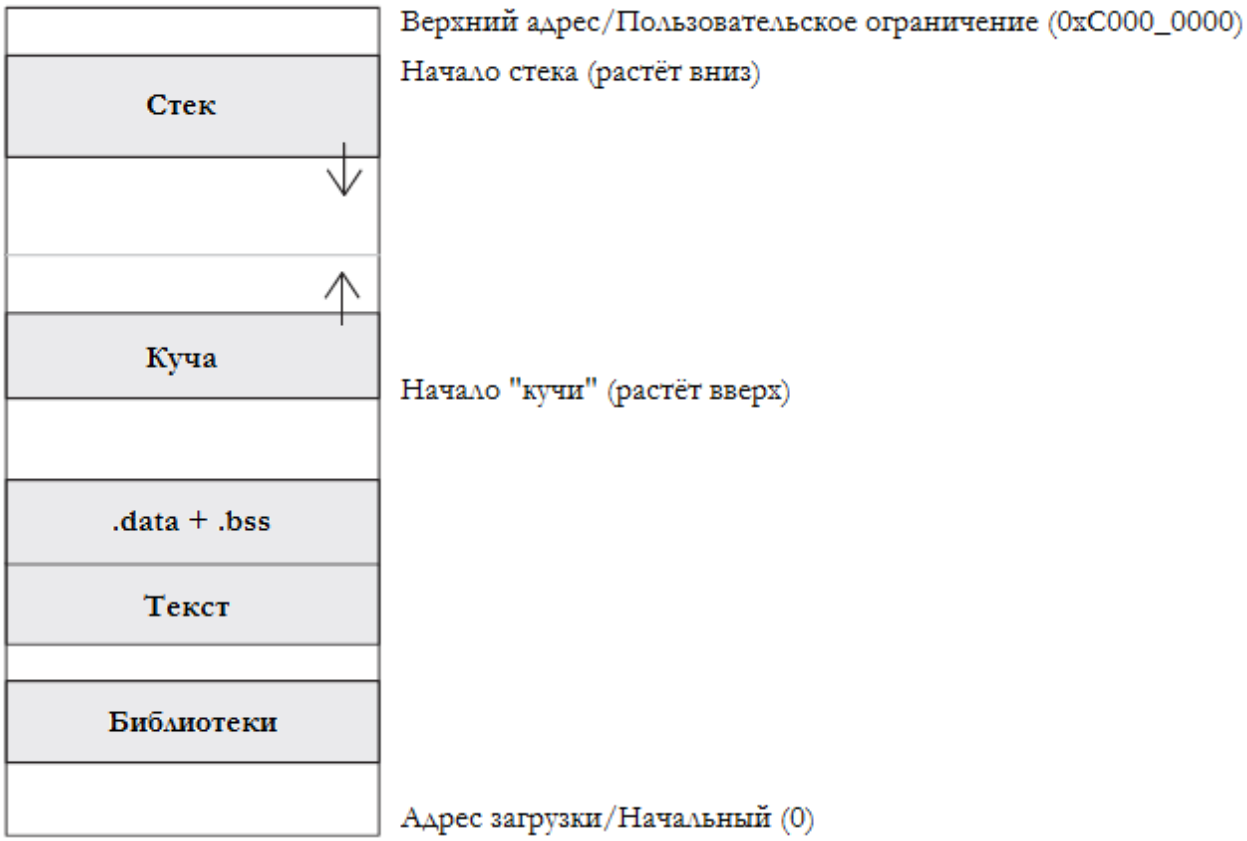

Рисунок 10.1 Карта памяти приложения для Linux.

Запустите несколько экземпляров программы и вывод будет в том же диапазоне для всех экземпляров приложения. Это становится возможным из-за MMU, который помогает в предоставлении отдельного виртуального адресного пространства для каждого процесса. Приложение имеет дело только с виртуальным адресом. Программное обеспечение

виртуальной памяти в ядре и оборудование MMU связывают виртуальный адрес с реальным физическим адресом.

Теперь же из-за отсутствия VM не может быть создано отдельное виртуальное адресное пространство для каждой программы. Поэтому в uClinux приложения вынуждены использовать совместно всё свободное адресное пространство как один большой непрерывный физический кусок. Программы загружаются в доступную свободную часть памяти, в любом произвольном месте памяти. Напомним, что на системах с MMU это произвольное расположение будет связано с нулевым виртуальным адресом в карте памяти процесса. В отличие от стандартного Linux это означает, что начальный адрес программы неизвестен (а не произвольный адрес) и адресация, используемая в инструкциях, не может быть абсолютной. (* Эта проблема называется проблемой неизвестного адреса.) Загрузчики uClinux занимаются этой дополнительной работой по модификации программы при запуске на основе доступного начального адреса. Компиляторам и компоновщикам также необходимо взаимодействовать с загрузчиком для оказания помощи в этой работе. uClinux имеет два различных метода для решения этой проблемы неизвестного адреса.

10.2.1 Полностью перемещаемые двоичные файлы (FRB)

Такой двоичный файл скомпилирован с текстом, начинающимся с нулевого адреса. Компилятор создаёт фиксированный позиционно-независимый код (Position Independent Code, PIC). Для помощи загрузчику для загрузки такого образа по любому произвольному адресу компоновщик добавляет в конце сегмента данных таблицу переадресации. Записи в таблице переадресации указывают места в файле, которые нуждаются в модификации. Загрузчик копирует эти сегменты текста и данных в оперативную память и пробегает по таблице переадресации, модифицируя каждую запись, добавляя начальный адрес соответствующего сегмента, доступный во время загрузки.

(*FRB - Fully Relocatable Binaries, Полностью перемещаемые бинарные файлы)

10.2.2 Позиционно независимый код (PIC)

В этом случае компилятор создаёт позиционно независимый код с помощью программного счётчика - относительно текстовых адресов. (* Например, вызов функции будет использовать “call 0x500”, означающий вызов функции, расположенный по адресу PC+0x500 или “jmp 0x20”, означающий переход по адресу PC+0x20.) Относительная адресация требует аппаратной поддержки, инструкций, которые способны интерпретировать адресацию относительно PC (Program Counter, счётчик команд, программный счётчик). Все данные адресуются с помощью таблицы, называемой глобальной таблицей смещений (Global Offset Table, GOT). GOT помещается в начало сегмента данных и содержит указатели адресов данных, используемых в коде. На некоторых архитектурах, таких как m68k, размер GOT ограничен.

Выполнение на месте (eXecute In Place, XIP)

Режим PIC также подходит для реализации выполнения на месте (eXecute-In-Place, XIP). В отличие от случая смены адресов, когда текст и данные необходимо перед исполнением скопировать в память (для модификации адресов переходов и вызовов), в двоичных файлов на основе PIC ничего исправлять не нужно. Программа может начать выполнение сразу, как только был настроен сегмент данных. (* Заметим, что для этого сегменты данных и текста должны быть разделены с помощью соответствующих аргументов компилятора.) XIP

использует это свойство и работает с текстом непосредственно на флеш/ПЗУ, на месте. Несколько экземпляров программы просто создают новые сегменты данных, а текстовый сегмент фактически является общим для всех экземпляров. В Таблице 10.1 приведены различия между FRB и PIC.

Table 10.1 Сравнение FRB и PIC

Полностью перемещаемый бинарный файл	*Позиционно независимый код*
XIP не возможен.	XIP возможен.
Несколько экземпляров одной программы приводят к потере памяти, так как для каждого экземпляра текстовые сегменты должны быть скопированы в память.	С XIP текстовый сегмент является общим для нескольких экземпляров, не требуя выделения памяти.
Время запуска большое, так как до запуска должны быть модифицированы адреса.	Небольшое время запуска.
Работает на всех платформах.	PIC требует поддержки на данной платформе(относительный режим адресации).

10.2.3 Файловый формат bFLT

Форматом исполняемого файла для стандартного Linux является ELF. uClinux вводит новый формат файлов, разработанный для решения следующих задач:

- Упрощение процесса загрузки и выполнения приложения.
- Создание небольшого и эффективно использующего память формата файла (заголовки ELF являются большими).
- Создание файлового формата, который поможет решить проблемы при загрузке программ в системах без MMU.
- Обеспечение хранения таблицы переадресации для FRB или глобальной таблицы смещений в случае исполняемых файлов PIC.

Форматом файла, используемым в uClinux, является двоичный FLAT (bFLT). Компиляторы и компоновщики uClinux имеют специальные флаги, которые помогают генерировать bFLT файл на основе FRB или PIC. Ядро uClinux также имеет новый загрузчик, который может интерпретировать заголовки bFLT. Ниже приведена структура 64-х байтного заголовка bFLT, находящаяся по адресу со смещением 0 любом файле bFLT.

```
struct flat_hdr {
  char magic[4];
  unsigned long rev;        /* Версия */
  unsigned long entry;      /* Смещение первой исполняемой
                               инструкции с текстовым сегментом
                               от начала файла */
  unsigned long data_start; /* Смещение сегмента данных
                               от начала файла */
  unsigned long data_end;   /* Смещение конца сегмента
```

```
                                данных от начала файла */
  unsigned long bss_end;    /* Смещение конца сегмента bss
                                от начала файла */

  /*
   * Предполагается, что участок между data_end и bss_end
   * формирует сегмент bss.
   */

  unsigned long stack_size;  /* Размер стека в байтах */
  unsigned long reloc_start; /* Смещение записей переадресации
                                от начала файла */
  unsigned long reloc_count; /* Количество записей переадресации */
  unsigned long flags;
  unsigned long filler[6];   /* Зарезервировано, устанавливается в 0 */
};
```

Строка **magic** в любом файле bFLT представляет из себя 4-х байтовую последовательность ASCII символов '**b**', '**f**', '**l**', '**t**' или **0x62**, **0x46**, **0x4C**, **0x54**. **rev** указывает номер версии файла. **entry** указывает на начальное смещение текстового сегмента от начала файла. Обычно это **0x40** (64), размер данного заголовка. Сразу же после текста расположен сегмент данных, а размер сегмента данных хранится в **data_start** и **data_end**. Сегмент **bss** начинается с **data_end** и заканчивается в **bss_end**. Заголовок bFLT хранит размер стека, выделенного для приложения, в **stack_size**. (* Размер стека может быть установлен в момент преобразования файла ELF в FLT.) Позже мы покажем, почему необходимо указать размер стека.

Поля **reloc_start** и **reloc_count** предоставляют информацию о начальном смещении и количестве записей переадресации. Напомним, что каждая запись переадресации является указателем на абсолютный адрес, который должен быть модифицирован. Новый адрес для записи рассчитывается путём сложения базового адреса соответствующего сегмента с абсолютным адресом, указанным в записи.

Загрузчик файлов bFLT в ядре реализован в файле **linux/fs/binfmt_flat.c**. Основной функцией является **load_flat_binary**. Эта функция отвечает за загрузку и выполнение файла bFLT в системе uClinux. Функция читает заголовок и выделяет необходимую память. На основании записей в поле **flags** объём памяти, выделяемый для бинарного файла PIC, будет размером со стек и данные (в том числе GOT). Если это бинарный файл не PIC формата, то это размер стека, данных (в том числе таблицы переадресации) и текста. Он также связывает необходимые текстовые сегменты и помечает страницы как исполняемые. Данный формат файлов также позволяет иметь сжатый **gzip** текстовый раздел, о чём указывается в поле **flags**. В этом случае загрузчик также заботится о распаковке текстовых разделов в оперативную память. После того, как все разделы связаны между собой, мы готовы к модификации адресов. Рисунок 10.2 показывает разделы файла bFLT, а Рисунок 10.3 показывает единый файл при загрузке в память. Отметим, что стек, попадающий в bss, приведёт к падению системы, поскольку обработчика ошибок нет.

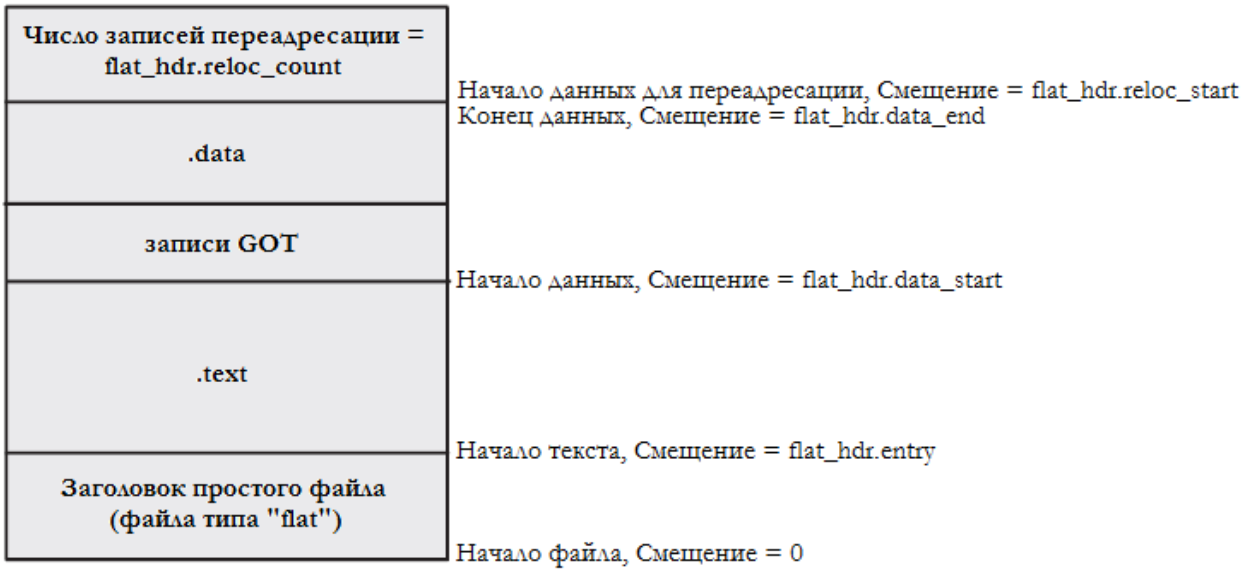

Рисунок 10.2 Разделы файла bFLT.

