Этот файл был взят с сайта

http://all-ebooks.com

Данный файл представлен исключительно в ознакомительных целях. После ознакомления с содержанием данного файла Вам следует его незамедлительно удалить. Сохраняя данный файл вы несете ответственность в соответствии с законодательством.
Любое коммерческое и иное использование кроме предварительного ознакомления запрещено.
Публикация данного документа не преследует за собой никакой коммерческой выгоды.
Эта книга способствует профессиональному росту читателей и является рекламой бумажных изданий.
Все авторские права принадлежат их уважаемым владельцам.
Если Вы являетесь автором данной книги и её распространение ущемляет Ваши авторские права или если Вы хотите внести изменения в данный документ или опубликовать новую книгу свяжитесь с нами по email.

РАЗРАБОТЧИКУ

Разработка ядра Linux

Второе издание

NOVELL PRESS

Роберт Лав

Главный инженер по разработке ядра
Группа Ximian Desktop, корпорации Novell

ВИЛЬЯМС

www.williamspublishing.com

Novell

Разработка ядра Linux

Второе издание

Linux Kernel Development

Second Edition

Robert Love

Novell®

Novell Press, 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Разработка ядра Linux

Второе издание

Роберт Лав

Москва • Санкт-Петербург • Киев
2006

ББК 32.973.26-018.2.75
Л13
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского *АА. Судакава*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspubiishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Лав, Роберт.

Л13 Разработка ядра Linux, 2-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильяме" 2006. — 448 с. : ил. — Парал. тит. англ.

ISBN 5-8459-1085-4 (рус.)

В книге детально рассмотрены основные подсистемы и функции ядер Linux серии 2.6, включая особенности построения, реализации и соответствующие программны интерфейсы. Рассмотренные вопросы включают: планирование выполнения процес сов, управление временем и таймеры ядра, интерфейс системных вызовов, особен ности адресации и управления памятью, страничный кэш, подсистему VFS, механиз мы синхронизации, проблемы переносимости и особенности отладки. Автор книги является разработчиком основных подсистем ядра Linux. Ядро рассматривается как с теоретической, так и с прикладной точек зрения, что может привлечь читателей различными интересами и потребностями.

Книга может быть рекомендована как начинающим, так и опытным разработчи кам программного обеспечения, а также в качестве дополнительных учебных мате риалов.

ББК 32.973.26-018.2.7

Все названия программных продуктов являются зарегистрированными торговыми марками со ответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разреше ния издательства Novell Press.

Aufhorued translation from the English language edition published by Novell Press, Copyright © 200 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmi ted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permi sion from the publisher.

All terms mentioned in this book that are known to be trademarks or service marks have been appro[rialely capitalized. Novell Press cannot attest to the accuracy of this information. Use of a term in this boo should not be regarded as affecting the validity of any trademark or service mark.

Russian language edition is published by Williams Publishing House according to the Agreement wit R&I Enterprises International, Copyright © 2006

ISBN 5-S459-10854 (рус.)
ISBN 0-Й72-Я2720-1 (англ.)

© Издательский дом "Вильяме", 2006
© 2005 by Pearson Education, Inc., 200

Оглавление

Предисловие 15
Введение 17
Об авторе 21
От издательства 22
Глава 1. Введение в ядро Linux 23
Глава 2. Начальные сведения о ядре Linux 33
Глава 3. Управление процессами 45
Глава 4. Планирование выполнения процессов 65
Глава 5. Системные вызовы 95
Глава 6. Прерывания и обработка прерываний 109
Глава 7. Обработка нижних половин и отложенные действия 131
Глава 8. Введение в синхронизацию выполнения кода ядра 163
Глава 9. Средства синхронизации в ядре 177
Глава 10. Таймеры и управление временем 207
Глава 11. Управление памятью 233
Глава 12. Виртуальная файловая система 265
Глава 13. Уровень блочного ввода-вывода 293
Глава 14. Адресное пространство процесса 311
Глава 15. Страничный кэш и обратная запись страниц 331
Глава 16. Модули 343
Глава 17. Объекты kobject и файловая система sysfs 355
Глава 18. Отладка 373
Глава 19. Переносимость 389
Глава 20. Заплаты, разработка и сообщество 405
Приложение А. Связанные списки 415
Приложение Б. Генератор случайных чисел ядра 423
Приложение В. Сложность алгоритмов 429
Приложение Г. Библиография и список литературы 433
Предметный указатель 437

Содержание

Предисловие 15
Введение 17
Итак... 18
Версия ядра 18
Читательская аудитория 18
Интернет-ресурс 19
Благодарности ко второму изданию 20
Об авторе 21
От издательства 22
Для читателей 22

Глава 1. Введение в ядро Linux 23

Потом пришел Линус: введение в Linux 25
Обзор операционных систем и ядер 26
Ядро Linux в сравнении с классическими ядрами Unix 29
Версии ядра Linux 31
Сообщество разработчиков ядра Linux 32
Перед тем как начать 32

Глава 2. Начальные сведения о ядре Linux 33

Получение исходного кода ядра 33
Инсталляция исходного кода ядра 33
Использование заплат 34
Дерево исходных кодов ядра 34
Сборка ядра 34
Уменьшение количества выводимых сообщений 37
Параллельная сборка 37
Инсталляция ядра 38
"Зверек другого рода" 38
Отсутствие библиотеки libc 39
Компилятор GNU C 39
Отсутствие защиты памяти 41
Нельзя просто использовать вычисления с плавающей точкой 42
Маленький стек фиксированного размера 42
Синхронизация и параллелизм 42
Переносимость — это важно 43
Резюме 43

Глава 3. Управление процессами 45

Дескриптор процесса и структура task structure . 46

Выделение дескриптора процесса 47
Хранениедескрипторапроцесса 48
Состояние процесса 50
Манипулирование текущим состоянием процесса 51
Контекст процесса 51
Дерево семейства процессов 52
Создание нового процесса 53
Копирование при записи 54
Реализация потоков в ядре Linux 57
Потоки в пространстве ядра 59
Завершение процесса 59
Удаление дескриптора процесса 61
Дилемма "беспризорного" процесса 61
Резюме 63

Глава 4. Планирование выполнения процессов 65

Стратегия планирования 67
Процессы, ограниченные скоростью ввода-вывода и скоростью процессора 67
Приоритет процесса 68
Квант времени 69
Вытеснение процесса 70
Стратегия планирования в действии 70
Алгоритм планирования 71
Очереди выполнения 72
Массивы приоритетов 74
Пересчет квантов времени 75
Вычисление приоритетов и квантов времени 78
Переход в приостановленное состояние и возврат к выполнению 81
Балансировка нагрузки 83
Вытеснение и переключение контекста 87
Вытеснение пространства пользователя 88
Вытеснение пространства ядра 88
Режим реального времени 89
Системные вызовы для управления планировщиком 91
Системные вызовы, связанные с управлением стратегией и приоритетом 91
Системные вызовы управления процессорной привязкой 92
Передача процессорного времени 92
В завершение о планировщике 93

Глава 5. Системные вызовы 95

API, POSIX и библиотека C 96
Вызовы syscall 97
Номера системных вызовов 98
Производительность системных вызовов 99
Обработка системных вызовов 99
Определение необходимого системного вызова 99
Передача параметров 100

Реализация системных вызовов 101
Проверка параметров 101
Контекст системного вызова 104
Окончательные шаги регистрации системного вызова 104
Доступ к системным вызовам из пространства пользователя 106
Почему не нужно создавать системные вызовы 107
В заключение о системных вызовах 108

Глава 6. Прерывания и обработка прерываний 109

Прерывания 109
Обработчики прерываний 111
Верхняя и нижняя половины 111
Регистрация обработчика прерывания 112
Освобождение обработчика прерывания 114
Написание обработчика прерывания 115
Совместно используемые обработчики 116
Настоящий обработчик прерывания 117
Контекст прерывания 119
Реализация системы обработки прерываний 121
Управление прерываниями 124
Запрещение и разрешение прерываний 125
Запрещение определенной линии прерывания 126
Состояние системы обработки прерываний 127
Не нужно прерывать, мы почти закончили! 128

Глава 7. Обработка нижних половин и отложенные действия 131

Нижние половины 132
Когда нужно использовать нижние половины 133
Многообразие нижних половин 133
Механизм отложенных прерываний (softirq) 136
Реализация отложенных прерываний 136
Использование отложенных прерываний 139
Тасклеты 141
Реализация тасклетов 141
Использование тасклетов 144
Демон ksoftirqd 146
Старый механизм BH 148
Очереди отложенных действий 149
Реализация очередей отложенных действий 150
Использование очередей отложенных действий 154
Старый механизм очередей заданий 157
Какие обработчики нижних половин необходимо использовать 157
Блокировки между обработчиками нижних половин 159
Запрещение обработки нижних половин 160
Внизу обработки нижних половин 162

Глава 8. Введение в синхронизацию выполнения кода ядра 163

Критические участки и состояние конкуренции за ресурсы 164
Зачем нужна защита 164
Блокировки 167
Откуда берется параллелизм 169
Что требует защиты 170
Взаимоблокировки 172
Конфликт при захвате блокировки и масштабируемость 174
Блокировки в вашем коде 176

Глава 9. Средства синхронизации в ядре 177

Атомарные операции 177
Целочисленные атомарные операции 178
Битовые атомарные операции 181
Спин-бдокировки 183
Другие средства работы со спин-блокировками 186
Спин-блокировки и обработчики нижних половин 187
Спин-блокировки чтения-записи 188
Семафоры 190
Создание и инициализация семафоров 193
Использование семафоров 193
Семафоры чтения-записи 195
Сравнение спин-блокировок и семафоров 196
Условные переменные 196
BLK: Большая блокировка ядра 197
Секвентные блокировки 199
Средства запрещения преемптивности 200
Барьеры и порядок выполнения 201
Резюмирование по синхронизации 205

Глава 10. Таймеры и управление временем 207

Информация о времени в ядре 208
Частота импульсов таймера: HZ 209
Идеальное значение параметра HZ 210
Переменная jiffies 213
Внутреннее представление переменной j iffies 214
Переполнение переменной jiffies 215
Пространство пользователя и параметр HZ 217
Аппаратные часы и таймеры 218
Часы реального времени 218
Системный таймер 218
Обработчик прерываний таймера 219
Абсолютное время 221
Таймеры 223
Использование таймеров 224
Состояния конкуренции, связанные с таймерами 226

Реализациятаймеров 226
Задержка выполнения 227
Задержка с помощью цикла 227
Короткие задержки 229
Функция schedule_timeout () 230
Время вышло 232

Глава 11. Управление памятью 233

Страницы памяти 233
Зоны 235
Получение страниц памяти 238
Получение страниц заполненных нулями 238
Освобождение страниц 239
Функция kmalloc () 240
Флаги gfp_mask 241
Функция kfree () 245
Функция vmalloc () 246
Уровень слябового распределителя памяти 248
Устройство слябового распределителя памяти 249
Интерфейс слябового распределителя памяти 252
Пример использования слябового распределителя памяти 254
Статическое выделение памяти в стеке 256
Честная игра со стеком 257
Отображение верхней памяти 257
Постоянное отображение 257
Временное отображение 258
Выделение памяти, связанной с определенным процессором 259
Новый интерфейс percpu 260
Работа с данными, связанными с процессорами, на этапе компиляции 260
Работа с данными, связанными с процессорами, на этапе выполнения 261
Когда лучше использовать данные, связанные с процессорами 263
Какой способ выделения памяти необходимо использовать 264

Глава 12. Виртуальная файловая система 265

Общий интерфейс к файловым системам 266
Уровень обобщенной файловой системы 266
Файловые системы Unix 267
Объекты VFS и их структуры данных 269
Другие объекты подсистемы VFS 270
Объект superblock 270
Операции суперблока 272
Объект inode 274
Операции с файловыми индексами 276
Объект dentry 278
Состояние элементов каталога 280
Кэш объектов dentry 280
Операции с элементами каталогов 281

Объект file 283
Файловые операции 284
Структуры данных, связанные с файловыми системами 288
Структуры данных, связанные с процессом 289
Файловые системы в операционной системе Linux 291