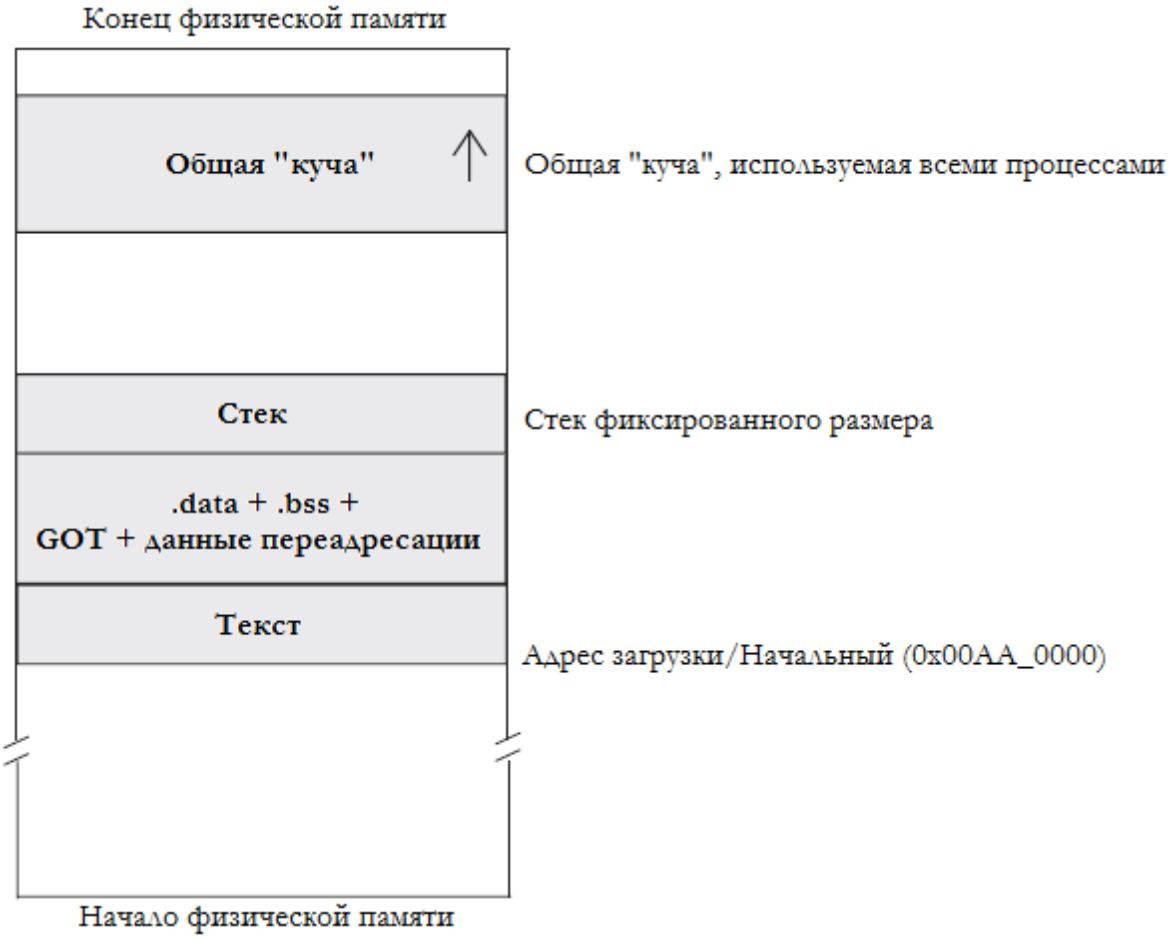

Рисунок 10.3 Файл bFLT, загруженный в память.

10.2.4 Загрузка файла bFLT

Загрузка простого ("flat") файлового формата обрабатывается в функции **load_flat_file**. Шаги в процедуре загрузки следующие:

1. Сначала функция читает разные поля заголовка и рассчитывает необходимую память и тип требующейся загрузки.

```
text_len  = ntohl(hdr->data_start);
data_len  = ntohl(hdr->data_end) - ntohl(hdr->data_start);
bss_len   = ntohl(hdr->bss_end) - ntohl(hdr->data_end);
stack_len = ntohl(hdr->stack_size);

if (extra_stack) {
  stack_len += *extra_stack;
  *extra_stack = stack_len;
}
relocs = ntohl(hdr->reloc_count);
flags  = ntohl(hdr->flags);
rev    = ntohl(hdr->rev);
    …
    …
/*
 * рассчитываем дополнительное пространство,
 * необходимое для отображения
 */
extra = max(bss_len + stack_len, relocs *
        sizeof(unsigned long));
```

2. Затем загрузчик отображает разделы файла в оперативную память или берёт их из флеш-памяти на основе установленных флагов в заголовке файла. Например, если файл имеет разделы со сжатыми данными или текстом, то загрузчик должен сначала считать такой файл в память и распаковать эти разделы. Распечатка 10.1 338 показывает каждый логический шаг в этом процессе.

3. Загрузчик настраивает структуру задачи процесса, указывая информацию о местонахождении стека, данных и текста.

```
current->mm->start_code = start_code;
current->mm->end_code = end_code;
current->mm->start_data = datapos;
current->mm->end_data = datapos + data_len;
```

4. Наконец, он выполняет модификацию адресов, как показано в Распечатке 10.2 340.

Проверка на наличие GOT производится с помощью переменной **flags**, полученной ранее из заголовка файла bFLT. Также отметим, что **datapos** является началом смещением таблицы GOT после отображения файла в память. Теперь загрузчику необходимо модифицировать каждую запись, присутствующую в GOT. Функция **calc_reloc** делает необходимое исправление адреса, как показано в следующем фрагменте:

```
static unsigned long
```

```
calc_reloc(unsigned long r, struct lib_info *p, int curid, int internalp)
{
    …
    …
  if (r < text_len) /* В текстовом сегменте */
    addr = r + start_code;
  else /* В сегменте данных */
    addr = r - text_len + start_data;

  return addr;
}
```

Расчёт прост. Если адрес, который должен быть модифицирован, находится в пределах текстового сегмента, то к нему добавляется начальный адрес текстового сегмента. В противном случае это адрес в сегменте данных и, следовательно, модификация адреса осуществляется на основе начального адреса сегмента данных. После модификации записей GOT, записи таблицы переадресации проходят и приводятся в порядок через функцию **calc_reloc**. Чтобы понять тонкости перемещения, рассмотрим пример.

Случай FRB (полностью перемещаемых бинарных файлов)

Создадим файл **sample.c** с пустой функцией **main**.

Sample.c

```
main {}
```

Компилируем и создаём файл типа "flat". О том, как компилировать файлы FRB с корректными параметрами компилятора, смотрите Раздел 10.7.1 353. Пусть выходной файл называется **sample**, а файл символов будет **symbol.gdb**. Заголовок файла "flat" выведен с использованием программы **flthdr**.

```
#flthdr sample

  Magic:        bFLT
  Rev:          4
  Entry:        0x48
  Data Start:   0x220
  Data End:     0x280
  BSS End:      0x290
  Stack Size:   0x1000
  Reloc Start:  0x280
  Reloc Count:  0x1c
  Flags:        0x1  ( Load-to-Ram )
```

Вывод может быть напрямую соотнесён с заголовком bFLT, описанным выше. Поскольку это перемещаемый бинарный файл, обратите внимание на установленный в заголовке флаг **Load-to-RAM** (загрузка в память). Также **Reloc Count** равен 0x1C, то есть для перемещения пустого файла **main** было создано 28 записей. Чтобы разгадать эту тайну, давайте посмотрим на файл символов, **symbol.gdb**.

```
# nm sample.gdb
  00000004 T _stext
  00000008 T _start
  00000014 T __exit
  0000001a t empty_func
  0000001a W atexit
  0000001c T main
  00000028 T __uClibc_init
  0000004a T __uClibc_start_main
  000000ba T __uClibc_main
  000000d0 T exit
    …
    …
  00000214 D _errno
  00000214 V errno
  00000218 D _h_errno
  00000218 V h_errno
  0000021c d p.3
    …
    …
  00000240 B __bss_start
  00000240 b initialized.10
  00000240 B _sbss
  00000240 D _edata
  00000250 B _ebss
  00000250 B end
```

Эта распечатка показывает различные символы, находящиеся в файле. Поскольку любое приложение должно быть связано с **libc**, то в этой распечатке символов все выведенные символы, кроме **0x1c T main()**, взяты из **libc**, в данном случае это **uClibc**. Данные и текст **libc** имеют в себе точки для перемещения, указанные в таблице переадресации: 28 записей для перерасчёта.

Теперь в файл **sample.c** добавим код.

Sample.c

```
int x=0xdeadbeef;
int *y=&x;

main () {
  *y++;
}
```

Снова скомпилируем и распечатаем заголовок файла.

```
#flthdr sample

  Magic:         bFLT
  Rev:           4
  Entry:         0x48
  Data Start:    0x220
  Data End:      0x280
  BSS End:       0x290
```

```
  Stack Size:   0x1000
  Reloc Start:  0x280
  Reloc Count:  0x1f
  Flags:        0x1 ( Load-to-Ram )
```

Обратите внимание на увеличение счётчика записей переадресации с 0x1c до 0x1f; созданы три дополнительные записи переадресации. В новом **sample.c** в разделе данных создана запись для перерасчёта адреса **y**, указывающего на **x**. Также увеличение содержимого **y** создаёт запись для перемещения в текстовом разделе.

Используя **nm** для данных **sample.gdb** получаем следующие адреса **x** и **y**:

```
  …
00000200 D x
00000204 D y
  …
```

В таблицу переадресации добавлена запись для **y**. Таблица содержит запись 204, которая должна быть перерасчитана. Также должен быть перерасчитан адрес, указываемый в 204 (то есть 200). Это то, что выполняется в Распечатке 10.2 340. Ниже приводится простая интерпретация кода перерасчёта адресов.

Сделаем распечатку **sample** с помощью **od**. Мы выводим здесь только таблицу переадресации, начиная с байта со смещением 280, как указано в заголовке этого файла.

```
#od -Ax -x sample -j280

  000280 0000 1e00 0000 2400 0000 2c00 0000 3600
  000290 0000 4000 0000 4600 0000 6400 0000 6c00
  0002a0 0000 7600 0000 7e00 0000 8c00 0000 9e00
    …
```

Теперь обратим внимание на вторую ненулевую запись 0x2400. Шаги, выполняемые для перерасчёта, следующие:

- Шаг 1:

  ```
  relval = ntohl(reloc[i]);
  addr = relval;
  ```

  0x2400 должно быть преобразовано в машинную последовательность байт и это 0x0024.

- Шаг 2:

  ```
  rp = (unsigned long *) calc_reloc(addr, libinfo, id, 1);
  ```

  Это означает, что запись 0x0024 должна быть перерасчитана. **calc_reloc** вернёт перерасчитанный адрес, который будет в момент выполнения, на основе начального адреса текста (поскольку 0x24 < text_len).

- Шаг 3:

  ```
  addr = *rp;
  ```

Содержимое (start_of_text + 0x0024) для файла **sample** на самом деле адрес **y** 0x204, который должен быть перерасчитан.

- Шаг 4:

```
addr = calc_reloc(addr, libinfo, id, 0);
*rp=addr;
```

Вызов **calc_reloc** для 0x204 и запись в память актуального значения.

Чтобы подвести итог, на Рисунке 10.4 показаны различные этапы выполнения перерасчёта. Для текстовой ссылки также будет создана аналогичная запись.

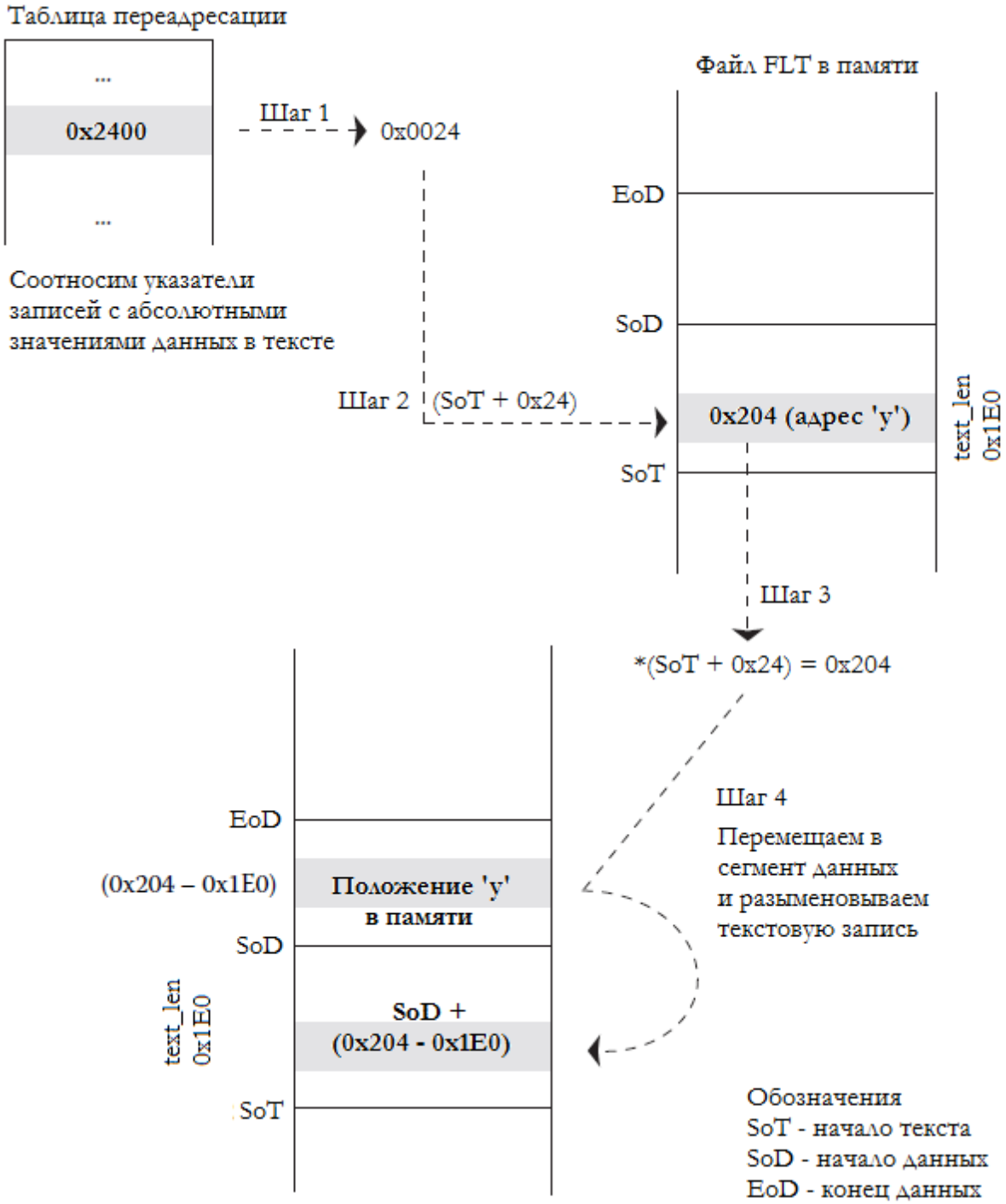

Рисунок 10.4 Перерасчёт адресов файла типа "flat".

Случай PIC

Снова возьмём файл **sample.c** с пустой функцией **main**.

Sample.c

```
main {}
```

Скомпилируем программу с включённым параметром PIC и изучим созданный код.

```
#flthdr sample

  Magic:         bFLT
  Rev:           4
  Entry:         0x48
  Data Start:    0x220
  Data End:      0x2e0
  BSS End:       0x2f0
  Stack Size:    0x1000
  Reloc Start:   0x2e0
  Reloc Count:   0x2
  Flags:         0x2 ( Has-PIC-GOT )
```

Обратите внимание, что есть только две записи для перерасчёта и поле **flags** указывает на наличие GOT. Напомним, что GOT присутствует в начале раздела данных с последней записью, указанной с помощью -1. Чтобы изучить содержимое нашего файла и увидеть наличие GOT, мы используем **od**.

```
#od -Ax -x sample

  000000 4662 544c 0000 0400 0000 4800 0000 2002
  000010 0000 e002 0000 f002 0000 0010 0000 e002
  000020 0000 0200 0000 0200 1342 678b 0000 0000
    …
    …
  000220 0000 0000 0000 0000 0000 0000 0000 6802
  000230 0000 7c02 0000 a002 0000 aa01 0000 7402
  000240 0000 6002 0000 6c02 0000 7002 0000 2001
  000250 0000 8802 0000 6402 0000 2800 0000 1c00
  000260 0000 0000 0000 0000 0000 0000 0000 c800
  000270 0000 0000 0000 0001 0000 4400 0000 0000
  000280 ffff ffff 0000 0000 0000 0000 0000 0000
  000290 0000 0000 0000 0000 0000 0000 0000 0000
    …
```

Обратим внимание на начало GOT по адресу 0x220, как указано в **Data Start**: 0x220, и конец GOT по адресу 0x280, определяемому по значению 0xFFFFFFFF, в общей сложности 16 действительных (ненулевых) записей GOT. Все эти записи GOT предназначены для стандартных символов **libc**.

Теперь в **sample.c** снова добавим наши строчки, скомпилируем и распечатаем заголовок.

Sample.c

```
int x=0xdeadbeef;
int *y=&x;

main () {
  *y++;
}
```

```
#flthdr sample

  Magic:         bFLT
  Rev:           4
  Entry:         0x48
  Data Start:    0x220
  Data End:      0x2e0
  BSS End:       0x2f0
  Stack Size:    0x1000
  Reloc Start:   0x2e0
  Reloc Count:   0x3
  Flags:         0x2  ( Has-PIC-GOT )
```

Сразу же замечаем увеличение счётчика **Reloc** на 1. Это перерасчёт для **x** по адресу в **y**. Как и ожидалось, ссылка на **y** в тексте создаёт запись в GOT. С помощью **od** сделаем распечатку **sample**.

```
000220 0000 0000 0000 0000 0000 0000 0000 7002
000230 0000 8402 0000 a002 0000 b601 0000 7c02
000240 0000 6802 0000 7402 0000 7802 0000 2c01
000250 0000 9002 0000 6c02 0000 3400 0000 1c00
000260 0000 6402 0000 0000 0000 0000 0000 0000
000270 0000 d400 0000 0000 0000 0c01 0000 5000
000280 ffff ffff 0000 0000 0000 0000 0000 0000
```

У нас есть в общей сложности 17 записей в GOT, то есть одна дополнительная запись. Нам необходимо определить ту запись GOT, которая соответствует **y**. Выполним **nm sample.gdb** и посмотрим переменные, как и раньше.

```
  …
00000260 D x
00000264 D y
  …
```

Выделенная жирным запись выше в таблице GOT показывает запись для **y**.

Загрузчик в Распечатке 10.2 340 выполняет проверку на флаг GOT и сначала перемещает записи GOT, а затем продолжает модифицировать записи. Рисунок 10.5 показывает, как перерасчитывается запись GOT.

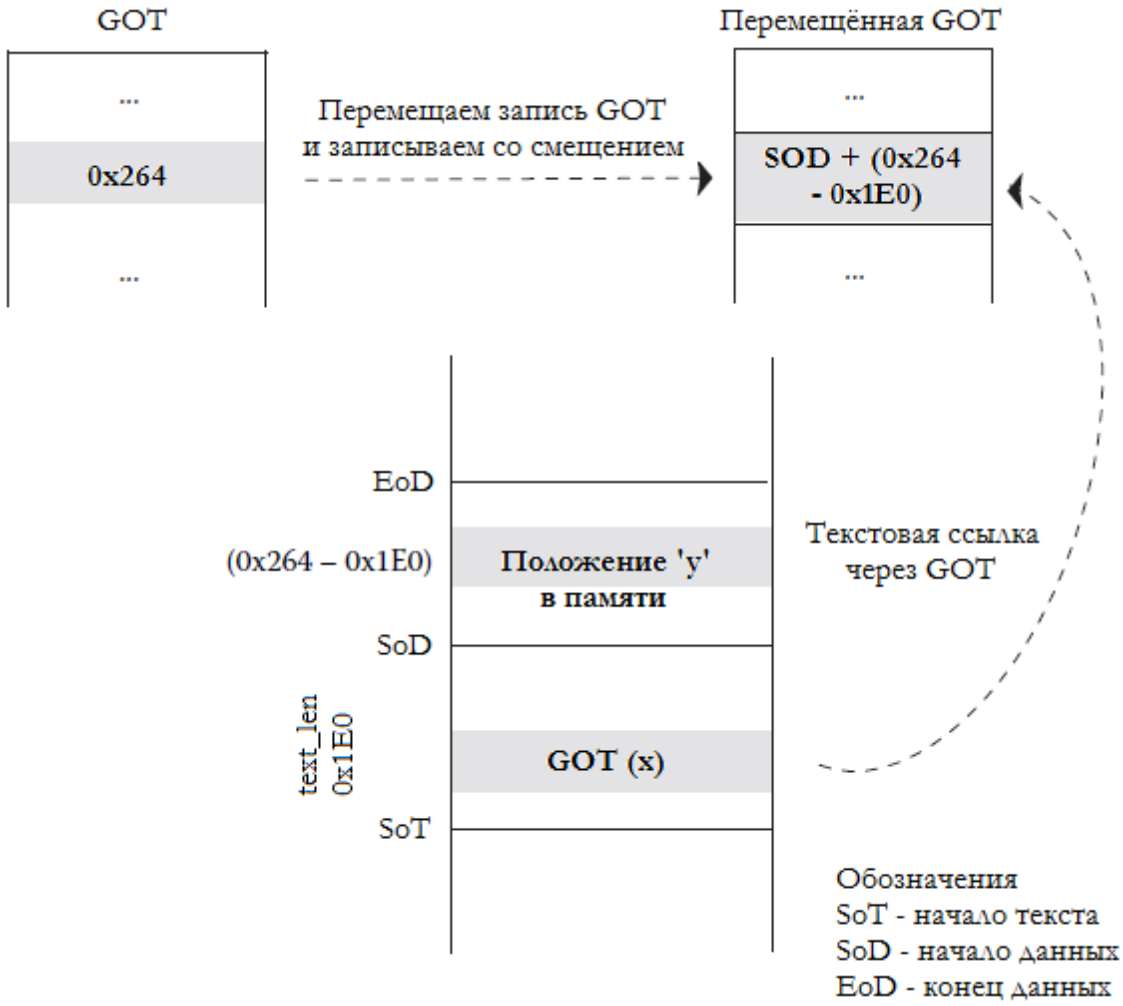

Рисунок 10.5 Перерасчёт записи GOT.

Подводим итог нашего обсуждения в этом разделе.

- uClinux использует формат файла bFLT (Binary FLAT) .
- bFLT может быть либо FRB, либо PIC.
- FRB имеет абсолютные ссылки на данные и текст, записи переадресации указывают на места, которые должны быть исправлены загрузчиком.
- PIC работает с адресацией относительно текста и, следовательно, требует поддержки на уровне платформы. Ссылки на данные в тексте делаются с помощью GOT. GOT содержит указатели данных, которые исправляются загрузчиком.
- XIP предусматривает запуск файлов с флеш-памяти, экономя таким образом на текстовом пространстве, которое в противном случае занимало бы место в оперативной памяти

Распечатка 10.1 Загрузчик bFLT

Распечатка 10.1

```
  …
if ((flags & (FLAT_FLAG_RAM|FLAT_FLAG_GZIP)) == 0) {
```

```
  …

/*
 * Случай отображения в ПЗУ: отображаем из файла для XIP
 */
textpos = do_mmap(bprm->file, 0, text_len, PROT_READ|PROT_EXEC,
                                                        0, 0);

  …

/*
 * Выделяем память для разделов данных, стека и данных переадресации.
 * Отметим, что это делается также для общих библиотек
 */
realdatastart = do_mmap(0, 0, data_len + extra +
                        MAX_SHARED_LIBS * sizeof(unsigned long),
                        PROT_READ|PROT_WRITE|PROT_EXEC, 0, 0);

  …
  …

/*
 * Читаем секцию .data из файла в память, распаковываем,
 * если необходимо
 */
#ifdef CONFIG_BINFMT_ZFLAT
if (flags & FLAT_FLAG_GZDATA) {
  result = decompress_exec(bprm, fpos, (char *) datapos,
                data_len + (relocs * sizeof(unsigned long)), 0);
}else
#endif
{
result = bprm->file->f_op->read(bprm->file, (char *) datapos,
                data_len + (relocs * sizeof(unsigned long)), &fpos);
}

  …
  …

} else {
/*
 * Случай отображений в RAM: Выделение памяти для всего
 * (текста, данных, стека, данных переадресации)
 */
textpos = do_mmap(0, 0, text_len + data_len + extra +
                  MAX_SHARED_LIBS * sizeof(unsigned long),
                  PROT_READ | PROT_EXEC | PROT_WRITE, 0, 0);

  …
  …

/*
 * Читаем разделы .text, .data из файла в ОЗУ, распаковываем,
```

```
 * если необходимо
 */
if (flags & FLAT_FLAG_GZIP) {
  result = decompress_exec(bprm, sizeof (struct flat_hdr),
           (((char *) textpos) + sizeof (struct flat_hdr)),
           (text_len + data_len + (relocs * sizeof(unsigned long))
           - sizeof (struct flat_hdr)), 0);
  …
result = bprm->file->f_op->read(bprm->file, (char *) textpos,
                                text_len, &fpos);
  …
```

Распечатка 10.2 Модификация адресов, выполняемая загрузчиком

Распечатка 10.2

```
/* Проверяем GOT и делаем модификацию адресов в GOT */

if (flags & FLAT_FLAG_GOTPIC) {
  for (rp = (unsigned long *)datapos; *rp != 0xffffffff; rp++) {
    unsigned long addr;
    if (*rp) {
      addr = calc_reloc(*rp, libinfo, id, 0);
      if (addr == RELOC_FAILED)
        return -ENOEXEC;
      *rp = addr;
    }
  }
}

/* Пробегаем по записям данных переадресации и также их модифицируем */

for (i=0; i < relocs; i++) {
  unsigned long addr, relval;

  /* Получаем адрес указателя для перерасчёта (конечно,
   * этот адрес сначала должен быть перемещён).
   */
  relval = ntohl(reloc[i]);
  addr = flat_get_relocate_addr(relval);
  rp = (unsigned long *) calc_reloc(addr, libinfo, id, 1);

  …
  …

  /* Получаем значение указателя. */
  addr = flat_get_addr_from_rp(rp, relval, flags);
  if (addr != 0) {
    /*
     * Выполняем перерасчёт адреса. PIC уже перемещён
     * в раздел данных в нужном порядке
     */
    if ((flags & FLAT_FLAG_GOTPIC) == 0)
```

```
        addr = ntohl(addr);

    addr = calc_reloc(addr, libinfo, id, 0);

    …
    …

    /* Записываем перерасчитанный указатель обратно. */
    flat_put_addr_at_rp(rp, addr, relval);
}
```

10.3 Управление памятью

В этом разделе мы узнаем об изменениях, сделанных в ядре и **libc**, касающихся управления памятью в uClinux. Мы разделим наше обсуждение на две части, на основе двух разделов памяти, "куче" (хипе) и стеке.