Глава 13. Уровень блочного ввода-вывода 293

Анатомия блочного устройства 294
Буферы и заголовки буферов 295
Структура bio 298
Сравнение старой и новой реализаций 300
Очереди запросов . 301
Запросы 301
Планировщики ввода-вывода 302
Задачи планировщика ввода-вывода 302
Лифтовой алгоритм Линуса 503
Планировщик ввода-вывода с лимитом по времени 304
Прогнозирующий планировщик ввода-вывода 307
Планировщик ввода-вывода с полностью равноправными очередями 308
Планировщик ввода-вывода noop 309
Выбор планировщика ввода-вывода 309
Резюме 310

Глава 14. Адресное пространство процесса 311

Дескриптор памяти 313
Выделение дескриптора памяти 315
Удаление дескриптора памяти 315
Структура mm_struct и потоки пространства ядра 316
Области памяти 316
Флаги областей VMA 317
Операции с областями VMA 319
Списки и деревья областей памяти 320
Области памяти в реальной жизни 321
Работа с областями памяти 322
Функция find_vma () 323
Функция find_vma_prev () 324
Функция find_VMA_intersection () 324
Функции mmap () и do_mmap (): создание интервала адресов 325
Системный вызов mmap () 326
Функции munmap () и do_munmap (): удаление интервала адресов 327
Системный вызов munmap () 327
Таблицы страниц 327
Заключение 329

Глава 15. Страничный кэш и обратная запись страниц 331

Страничный кэш 332
Объект address_space 332

Базисное дерево 335
Старая хеш-таблица страниц 336
Буферный кэш 336
Демон pdflush 337
Демоны bdflush и kupdated 339
Предотвращение перегруженности: для чего нужны несколько потоков 339
Коротко о главном 341

Глава 16. Модули 343

Модуль "Hello,Worldl" 343
Сборка модулей 345
Использование дерева каталогов исходных кодов ядра 345
Компиляция вне дерева исходных кодов ядра 347
Инсталляция модулей 347
Генерация зависимостей между модулями 347
Загрузка модулей 348
Управление конфигурационными параметрами 349
Параметры модулей 351
Экспортируемые символы 353
Вокруг модулей 354

Глава 17. Объекты kobject и файловая система sysfs 355

Объекты kobject 356
Типы ktype 357
Множества объектов kset 358
Подсистемы 358
Путаница со структурами 359
Управление и манипуляции с объектами kobject 360
Счетчики ссылок 361
Структуры kref 362
Файловая система sysfs 363
Добавление и удаление объектов на файловой системе sysfs 365
Добавление файлов на файловой системе sysfs 366
Уровень событий ядра 369
Кратко об объектах kobj ect и файловой системе sysfs 371

Глава 18. Отладка 373

С чего необходимо начать 373
Дефекты ядра 374
Функция printk() 375
Устойчивость функции printk() 375
Уровни вывода сообщений ядра 376
Буфер сообщений ядра 377
Демоны syslogd и klogd 377
Замечание относительно функции printk() и разработки ядра 378
Сообщения Oops 378
Утилита ksymoops 380

Функция kallsyms 380
Конфигурационные параметры отладки ядра 381
Отладка атомарных операций 381
Генерация ошибок и выдача информации 382
Магическая клавиша SysRq 382
Сага об отладчике ядра 384
Использование отладчика gdb 384
Отладчик kgdb 385
Отладчик kdb 385
Исследование и тестирование системы 385
Использование идентификатора UID в качестве условия 385
Использование условных переменных 386
Использование статистики 386
Ограничение частоты следования событий при отладке 387
Нахождение исполняемых образов с изменениями приводящими к ошибкам 388
Если ничто не помогает — обратитесь к сообществу 388

Глава 19. Переносимость 389

История переносимости Linux 390
Размер машинного слова и типы данных 391
Скрытые типы данных 394
Специальные типы данных 394
Типы с явным указанием размера 395
Знак типа данных char 396
Выравнивание данных 396
Как избежать проблем с выравниванием 397
Выравнивание нестандартных типов данных 397
Заполнение структур 398
Порядок следования байтов 399
История терминов big-endian и little-endian 401
Порядок байтов в ядре 401
Таймер 401
Размер страницы памяти 402
Порядок выполнения операций процессором 403
Многопроцессорность, преемптивность и верхняя память 403
Пару слов о переносимости 404

Глава 20. Заплаты, разработка и сообщество 405

Сообщество 405
Стиль написания исходного кода 406
Отступы 406
Фигурные скобки 406
Длинные строки 407
Имена 408
Функции 408
Комментарии 408
Использование директивы typedef 410

Использование того, что уже есть 410
Никаких директив ifdef в исходном коде 410
Инициализация структур 411
Исправление ранее написанного кода 411
Организация команды разработчиков 411
Отправка сообщений об ошибках 412
Генерация заплат 412
Представление заплат 413
Заключение 414

Приложение А. Связанные списки 415

Кольцевые связанные списки 416
Перемещение по связанному списку 416
Реализация связанных списков в ядре Linux 417
Структура элемента списка 417
Работа со связанными списками 419
Перемещение по связанным спискам 421

Приложение Б. Генератор случайных чисел ядра 423

Принцип работы и реализация 424
Проблема с загрузкой системы 426
Интерфейсы для ввода энтропии 426
Интерфейсы для вывода энтропии 427

Приложение В. Сложность алгоритмов 429

Алгоритмы 429
Множество О 430
Множество большого-тета 430
Объединяем все вместе 431
Опасность, связанная со сложностью алгоритмов 431

Приложение Г. Библиография и список литературы 433

Книги по основам построения операционных систем 433
Книги о ядрах Unix 434
Книги о ядрах Linux 434
Книги о ядрах других операционных систем 435
Книги по API Unix 435
Другие работы 435
Web-сайты 436

Предметный указатель 437

Посвящается Дорис (Doris) и Хэлен (Helen)

Предисловие

В связи с тем, что ядро и приложения операционной системы Linux используются все более широко, возрастает число разработчиков системного программного обеспечения, желающих заняться разработкой и поддержкой операционной системы Linux. Некоторые из этих инженеров руководствуются исключительно собственным интересом, некоторые работают в компаниях, которые занимаются операционной системой Linux, некоторые работают на производителей компьютерных аппаратных средств, некоторые заняты в проектах по разработке программного обеспечения на дому.

Однако все они сталкиваются с общей проблемой: кривая затрат на изучение ядра становится все длиннее и круче. Система становится все более сложной и, кроме того, очень большой по объему. Годы проходят, и нынешние члены команды разработчиков ядра приобретают все более широкие и глубокие знания, что увеличивает разрыв между ними и разработчиками-новичками.

Я уверен, что понимание основного кода ядра Linux уже сейчас является проблемой, приводящей к ухудшению качества ядра, и в будущем эта проблема станет еще более серьезной. Все, кому нравится операционная система Linux, несомненно, заинтересованы в увеличении числа разработчиков, которые смогут внести свой вклад в развитие ядра этой операционной системы.

Один из возможных подходов к решению данной проблемы — ясность исходного кода: удобные интерфейсы, четкая структура, следование принципу "Делать мало, но делать хорошо" и т.д. Такое решение предложено Линусом Торвальдсом (Linus Torvalds).

Подход, который предлагаю я, состоит в использовании большего числа комментариев внутри исходного кода, что поможет читателю понять, чего хотел достичь программист. (Процесс выявления расхождений между целью и реализацией известен как *отладка*. Этот процесс значительно затрудняется, если не известно, чего хотели достичь.)

Однако комментарии все же не дают представления о том, для чего предназначено большинство подсистем и как разработчики приступали к их реализации.

Именно печатное слово лучше всего подходит для стартовой точки такого понимания.

Вклад Роберта Лава (Robert Love) состоит в предоставлении возможности, благодаря которой опытные разработчики смогут получить полную информацию о том, какие задачи должны выполнять различные подсистемы ядра и каким образом предполагается выполнение этих задач. Этой информации должно быть достаточно для многих людей: для любопытных, для разработчиков прикладного программного обеспечения, для тех, кто хочет ознакомиться с устройством ядра, и т.д.

Кроме того, данная книга является ступенькой, которая может перенести начинающих разработчиков на новый уровень, где изменения в ядро вносятся для того, чтобы достичь определенной цели. Я хотел бы посоветовать начинающим разработчикам, чтобы они не боялись испачкать свои руки: наилучший способ понять какую-либо часть ядра — это внести в нее изменения. Внесение изменений повышает понимание разработчика до уровня, которого нельзя достичь простым чтением кода ядра.

Серьезный разработчик ядра присоединится к спискам рассылки разработчиков и будет контактировать с другими коллегами. Это основной способ, позволяющий разработчикам учиться и быть на высоком уровне. Роберт очень хорошо осветил механизмы и культуру этой важной части жизни сообщества разработчиков ядра.

Пользуйтесь книгой Роберта и учитесь по ней! Может быть, и вы решите сделать следующий шаг и вступить в сообщество разработчиков ядра, куда мы вас и приглашаем. Людей ценят по важности их дел, поэтому, помогая развитию операционной системы Linux, знайте, что ваша работа— небольшая, но непосредственная помощь десяткам или даже сотням миллионов людей.

Эндрю Мортон (Andrew Morton)
Open Source Development Labs

Введение

Когда я сделал первую попытку превратить свой опыт работы с ядром Linux в текст книги, понял, что не знаю, как двигаться дальше. Не хотелось просто писать еще одну книгу о ядре операционной системы. Конечно, на эту тему *не так уж и много* книг, но все же я хотел сделать что-то такое, благодаря чему моя книга была бы особенной. Как достичь этой цели? Я не могу успокоиться, пока не сделаю что-нибудь особенное, лучшее в своем роде.

Наконец я решил, что смогу предложить достаточно уникальный подход к данной теме. Моя работа — изучение и разработка ядра операционной системы. Мое увлечение — изучение и разработка ядра операционной системы. Моя любовь — ядро операционной системы. Конечно, за многие годы я успел собрать много интересных анекдотов и полезных советов. С моим опытом я смог бы написать книгу о том, как нужно разрабатывать программный код ядра и как этого делать *не нужно*. Прежде всего, эта книга об устройстве и практической реализации ядра операционной системы Linux. В ней информация представлена так, чтобы получить достаточно знаний для решения реальных практических задач и чтобы эти задачи решать правильно. Я человек прагматичный, и книга имеет практический уклон. Она должна быть полезной, интересной и легко читаться.

Я надеюсь, что читатели, после прочтения этой книги, получат хорошее понимание тек правил (писанных и неписаных), которые действуют в ядре операционной системы. Я также надеюсь, что читатели сразу после прочтения этой книги смогут начать действовать и писать полезный, правильный и хороший код ядра. Конечно, эту книгу можно читать и просто ради интереса.

Это то, что касалось еще первого издания книги. Однако время идет и снова приходится возвращаться к рассмотренным вопросам. В этом издании представлено несколько больше информации по сравнению с первым: материал серьезно пересмотрен и доработан, появились новые разделы и главы. С момента выхода первого издания в ядро были внесены изменения. Однако, что более важно, сообщество разработчиков ядра Linux приняло решение[1] в ближайшем будущем не начинать разработку серии ядра 2.7. Было решено заняться стабилизацией серии ядра 2.6. Стабилизация включает в себя много моментов, тем не менее есть один важный, который касается данной книги, — книга, которая посвящена ядру серии 2.6, остается актуальной. Если изменения происходят не слишком быстро, то существует большой шанс, что "моментальный снимок" ядра останется актуальным и в будущем. В конце концов, книга сможет вырасти и стать канонической документацией по ядру. Я надеюсь, что именно такая книга и находится у вас в руках.

Как бы там ни было, книга написана, и я надеюсь, что она вам понравится.

[1] Это решение было принято на саммите разработчиков ядра Linux (Linux Kernel Development Summit), который состоялся летом 2004 года в г. Оттава, Канада.

Итак...

Разработка программного кода ядра операционной системы не требует наличия гениальной, волшебной или густой бороды Unix-хакера. Хотя ядро операционной системы и имеет некоторые свои особенности, оно незначительно отличается от любого большого программного продукта. Так же как и в случае любой сложной программы, здесь есть, что изучать, но в программировании ядра не намного больше священных или непонятных вещей, чем в создании любой другой программы.