10.3.1 Куча

Выделение памяти

Библиотечными вызовами, использующимися для выделения/освобождения памяти в "куче", являются **malloc**, **realloc**, **calloc** и **free**. Основной функцией является **malloc**, и нам необходимо понять, как работает **malloc** на стандартном Linux, и почему она не может быть использована в uClinux.

malloc обеспечивает динамическое выделение памяти в "куче" в процессе работы. По существу, чтобы управлять размером адресного пространства процесса, она использует низкоуровневые системные вызовы **sbrk()** / **brk()**. **sbrk()** добавляет память в конец адресного пространства процесса, увеличивая тем самым размер. **brk()**, с другой стороны, можно задать произвольный размер в пространстве процесса. Они эффективно используются библиотечными вызовами **malloc** / **free** для выделения/освобождения памяти для приложений, соответственно. Пространство процесса в стандартном Linux является виртуальным адресным пространством и, следовательно, добавление большей памяти выполняется простой настройкой структур виртуальной памяти в ядре, которое обеспечивает необходимую связь с физическими адресами. Так что **malloc** просто увеличивает размер виртуальной памяти, а **free** уменьшает размер виртуальной памяти по мере необходимости. Например, рассмотрим приложение размером 64 K (включая размер bss) с начальным размером "кучи" равным 0. При этом используется от 0 до 64 K в виртуальной памяти (общий размер виртуальной памяти = 64 K + размер стека), как показано на Рисунке 10.6.

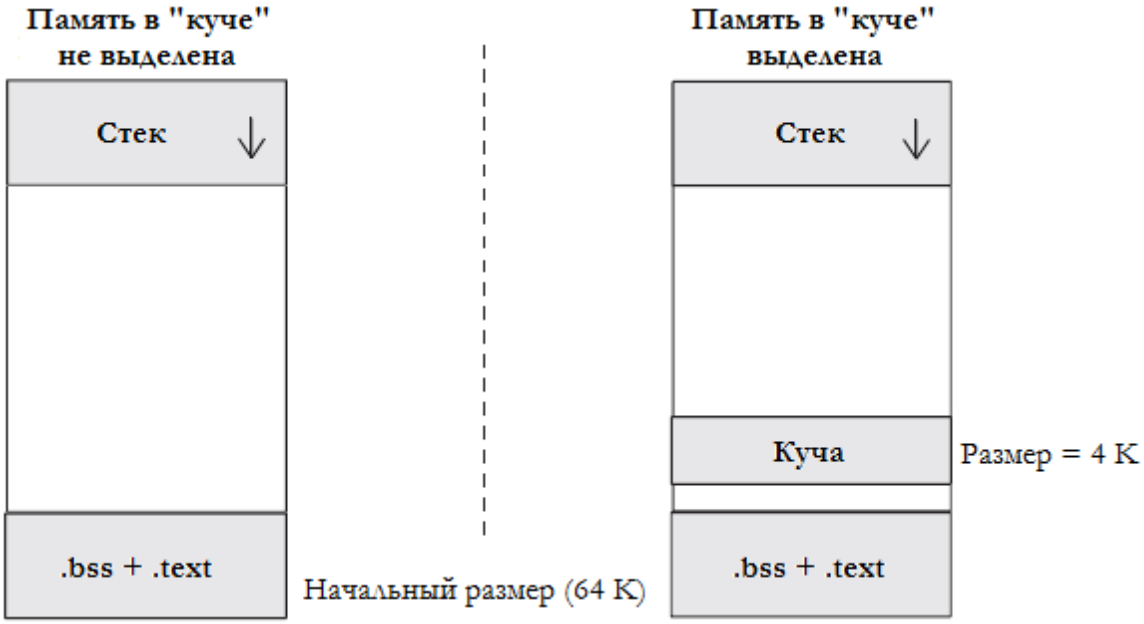

Рисунок 10.6 Выделение памяти в "куче".

Теперь предположим, что приложение делает вызов **malloc(4096)**. Это увеличит размер виртуальной памяти до 68 K, как показано на Рисунке 10.6. Фактическая физическая память выделяется только на фактическое время использования в обработчике ошибки отсутствия страницы в памяти.

В uClinux нет виртуальной памяти, так что добавлять или изменять границы адресного пространства процесса невозможно. Поэтому пришлось обеспечивать альтернативные решения. Такое альтернативное решение должно позволить обходиться очень небольшими изменениями или вообще их отсутствием при переносе приложений на uClinux. Такие простые решения, как отсутствие в uClinux "кучи", предварительное выделение всей необходимой памяти с использованием статических массивов, предоставление "кучи" фиксированного размера, и так далее, исключены. Это потребует огромных усилий по переносу и даже реорганизацию большинства приложений, что не желательно.

В uClinux такое решение обеспечивается использованием общесистемного пула свободной памяти для выделения памяти в "куче" всеми процессами. Поскольку все процессы выделяют память в одном пуле, один процесс может забрать всю доступную память и в результате случится нехватка общесистемной памяти. Это системный недостаток любой конструкции без MMU.

Простейшая реализация **malloc** использует прямые вызовы **mmap()** / **munmap()**. Весь требуемый объём памяти запрашивается непосредственно с помощью распределителя памяти ядра. Реализация приводится ниже.

- Делаем системный вызов **mmap()**, чтобы получить память из пула памяти ядра

```
void * malloc(size_t size) {
  …
  result = mmap((void *) 0, size, PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, 0, 0);
  if (result == MAP_FAILED)
    return 0;
  return(result);
}
```

- Чтобы вернуть использованную память, вызываем **munmap()**

Проблемы такого подхода становятся очевидными из его реализации, показанной в Распечатке 10.3 345, поскольку вызов **mmap** связан с распределителем памяти ядра, **kmalloc**, чтобы получить свободную память. Возвращённая память сохраняется в связном списке указателей для учёта. Учёт необходим для того, чтобы система могла отслеживать память, выделенную каждому процессу и, следовательно, они связаны с структурой процесса, так что она может быть освобождена, когда процесс заканчивает работу. Накладные расходы для структур данных **tblock** и **rblock** составляют 56 байт на одно выделение. Приложения, требующие небольших кусков через **malloc**, очевидно, зря потратят память.

Также, **kmalloc** возвращает куски памяти размером с округлением до степени 2, ограниченные максимальным размером 1 Мб. Например, запрос на выделение 1.5 К (1536) приведёт к потере 0.5 К (512), так как ближайшая доступная степень 2 только 2 К (2048). Таким образом, распределитель очень неэффективен и ограничен.

56-ти байтовые накладные расходы могут быть уменьшены, если сделать API **malloc** умнее. Например, можно либо делать групповые выделения, либо выделять большие куски, а затем управлять этими кусками с помощью меньших структур данных. Эта проблема была также решена в ядре 2.6 и структуры **rblock** и **tblock** были удалены. Вторая проблема требует переписать схему выделения памяти ядра. В ядре uClinux имеется модифицированный метод распределения памяти в ядре. Это помогает уменьшить накладные расходы при выделении памяти. Новая функция **kmalloc** использует степень 2-ки для запросов памяти размером до размера 1-ой страницы (например, 4 КБ или 4096 байт), а для размеров более 1-ой страницы он округляется до границы ближайшей страницы. Например, рассмотрим выделение 100 К с использованием стандартной **kmalloc**. Это приведёт к выделению 128 К и потере 28 К. Однако, используя новую **kmalloc** можно выделить точно 100 К.

uClibc предоставляет три различные реализации **malloc**. Перечислим преимущества и недостатки каждой из их.

- **malloc-simple [uClinux-dist/uClibc/libc/stdlib/malloc-simple]**

```
malloc(size_t size) {
  …
  …
  result = mmap((void *) 0, size, PROT_READ | PROT_WRITE,
                MAP_SHARED | MAP_ANONYMOUS, -1, 0);

  if (result == MAP_FAILED)
    return 0;
  return(result);
}
```

Это простейшая форма **malloc**, используемая в uClinux. Это простая реализация в одну строчку с быстрыми и прямыми выделениями. Недостатком является 56-ти байтовые накладные расходы, которая проявляются при использовании приложений, которые требуют большого числа выделений памяти малого размера.

- **malloc [uClinux-dist/uClibc/libc/stdlib/malloc]:**

В этой реализации **malloc** имеет внутреннюю "кучу", выделенную в статической области. Функция **malloc_from_heap()**, основываясь на требуемом размере, принимает решение о выделении памяти из этой статической области или прибегает к **mmap**. В этом подходе запросы малого размера не имеют при распределении проблемы накладных расходов, так как выделение памяти происходит из внутренней "кучи". Реализация приведена в Распечатке 10.4 346.

- **malloc-standard [uClinux-dist/uClibc/libc/stdlib/malloc-standard]:**

  Это самая сложная из всех реализаций **malloc** и используемая в стандартном вызове **libc**. По существу, библиотека **malloc** поддерживает выделения памяти малого размера внутри себя с помощью кусочков разного размера. Если запрос на выделение укладывается в кусочек определённого размера, то кусочек удаляется из списка свободных и маркируется как использованный (пока он не будет освобождён или приложение не завершится). Если запрашиваемый размер выделения больше, чем имеющийся размер кусочка, или если все уже использованы, библиотека вызывает **mmap**, предоставляющий размер выделения больший, чем пороговый размер. Этот пороговый размер регулируется таким образом, чтобы минимизировать накладные расходы при выделении памяти через **mmap**. В противном случае для увеличения размера "кучи" процесса используется **brk**, и в случае неудачи он в конечном счёте возвращается к **mmap**. При таком подходе очень эффективно управляемые кусочки сокращают накладные расходы при выделении памяти, добавляемые **mmap**, но это решение доступно только на системах с MMU.

Фрагментация памяти

Количество доступной свободной памяти не гарантирует такое же количество выделенной памяти. Например, если система имеет 100 К свободной памяти, это не обязательно означает, что **malloc(100K)** будет успешным. Это происходит из-за проблемы, называемой ***фрагментацией памяти***. Доступная память может быть не непрерывной и, следовательно, запрос на выделение может быть не выполнен. Выделение памяти в системе начинается только на границе страницы. Свободная страница запрашивается из списка доступной свободной памяти и добавляется в список используемой. Любой последующий запрос памяти, который помещается внутри оставшегося пространства, выполняет выделение памяти из той же страницы. И как только страница помечена как использованная, она может быть возвращена в системный список свободной памяти только тогда, когда все выделения памяти, использующие эту страницу, освободили память.

Например, предположим, что программа выполнила 16 256-ти байтовых выделений памяти. Предполагая накладные расходы в схеме выделения памяти равными нулю, это соответствует размеру одной страницы, в нашем случае 4 К = 16*256. Теперь до тех пор, пока приложение не освободит все 16 указателей, используемая страница не может быть освобождена. Таким образом, может быть ситуация, когда в системе есть достаточно памяти, но всё же её недостаточно для обслуживания текущего запроса на выделение.

Дефрагментация памяти

В стандартном Linux все приложения пользовательского пространства используют виртуальные адреса. Дефрагментация памяти проста, в том смысле, что необходимо изменить виртуальное отображения процесса, чтобы указать на новые места физического

расположения. Даже если изменилось физическое местонахождение памяти, программы всё ещё используют виртуальный адрес и в состоянии нормально работать даже после перемещения. Без виртуальной памяти это становится невозможным.

На uClinux не существует решения для дефрагментации и разработчики должны быть осведомлены об этой ситуации и избегать моделей изменения распределения памяти в приложениях.

Распечатка 10.3 Реализация mmap в uClinux

Распечатка 10.3

```
do_mmap_pgoff() {

    …
    …

  if (file) {
    error = file->f_op->mmap(file, &vma);
    if (!error)
      return vma.vm_start;
    …
  }

    …

  tblock = (struct mm_tblock_struct *)
  kmalloc(sizeof(struct mm_tblock_struct),GFP_KERNEL);
    …
    …
  tblock->rblock = (struct mm_rblock_struct *)
  kmalloc(sizeof(struct mm_rblock_struct), GFP_KERNEL);

    …
    …
  result = kmalloc(len, GFP_KERNEL);
    …
    …

  /* Подсчёт ссылок */
  tblock->rblock->refcount = 1;
  tblock->rblock->kblock = result;
  tblock->rblock->size = len;
    …
    …
  /* Подключаем блок к списку блоков структуры задачи */
  tblock->next = current->mm->context.tblock.next;
  current->mm->context.tblock.next = tblock;
  current->mm->total_vm += len >> PAGE_SHIFT;

  return (unsigned long)result;
}
```

Распечатка 10.4 Реализация malloc в uClibc с помощью "кучи"

Распечатка 10.4

```
HEAP_DECLARE_STATIC_FREE_AREA (initial_fa, 256);
struct heap __malloc_heap = HEAP_INIT_WITH_FA (initial_fa);

void * malloc(size_t size) {
  …
  …
  mem = malloc_from_heap (size, &__malloc_heap);
  if (unlikely (!mem))
  {
  oom:
    __set_errno (ENOMEM);
    return 0;
  }
  return mem;
}

static void *
malloc_from_heap (size_t size, struct heap *heap){

  /* Сначала попробуем выделить из области внутренней "кучи" */
  __heap_lock (heap);

  mem = __heap_alloc (heap, &size);

  __heap_unlock (heap);

  /* при неудаче вызываем mmap */
  if (unlikely (! mem)) {
    /* Из "кучи" выделить не смогли, */
    block = mmap (0, block_size, PROT_READ | PROT_WRITE,
                  MAP_SHARED | MAP_ANONYMOUS, 0, 0);
  }
}
```

10.3.2 Стек

Программный стек на системах со стандартным Linux растёт по запросу. Это стало возможным благодаря интеллектуальному обработчику ошибки страницы. Стек растёт вниз от верхней части сегмента данных пространства пользователя. Растущий стек ограничен только собственной растущей "кучей" программы, которая растёт в обратном направлении, начиная с конца bss.

На системах без MMU нет возможности для реализации обработчика ошибки страницы и, следовательно, рост стека по требованию невозможен. Единственным предлагаемым здесь uClinux решением является фиксированный размер стека. Этот размер стека задаётся в момент компиляции и сохраняется как часть образа исполняемого приложения.

Как видно из предыдущего раздела, заголовок bFLT имеет запись под названием **stack_size** для хранения размера стека, резервируемого загрузчиком при запуске программы. Теперь разработчик должен позаботится, чтобы сделать хорошее предположение о максимальном размере стека, необходимом программе, и сделать его доступным в момент создания двоичного образа.

10.4 Отображение файла / памяти — тонкости mmap() в uClinux

В Linux области памяти отображаются в адресное пространство процесса с помощью системного вызова **mmap()**. Отображённая область управляется с помощью флагов защиты, таких как **PROT_READ** и **PROT_WRITE**, а способ использования, закрытое (private) или разделяемое (shared) отображение, указывается с помощью флагов **MAP_PRIVATE** и **MAP_SHARED**. Например, загрузчик отображает текстовые области разделяемых библиотеки как только читаемые общие, разделяемые (**PROT_READ** и **MAP_SHARED**) области в пространстве процесса приложения.

mmap() внутри себя заполняет структуры данных виртуальной памяти, которые описывают параметры каждой отображённой области, и оставляет остальное для обработчика ошибки страницы. Обработчик ошибки страницы выполняет необходимые действия, используя обстоятельства ошибки и флаги в структурах данных виртуальной памяти. Например, он мог бы выделить новые страницы, если находит такую область действительной и отсутствуют записи о страницах. Или он мог бы выбрать расширение области, если эта область помечена как расширяемая (например, стек).

В uClinux, без обработчика ошибки страницы, предоставляемые **mmap** функциональные возможности очень примитивны. Следующие две формы **mmap** не могут быть реализованы из-за отсутствия MMU. Мы кратко обсудим причины.

- **mmap(MAP_PRIVATE, PROT_WRITE)** не реализована. Этот тип отображения создаёт отображение файла с разрешённой записью в область виртуальных адресов процесса. Это означает, что страницы должны быть выделены в момент записи и удерживаются одним процессом, и этот процесс в одиночку видит изменения. Обработчик ошибки страницы заботится о выделении страниц когда и как только они записываются процессом и, таким образом, память выделяется только для тех страниц файла, которые изменены процессом.
- **mmap(MAP_SHARED, PROT_WRITE, file)** не реализована. Стандартная реализация в Linux для этого типа вызова **mmap** создаёт общие страницы памяти для разных процессов и записи на диске в эту область будут синхронизированы. Страницы отображаются, как и ранее, срабатыванием ошибки страницы памяти после первой записи на страницу и помечаются как "грязные" (изменённые). Позже ядро запишет изменённые страницы на диск и пометит их готовыми к следующей обработке ошибки страницы в случае более поздней записи.

Оба приведённых выше варианта требуют работающего обработчика ошибок страницы, доступного только при аппаратном MMU. Следовательно, в uClinux это не может быть реализовано. В uClinux реализацией в ядре **mmap** является **do_mmap_pgoff()**. **do_mmap_pgoff()** для случаев без MMU выполняет следующие действия:

1. Проверяет наличие флагов защиты и отображения для случаев, которые не реализованы.

```
  …
if ((flags & MAP_SHARED) && (prot & PROT_WRITE) && (file)) {
  printk("MAP_SHARED not supported (cannot write mappings
         to disk)\n");
  return -EINVAL;
}
```

```
if ((prot & PROT_WRITE) && (flags & MAP_PRIVATE)) {
  printk("Private writable mappings not supported\n");
  return -EINVAL;
}
  …
```

2. Если предоставлен файловый указатель и если файловые операции этого файла поддерживаются функцией **mmap**, то вызывается соответствующая файлу **mmap**.

```
  …
if (file && file->f_ops->mmap)
  file->f_ops->mmap(file, &vma);
return vma.vm_start;
  …
```

3. Если указатель на файл не предоставлен, то выделяется запрашиваемая память из распределителя памяти ядра с помощью **kmalloc**.

```
  …
  …
result = kmalloc(len, GFP_KERNEL);
  …
  …
```

10.5 Создание процесса

Создание процесса в Linux осуществляется с помощью системного вызова **fork()**. **fork()** создаёт для вызывающей стороны новый дочерний процесс. Как только **fork** возвращается, порождающий и порождённый становятся двумя независимыми субъектами, имеющими разные PID-ы. Теоретически, то, что необходимо сделать **fork()**, это создать точную копию всех структур данных родительского процесса, включая страницы памяти родительского процесса, доступные только ему. В Linux это дублирование страниц памяти родительского процесса является отложенным. Вместо этого порождающий и дочерний процессы совместно используют одни и те же страницы в памяти, пока один из них не попытается изменить общие страницы. Такой подход называется COW (Copy on Write, копирование при записи). Теперь давайте посмотрим, как **fork()** достигает этого. Мы обсудим реализацию **fork** в Linux 2.6. В Linux 2.4 API называются по другому, но функциональность осталась прежней.

1. Создаётся новая структура задачи процесса для дочернего процесса.

```
p = dup_task_struct(current);
```

   Это создаст новую структуру задачи и скопирует некоторые указатели из **current**.

2. Получается PID для дочернего процесса.

```
p->pid = alloc_pidmap();
```

3. Из родительского в дочерний процесс копируются файловые дескрипторы, обработчики сигналов, политики планировщика и так далее.

```
/* копируем всю информацию процесса */
  …
  …
// Копируем файловые дескрипторы
if ((retval = copy_files(clone_flags, p)))
  goto bad_fork_cleanup_semundo;
if ((retval = copy_fs(clone_flags, p)))
  goto bad_fork_cleanup_files;

// Копируем обработчики сигналов
if ((retval = copy_sighand(clone_flags, p)))
  goto bad_fork_cleanup_fs;

// Копируем информацию о сигналах
if ((retval = copy_signal(clone_flags, p)))
  goto bad_fork_cleanup_sighand;

// Копируем страницы памяти
if ((retval = copy_mm(clone_flags, p)))
  goto bad_fork_cleanup_signal;
  …
  …
```

4. Порождённый процесс добавляется в очередь планировщика задач и выполняется возврат.

```
  …
  …
/* Выполняем относящуюся к планировщику настройку */
sched_fork(p);
  …
  …

if (!(clone_flags & CLONE_STOPPED))
  wake_up_new_task(p, clone_flags);
else
  p->state = TASK_STOPPED;
  …
  …
```

   Это изменит состояние процесса на **TASK_RUNNING** и включит данный процесс в список процесс работоспособных процессов, содержащийся в планировщике задач.

Как только **fork** возвращается, дочерний процесс становится работоспособным процессом и будет запланирован в соответствии с политикой планирования родительского процесса. Обратите внимание, что в **fork** копируются только структуры данных родительского процесса. Его сегменты текста, данных и стека не копируются. **fork** пометила такие страницы как COW для последующего выделения памяти по требованию.

На шаге 3 функция **copy_mm()** по существу помечает страницы родителя как общие только читаемые между родителем и потомком. Атрибут "только чтение" гарантирует, что содержимое памяти не может быть изменено, пока оно является общим. Всякий раз, когда любой из двух процессов пытается записать в эту страницу, обработчик ошибки страницы

определяет COW страницу с помощью специальной проверки дескрипторов страницы. Страница, соответствующая ошибке, дублируется и помечается как доступная для записи в процессе, который пытался записать. Исходная страница остаётся защищённой от записи, пока другой процесс не попытается выполнить запись, в течение которой эта страница помечается доступной для записи только после проверки, что она уже не используется каким-то другим процессом.

Как показано выше, процесс дублирования в Linux, осуществляемый через COW, реализован с помощью обработчика ошибки страницы и, следовательно, uClinux не поддерживает **fork()**. Также для родителя и потомка невозможно иметь аналогичное виртуальное адресное пространство, как это ожидается от **fork**. Вместо использования для создания дочернего процесса **fork()**, разработчики uClinux предлагают использование **vfork()** вместе с **exec()**. Системный вызов **vfork()** создаёт дочерний процесс и блокирует выполнение родительского процесса, пока потомок не завершит работу или не выполнит новую программу. Это гарантирует, что родительскому и дочернему процессам не требуется иметь общий доступ к страницам памяти.

10.6 Совместно используемые библиотеки

Динамический компоновщик, который заботится о загрузке совместно используемой библиотеки, в значительной степени базируется на MMU и виртуальной адресации. Совместно используемая библиотека загружается в ОЗУ, когда приложение использует её в первый раз. Другие программы, использующие ту же библиотеку, которые запускаются позже (но до завершения первого приложения) получают местоположение текста, отображённое в их виртуальное адресное пространство. Иными словами, в физической памяти присутствует только одна копия текста библиотеки. Все последующие ссылки представляют собой только виртуальные записи, которые указывают на эту единственную физическую копию. Также отметим, что общим является только текст; для данных по-прежнему должна быть выделена память для каждого процесса. Общие страницы освобождаются, когда завершает работу последнее приложение, использующее библиотеку.

Без MMU не представляется возможным отображать одну и ту же физическую память на отдельное адресное пространство процесса. Поэтому для реализации общих библиотек uClinux использует другой метод.

10.6.1 Реализация совместно используемых библиотек в uClinux (libN.so)

Как мы уже видели, uClinux использует для исполняемых файлов формат bFLT. Загрузчик ядра uClinux выполняет работу по загрузке приложения в доступное место в памяти, настраивая необходимые разделы данных и кода. Напомним также, что если загружаемый файл bFLT имеет возможность выполнения на месте (XIP), то загрузчик ядра будет использовать один и тот же текстовый сегмент для несколько экземпляров приложения, а также создаст отдельные разделы данных для каждого экземпляра. Эта поддержка XIP используется в качестве базы для реализации в uClinux общих библиотек. Если текст не выполняем на месте, то общие библиотеки не могут работать. Это происходит потому, что общие текстовые страницы не могут существовать в памяти без поддержки MMU. Поэтому идея состоит в том, чтобы хранить их в общедоступном месте (флеш-памяти), и чтобы все программы обращались непосредственно к одному месту.

Для uClinux существует несколько реализаций различных типов совместно используемых библиотек. Мы обсудим реализацию динамических библиотек для m68k, которая использует файлы bFLT, известный в народе как метод libN.so. В этом методе общие библиотеки представляют собой бинарными файлы bFLT, но такие, которые содержат внутри

определённый идентификатор библиотеки (ID). Это требует изменений в компиляторе, а также загрузчике. Каждый символ, на который есть ссылка в исполняемом модуле, имеет добавленный к нему определённый идентификатор библиотеки. Когда загрузчик должен обработать символ, он ищет ссылку с помощью ID, содержащегося в символе, и таким образом определяет необходимую библиотеку. Для упрощения поиска имена совместно используемых библиотек следуют определённому шаблону. Библиотека, которая имеет **ID=X**, имеет имя **libX.so**. Так что когда загрузчик должен обработать символ с **ID=X** в нём, он просто загружает файл **/lib/libX.so**.

Компилятор создаёт отдельный сегмент GOT и данных для приложения и для каждой библиотеки. Когда программа загружается, он обеспечивает, чтобы отдельные сегменты данных (в разделяемых библиотеках) были доступными с фиксированными смещениями от базового адреса. Уникальный идентификационный номер, выделенный для каждой библиотеки, используется для определения смещения, чтобы найти определённый сегмент данных. Приложение также должно использовать один и тот же метод обращений с использованием идентификатора и значение идентификатора 0 зарезервировано для приложения.

Поддержку совместно используемой библиотеки в ядре осуществляет компонент, выбираемый с помощью флага **CONFIG_BINFMT_SHARED_FLAT**. Основной структурой для поддержки разделяемой библиотеки в uClinux является показанная ниже структура **lib_info**.

```
struct lib_info {
  struct {
    unsigned long start_code; /* Начало текстового сегмента */
    unsigned long start_data; /* Начало сегмента данных */
    unsigned long start_brk;  /* Конец сегмента данных */
    unsigned long text_len;   /* Длина текстового сегмента */
    unsigned long entry;      /* Начальный адрес для этого
                                 модуля */
    unsigned long build_date; /* Когда он был скомпилирован */
    short loaded;             /* Загружена ли эта библиотека? */
  } lib_list[MAX_SHARED_LIBS];
};
```

Макрос **MAX_SHARED_LIBS** определяется как 4, если **CONFIG_BINFMT_SHARED_FLAT** установлен, либо по умолчанию используется значение 1. Эта структура используется для хранения списка библиотек, загруженных для каждого приложения. Хотя теоретический предел максимального числа библиотек составляет 255 - 1, ядро определяет его как 4, что позволяет использовать 3 разделяемые библиотеки для каждого приложения. Значение ID равное 0 используется как относящееся к загружаемому приложению. Распознавание символа приложения, использующего разделяемую библиотеку, происходит внутри функции **calc_reloc**, показанной в Распечатке 10.5 352.