Очень важно, чтобы вы читали программный код. Доступность открытого исходного кода операционной системы Linux — это подарок, который встречается очень редко. Однако недостаточно *только* читать исходный код. Необходимо взяться за дело серьезно и изменять этот программный код. Находите ошибки и исправляйте их! Улучшайте драйверы для своего аппаратного обеспечения! Находите слабые места и закрывайте их! У вас все получится, если вы будете сами *писать* программный код.

Версия ядра

Эта книга посвящена ядрам Linux серии 2.6 и базируется на версии ядра 2.6.10. Ядро — это "движущийся объект", и никакая книга не в состоянии передать динамику во все моменты времени. Тем не менее базовые внутренние структуры ядра уже сформировались, и основные усилия по представлению материала были направлены на то, чтобы этот материал можно было использовать и в будущем.

Читательская аудитория

Эта книга предназначена для разработчиков программного обеспечения, которые заинтересованы в понимании ядра операционной системы Linux. Тем не менее это *не* построчные комментарии исходного кода ядра. Это также не руководство по разработке драйверов и не справочник по программному интерфейсу (API) ядра (кстати, формализованного API ядра Linux никогда не было). Целью книги является предоставление достаточной информации об устройстве и реализации ядра для того, чтобы подготовленный программист смог начать разработку программного кода. Разработка ядра может быть увлекательным и полезным занятием, и я хочу ознакомить читателя с этой сферой деятельности по возможности быстро. В книге обсуждаются как вопросы теории, так и практические приложения, она обращена к людям, которые интересуются и тем, и другим. Я всегда придерживался мнения, что для понимания практических приложений необходима теория, тем не менее я считаю, что эта книга не сильно углубляется в оба этих направления. Я надеюсь, что, независимо от мотиваций необходимости понимания ядра операционной системы Linux, эта книга сможет объяснить особенности устройства и реализации в достаточной степени.

Таким образом, данная книга освещает как использование основных подсистем ядра, так и особенности их устройства и реализации. Я думаю, что эти вопросы важны и достойны обсуждения. Хороший пример — глава 7, "Обработка нижних половин и отложенные действия", посвященная обработчикам *нижних половин* (bottom half).

В этой главе рассказывается о принципах работы и об особенностях реализации механизмов обработки нижних половин (эта часть может быть интересна разработчикам основных механизмов ядра), а также о том, как на практике использовать экспортируемый интерфейс ядра для реализации собственных обработчиков bottom half (это может быть интересно для разработчиков драйверов устройств). На самом деле мне кажется, что обе эти стороны обсуждения будут интересны для всех групп разработчиков. Разработчик основных механизмов ядра, который, конечно, должен понимать принципы работы внутренних частей ядра, должен также понимать и то, как интерфейсы ядра будут использоваться на практике. В то же самое время разработчик драйверов устройств получит большую пользу от хорошего понимания того, что стоит за этим интерфейсом.

Все это сродни изучению программного интерфейса некоторой библиотеки наряду с изучением того, как эта библиотека реализована. На первый взгляд, разработчик прикладных программ должен понимать лишь интерфейс (API). И действительно, интерфейсы часто предлагают рассматривать в виде черного ящика. Разработчик библиотеки, наоборот, обычно интересуется лишь принципом работы и реализации функций библиотеки. Я уверен, что обе группы разработчиков должны потратить некоторое время на изучение другой стороны предмета. Разработчик программ, который хорошо понимает операционную систему, сможет значительно лучше эту операционную систему использовать. Аналогично разработчик библиотеки должен иметь хотя бы малое представление о том, что происходит в реальной жизни, и, в частности, о тех программах, в которых будет использоваться его библиотека. Поэтому я старался коснуться как устройства, так и использования подсистем ядра не только в связи с тем, что эта книга может быть полезна для одной или другой группы разработчиков, а в надежде, что *весь материал* книги будет полезен для всех разработчиков.

Предполагается, что читатель знаком с языком программирования C и операционной системой Linux. Некоторые знания принципов построения операционных систем также желательны. Я старался объяснять все понятия, однако в случае проблем в списке литературы можно найти несколько отличных книг, которые посвящены основам построения операционных систем.

Эта книга будет полезна для студентов, изучающих основы построения операционных систем, в качестве *прикладного* пособия и вводного материала по соответствующей теории. Книга пригодна как для расширенных специальных курсов, так и для общих специальных курсов, причем в последнем случае без дополнительных материалов. Я прошу потенциальных учебных инструкторов связаться со мной; я буду очень рад оказать помощь.

Интернет-ресурс

Автор поддерживает Интернет-сайт `http://tech9.net/rml/kernel_book/`, содержащий информацию о данной книге, включая ошибки, расширенные и исправленные разделы, а также информацию о будущих изданиях. Всем читателям рекомендуется посетить этот сайт.

Благодарности ко второму изданию

Как и большинство авторов, я писал эту книгу, не сидя в пещере (что само по себе хорошо, потому что в пещерах могут водиться медведи), и, следовательно, многие люди оказали мне поддержку в создании рукописи своим сердцем и умом. Поскольку невозможно привести полный список этих людей, я хочу поблагодарить всех своих друзей и коллег за помощь, поддержку и конструктивную критику.

В первую очередь, я хотел бы высказать благодарность моему редактору Скотту Мейерсу (Scott Meyers) за руководство, благодаря которому второе издание книги превратилось из идеи в конечный продукт. Мне снова было очень приятно работать с Джоржем Недеффом (Georg Nedeff), производственным редактором, который во всем обеспечивал порядок. Особая благодарность литературному редактору Марго Кэтс (Margo Catts). Мы можем только желать, чтобы наше владение ядром было так же совершенно, как ее владение печатным словом.

Отдельное спасибо техническим редакторам этого издания Адаму Белею (Adam Belay), Мартину Пулу (Martin Pool) и Крису Ривере (Chris Rivera). Их знания и исправления помогли сделать эту книгу неизмеримо лучше. Если, несмотря на их неоценимые усилия, все же остались ошибки, то это вина автора. Такое же большое спасибо Заку Брауну (Zak Brown), который приложил огромные усилия к техническому редактированию первого издания.

Многие разработчики ядра отвечали на вопросы, предоставляли поддержку или просто писали программный код, интересный настолько, что по нему можно было бы написать отдельную книгу. Среди них Андреа Аркангели (Andrea Arcangely), Алан Кокс (Alan Cox), Грег Кроах-Хартман (Greg Kroah-Hartman), Даниэл Филлипс (Daniel Phillips), Дэвид Миллер (David Miller), Патрик Мочел (Patrick Mochel), Эндрю Мортон (Andrew Morton), Звене Мвейкамбо (Zwane Mwaikambo), Ник Пиггин (Nick Piggin) и Линус Торвальдс (Linus Torvalds). Особое спасибо тайному сообществу ядра (хотя никакого тайного сообщества нет).

Я хочу выразить свою любовь и признательность многим людям. Среди них Пол Амичи (Paul Amichi), Кейт Бэрбег (Keith Barbag), Дейв Эггерс (Dave Eggers), Ричард Эриксон (Richard Erickson), Нат Фридман {Nat Friedman), Дастин Холл (Dostin Hall), Джойс Хокинс (Joyce Hawkins), Мигуэль де Иказа (Miguel de Icaza), Джимми Крел (Jimmy Krehl), Дорис Лав (Doris Love), Джонатан Лав (Jonathan Love), Патрик ЛеКлер (Patrick LeClair), Линда Лав (Linda Love), Рэнди О'Дауд (Randy O'Dowd), Сальваторэ Рибаудо (Salvatore Ribaudo) и его чудесная мама, Крис Ривера (Chris Rivera), Джой Шау (Joey Shaw), Джэрэми ВанДорен (Jeremy VanDoren) и его семья, Стив Вейсберг (Steve Weisberg) и Хелен Винснант (Helen Whinsnant).

И в заключение, спасибо за все моим родителям.

Желаю большого хакерского счастья!

Роберт Лав,
г. Кембридж, штат, Массачусетс.

Об авторе

Роберт Лав (Robert Love) использует операционную систему Linux с первых дней ее существования. Он является страстным активистом сообществ разработчиков ядра и GNOME. Сейчас Роберт работает главным инженером по разработке ядра группы разработчиков Ximian Desktop компании Novell. До этого он работал инженером по разработке ядра компании Мота Vista Software.

Проекты по разработке ядра, которыми занимался автор, включают планировщик выполнения процессов, преемптивное (вытесняемое) ядро (preemptive kernel), уровень событий ядра, улучшение поддержки виртуальной памяти (VM), улучшение поддержки многопроцессорного оборудования. Роберт является автором утилит schedutils и менеджера томов GNOME. Роберт Лав читает лекции и пишет статьи по основам построения ядра операционной системы и получает приглашения редактировать статьи в издании *Linux Journal*.

Автор получил степень бакалавра по математике и вычислительной технике в университете штата Флорида. Хотя Роберт и родился в южной Флориде, своим домом он считает Кембридж, штат Массачусетс. Роберт увлекается футболом, фотографией и любит готовить.

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Информация для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

Для читателей

Более подробную информацию об этой и других книгах издательства Sams Publishing можно получить на Интернет-сайте www. nowellpress.com. Для поиска информации о книгах введите в поисковое поле код ISBN (без соединительных черточек) или название книги.

1

Введение в ядро Linux

Даже после трех десятилетий использования операционная система (ОС) Unix все еще считается одной из самых мощных и элегантных среди всех существующих операционных систем. Со времени создания операционной системы Unix в 1969 году, это детище Денниса Ритчи (Dennis Ritchie) и Кена Томпсона (Ken Thompson) стало легендарным творением, системой, принцип работы которой выдержал испытание временем и имя которой оказалось почти незапятнанным.

Операционная система Unix выросла из Multics — многопользовательской операционной системы, проект по созданию которой потерпел неудачу в корпорации Bell Laboratories. По прекращении проекта Multics, сотрудники центра Bell Laboratories Computer Sciences Research Center прекратили работу и так и не создали дееспособной диалоговой операционной системы. Летом 1969 года программисты корпорации Bell Labs разработали проект файловой системы, которая в конце концов была включена в операционную систему Unix. Томпсон осуществил реализацию операционной системы для реально не используемой платформы PDP-7. В 1971 году операционная система Unix была перенесена на платформу PDP-11, а в 1973 году переписана с использованием языка программирования C, что было беспрецедентным шагом в то время, но этот шаг стал основой для будущей переносимости. Первая версия операционной системы Unix, которая использовалась вне стен Bell Labs, называлась Unix System версии 6, ее обычно называют V6.

Другие компании перенесли операционную систему Unix на новые типы машин. Версии, полученные в результате переноса, содержали улучшения, которые позже привели к появлению нескольких разновидностей этой операционной системы, В 1977 году корпорация Bell Labs выпустила комбинацию этих вариантов в виде одной операционной системы Unix System III, а в 1982 году корпорация AT&T представила версию System V[1].

Простота устройства операционной системы Unix, а также тот факт, что эта система распространялась вместе со своим исходным кодом, привели к тому, что дальнейшие разработки начали проводиться в других организациях. Наиболее важным среди таких разработчиков был Калифорнийский университет в городе Беркли (University of California at Berkeley).

[1] Как насчет версии System IV? Ходят слухи, что это внутренняя экспериментальная версия.

Варианты операционной системы Unix из Беркли именовались Berkeley Software Distributions (BSD). Первая версия операционной системы Unix, разработанная в Беркли в 1981 году, называлась 3BSD. Следом за ней появились выпуски серии 4BSD: 4.0BSD, 4.1BSD, 4.2BSD и 4.3BSD. В этих версиях операционной системы Unix была добавлена виртуальная память, замещение страниц по требованию (demand paging) и стек протоколов TCP/IP. Последней официальной версией ОС Unix из Беркли была 4.4BSD, выпущенная в 1993 году, которая содержала переписанную систему управления виртуальной памятью. Сейчас разработка линии BSD продолжается в операционных системах Darwin, Dragonfly BSD, FreeBSD, NetBSD и OpenBSD.

В 1980-1990-х годах многие компании, разработчики рабочих станций и серверов, предложили свои коммерческие версии операционной системы Unix. Эти операционные системы обычно базировались на реализациях AT&T или Беркли и поддерживали дополнительные профессиональные возможности, которые обеспечивала соответствующая аппаратная платформа. Среди таких систем были Tru64 компании Digital, HP-UX компании Hewlett Packard, AIX компании IBM, DYNIX/ptx компании Sequent, IRIX компании SGI, Solaris компании Sun.

Первоначальное элегантное устройство операционной системы Unix в соединении с многолетними нововведениями и улучшениями, которые за ними последовали, сделали систему Unix мощной, устойчивой и стабильной. Очень небольшое количество характеристик ОС Unix ответственны за ее устойчивость. Во-первых, операционная система Unix проста: в то время как в некоторых операционных системах реалилованы тысячи системных вызовов и эти системы имеют недостаточно ясное назначение, Unix-подобные операционные системы обычно имеют только несколько сотен системных вызовов и достаточно четкий дизайн. Во-вторых, в операционной системе Unix *все представляется в виде файлов*[2]. Такая особенность позволяет упростить работу с данными и устройствами, а также обеспечить это посредством простых системных вызовов: `open()`, `read()`, `write()`, `ioctl()` и `close()`. В-третьи, ядро и системные утилиты Операционной системы Unix написаны на языке программирования C - это свойство делает Unix удивительно переносимой и доступной для широкого круга разработчиков операционной системой.

Для ОС Unix характерно очень малое время создания нового процесса и уникальный системный вызов `fork()`. И наконец, операционная система Unix предоставляет простые и в то же время устойчивые средства межпроцессного взаимодействия, которые, в сочетании с быстрым созданием процессов, позволяют создавать простые утилиты, которые *умеют выполнять всего одну функцию, но делают это хорошо,* и могут быть связаны вместе для выполнения более сложных задач.

Сегодня Unix — современная операционная система, которая поддерживает многозадачность, многопоточность, виртуальную память, замещение страниц по требованию, библиотеки совместного использования, загружаемые по требованию, и сеть TCP/IP. Многие варианты операционной системы Unix поддерживают масштабирование до сотен процессоров, в то время как другие варианты ОС Unix работают на миниатюрных устройствах в качестве встраиваемых систем. Хотя разработка Unix больше не является исследовательским проектом, все же продолжаются разработки (с целью получить дополнительные преимущества) с использованием возможностей

[2]Да, конечно, не все, но многое представлено в виде файла. В современных операционных системах, таких как Plan9 (наследник Unix), практически все представляется в виде файлом.

операционной системы Unix, которая при этом остается практичной операционной системой общего назначения.

Операционная система Unix обязана своим успехом простоте и элегантности построения. В основе ее сегодняшней мощности лежат давние идеи Денниса Ритчи, Кена Томпсона и других разработчиков, обеспечившие возможность операционной системе Unix бескомпромиссно развиваться.

Потом пришел Линус: введение в Linux

Операционная система Linux была разработана Линусом Торвальдсом (Linus Torvalds) в 1991 году как операционная система для компьютеров, работающих на новом в то время микропроцессоре Intel 80386. Тогда Линус Торвальдс был студентом университета в Хельсинки и был крайне возмущен отсутствием мощной и в то же время свободно доступной Unix-подобной операционной системы. Операционная система DOS, продукт корпорации Microsoft, была для Торвальдса полезна только лишь, чтобы поиграть в игрушку "Принц Персии", и не для чего больше. Линус пользовался операционной системой Minix, недорогой Unix-подобной операционной системой, которая была создана в качестве учебного пособия. В этой операционной системе ему не нравилось отсутствие возможности легко вносить и распространять изменения исходного кода (это запрещалось лицензией ОС Minix), а также технические решения, которые использовал автор ОС Minix.

Поставленный перед такой проблемой, Линус решил написать свою операционную систему. Начал он с написания простого эмулятора терминала, который он подключал к большим Unix-системам в университете. Его эмулятор терминала постепенно рос, развивался и улучшался. Постепенно *у* Линуса появилась еще не совсем зрелая, но полноценная Unix-система. В 1991 голу он опубликовал в Интернет ее первую версию.

По некоторым неясным причинам, использование операционной системы Linux и количество ее пользователей начали стремительно расти. Более важным для успеха Linux стало то, что эта операционная система привлекла многих разработчиков, которые начали изменять, исправлять и улучшать код. Благодаря соответствующему лицензионному соглашению, ОС Linux быстро стала совместным проектом, который разрабатывается многими людьми.

Сейчас Linux — это развитая операционная система, работающая на аппаратных платформах AMD x86-64, ARM, Compaq Alpha, CRIS, DEC VAX, H8/300, Hitachi SuperH, HP PA-RISC, IBM S/390, Incel IA-64, MIPS, Motorola 68000, PowerPC, SPARC, UltraSPARC и v850. Она работает в различных системах, как размером с часы, так и на больших супер-компьютерных кластерах. Сегодня коммерческий интерес к операционной системе Linux достаточно высок. Как новые корпорации, ориентирующиеся исключительно на Linux {Monta Vista или Red Hal), так и старые (IBM, Novell) предлагают решения на основе этой ОС для встраиваемых систем, десктопов и серверов.

Операционная система Linux является клоном Unix, по ОС Linux— это не Unix. Хотя в ОС Linux позаимствовано много идей от Unix, в Linux реализован API ОС Unix (как это определено в стандарте POSIX и спецификации Single Unix Specification), все же система Linux не является производной от исходного кода Unix, как это имеет место для других Unix-систем, Там, где это желательно, были сделаны отклонения от пути, по которому шли другие разработчики, однако это не

подрывает основные принципы построения операционной системы Unix и не нарушает программные интерфейсы.

Одна из наиболее интересных особенностей операционной системы Linux — то, что это не коммерческий продукт; наоборот, это совместный проект, который выполняется через всемирную сеть Интернет. Конечно, Линус остается создателем Linux и занимается *поддержкой* ядра, но работа продолжается группой мало связанных между собой разработчиков. Фактически кто угодно может внести свой вклад в операционную систему Linux. Ядро Linux, так же как и большая часть операционной системы, является *свободно распространяемым* программным обеспечением и имеет *открытый исходный код*[3].

В частности, ядро Linux выпускается под лицензией GNU General Public License (GPL) версии 2.0. В результате каждый имеет право загружать исходный код и вносить в него любые изменения. Единственная оговорка — любое распространение внесенных вами изменений должно производиться на тех же условиях, которыми пользовались вы при получении исходного кода, включая доступность самого исходного программного кода[4].

Операционная система Linux предоставляет много возможностей для многих людей. Основными частями системы являются ядро, библиотека функций языка C, компилятор, набор инструментов, основные системные утилиты, такие как программа для входа в систему (login) и обработчик команд пользователя (shell). В операционную систему Linux может быть включена современная реализация системы X Windows, включая полно-функциональную среду офисных приложений (desktop environment), такую как, например, GNOME. Для ОС Linux существуют тысячи свободных и коммерческих программ. В этой книге под понятием *Linux, в* основном, имеется в виду *ядро Linux*. Там, где это может привести к неопределенностям, будет указано, что имеется в виду под понятием Linux — вся система или только ядро. Строго говоря, термин Linux относится только к ядру.

Обзор операционных систем и ядер

Из-за неуклонного роста возможностей и не очень качественного построения некоторых современных операционных систем, понятие операционной системы стало несколько неопределенным. Многие пользователи считают, что то, что они видят на экране, — и есть операционная система. Обычно, и в этой книге тоже, под *операционной системой* понимается часть компьютерной системы, которая отвечает за основные функции использования и администрирования. Это включает в себя ядро и драйверы устройств, системный загрузчик (boot loader), командный процессор и другие интерфейсы пользователя, а также базовую файловую систему и системные утилиты. В общем, только *необходимые* компоненты. Термин *система* обозначает операционную систему и все пользовательские программы, которые работают под ее управлением.

Конечно, основной темой этой книги будет *ядро* операционной системы. Интерфейс пользователя — это внешняя часть операционной системы, а ядро — вну-

[3] Для тех, кому интересно, дискуссии по поводу отличия свободного кода от открытого доступна в Интернет по адресам http://www.fsf.org и http://www.opensource.org.

[4] Вероятно, вам нужно прочесть лицензию GNU GPL, если вы еще не читали ее. В файле COPYING, в исходном коде ядра, есть копия этой лицензии. В Интернет лицензия доступна по адресу http://www.fsf.org.

тренняя. В своей основе ядро — это программное обеспечение, которое предоставляет базовые функции для всех остальных частей операционной системы, занимается управлением аппаратурой и распределяет системные ресурсы. Ядро часто называют *основной частью* (core) или *контроллером* операционной системы. Типичные компоненты ядра — обработчики прерываний, которые обслуживают запросы на прерывания, планировщик, который распределяет процессорное время между многими процессами, система управления памятью, которая управляет адресным пространством процессов, и системные службы, такие как сетевая подсистема и подсистема межпроцессного взаимодействия. В современных системах с устройствами управления защищенной памятью ядро обычно занимает привилегированное положение по отношению к пользовательским программам. Это включает доступ ко всем областям защищенной памяти и полный доступ к аппаратному обеспечению. Состояние системы, в котором находится ядро, и область памяти, в которой находится ядро, вместе называются *пространством ядра* (или режимом ядра, kernel-space). Соответственно, пользовательские программы выполняются в *пространствах задач* (пользовательский режим, режим задач, user-space). Пользовательским программам доступно лишь некоторое подмножество машинных ресурсов, они не могут выполнять некоторые системные функции, напрямую обращаться к аппаратуре и делать другие недозволенные вещи. При выполнении программного кода ядра система находится в пространстве (режиме) ядра, в отличие от нормального выполнения пользовательских программ, которое происходит в режиме задачи.

Прикладные программы, работающие в системе, взаимодействуют с ядром с помощью интерфейса *системных вызовов* (system call) (рис. 1.1). Прикладная программа обычно вызывает функции различных библиотек, например *библиотеки функций* языка С, которые, в свою очередь, обращаются к интерфейсу системных вызовов для того, чтобы отдать приказ ядру выполнить определенные действия от их имени. Некоторые библиотечные вызовы предоставляют функции, для которых отсутствует системный вызов, и поэтому обращение к ядру — это только один этап в более сложной функции. Давайте рассмотрим всем известную функцию `printf()`. Эта функции обеспечивает форматирование и буферизацию данных и лишь после этого один раз обращается к системному вызову `write()` для вывода данных на консоль. Некоторые библиотечные функции соответствуют функциям ядра один к одному. Например, библиотечная функция `open()` не делает ничего, кроме выполнения системного вызова `open()`. В то же время некоторые библиотечные функции, как, например, `strcpy()`, надо полагать, вообще не используют обращения к ядру. Когда прикладная программа выполняет системный вызов, то говорят, что *ядро выполняет работу от имени прикладной программы.* Более того, говорят, что прикладная программа *выполняет системный вызов в пространстве ядра,* а ядро выполняется в *контексте процесса.* Такой тип взаимодействия, когда прикладная программа *входит в* ядро через интерфейс системных вызовов, является фундаментальным способом выполнения задач.

В функции ядра входит также управление системным аппаратным обеспечением. Практически все платформы, включая те, на которых работает операционная система Linux, используют *прерывания* (interrupt). Когда аппаратному устройству необходимо как-то взаимодействовать с системой, оно генерирует прерывание, которое прерывает работу ядра в асинхронном режиме[5].