Обратите внимание на извлечение значения **id**, в которое будет переведено **r**, использующее **id = (r >> 24) & 0xff**. Изменения в компоновщике позволяют размещать значение **id** в старшем байте адреса символа. Например, если символ **foo** не определён в приложении, но определён в библиотеке с **ID=3**, то такой символ в данном приложении будет иметь запись форме **0x03XX_XXXX** = (**0x0300_0000** | адрес **foo()** в **lib3.so**). Таким образом, все внешние по отношению к приложению символы в своём старшем байте будут иметь **id**, соответствующий библиотеке.

Фактическая загрузка библиотеки в программную память происходит внутри функции **load_flat_shared_library**. Эта функция загружает файл **/lib/libX.so** используя обычную

функции загрузчика файла типа "flat" **load_flat_file**, передавая соответствующее значение **id=X**. (* Обратите внимание, что во время выполнения двоичных файлов типа "flat", exec() использует эту же функцию с id=0.) Загрузчик двоичного файла "flat" поддерживает массив **lib_info.lib_list[]**, чтобы отслеживать все загружаемые файлы, включая приложение, которое загружается по адресу 0. Загрузчик обеспечивает, чтобы были доступны все неопознанные/внешние символы, необходимые для исполнения.

Вышеописанная реализация является одной наиболее широко используемой для процессоров на основе m68k и была предложена Полом Дейлом. Для uClinux доступны и другие реализации совместно используемой библиотеки. XFLAT от Cadenux представляет собой другую реализацию совместно используемой библиотеки, широко используемую на процессорах ARM. FRV-uClinux также имеет реализацию разделяемой библиотеки. Каждая реализация было сделано с разными целями, решает разные проблемы.

Распечатка 10.5 Распознавание символов совместно используемой библиотеки

Распечатка 10.5

```
static unsigned long
calc_reloc(unsigned long r, struct lib_info *p, int curid,
           int internalp)
{
  …
  …
#ifdef CONFIG_BINFMT_SHARED_FLAT
  if (r == 0)
    id = curid;   /* 0 - всегда ссылка на себя */
  else {
    id = (r >> 24) & 0xff; /* Получаем ID для перемещения */
    r &= 0x00ffffff;       /* Приводим ID в порядок */
  }
  if (id >= MAX_SHARED_LIBS) {
    printk("BINFMT_FLAT: reference 0x%x to shared library %d",
           (unsigned) r, id);
    goto failed;
  }
  if (curid != id) {
    …
    …
  }else if ( ! p->lib_list[id].loaded &&
             load_flat_shared_library(id, p) > (unsigned long) -4096) {
    printk("BINFMT_FLAT: failed to load library %d", id);
    goto failed;
  }
  …
  …
#else
  id = 0;
#endif
  …
  …
}
```

10.7 Перенос приложений на uClinux

В этом разделе мы рассмотрим шаги, необходимые для создания программ для uClinux и совместно используемых библиотек, и что должно быть учтено до переноса приложений из стандартного Linux в uClinux.

10.7.1 Созданий программ для uClinux

Исполняемым файлом в uClinux является двоичный файл формата "flat". Обычный файл ELF не поддерживается на uClinux. Набор инструментов для uClinux предоставляет специальную программу для преобразования файла ELF в файл bFLT. Не все файлы ELF могут быть преобразованы в BFLT. Для этого сгенерированный код должен быть позиционно независимым. uClinux имеет два варианта позиционно-независимых двоичных файлов: полностью перемещаемые двоичные файлы и двоичные файлы PIC. Список команд компилятора для создания разных форм файла bFLT, использующих набор инструментов m68k, приведён ниже.

Создание полностью перемещаемых двоичных файлов

- Компилируем файл. Это создаст **sample.o**.

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -c -o sample.o sample.c
```

- Компонуем и создаём файл "flt". Этот шаг создаст исполняемый файл **sample** и файл символов **sample.gdb**.

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -Wl,-elf2flt -Wl,-move-rodata -nostartfiles
/home/sriramn/work/uclinux/uClinux-dist/lib/crt0.o
-L/home/sriramn/work/uclinux/uClinux-dist/lib o sample
sample.o -lc
```

Создание двоичных файлов PIC

- Компилируем файл.

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -msep-data -c -o sample.o sample.c
```

- Компонуем и создаём файл "flt".

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -msep-data -Wl,-elf2flt -Wl,-move-rodata
-nostartfiles
/home/sriramn/work/uclinux/uClinux-dist/lib/crt0.o
-L/home/sriramn/work/uclinux/uClinux-dist/lib -o testhw1
sample.o -lc
```

- Обратите внимание, что **-msep-data** заставляет выполнить **-fPIC** внутренне. А **-elf2flt**, передаваемая компоновщику, заставляет выполнить преобразование из формате ELF в bFLT. Кроме того, **-msep-data** включает XIP.

- Чтобы изменить размер стека файла bFLT, используем команду

```
elf2flt –s <stack _size> test.flt
```

- Чтобы сжать файл (всё кроме заголовков), используем

```
elft2flt –z –o test.flt test.elf
```

Сжатые образы не являются доступными для исполнения на месте, так как перед исполнением они должны быть распакованы в ОЗУ.

10.7.2 Создание совместно используемых библиотек для uClinux

Совместно используемые библиотеки в uClinux представляют собой обычные файлы bFLT, созданные с помощью специальных флагов компилятора. Флаги компилятора помогают определить символьные ссылки с помощью фиксированного числа ID библиотеки. Шаги, необходимые для создания и использования разделяемых библиотек, перечислены ниже. В этом примере, чтобы создать общую библиотеку **libtest**, мы используем **a.c** и **b.c**.

Файл: **a.c**

```
void a()
{
  printf("I am a\n");
}
```

Файл: **b.c**

```
void b()
{
  printf("I am b\n");
}
```

- Компилируем каждый файл. Обратите внимание, что используется флаг **-mid-**

sharedlibrary.

```
m68k-elf-gcc -Wall -Wstrict-prototypes -Wno-trigraphs
-fno-strict-aliasing -Os -g -fomit-frame-pointer
-m68000 -fno-common -Wall -fno-builtin -DEMBED
-mid-shared-library -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-Dlinux -D__linux__ -D__uClinux__ -Dunix -msoft-float
-fno-builtin a.c -c -o a.o
m68k-elf-gcc -Wall -Wstrict-prototypes -Wno-trigraphs
-fno-strict-aliasing -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -fno-builtin -DEMBED -mid-shared-library -
nostdinc -I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-Dlinux -D__linux__ -D__uClinux__ -Dunix -msoft-float
-fno-builtin b.c -c -o b.o
```

- Создаём архив.

```
m68k-elf-ar r libtest.a a.o b.o
m68k-elf-ranlib libtest.a
```

- Создаём двоичный файл типа "flat" с необходимым идентификатором библиотеки (здесь мы используем 2, **libc** имеет id =1). Обратите внимание на добавление файла **uClibc/lib/main.o** для пустой функции **main** и опцию **-shared-lib-id=2**.

```
m68k-elf-gcc -nostartfiles -o libtest -Os -g
-fomit-frame-pointer -m68000 -fno-common -Wall
-fno-builtin -DEMBED -mid-shared-library -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-Dlinux -D__linux__ -D__uClinux__ -Dunix -Wl,-elf2flt
-nostdlib -Wl,-shared-lib-id,2
/home/sriramn/work/uclinux/uClinux-dist/uClibc/lib/main.o
-Wl,-R,/home/sriramn/work/uclinux/uClinux-dist/lib/libc.gdb
-lc -lgcc -Wl,--whole-archive,libtest.a,--no-whole-archive
```

- Удаляем символы запуска. Так как должна быть возможность использовать библиотеку с другим приложением, символы запуска, добавляемые через среду исполнения языка Си, такие как **_main**, **_start** и тому подобные, должны быть удалены. Это делается с помощью следующей команды.

```
m68k-elf-objcopy -L _GLOBAL_OFFSET_Table_ -L main -L __main
-L _start -L __uClibc_main -L __uClibc_start_main
-L lib_main -L _exit_dummy_ref
-L __do_global_dtors -L __do_global_ctors
-L __CTOR_LIST__ -L __DTOR_LIST__
-L _current_shared_library_a5_offset_
libtest.gdb
```

- Устанавливаем эту библиотеку в **rootfs** под соответствующим именем.

```
cp libtest.gdb romfs/lib/lib2.so
```

Чтобы проанализировать созданные символы, выполним для этого символьного файла **nm**.

```
#nm libtest.gdb | sort
  0100001c A __assert
  01000098 A isalnum
  010000b8 A isalpha
  010000d8 A isascii
    …
    …
  010355c4 A __ti19__pointer_type_info
  010355d0 A __ti16__ptmd_type_info
  010355dc A __ti19__builtin_type_info
  020000cc T a
  020000e4 T b
  02000100 D __data_start
  02000100 D data_start
```

Обратите внимание на наличие символов **01xxxxxx** из **libc** (**lib1.so**) и символы нашей библиотеки **lib2.so**, начинающиеся с **02xxxxxx**.

10.7.3 Использование совместно используемой библиотеки в приложении

Теперь рассмотрим, как использовать созданную библиотеку в приложении. Компоновщик должен знать о внешних ссылках в приложении, чтобы он мог в сгенерированном файле bFLT пометить их как ссылки на разделяемые библиотеки. Для компиляции программы мы должны сделать два следующих шага.

Файл: **use.c**

```
extern void a();
extern void b();

main()
{
  a();
  b();
}
```

- Компилируем **use.c** (обратите внимание на **shared-library-id=0**).

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -mid-shared-library -mshared-library-id=0 -c
-o use.o use.c
```

- Связываем (компонуем) **use.c** с **libc** и **libtest**.

```
m68k-elf-gcc -m68000 -Os -g -fomit-frame-pointer -m68000
-fno-common -Wall -Dlinux -D__linux__ -Dunix
-D__uClinux__ -DEMBED -nostdinc
-I/home/sriramn/work/uclinux/uClinux-dist/include
-I/home/sriramn/work/uclinux/uClinux-dist/include/include
-fno-builtin -mid-shared-library -mshared-library-id=0
-Wl,-elf2flt -Wl,-move-rodata -Wl,-shared-lib-id,0
-nostartfiles
/home/sriramn/work/uclinux/uClinux-dist/lib/crt0.o
-L/home/sriramn/work/uclinux/uClinux-dist/lib -L. -o use
use.o -Wl,-R,
/home/sriramn/work/uclinux/uClinux-dist/lib/libc.gdb -lc
-Wl,-R,libtest.gdb -ltest
```

И снова для **use.gdb** выполняем **nm**, чтобы увидеть наличие библиотеки.

```
#nm use.gdb | sort
  00000004 T _stext
  00000008 T _start
  00000014 T __exit
  0000001a t empty_func
  0000001c T main
    …
    …
  00000260 B end
  00000260 B _end
  0100001c A __assert
  01000098 A isalnum
  010000b8 A isalpha
  010000d8 A isascii
  010000ec A iscntrl
    …
    …
  010355d0 A __ti16__ptmd_type_info
  010355dc A __ti19__builtin_type_info
  020000cc A a
  020000e4 A b
```

После понимания шагов, необходимых для создания программ для uClinux, перед переносом приложений из стандартного Linux для uClinux, вы также должны понять различные ограничения uClinux.

10.7.4 Ограничения на память

uClinux не обеспечивает динамический стек. Исполняемые приложения имеют заранее заданный размер стека, установленный во время компиляции с помощью **elf2flt**. Программисты должны избежать больших выделений памяти на стеке. Вместо этого используйте "кучу" или, если требование не является динамическим, переместите его в раздел bss.

Программы C++ используют **malloc** даже для деклараций встроенных типов данных через оператор **new**. Многие приложения, написанные на C++, имеют проблемы с запуском

на uClinux. Отсутствие MMU и интеллектуального **malloc** приводит к неразрешимой проблеме фрагментации памяти, что делает систему бесполезной. Следовательно, C++ не рекомендуется на системе без MMU. Изменяйте любое приложение, которое должно выделять малые куски памяти через **malloc** или, если возможно, пишите зависимые от приложения методы распределения памяти, которые будут сами управлять заранее выделенной областью памяти.

10.7.5 Ограничения mmap

Функция **mmap()** в uClinux очень примитивна в своей функциональности и программы, которые зависят от поведения **mmap()**, могут завершиться ошибкой. Таким образом, мы приведем здесь список вызовов **mmap()**, которые не будут работать, и те, которые работают с ограничениями, если таковые имеются.

- Общедоступное отображение с разрешённой записью в uClinux невозможно.

```
mmap(MAP_SHARED, PROT_WRITE, file)
```

- Любое защищённое отображение с разрешённой записью на uClinux не поддерживается.