[5] Инымисловами, заранее неизвестно, в какой момент времени это событие произойдет и в каком состоянии будет система в этот момент времени. - Прим*. *перев.*

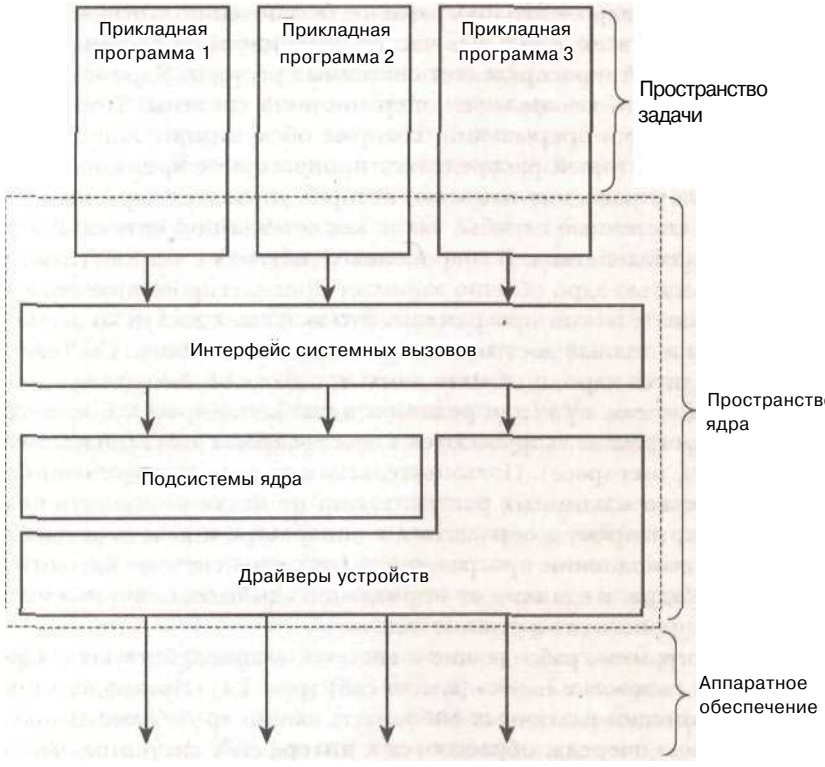


Рис. *1.1. Взаимодействие между прикладными программами, ядром и аппаратным* ***обеспечением.*** .

Обычно каждому типу прерываний соответствует номер. Ядро использует номер прерывания для выполнения специального обработчика прерывания (interrupt handler), который обрабатывает прерывание и отправляет на него ответ. Например, при вводе символа с клавиатуры, контроллер клавиатуры генерирует прерывание, чтобы дать знать системе, что в буфере клавиатуры есть новые данные. Ядро определяет номер прерывания, которое пришло в систему и выполняет соответствующий обработчик прерывания. Обработчик прерывания обрабатывает данные, поступившие с клавиатуры, и даст знать контроллеру клавиатуры, что ядро готово для приема новых данных. Для обеспечения синхронизации выполнения ядро обычно может запрещать прерывания: или все прерывания, или только прерывание с определенным номером. Во многих операционных системах обработчики прерываний не выполняются в контексте процессов. Они выполняются в специальном *контексте прерывания* (interrup context), который не связан ни с одним процессом. Этот специальный контекст существует то только для того, чтобы дать обработчику прерывания возможность быстро отреагировать на прерывание и закончить работу.

Контексты выполнения заданий полностью определяют всю широту возможных действий ядра. Фактически, можно заключить, что в операционной системе Linux процессор в любой момент времени выполняет один из трех типов действий.

- Работа от имени определенного процесса в режиме ядра в контексте процесса.

- Работа по обработке прерывания в режиме ядра в контексте прерывания, не связанном с процессами.
- Выполнение кода пользовательской программы в режиме задачи.

Ядро Linux в сравнении с классическими ядрами Unix

Благодаря общему происхождению и одинаковому API, современные ядра Unix имеют некоторые общие характерные черты. За небольшими исключениями ядра Unix представляют собой монолитные статические бинарные файлы. Это значит, что они существуют в виде больших исполняемых образов, которые выполняются один раз и используют одну копию адресного пространства. Для работы операционной системы Unix обычно требуется система с контроллером управления страничной адресацией памяти (memory management unit); это аппаратное обеспечение позволяет обеспечить защиту памяти в системе и предоставить каждому процессу уникальное виртуальное адресное пространство. В списке литературы приведены мои любимые книги по устройству классических ядер операционной системы Unix.

Сравнение решений на основе монолитного ядра и микроядра

Операционные системы, в соответствии с особенностями построения, можно разделить на две большие группы: с монолитным ядром и с микроядром. (Есть еще третий тип — экзоядро, которое пока еще используется, в основном, только в исследовательских операционных системах, но уже начинает пробивать дорогу в большой мир.)

Монолитное ядро является самым простым, и до 1980-х годов все ядра строились именно таким образом. Монолитное ядро реализовано в виде одного большого процесса, который выполняется в одном адресном пространстве, Такие ядра обычно хранятся на диске в виде одного большого статического бинарного файла. Все службы ядра существуют и выполняются в одном большом адресном пространстве ядра. Взаимодействия в ядре выполняются очень просто, потому что все, что выполняется в режиме ядра, — выполняется в одном адресном пространстве. Ядро может вызывать функции непосредственно, как это делает пользовательское приложение. Сторонники такой модели обычно указывают на простоту и высокую производительность монолитных ядер.

Микроядра не реализуются в виде одного большого процесса. Все функции ядра разделяются на несколько процессов, которые обычно называют серверами. В идеале, в привилегированном режиме работают только те серверы, которым абсолютно необходим привилегированный режим. Остальные серверы работают в пространстве пользователя. Все серверы, тем не менее, поддерживаются независимыми друг от друга и выполняются каждый в своем адресном пространстве. Следовательно, прямой вызов функций, как в случае монолитного ядра, невозможен. Все взаимодействия внутри микроядра выполняются с помощью передачи сообщений. Механизм межпроцессного взаимодействия (Inter Process Comrrmnication, IPC) встраивается в систему, и различные серверы взаимодействуют между собой и обращаются к "службам" друг друга путем отправки сообщений через механизм IPC. Разделение серверов позволяет предотвратить возможность выхода из строя одного сервера при выходе из строя другого.

Кроме того, модульность системы позволяет одному серверу вытеснять из памяти другого. Поскольку механизм IPC требует больше накладных расходов, чем обычный вызов функции, и при этом может потребоваться переключение контекста из пространства пользователя в пространство ядра и наоборот, то передача сообщений приводит к падению производительности по сравнению с монолитными ядрами, в которых используются обычные вызовы функций.

В современных операционных системах с микроядром, большинство серверов выполняется в пространстве ядра, чтобы избавиться от накладных расходов, связанных с переключением контекста, кроме того, это дает потенциальную возможность прямого вызова функций. Ядро операционной системы Windows NT, а также ядро Mach (на котором базируется часть операционной системы Mac OS X) - это примеры микроядер. В последних версиях как Windows NT, так и Mac OS X все серверы выполняются только в пространстве ядра, что является отходом от первоначальной концепции микроядра.

Ядро ОС Linux монолитное, т.е. оно выполняется в одном адресном пространстве, в режиме ядра. Тем не менее ядро Linux позаимствовало некоторые хорошие свойства микроядерной модели: в нем используется преемптивное ядро, поддерживаются потоки пространства ядра и возможность динамической загрузки в ядро внешних бинарных файлов (модулей ядра). Ядро Linux не использует никаких функций микроядерной модели, которые приводят к снижению производительности: все выполняется в режиме ядра с непосредственным вызовом функций, вместо передачи сообщений. Следовательно, операционная система Linux — модульная, многопоточная, а выполнение самого ядра можно планировать.

Прагматизм снова победил.

По мере того как Линус и другие разработчики вносили свой вклад в ядро Linux, они принимали решения о том, как развивать ОС Linux без пренебрежения корнями, связанными с Unix (и, что более важно, без пренебрежения API ОС Unix). Поскольку операционная система Linux не базируется на какой-либо версии ОС Unix, Линус и компания имели возможность найти и выбрать наилучшее решение для любой проблемы и даже со временем изобрести новые решения! Ниже приводится анализ характеристик ядра Linux, которые отличают его от других разновидностей Unix.

- Ядро Linux поддерживает динамическую загрузку модулей ядра. Хотя ядро Linux и является монолитным, оно дополнительно поддерживает динамическую загрузку и выгрузку исполняемого кода ядра по необходимости.
- Ядро Linux поддерживает симметричную многопроцессорную обработку (SMP). Хотя большинство коммерческих вариантов операционной системы Unix сейчас поддерживает SMP, большинство традиционных реализаций ОС Unix такой поддержки не имеет.
- Ядро Linux является преемптивным. В отличие от традиционных вариантов ОС Unix, ядро Linux в состоянии вытеснить выполняющееся задание, даже если это задание работает в режиме ядра. Среди коммерческих реализаций ОС Unix преемптивное ядро имеют только операционные системы Solaris и IRIX.
- В ядре Linux используется интересный подход для поддержки многопоточности (threads): потоки ни чем не отличаются от обычных процессов. С точки зрения ядра все процессы одинаковы, просто некоторые из них имеют общие ресурсы.
- В ядре Linux отсутствуют некоторые функции ОС Unix, которые считаются плохо реализованными, как, например, поддержка интерфейса STREAMS, или отвечают "глупым" стандартам.
- Ядро Linux является полностью открытым во всех смыслах этого слова. Набор функций, реализованных в ядре Linux, — это результат свободной и открытой модели разработки операционной системы Linux. Если какая-либо функция ядра считается маловажной или некачественной, то разработчики ядра

не обязаны ее реализовать. В противоположность этому, внесение изменений при разработке ядра Linux занимает "элитарную" позицию: изменения должны решать определенную практическую задачу, должны быть логичными и иметь понятную четкую реализацию. Следовательно, функции некоторых современных вариантов ОС Unix, такие как память ядра со страничной реализацией, не были реализованы. Несмотря на имеющиеся различия, Linux является операционной системой со строгим наследованием традиций ОС Unix.

Версии ядра Linux

Ядро Linux поставляется в двух вариантах-: стабильном (stable) и разрабатываемом (development). Версии стабильного ядра - это выпуски продукции промышленного уровня, которая готова для широкого использования. Новые стабильные версии ядра обычно выпускаются для исправления ошибок и для предоставления новых драйверов устройств. Разрабатываемые версии ядра, наоборот, подвержены быстрым изменениям. По мере того как разработчики экспериментируют с новыми решениями, часто вносятся радикальные изменения в ядро.

Ядра Linux стабильных и разрабатываемых версий можно отличить друг от друга с помощью простой схемы присваивания имен (рис. 1.2.). Три числа, которые разделяются точкой, определяют версию ядра. Первое число - значение старшей (major) версии, второе - значение младшей (minor), третье число - значение редакции (выпуска, revision). Значение младшей версии также определяет, является ли ядро стабильным или разрабатываемым; если это значение четное, то ядро стабильное, а если нечетное, то разрабатываемое. Так, например, версия 2.6.0 определяет стабильное ядро. Ядро имеет старшую версию 2, младшую версию 6 и редакцию 0. Первые два числа также определяют "серию ядер", в данном случае серия ядер — 2.6.

Рис. 1.2. Соглашение о присваивании имен ядрам

Разработка ядра соответствует различным фазам. Вначале разработчики ядра работают над новыми функциями, что напоминает хаос. Через определенное время ядро оказывается сформировавшимся, и в конце концов объявляется замораживание функций.

Начиная с этого момента никакие новые функции не могут быть добавлены в ядро. Однако работа над существующими функциями может быть продолжена. После того как ядро становится почти стабильным, осуществляется замораживание кода. В этом случае допускаются только исправления ошибок. Вскоре после этого (можно надеяться) ядро выпускается в виде первой, новой, стабильной версии. Например, при стабилизации серии ядер 2.5 получается серия 2.6.

Все это не правда

По крайней мере - не совсем. Приведенное только что описание процесса разработки ядра технически правильное. Раньше процесс происходил именно так, как описано. Тем не менее летом 2004 года на ежегодном саммите для приглашенных разработчиков ядра Linux было принято решение продолжить разработку серии 2.6 ядра Linux и в ближайшем будущем не переходить на серию разрабатываемого ядра 2.7. Такое решение было принято потому, что ядро 2.6 получилось хорошим; оно, в основном, стабильно и на горизонте нет никаких новых функций, которые требуют серьезного вторжения в ядро.

Кроме того, и, возможно, это главное — существующая система поддержки, которая обеспечивается Линусом Тораальдсом и Эндрю Мортоном, работает чрезвычайно хорошо. Разработчики ядра уверены, что процесс разработки может продолжаться таким образом, что серия ядер 2.6 будет оставаться стабильной и в ней будут появляться новые возможности. Время рассудит, но уже сейчас результаты выглядят хорошо.

Эта книга базируется на ядрах стабильной серии 2.6.

Сообщество разработчиков ядра Linux

Когда вы начинаете разрабатывать код ядра Linux, вы становитесь частью глобального сообщества разработчиков ядра Linux. Главный форум этого сообщества — *список рассылки разработчиков ядра Linux* (linux-kernel mailing list). Информация по поводу подписки на этот форум доступна по адресу `http://vger.kernel.org`. Следует заметить, что это достаточно перегруженный сообщения список рассылки (количество сообщений порядка 300 в день) и что другие читатели этого списка (разработчики ядра, включая Линуса) не очень склонны заниматься ерундой. Однако этот список рассылки может оказать неоценимую помощь в процессе разработки; здесь вы сможете найти тестологов, получить экспертную оценку и задать вопросы.

В последних главах приведен обзор процесса разработки ядра и более полное описание того, как успешно принимать участие в деятельности сообщества разработчиков ядра.

Перед тем как начать

Эта книга посвящена ядру Linux: как оно работает, почему оно работает и чему следует уделить внимание. Далее будут описаны принципы работы и реализация основных подсистем ядра, а также интерфейсы и программная семантика. Эта книга касается практических вопросов, и в ней используется подход на основании золотой серединки указанных выше направлений. Такой интересный подход в сочетании с анекдотами из личной практики автора и советами по хакерским приемам позволяет быть уверенным в том, что книга станет хорошим стартом.

Я надеюсь, что у читателей есть доступ к системе Linux и дереву исходного кода ядра. В идеале предполагается, что читатель— это пользователь операционной системы Linux, который уже "копался" в исходном программном коде, но все же нуждается в некоторой помощи для того, чтобы все связать воедино. В принципе, читатель может и не быть пользователем Linux, но хочет разобраться в устройстве ядра из чистого любопытства. Тем не менее, для того чтобы самому научиться писать программы — исходный код незаменим. Исходный программный код *свободно* доступен— пользуйтесь им!

Удачи!

2

Начальные сведения о ядре Linux

В этой главе будут рассмотрены основные вопросы, связанные с ядром Linux: где получить исходный код, как его компилировать и как инсталлировать новое ядро. После этого рассмотрим некоторые допущения, связанные с ядром Linux, отличия между ядром и пользовательскими программам, а также общие методы, которые используются в ядре.

Ядро имеет интересные особенности, которые отличают его от других программ, но нет таких вещей, в которых нельзя разобраться. Давайте этим займемся.

.

Получение исходного кода ядра

Исходный программный код последней версии ядра всегда доступен как в виде полного архива в формате tar (tarball), так и виде инкрементной заплаты по адресу http://www.kernel.org.

Если нет необходимости по той или другой причине работать со старыми версиями ядра, то всегда нужно использовать самую последнюю версию. Архив kernel.org — это то место, где можно найти как само ядро, так и заплаты к нему от ведущих разработчиков.

Инсталляция исходного кода ядра

Архив исходного кода ядра в формате tar распространяется в сжатых форматах GNU zip (gzip) и bzip2. Формат bzip2 наиболее предпочтителен, так как обеспечивает больший коэффициент сжатия по сравнению с форматом gzip. Архив ядра в формате bzip2 имеет имя linux-x.y.z.tar.bz2, где x, y, z — это номер соответствующей версии исходного кода ядра. После загрузки исходного кода его можно декомпрессировать очень просто. Если tar-архив сжат с помощью GNU zip, то необходимо выполнить следующую команду.

```
$ tar xvzf linux-x.y.z.tar.gz
```

Если сжатие выполнено с помощью bzip2, то команда должна иметь следующий вид.

```
$ tar xvjf linux-x.y.z.tar.bz2
```

Обе эти команды позволяют декомпрессировать и развернуть дерево исходных кодов ядра в каталог с именем linux-x.y.z.

Где лучше инсталлировать и изменять исходный код

Исходный код ядра обычно инсталлируется в каталог /usr/src/linux. Заметим, что это дерево исходного кода нельзя использовать для разработок. Версия ядра, с которой была скомпилирована ваша *библиотека C,* часто связывается с этим деревом каталогов. Кроме того, чтобы вносить изменения в ядро, не обязательно иметь права пользователя root, вместо этого лучше работать в вашем домашнем каталоге и использовать права пользователя root только для инсталляции ядра. Даже при инсталляции нового ядра каталог /usr/src/linux лучше оставлять без изменений.

Использование заплат

В сообществе разработчиков ядра Linux заплаты (patch) — это основной язык *общения*. Вы будете распространять ваши изменения исходного кода ядра в виде заплат и получать изменения кода от других разработчиков тоже в виде заплат. При данном рассмотрении наиболее важными являются *инкрементные заплаты* (incremental patch), которые позволяют перейти от одной версии ядра к другой. Вместо того чтобы загружать большой архив ядра, можно просто применить инкрементную заплату и перейти от имеющейся версии к следующей. Это позволяет сэкономить время и пропускную способность каналов связи. Для того чтобы применить инкрементную заплату, находясь в каталоге дерева исходных кодов ядра, нужно просто выполнить следующую команду.

```
$ patch -p1 < ../patch-x.y.z
```

Обычно заплата для перехода на некоторую версию ядра должна применяться к предыдущей версии ядра.

В следующих главах использование заплат рассматривается более подробно.

Дерево исходных кодов ядра

Дерево исходных кодов ядра содержит ряд каталогов, большинство из которых также содержит подкаталоги. Каталоги, которые находятся в корне дерева исходных кодов, и их описание приведены в табл. 2.1.

Некоторые файлы, которые находятся в корне дерева исходных кодов, также заслуживают внимания. Файл COPYING — это лицензия ядра (GNU GPL v2). Файл CREDITS — это список разработчиков, которые внесли большой вклад в разработку ядра. Файл MAINTAINERS — список людей, которые занимаются поддержкой подсистем и драйверов ядра. И наконец, Makefile — это основной сборочный файл ядра.

Сборка ядра

Сборка ядра достаточно проста. Это может показаться удивительным, но она даже более проста, чем компиляция и инсталляция других системных компонентов, как, например библиотеки glibc. В ядрах серии 2.6 встроена новая система конфигурации и компиляции, которая позволяет сделать эту задачу еще проще и является долгожданным улучшением по сравнению с серией ядер 2.4.

Таблица 2.1. Каталоги в корне дерева исходных кодов ядра

Каталог	Описание
arch	Специфичный для аппаратной платформы исходный код
crypto	Криптографический API
Documentation	Документация исходного кода ядра
drivers	Драйверы устройств
fs	Подсистема VFS и отдельные файловые системы
include	Заголовочные файлы ядра
init	Загрузка и инициализация ядра
ipc	Код межпроцессного взаимодействия
kernel	Основные подсистемы, такие как планировщик
lib	Вспомогательные подпрограммы
mm	Подсистема управления памятью и поддержка виртуальной памяти
net	Сетевая подсистема
scripts	Сценарии компиляции ядра
security	Модуль безопасности Linux
sound	Звуковая подсистема
usr	Начальный код пространства пользователя (initramfs)

Так как доступен исходный код ядра Linux, то, это означает, что есть возможность сконфигурировать ядро перед компиляцией. Есть возможность скомпилировать поддержку только необходимых драйверов и функций. Конфигурация ядра— необходимый этап перед тем, как его компилировать. Поскольку в ядре бесчисленное количество функций и вариантов поддерживаемого аппаратного обеспечения, возможностей по конфигурации, мягко говоря, *много.* Конфигурация управляется с помощью опций конфигурации в виде CONFIG_FEATURE. Например, поддержка симметричной многопроцессорной обработки (Symmetric multiprocessing, SMP) устанавливается с помощью опции CONFIG SMP. Если этот параметр установлен, то поддержка функций SMP включена. Если этот параметр не установлен, то функции поддержки SMP отключены. Все конфигурационные параметры хранятся в файле .config в корневом каталоге дерева исходного кода ядра и устанавливаются одной из конфигурационных программ, например, с помощью команды make xconfig. Конфигурационные параметры используются как для определения того, какие файлы должны быть скомпилированы во время сборки ядра, так и для управления процессом компиляции через директивы препроцессора.

Конфигурационные переменные бывают двух видов: логические *(boolean)* и переменные с тремя состояниями (*instate).* Логические переменные могут принимать значения *yes* и *no.* Такие переменные конфигурации ядра, как CONFIG_PREEMPT, обычно являются логическими. Конфигурационная переменная с тремя состояниями может принимать значения *yes, no* и module. Значение *module* отвечает конфигурационному параметру, который установлен, но соответствующий код должен компилироваться как модуль (т.е. как отдельный объект, который загружается динамически). Драйверы устройств обычно представляются конфигурационными переменными с тремя состояниями.

Конфигурационные параметры могут иметь целочисленный, или строковый, тип. Эти параметры не контролируют процесс сборки, а позволяют указать значения, которые встраиваются в исходный код ядра с помощью препроцессора. Например, с помощью конфигурационного параметра можно указать размер статически выделенного массива.

Ядра, которые включаются в поставки ОС Linux такими производителями, как Novell и Redhat, компилируются как часть дистрибутива. В таких ядрах обычно имеется большой набор различных функций и практически полный набор всех драйверов устройств в виде загружаемых модулей. Это позволяет получить хорошее базовое ядро и поддержку широкого диапазона оборудования. К сожалению, как разработчикам ядра, вам потребуется компилировать свои ядра и самим разбираться, какие модули включать, а какие нет.

В ядре поддерживается несколько инструментов, которые позволяют выполнять конфигурацию. Наиболее простой инструмент — это текстовая утилита командной строки:

```
make config
```

Эта утилита просматривает все параметры один за другим и интерактивно запрашивает у пользователя, какое значение соответствующего параметра установить — *yes, no* или module {для переменной с тремя состояниями). Эта операция требует *длительного* времени, и если у вас не почасовая оплата, то лучше использовать утилиту на основе интерфейса *ncurses:*

```
make menuconfig
```

или графическую утилиту на основе системы *X11:*

```
make xconfig
```

или еще более удобную графическую утилиту, основанную на библиотеке *gtk+*

```
make gconfig
```

Эти утилиты позволяют разделить все параметры по категориям, таким как Processor Features (Свойства процессора) и Network Devices (Сетевые устройства). Пользователи могут перемещаться по категориям и, конечно, изменять значения конфигурационных параметров. Команда

```
$ make defconfig
```

позволяет создать конфигурационный файл, который будет содержать параметры, используемые по умолчанию для текущей аппаратной платформы. Хотя эти параметры и достаточно общие (ходят слухи, что для аппаратной платформы i386 используется конфигурация Линуса), они являются хорошей стартовой точкой, если вы никогда перед этим не занимались конфигурацией ядра. Чтобы все сделать быстро, необходимо выполнить эту команду, а потом проверить, включена ли поддержка всех нужных аппаратных устройств.

Конфигурационные параметры содержатся в корне дерева каталогов исходного кода ядра в файле с именем `.config`. Для вас может показаться более простым, так же как и для большинства разработчиков, непосредственно редактировать этот конфигурационный файл. Достаточно легко проводить поиск в этом файле и изменять значение конфигурационных параметров. После внесения изменений в конфигура-

ционный файл или при использовании существующего конфигурационного файла для нового дерева каталогов исходного кода ядра, необходимо активизировать и обновить конфигурацию с помощью команды:

```
make oldconfig
```

Кстати, перед сборкой ядра эту команду также необходимо выполнить. После того как конфигурация ядра выполнена, можно выполнить сборку с помощью команды:

```
make
```

В отличие от предыдущих серий ядер, в версии 2.6 больше нет необходимости выполнять команду make dep перед сборкой ядра, так как создание дерева зависимостей выполняется автоматически. Также не нужно указывать цель сборки, например *bzlmage,* как это было необходимо для более ранних версий. Правило, записанное в файле с именем Makefile, которое используется по умолчанию, в состоянии обработать все!