```
mmap(MAP_PRIVATE, PROT_WRITE, файл или не файл)
```

- Незащищённое общее отображение для не файлового дескриптора возвращает требуемый размер. Оно похоже на таковое на системах с MMU, но память представляет собой прямой адрес ядра. **mmap(MAP_SHARED, 0, nofile, size)** выделяет объём памяти через распределителя ядра.

10.7.6 Ограничения на уровне процесса

Не существует реализованного в uClinux вызова **fork()**. Поэтому, чтобы сделать программы (которые требуют **fork()**) работающими, необходимо в дополнение к вызову **fork()** использовать вызов **vfork()** с последующим **exec()**. Дочерний процесс не должен изменять данные родительского процесса, пока не вызвана **exec()**. Распечатка 10.6 358 показывает обычную программу Linux, использующую **fork()**, и как она может быть перенесена на uClinux.

Распечатка 10.6 Перенос приложений на uClinux

Распечатка 10.6

```
/* Пример программы для обычного Linux */

/* fork.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

int main(int argc, char *argv[])
```

```
{
  pid_t pid;
  /*
   * Если fork возвращает -1, это означает, что вызов fork
   * закончился неудачей при создании дочернего процесса.
   * В родительский процесс возвращается pid дочернего процесса.
   * В дочернем процессе возвращается 0.
   */
  if ((pid = fork()) < 0) {
    printf("Fork() failed\n");
    exit(1);
  }else if (pid != 0) {
    /* Код родителя */
    printf("Parent exiting.\n");
    exit(0);
  }
  /* Код потомка */
  printf("Starting child...\n");
  while (1) {
    sleep(2);
    printf("...Child running\n");
  }
}
```

```
/* Показанное выше приложение может быть перенесено на uClinux */

/* vfork.c */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>

int main(int argc, char *argv[])
{
  pid_t pid;
  int c_argc = 0;
  char *c_argv[3];
  char child=0;

  /* Определяя потомка, мы используем системный аргумент argv[1]=child */
  if (argc > 2 && !strcmp(argv[1],"child")) child=1;

  /* используем vfork(), возвращаемые значения аналогичны fork() */
  if (!child) {
    if ((pid = vfork()) < 0) {
      printf("vfork() failed\n");
      exit(1);
    } else if (pid != 0) {
      /* Код родителя */
      printf("Parent exiting.\n");
      exit(0);
    }
```

```
    /*
     * Здесь вызывается exec потомка. Для идентификации потомка
     * мы передаём специальный аргумент
     */
    c_argv[c_argc++] = argv[0];
    c_argv[c_argc++] = "child";
    c_argv[c_argc++] = NULL;
    execv(c_argv[0], c_argv);

    /* Заметим, что если всё успешно, execv никогда не вернётся */
    printf("execv() failed\n");
    exit(1);

  }else { // Код потомка
    printf("Starting child...\n");
    while (1) {
      sleep(2);
      printf("...Child running\n");
    }
  }
}
```

10.8 XIP — eXecute In Place, выполнение на месте

В стандартном Linux программы обычно загружаются и выполняются в системной памяти. Загрузчик загружает разделы текста с носителя (дискового или флеш-памяти) в эту память. Другие страницы получаются по запросу страниц, если это требуется, с помощью обработчика ошибки страницы. В случае uClinux, так как обработчик ошибки страницы невозможен, весь текстовый раздел должен быть сразу считан в оперативную память загрузчиком. Загрузчик файлов типа "flat" выделяет память размером с текст вместе со стеком, данными и таблицей переадресации.

В системе с небольшим количеством памяти uClinux предоставляет альтернативу, XIP. С XIP становится возможным выполнять код с накопителя без необходимости загружать его в ОЗУ. Загрузчик использует непосредственно указатель на память накопителя и это позволяет сэкономить на выделении памяти, которое в противном случае пришлось бы делать для текстового раздела. Обратите внимание, что для исполнения память для данных и стека всё же должна быть выделена. XIP имеет некоторые ограничения или требования, которые должны учитываться при разработке. Перечислим их здесь.

10.8.1 Аппаратные требования

- ***Поддержка для PIC в процессоре***: XIP возможно, только если процессор имеет поддержку для создания позиционно-независимого кода. Он должен быть в состоянии выполнять адресацию относительно счётчика команд. Это необходимо, чтобы избежать в тексте (жёстких) ссылок на адреса. В отсутствие PIC сгенерированный код будет иметь адресные смещения от нуля и загрузчику придётся модифицировать адреса, основываясь на адресе загрузки и, следовательно, потребуется загрузка текста в ОЗУ, нарушая цель использования XIP.

- ***Только NOR Flash***: Есть два типа устройств флеш-памяти, NOR и NAND, оба широко используются во встраиваемых системах. NOR Flash позволяет обращаться ко всем

секторам случайным образом и читается так же, как SRAM. NAND, с другой стороны, для чтения из памяти требует программирования некоторых управляющих регистров. Обычно, чтобы прочитать содержимое NAND Flash требуется драйвер флеш-памяти. Когда программа выполняется с флэш-памяти, это значит, что указатель команд или программный счетчик просто увеличивается для извлечения следующей инструкции, указывая на следующее слово во флэш-памяти. Невозможно для извлечения следующей инструкции запускать код драйвера. Следовательно, для XIP подходит только NOR Flash.

10.8.2 Программные требования

- ***Поддержка файловой системы***: ROMFS (нельзя использовать с CRAMFS или JFFS2). Файловая система, используемая в сочетании с XIP, должна быть простой и не может быть сжатой. Если файлы сжаты, они должны быть распакованы в страницы, находящиеся в оперативной памяти. Следовательно, это исключает любую файловую систему, которая пытается использовать сжатие. Хороший подход - это поместить все исполняемые XIP файлы в раздел ROMFS и иметь разделы CRAMFS, JFFS2 для других сжатых файлов на основе требований к системе.

10.9 Сборка дистрибутива uClinux

В этом разделе мы обсудим, как собрать дистрибутив uClinux. Процедура сборки uClinux довольно проста. Система сборки хорошо интегрирована и контролируется системой GNU **make**. Высокоуровневая сборка объединяет следующее.

- Выбор платформы/поставщика
- Выбор версии ядра и сборку
- Выбор библиотеки языка Си и сборку
- Сборку библиотек поддержки (libmath, libz и так далее)
- Выбор пользовательских приложений (Busybox, Tinylogin и так далее)
- Сборку корневой файловой системы
- Создание окончательного образа ROM/flash для целевой платформы

Мы пройдём через различные этапы и меню конфигураций. Загрузите последнюю версию tar архива с http://www.uclinux.org. Распакуйте дистрибутив в какой-нибудь каталог.

```
#tar jxvf uClinux-dist-20041215.tar.bz2
```

Это распакует файлы дистрибутива в подкаталог **uClinux-dist**. Шаги для построения следующие:

1. ***Сборка конфигурации***: перейдите в каталог дистрибутива и выполните

   ```
   make config (or)
   make menuconfig (or)
   make xconfig
   ```

   Это запрашивает высокоуровневое мню. Рисунок 10.7 показывает меню верхнего уровня для **make xconfig**.

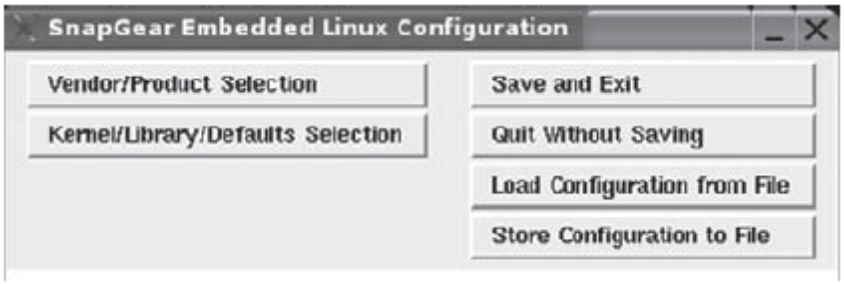

Рисунок 10.7 make xconfig.

2. ***Выбор платформы***: меню **Vendor/Product Selection** (Выбор поставщика/продукта) содержит список всех платформ, для которых доступен uClinux. Выберите подходящего поставщика и продукт из списка доступных. Рисунок 10.8 показывает меню **Vendor/Product Selection** для **make xconfig**.

Рисунок 10.8 Выбор поставщика/продукта.

3. ***Выбор ядра/библиотеки***: меню **Kernel/Library Selection** показано на Рисунке 10.9. Дистрибутив uClinux поддерживает три версии ядра Linux: 2.0, 2.4 и 2.6. Выберите подходящее ядра на основе необходимых функций и требований проекта. uClinux предоставляет три варианта библиотеки языка Си: **glibc**, **uClibc** и **uC-libc**. **uC-libc** - это старая версия **uClibc**. **uClibc** написана для встраиваемых систем и, следовательно, является наиболее подходящей. **uClibc** будет достаточно для большинства систем. Установите **Customize Kernel Settings** (Настройка параметров ядра) в **y** (да), если вам необходимо изменить настройки ядра. Если вам необходимо

изменить приложения пространства пользователя и библиотеки поддержки, установите **Customize Vendor/User Settings** (Настройка параметров поставщика/пользователя) в **y** (да).

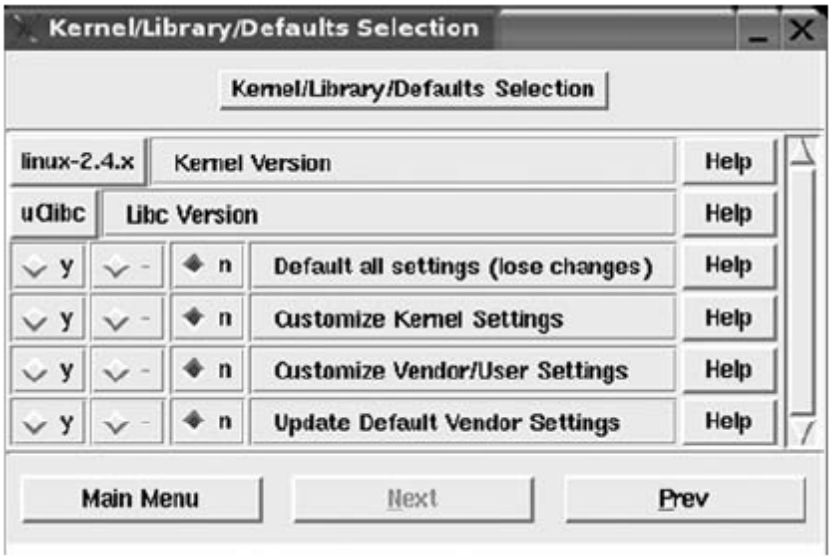

Рисунок 10.9 Выбор ядра/библиотеки.

4. ***Конфигурация пользовательского пространства***: установка **Customize Vendor/User Settings** в **y** предоставляет меню конфигурации пользовательского пространства. Это меню предоставляет выбор приложений, которые необходимы на целевой платформе. Также могут быть выбраны необходимые для сборки библиотеки. Меню показано на Рисунке 10.10.

Рисунок 10.10 Выбор приложения.

5. ***Зависимости и сборка***:

```
# make dep
# make
```

6. Это завершает процедуру сборки. Процедура сборки соберёт все файлы вместе в окончательный образ ROM/flash в каталоге **images**.

Приложение A, Ускорение запуска

В большинстве встраиваемых систем быстрая загрузка является важным системным требованием. Поскольку Linux завоёвывает прочные позиции на рынке встраиваемых систем, как можно более быстрая загрузка для основанной на Linux встроенной системы становится всё более важной. Тем не менее, уменьшение времени загрузки для встроенной системы Linux не является простой задачей по следующим причинам:

- Если мы определим включённое состояние как состояние, в котором основные службы (в зависимости от функциональных возможностей системы) доступны, то разные встраиваемые системы имеют разные требования для включённого состояния. Проанализируем маршрутизатор и портативное устройство для определения включённого состояния. Для маршрутизатора включённое состояние становится полезным, когда маршрутизатор сконфигурировал все сетевые интерфейсы, запустил все необходимые сетевые протоколы и настроил различные таблицы маршрутизации. Однако, для портативного устройства такое состояние является полезным, когда для пользователя доступны оконная система и устройства ввода/вывода. В отличие от маршрутизатора, для портативного устройства инициализация подключённого к сети стека не должна быть критерием достижения включённого состояния; наоборот, сетевой стек можно проинициализировать в более поздний момент времени. Везде в этом разделе время загрузки означает время, затраченное системой на достижение включённого состояния после подачи питания.
- Linux развивался начиная с рынка настольных компьютеров и серверов, для которого время загрузки не является важным требованием. На настольных компьютерах и серверах загрузка Linux занимает несколько минут; это совершенно неприемлемо для встраиваемых систем.
- С увеличением вычислительной мощности встраиваемых систем размер программного обеспечения встраиваемых систем также многократно увеличился. Больший набор программного обеспечения просто означает большее время загрузки.

Следовательно, уменьшение времени загрузки является индивидуальным процессом для каждой встроенной системы. В этом разделе описываются некоторые общие методы для сокращения времени загрузки. Однако, каждый из этих методов обычно является компромиссом между затратами памяти или скоростью; это также обсуждается. Время загрузки систем на основе Linux можно разделить на три этапа, как показано в Таблице A.1.

Таблица A.1 Этапы загрузки Linux

Состояние	*Описание*	*Что отнимает много времени*
Загрузчик	Загрузчик выполняет POST, запускает экран для взаимодействия с пользователем м загружает в память ядро Linux для инициализации.	POST, обнаружение ядра, копирование его в память, распаковка ядра.
Включение ядра	Ядро должно проинициализировать оборудование, настроить разные подсистемы и драйверы,	Инициализация драйверов, монтирование файловой системы.

	смонтировать корневую файловую подсистему и передать управление в пространство пользователя.	
Включение пользовательского пространства	Запуск разнообразных служб системы.	Службы, стартующие последовательно, службы, начальный запуск которых может быть выполнен позже, загрузка модулей ядра.

Методы сокращения времени инициализации загрузчика

- ***POST***: Power-On Self Test (самотестирование при включении питания) может выполняться только во время выполнения холодной загрузки; во время теплого старта его можно пропустить и, следовательно, это может привести к уменьшению времени загрузки.
- ***Обнаружение, распаковка и загрузка памяти, содержащей ядро***: этот шаг может быть одним из наиболее длительных операций, так как это может занять от 500 мс до 1 с в зависимости от размера ядра. Чем больше размер ядра, тем будет требоваться больше времени на распаковку и копирование его в память. Если ядро является частью файловой системы Linux, хранящийся во флэш-памяти, то поиск ядра и анализ заголовков файлов (например, заголовков ELF) может занять много времени. Хранение образа ядра в разделе без файловой системы поможет обойти это. Чтобы избежать распаковку, файл может быть сохранён в несжатом формате, но это было бы за счёт дорогого пространства флэш-памяти. Чтобы избежать копирования ядра в память, ядро может быть подготовлено для XIP. eXecute In place (выполнение на месте) является методом, посредством которого ядро выполняется непосредственно из flash-памяти. XIP подробно обсуждается в Главе 10 322. Помимо сокращения времени запуска, другое достоинство XIP в том, что это бережёт память, потому что текстовый раздел ядра не копируется в память. Однако, недостатками использования XIP являются:
  - XIP замедляет время выполнения ядра, потому что он выполняется из flash-памяти.
  - Поскольку выполняемый на месте образ не может быть сжат, это будет означать, что для хранения несжатого образа потребуется больше flash-памяти.
  - Использование выполняемого на месте ядра потребует изменений в коде драйвера flash-памяти, поскольку такие операции, как стирание flash-памяти и запись, не могут быть выполнены, когда ядро выполняется из flash. Такими изменениями в драйвере flash-памяти обычно будут копирование части ядра в память и выполнение его с отключенными прерываниями. Однако, этого не требуется, если ядро выполняется на месте из flash-памяти, которая имеет файловую систему только для чтения, такую как с CRAMFS.
- Но если XIP является слишком дорогой операцией, то крайне важно, чтобы образ ядра был бы как можно меньше, чтобы избежать увеличения времени копирования. Чтобы сократить время копирования, можно использовать некоторые дополнительные приемы, такие как использование DMA для передачи образа с флэш-памяти.

Настройка ядра для уменьшения времени запуска

- ***Отключение вывода сообщений в ядре***: если печать ядра направлена в последовательную консоль (которая является медленным устройством), то вывод может привести к большой задержке во время инициализации ядра. Чтобы избежать этого, вывод сообщений ядра можно отключить с помощью опции командной строки ядра **quiet**. Однако, эти сообщения можно просмотреть позже командой **dmesg**.
- ***Жёсткое кодирование в ядре значения* loops_per_jiffies**: как уже говорилось в Главе 2 1, инициализация ядра включает в себя вызов функции **calibrate_delay()** для вычисления значения **loops_per_jiffies**. В зависимости от архитектуры процессора это может занять до 500 мс. Если частота процессора может быть известна в момент компиляции ядра, то это значение может быть жёстко прописано внутри ядра, либо может быть добавлена опция командной строки ядра, чтобы передать значение этой переменной во время загрузки ядра.
- ***Сокращение времени инициализации драйвера***: разные драйверы имеют разное время инициализации. Это может быть вызвано циклами в драйвере. Вызвать увеличение время загрузки может также зондирования аппаратных устройств на некоторых шинах, таких как PCI. Тонкая настройка времени инициализации в таких случаях означала бы изменения в драйверах, такие как предустановка в них значений, уже известных на момент сборки ядра.
- ***Использование подходящей корневой файловой системы***: разные файловые системы имеют разное время инициализации. Время загрузки для журналируемых файловых систем, таких как JFFS2, чрезвычайно большое, поскольку они сканируют для поиска записей во время инициализации всю flash-память. Файловые системы только для чтения, такие как с CRAMFS и ROMFS, имеют более короткое время инициализации.

Настройка пользовательского пространства для уменьшения времени запуска

- ***Загрузка модулей***: чем больше количество модулей ядра, тем больше времени уходит для загрузки этих модулей. В случае, если модулей ядра много, объединение их в единый модуль может эффективно уменьшить время загрузки модулей.
- ***Одновременный запуск служб***: как говорилось в Главе 2 1, для включения системных служб используются системные RC скрипты, которые запускаются последовательно. Параллельным запуском служб может быть достигнуто значительное улучшение. Однако, если это будет делаться, следует позаботиться о зависимости между службами.

Измерение времени загрузки

Существует много методов для измерения времени загрузки системы. В этом разделе мы обсудим измерение времени загрузки с помощью ***Instrumented printks*** (printk, снабжённый средствами измерения). Патч можно загрузить с www.celinuxforum.org. Патч добавляет поддержку отображения временных меток вместе с выводом **printk**. Примените патч к ядру и включите ***Show timing information on printks*** в разделе ***Kernel hacking***. Соберите и загрузите новое ядро. Пример вывода показан в Распечатке A.1.

Основой этого патча является функция **sched_clock**. Как уже говорилось в Главе 8 240,

высокая точность при измерении достигается, если для лучшей поддержки функции **sched_clock** обеспечивается BSP.

Распечатка A.1 Пример вывода Printk при измерении

```
[4294667.296000] Linux version 2.6.8.1 (root@amol) (gcc version
                 3.2.2 20030222 (Red Hat Linux 3.2.2-5)) #4 Wed
                 Mar 9 15:22:08 IST 2005
[4294667.296000] BIOS-provided physical RAM map:
[4294667.296000] BIOS-e820: 0000000000000000 - 000000000009fc00
                 (usable)
[4294667.296000] BIOS-e820: 000000000009fc00 - 00000000000a0000
                 (reserved)
[4294667.296000] BIOS-e820: 00000000000e6000 - 0000000000100000
                 (reserved)
[4294667.296000] BIOS-e820: 0000000000100000 - 000000000ef2fc00
                 (usable)
            …
            …
[4294671.443000] Dentry cache hash table entries: 32768 (order:
                 5, 131072 bytes)
[4294671.444000] Inode-cache hash table entries: 16384 (order: 4,
                 65536 bytes)
[4294671.570000] Memory: 237288k/244924k available (2099k kernel
                 code, 6940k reserved, 673k data, 172k init,
                 0k highmem)
[4294671.570000] Checking if this processor honors the WP bit
                 even in supervisor mode... Ok.
[4294671.570000] Calibrating delay loop... 5488.64 BogoMIPS
           …
           …
```

Приложение Б, GPL и встраиваемый Linux

Linux и большинство приложений, библиотек, драйверов и тому додобного с открытым исходным кодом распространяются под лицензией GNU GPL. В прежние времена компании сопротивлялись переходу на встраиваемый Linux из-за его лицензии GPL. Они опасались, что переход на Linux может заставить их сделать свою интеллектуальную собственность общедоступной. Со временем компании получили более глубокое представление о GPL и они поняли, что на встроенном Linux патентованное программное обеспечение может быть всегда в целости и сохранности.

Вы можете решить в своей разработке, использовать ли в продукте какие-то приложения с открытым кодом. Вы не должны считать, что все программы с открытым исходным кодом подпадает под GPL. Кроме GPL есть и другие лицензии, такие как Mozilla Public License (MPL), Малая GPL (Lesser GPL, LGPL), Apache License, BSD License, и так далее. (* Полный список есть на http://www.gnu.org/philosophy/license-list.html. GPL лицензии могут быть загружены с http://www.fsf.org/licensing.) Мы настоятельно рекомендуем вам связываться с юрисконсультом по открытому исходному коду и уточнять, не нарушает ли вы те или иные положения лицензий.

В этом приложении мы обсудим, как сохранить в целости и сохранности патентованное программное обеспечение на Linux. Сначала мы обсудим приложения пользовательского

пространства, а затем ядро.

Приложения пользовательского пространства

Линус Торвальдс сделал разъяснения по поводу программ пользовательского пространства, которые работают на Linux.

> *Это авторское право *не* покрывает пользовательские программы, которые используют службы ядра с помощью обычных системных вызовов - это просто считается нормальным использованием ядра и *не* подпадает под заголовок "производный продукт".* (* Смотрите файл COPYING в исходных текстах ядра.)

Это означает, что вы можете писать код, пользуясь свежей базой кода и используя службы Linux, и сохранить права на ваш код. Это не подпадает под GPL и вы не должны публиковать исходный код. Но вы должны убедиться, что в ваших программах пользовательского пространства вы неумышленно не используете какое-либо программное обеспечение с GPL. Следует позаботиться о следующем:

- Вы не должны использовать в приложении исходный код любой программы под лицензией GPL.
- Вы не должны компоновать своё приложение с любой библиотекой с GPL статически или динамически. Вы можете компоновать ваше приложение с библиотеками с LGPL. Большинство ключевых библиотек в Linux, таких как **libc**, **pthreads**, и так далее, распространяются под лицензией LGPL. Вы можете подключать в вашу программу библиотеки с LGPL без каких-либо обязательств по публикации исходного кода приложения.

Это позволило использовать механизмы IPC между GPL и не GPL программами. Например, вы можете загрузить DHCP сервер, распространяемый под лицензией GPL и написать собственный клиент DHCP. Вы не обязаны выпустить свой DHCP клиент под GPL. Тем не менее, какие-либо изменения, сделанные вами в любом приложении с GPL и которые используют механизмы IPC, чтобы обойти GPL, очень опасны. В таких случаях вы должны посоветоваться с юрисконсультом.

Обратите внимание, что GPL применяется только когда речь идёт о распространении программы или продукта. Вы можете использовать любые программы, драйверы и так далее с GPL любым способом, как вы хотите, пока это является внутренним пользованием, а не распространением. Например, вы можете использовать отладчики и профилировщики с открытым исходным для отладки ваших собственных программ. Вы также можете вносить изменения в них, не публикуя код, если они предназначены для внутреннего использования.

Таким образом, в Linux всегда можно сохранять права собственности на пользовательские приложения. Вам нужно всего лишь принять некоторые меры предосторожности при разработке приложений.

Ядро

Существует общее мнение, что загружаемые модули ядра, использующие стандартные интерфейсы, экспортируемые ядром, могут не раскрывать код и не подпадают под GPL. Например, при условии использования стандартных интерфейсов, экспортируемых ядром, вы можете иметь собственные драйверы устройств, реализованные в виде модулей ядра

Linux и не публиковать исходный код драйверов. Тем не менее, это одна из серых зон и вы должны проконсультироваться со своим юрисконсультом.

Распечатка Б.1 показывает отрывок из письма от Линуса Торвальдса в список рассылки ядра относительно его взгляда на загружаемые модули ядра и GPL.

Распечатка Б.1 Письмо Линуса Торвальдса относительно GPL и двоичных модулей ядра

```
From: Linus Torvalds

Subject: Re: Linux GPL and binary module exception clause?

Date: 2003-12-03 16:10:18 PST

...

И в самом деле, когда речь идёт о модулях, вопрос о GPL является точно таким
же.
Ядро _использует_ GPL. Никаких "если", "однако" и "возможно" об этом.
В результате, всё, что является производным продуктом, должно быть под
лицензией
GPL. Это очень просто.

Теперь, вопрос о "производном продукте" в авторском праве это единственная
вещь,
которая приводит к каким-то серым областям. Есть области, которые вовсе
не являются серыми: пространство пользователя, очевидно, не производный
продукт,
в то время как патчи ядра, очевидно, _являются_ производными продуктами.
```

Таким образом, предметами заботы должны быть следующие моменты:

- Обратитесь к юрисконсульту, чтобы узнать, можно ли использовать загружаемые модули ядра для защиты вашего собственного программного обеспечения.
- Любые изменения, сделанные в ядре Linux, подпадают под GPL.
- В ядре не сделано никакого экспорта неэкспортируемого интерфейса ядра для поддержки вашего загружаемого модуля.
- Любые изменения, сделанные в ядре в виде патча ядра, подпадают под GPL.

Что следует помнить

- В качестве руководителя проекта вы должны убедиться, что разработчики понимают GPL и другие лицензии, вовлечённые в проект.
- Во время разработки разработчики должны быть аккуратными при использовании для своего проекта частей программного обеспечения (в виде библиотеки или какого-либо исходного кода), доступного в Сети. Они не должны случайно нарушить какие-либо из вовлечённых лицензий. Как правильно говорят: лучше предотвратить, чем лечить.
- Важным пунктом, о котором вы должны заботиться в программном обеспечении, является нарушение патента. Возможно, что вы используете какой-то стандартный исходный код, который нарушает какой-то программный патент. Эти нарушения очень трудно уловить и вы должны быть предельно осторожны.

Но есть серая зона, в частности, что-то вроде "драйвер, который был первоначально написан для другой операционной системы" (то есть по происхождению явно не производный продукт Linux). В какой именно момент он становится производным продуктом ядра (и, следовательно, подпадает под GPL)?

ЭТО серая зона, и _это_ та область, где лично я считаю, что некоторые модули могут не считаться производным продуктом просто потому, что они не были предназначены для Linux и не зависят ни от какого особого поведения Linux.

В основном:

- Всё, что было написано имея в виду Linux (независимо от того, работает ли это _также_ на других операционных системах или нет), очевидно, частично производный продукт.
- Всё, что имеет знание и играет с фундаментальным внутренним поведением Linux, явно является производным продуктом. Если вам необходимо работать с кодом ядра, это производный продукт, в этом нет сомнений.

Исторически сложилось, что есть вещи, подобные оригинальному модулю файловой системы Эндрю (AFS): стандартной файловой системы, которая первоначально действительно не была написана для Linux, а всего лишь реализует поддержку файловой системы UNIX. Является ли она производным продуктом только потому, что перенесена на Linux, который разумно имел аналогичный интерфейс VFS для других Unix систем? Лично я не чувствую, что мог бы сделать подобный вызов в суд. Может быть, это было бы возможно, но это, безусловно, является серой зоной.

Лично я думаю, что данный случай не является производным продуктом, и я был готов сказать так парням AFS.

Означает ли это, что любой модуль ядра не является автоматически производным продуктом? НЕТ! Это не имеет ничего общего с модулями как таковыми, кроме того, совершенно ясно, что не-модули являются производными продуктами (если они так важны для ядра, что вы не можете загрузить их в качестве модуля, они явно производный продукт только в силу того, что очень тесно связаны с ядром - и потому, что GPL прямо упоминает компоновку).

Так что быть модулем не значит не быть производным продуктом. Это всего лишь один из признаков того, что _возможно_ есть другие аргументы, почему он не является производным.

Индекс