Уменьшение количества выводимых сообщений

Для того чтобы уменьшить шум, связанный с сообщениями, которые выдаются во время сборки, но в то же время видеть предупреждения и сообщения об ошибках, можно использовать такую хитрость, как перенаправление стандартного вывода команды make (1):

```
make > "имя_некоторого_файла"
```

Если вдруг окажется необходимым просмотреть выводимые сообщения, можно воспользоваться соответствующим файлом. Но обычно, если предупреждения или сообщения об ошибках *выводятся* на экран, в этом нет необходимости.

На самом деле я выполняю следующую команду

```
make > /dev/null,
```

что позволяет совсем избавиться от ненужных сообщений.

Параллельная сборка

Программа make (1) предоставляет возможность разбить процесс сборки на несколько *заданий.* Каждое из этих заданий выполняется отдельно от остальных и параллельно с остальными, существенно ускоряя процесс сборки на многопроцессорных системах. Это также позволяет более оптимально использовать процессор, Поскольку время компиляции большого дерева исходного кода также включает время ожидания завершения ввода-вывода (время, в течение которого процесс ждет завершения операций ввода-вывода).

По умолчанию утилита make (1) запускает только одну задачу, поскольку часто файлы сборки содержат некорректную информацию о зависимостях. При неправильной информации о зависимостях несколько заданий могут начать "наступать друг другу на ноги", что приведет к ошибкам компиляции. Конечно же, в файле сборки ядра таких ошибок нет. Для компиляции ядра с использованием параллельной сборки необходимо выполнить следующую команду.

```
$ make -jn
```

где n — количество заданий, которые необходимо запустить.

Обычно запускается один или два процесса на процессор. Например, на двухпроцессорной машине можно использовать следующий запуск.

```
$ make -j4
```

Используя такие отличные утилиты, как distcc (1) и ccache(1), можно еще более существенно уменьшить время компиляции ядра.

Инсталляция ядра

После того как ядро собрано, его необходимо инсталлировать. Процесс инсталляции существенно зависит от платформы и типа системного загрузчика. Для того чтобы узнать, в какой каталог должен быть скопирован образ ядра и как установить его для загрузки, необходимо обратиться к руководству по используемому системному загрузчику. На случай если новое ядро будет иметь проблемы с работоспособностью, всегда следует сохранить одну или две копии старых ядер, которые гарантированно работоспособны!

Например, для платформы x86, при использовании системного загрузчика grub можно скопировать загружаемый образ ядра из файла arch/i386/boot/bzlmage в каталог /boot и отредактировать файл /etc/grub/grub.conf для указания записи, которая соответствует новому ядру. В системах, где для загрузки используется загрузчик LILO, необходимо соответственно отредактировать файл /etc/lilo.conf и запустить утилиту lilo (8).

Инсталляция модулей ядра автоматизирована и не зависит от аппаратной платформы. Просто нужно запустить следующую команду с правами пользователя root.

```
$ make modules_install
```

В процессе компиляции в корневом каталоге дерева исходного кода ядра также создается файл System.map. В этом файле содержится таблица соответствия символов ядра их начальным адресам в памяти. Эта таблица используется при отладке для перевода адресов памяти в имена функций и переменных.

"Зверек другого рода"

Ядро имеет некоторые отличия в сравнении с обычными пользовательскими приложениями, эти отличия хотя и не обязательно приводят к серьезным усложнениям при программировании, но все же создают специфические проблемы при разработке ядра.

Эти отличия делают ядро *зверьком другого рода*. Некоторые из старых правил при этом остаются в силе, а некоторые правила являются полностью новыми. Хотя часть различий очевидна (все знают, что ядро может делать все, что пожелает), другие различия не так очевидны. Наиболее важные отличия описаны ниже.

- Ядро не имеет доступа к библиотеке функций языка С.
- Ядро программируется с использованием компилятора GNU C.
- В ядре нет такой защиты памяти, как в режиме пользователя.
- В ядре нельзя легко использовать вычисления с плавающей точкой.
- Ядро использует стек небольшого фиксированного размера.

- Поскольку в ядре используются асинхронные прерывания, ядро является преемптивным и в ядре имеется поддержка SMP, то в ядре необходимо учитывать наличие параллелизма и использовать синхронизацию.
- Переносимость очень важна.

Давайте рассмотрим более детально все эти проблемы, так как все разработчики ядра должны постоянно помнить о них.

Отсутствие библиотеки libc

В отличие от обычных пользовательских приложений, ядро не компонуется со стандартной библиотекой функций языка С (и ни с какой другой библиотекой такого же типа). Для этого есть несколько причин, включая некоторые ситуации с дилеммой о курице и яйце, однако первопричина — скорость выполнения и объем кода. Полная библиотека функций языка С, и даже только самая необходимая ее часть, очень большая и неэффективная для ядра.

При этом не нужно расстраиваться, так как многие из функций библиотеки языка С реализованы в ядре. Например, обычные функции работы со строками описаны в файле `lib/string.c`. Необходимо лишь подключить заголовочный файл `<linux/string.h>` и пользоваться этими функциями.

> **Заголовочные файлы**
>
> Заметим, что упомянутые заголовочные файлы и заголовочные файлы, которые будут упоминаться далее в этой книге, принадлежат дереву исходного кода ядра. В файлах исходного кода ядра нельзя подключать заголовочные файлы извне этого дерева каталогов, так же как и нельзя использовать внешние библиотеки,

Отсутствует наиболее известная функция `printf()`. Ядро не имеет доступа к функции `printf()`, однако ему доступна функция `printk()`. Функция `printk()` копирует форматированную строку в буфер системных сообщений ядра (kernel log buffer), который обычно читается с помощью программы syslog. Использование этой функции аналогично использованию `printf()`:

```
printk("Hello world! Строка: %s и целое число: %d\n", a_string, an_integer);
```

Одно важное отличие между `printf()` и `printk()` состоит в том, что в функции `printk()` можно использовать флаг уровня вывода. Этот флаг используется программой syslog для того, чтобы определить, нужно ли показывать сообщение ядра. Вот пример использования уровня вывода:

```
printk(KERN_ERR "Это была ошибка!\n");
```

Функция `printk()` будет использоваться на протяжении всей книги. В следующих главах приведено больше информации о функции `printk()`.

Компилятор GNU C

Как и все "уважающие себя" ядра Unix, ядро Linux написано на языке С. Может быть, это покажется неожиданным, но ядро Linux написано не на чистом языке С в стандарте ANSI С. Наоборот, где это возможно, разработчики ядра используют раз-

личные расширения языка, которые доступны с помощью средств компиляции *gcc* (GNU Compiler Collection — коллекция компиляторов GNU, в которой содержится компилятор C, используемый для компиляции ядра).

Разработчики ядра используют как расширения языка C ISO C99[1] так и расширения GNU C. Эти изменения связывают ядро Linux с компилятором gcc, хотя современные компиляторы, такие как Imel C, имеют достаточную поддержку возможностей компилятора gcc для того, чтобы ими тоже можно было компилировать ядро Linux. В ядре не используются какие-либо особенные расширения стандарта C99, и кроме того, поскольку стандарт C99 является официальной редакцией языка C, эти расширения редко приводят к возникновению ошибок в других частях кода. Более интересные и, возможно, менее знакомые отклонения от стандарта языка ANSI C связаны с расширениями GNU C. Давайте рассмотрим некоторые наиболее интересные расширения, которые могут встретиться в программном коде ядра.

Функции с подстановкой тела

Компилятор GNU C поддерживает функции с подстановкой тела (inline functions). Исполняемый код функции с подстановкой тела, как следует из названия, вставляется во все места программы, где указан вызов функции. Это позволяет избежать дополнительных затрат на вызов функции и возврат из функции (сохранение и восстановление регистров) и потенциально позволяет повысить уровень оптимизации, так как компилятор может оптимизировать код вызывающей и вызываемой функций вместе. Обратной стороной такой подстановки (ничто в этой жизни не дается даром) является увеличение объема кода, увеличение используемой памяти и уменьшение эффективности использования процессорного кэша инструкций. Разработчики ядра используют функции с подстановкой тела для небольших функций, критичных ко времени выполнения. Использовать подстановку тела для больших функций, особенно когда они вызываются больше одного раза или не слишком критичны ко времени выполнения, не рекомендуется.

Функции с подстановкой тела объявляются с помощью ключевых слов `static` и `inline` в декларации функции. Например,

```
static inline void dog(unsigned long tail_size);
```

Декларация функции должна быть описана перед любым ее вызовом, иначе подстановка тела не будет произведена. Стандартный прием — это размещение функций с подстановкой тела в заголовочных файлах. Поскольку функция объявляется как статическая (`static`), экземпляр функции без подстановки тела не создается. Если функция с подстановкой тела используется только в одном файле, то она может быть размещена в верхней части этого файла.

В ядре использованию функций с подстановкой тела следует отдавать преимущество по сравнению с использованием сложных макросов.

[1] Стандарт ISO C99 — это последняя основная версия редакции стандарта ISO C. Редакция C99 содержит многочисленные улучшения предыдущей основной редакции этого стандарта. Стандарт ISO C99 вводит поименную инициализацию полей структур и тип *complex*.

Встроенный ассемблер

Компилятор gcc C позволяет встраивать инструкции языка ассемблера в обычные функции языка C. Эта возможность, конечно, должна использоваться только в тех частях ядра, которые уникальны для определенной аппаратной платформы.

Для встраивания ассемблерного кода используется директива компилятора asm().

Ядро Limix написано на смеси языков ассемблера и C. Язык ассемблера используется в низкоуровневых подсистемах и на участках кода, где нужна большая скорость выполнения. Большая часть коду ядра написана на языке программирования C.

Аннотация ветвлений

Компилятор gnu C имеет встроенные директивы, позволяющие оптимизировать различные ветви условных операторов, которые наиболее или наименее вероятны. Компилятор использует эти директивы для соответственной оптимизации кода. В ядре эти директивы заключаются в макросы likely() и unlikely(), которые легко использовать. Например, если используется оператор if следующего вида:

```
if  (foo) {
/*..*/
}
```

то для того, чтобы отметить этот путь выполнения как маловероятный, необходимо указать:

```
/* предполагается, что значение переменной foo равно нулю ..*/
if (unllkely(ffoo)) {

/*..*/
}
```

И наоборот, чтобы отметить этот путь выполнения как наиболее вероятный

```
/* предполагается, что значение переменной foo не равно нулю ..*/
if (likely(foo)) {
/*..*/
}
```

Эти директивы необходимо использовать только в случае, когда направление ветвления с большой вероятностью известно априори или когда необходима оптимизация какой-либо части кода за счет другой части. Важно помнить, что эти директивы дают увеличение производительности, когда направление ветвления предсказано правильно, однако приводят к потере производительности при неправильном предсказании. Наиболее часто директивы unlikely () и likely () используются для проверки ошибок.

Отсутствие защиты памяти

Когда прикладная программа предпринимает незаконную попытку обращения к памяти, ядро может перехватить эту ошибку и аварийно завершить соответствующий процесс. Если ядро предпринимает попытку некорректного обращения к памяти, то результаты могут быть менее контролируемы. Нарушение правил доступа к памяти в режиме ядра приводит к ошибке *oops,* которая является наиболее часто встреча-

ющейся ошибкой ядра. Не стоит говорить, что нельзя обращаться к запрещенным областям памяти, разыменовывать указатели со значением NULL и так далее, однако в ядре ставки значительно выше!

Кроме того, память ядра не использует замещение страниц. Поэтому каждый байт памяти, который использован в ядре, — это еще один байт доступной физической памяти. Это необходимо помнить всякий раз, когда добавляются новые *функции ядра.*

Нельзя просто использовать вычисления с плавающей точкой

Когда пользовательская программа использует вычисления с плавающей точкой, ядро управляет переходом из режима работы с целыми числами в режим работы с плавающей точкой. Операции, которые ядро должно выполнить для использования инструкций работы с плавающей точкой, зависят от аппаратной платформы.

В отличие от режима задачи, в режиме ядра нет такой роскоши, как прямое использование вычислений с плавающей точкой. Активизация режима вычислений с плавающей точкой в режиме ядра требует сохранения и восстановления регистров устройства поддержки вычислений с плавающей точкой вручную, кроме прочих рутинных операций. Если коротко, то можно посоветовать: *не нужно этого делать;* никаких вычислений с плавающей точкой в режиме ядра.

Маленький стек фиксированного размера

Пользовательские программы могут "отдохнуть" вместе со своими тоннами статически выделяемых переменных в стеке, включая структуры большого размера и многоэлементные массивы. Такое поведение является законным в режиме задачи, так как область стека пользовательских программ может динамически увеличиваться в размере (разработчики, которые писали программы под старые и не очень интеллектуальные операционные системы, как, например, DOS, могут вспомнить то время, когда даже стек пользовательских программ имел фиксированный размер).

Стек, доступный в режиме ядра, не является ни большим, ни динамически изменяемым, он мал по объему и имеет фиксированный размер. Размер стека зависит от аппаратной платформы. Для платформы x86 размер стека может быть сконфигурирован на этапе компиляции и быть равным 4 или 8 Кбайт. Исторически так сложилось, что размер стека ядра равен двум страницам памяти, что соответствует 8 Кбайт для 32-разрядных аппаратных платформ и 16 Кбайт — для 64-разрядных. Этот размер фиксирован. Каждый процесс получает свою область стека.

Более подробное обсуждение использования стека в режиме ядра смотрите в следующих главах.

Синхронизация и параллелизм

Ядро подвержено состояниям конкуренции за ресурсы (race condition). В отличие от однопоточной пользовательской программы, ряд свойств ядра позволяет осуществлять параллельные обращения к ресурсам общего доступа, и поэтому требуется выполнять синхронизацию для предотвращения состояний конкуренции за ресурсы. В частности, возможны следующие ситуации.

- Ядро Linux поддерживает многопроцессорную обработку. Поэтому, без соответствующей защиты, код ядра может выполняться на одном, двух или большем количестве процессоров и при этом одновременно обращаться к одному ресурсу.
- Прерывания возникают асинхронно по отношению к исполняемому коду. Поэтому, без соответствующей защиты, прерывания могут возникнуть во время обращения к ресурсу общего доступа, и обработчик прерывания может тоже обратиться к этому же ресурсу.
- Ядро Linux является преемптивным. Поэтому, без соответствующей защиты, исполняемый код ядра может быть вытеснен в пользу другого кода ядра, который тоже может обращаться к некоторому общему ресурсу.

Стандартное решение для предотвращения состояния конкуренции за ресурсы (состояния гонок) — это использование спин-блокировок и семафоров.

Более полное обсуждение вопросов синхронизации и параллелизма приведено в следующих главах.

Переносимость — это важно

При разработке пользовательских программ переносимость *не всегда* является целью, однако операционная система Linux является переносимой и должна оставаться такой. Это означает, что платформо-независимый код, написанный на языке С, должен компилироваться без ошибок и правильно выполняться на большом количестве систем.

Несколько правил, такие как не создавать зависимости от порядка следования байтов, обеспечивать возможность использования кода для 64-битовых систем, не привязываться к размеру страницы памяти или машинного слова и другие— имеют большое значение. Эти вопросы более подробно освещаются в одной из следующих глав.

Резюме

Да, ядро— это действительно нечто иное: отсутствует защита памяти, нет проверенной библиотеки функций языка С, маленький стек, большое дерево исходного кода. Ядро Linux играет по своим правилам и занимается серьезными вещами. Тем не менее, ядро— это всего лишь программа; оно, по сути, не сильно отличается от других обычных программ. Не нужно его бояться.

Понимание того, что ядро не так уж страшно, как кажется, может стать первым шагом к пониманию того, что все *имеет свой смысл.* Однако чтобы достичь этой утопии, необходимо стараться, читать исходный код, изменять его и не падать духом.

Вводный материал, который был представлен в первой главе, и базовые моменты, которые описаны в текущей, надеюсь, станут хорошим фундаментом для тех знаний, которые будут получены при прочтении всей книги. В следующих разделах будут рассмотрены конкретные подсистемы ядра и принципы их работы.

3

Управление процессами

Процесс - одно из самых важных абстрактных понятий в Unix-подобных операционных системах[1]. По сути, процесс— это программа, т.е. объектный код, хранящийся на каком-либо носителе информации и находящийся в состоянии исполнения. Однако процесс — это не только исполняемый программный код, который для операционной системы Unix часто называется *text section (сегмент текста* или *сегмент кода).* Процессы также включают в себя *сегмент данных (data section),* содержащий глобальные переменные; набор ресурсов, таких как открытые файлы и ожидающие на обработку сигналы; адресное пространство и один или более *потоков выполнения.* Процесс — это живой результат выполнения программного кода.

Потоки выполнения, которые часто для сокращения называют просто потоками (*thread),* представляют собой объекты, выполняющие определенные операции внутри процесса. Каждый поток включает в себя уникальный счетчик команд (program counter), стек выполнения и набор регистров процессора. Ядро планирует выполнение отдельных потоков, а не процессов. В традиционных Unix-подобных операционных системах каждый процесс содержал только один поток. Однако в современных системах многопоточные программы используются очень широко. Как будет показано далее, в операционной системе Linux используется уникальная реализация потоков — между процессами и потоками нет никакой разницы. Поток в операционной системе Linux — это специальный тип процесса.

В современных операционных системах процессы предусматривают наличие двух виртуальных ресурсов: виртуального процессора и виртуальной памяти. Виртуальный процессор создает для процесса иллюзию, что этот процесс монопольно использует всю компьютерную систему, за исключением, может быть, только того, что физическим процессором совместно пользуются десятки других процессов. В главе 4, "Планирование выполнения процессов", эта виртуализация обсуждается более подробно. Виртуальная память предоставляет процессу иллюзию того, что он один располагает всей памятью компьютерной системы. Виртуальной памяти посвящена глава 11, "Управление памятью". Потоки *совместно* используют одну и ту же виртуальную память, хотя каждый поток получает свой виртуальный процессор.

---

[1] Другая абстракция — это файл.

Следует подчеркнуть, что сама по себе программа процессом не является; процесс — это *выполняющаяся* программа плюс набор соответствующих ресурсов. Конечно, может существовать два процесса, которые исполняют *одну и ту же* программу. В действительности может даже существовать два или больше процессов, которые совместно используют одни и те же ресурсы, такие как открытые файлы, или адресное пространство. Процесс начинает свое существование с момента создания, что впрочем не удивительно. В операционной системе Linux такое создание выполняется с помощью системного вызова `fork()` (буквально, ветвление или вилка), который создает новый процесс путем полного копирования уже существующего. Процесс, который вызвал системную функцию `fork()`, называется *порождающим (родительским, pannt),* новый процесс именуют *порожденным (дочерний, child).* Родительский процесс после этого продолжает выполнение, а порожденный процесс начинает выполняться с места возврата из системного вызова. Часто после разветвления в одном из процессов желательно выполнить какую-нибудь другую программу. Семейство функций `exec*()` позволяет создать новое адресное пространство и загрузить в него новую программу. В современных ядрах Linux функция `fork()` реализована через системный вызов `clone()`, который будет рассмотрен в следующем разделе.

Выход из программы осуществляется с помощью системного вызова `exit()`. Эта функция завершает процесс и освобождает все занятые им ресурсы. Родительский процесс может запросить о состоянии порожденных им процессов с помощью системного вызова `wait4()`[2], который заставляет один процесс ожидать завершения другого. Когда процесс завершается, он переходит в специальное состояние *зомби (zombie),* которое используется для представления завершенного процесса до того момента, пока порождающий его процесс не вызовет системную функцию `wait()` или `waitpid()`.

Иное название для процесса — *задание или задача (task).* О процессах в ядре операционной системы Linux говорят как о задачах. В этой книге оба понятия взаимозаменяемы, хотя по возможности для представления работающей программы в ядре будет использоваться термин *задача,* а для представления в режиме пользователя — термин *процесс.*

Дескриптор процесса и структура task structure

Ядро хранит информацию о всех процессах в двухсвязном списке, который называется *task list*[3] *(список задач).* Каждый элемент этого списка является *дескриптором процесса* и имеет тип структуры `struct task_struct`, которая описана в файле `include/linux/sched.h`. Дескриптор процесса содержит всю информацию об определенном процессе.

[2] В ядре реализован системный вызов `wait4()`. В операционной системе Linux через библиотеку функций языка С доступны функции `wait()`, `waitpid()`, `wait3()` и `wait4()`. Все эти функции возвращают информацию о состоянии завершившегося процесса, хотя в несколько разной семантике.

[3] Иногда в литературе по построению операционных систем этот список называется `task array` (массив задач). Поскольку в ядре Linux используется связанный список, а не статический массив, его называют *task list*.

Структура task_struct — достаточно большая структура данных размером порядка 1,7 Кбайт на 32-разрядной машине. Однако этот размер не такой уж большой, учитывая, что в данной структуре содержится вся информация о процессе, которая необходима ядру. Дескриптор процесса содержит данные, которые описывают выполняющуюся программу, — открытые файлы, адресное пространство процесса, ожидающие на обработку сигналы, состояние процесса и многое другое (рис. 3.1).

Выделение дескриптора процесса

Память для структуры task_struct выделяется с помощью подсистемы выделения памяти, которая называется *слябовый распределитель (slab allocator),* для возможности повторного использования объектов и раскрашивания кэша (cache coloring) (см. главу 11, "Управление памятью"). В ядрах до серии 2.6 структура task_struct хранилась в конце стека ядра каждого процесса. Это позволяет для аппаратных платформ, у которых достаточно мало регистров процессора (как, например, платформа x86), вычислять местоположение дескриптора процесса, только зная значение регистра *указателя стека (stack pointer),* без использования дополнительных регистров для хранения самого адреса этого местоположения. Так как теперь дескриптор процесса создается с помощью слябового распределителя, была введена новая структура thread_info, которая хранится в области дна стека (для платформ, у которых стек растет в сторону уменьшения значения адреса памяти) или в области вершины стека (для платформ, у которых стек растет в сторону увеличения значения адреса памяти)[4] (рис. 3.2.).

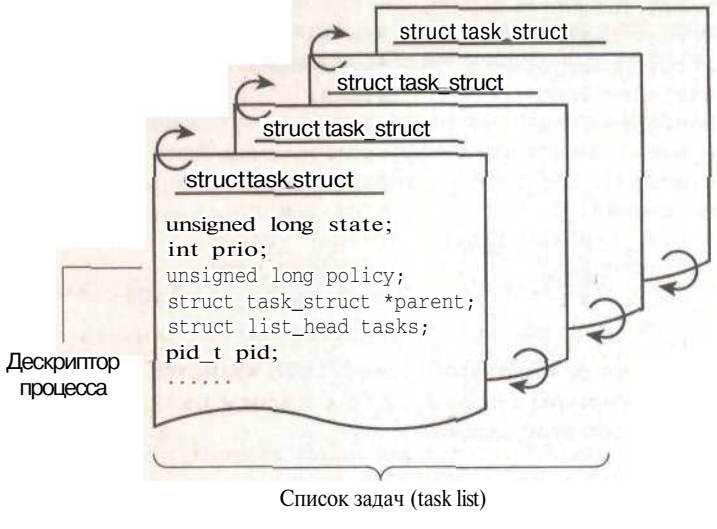


Рис. 3.1. Дескриптор процесса и список задач

[4] Причиной создания структуры *thread_info* было не только наличие аппаратных платформ, обедненных регистрами процессора, но и то, что положение этой структуры позволяет достаточно просто рассчитыпать смешения адресов для значений ее нолей при использовании языка ассемблера.

Рис 3.2. Дескриптор процесса и стек ядра

Структура `struct thread_info` для платформы x86 определена в файле <asm/thread_info.h> в следующем виде.

```
struct thread_info {
        struct task_struct      *task;
        struct exec_domain      *exec_domain;
        unsigned long           flags;
        unsigned long           status;
        __u32                   cpu;
        __s32                   preempt_count;
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
        unsigned long           previous_esp;
        __u8                    supervisorytack[0];
};
```

Для каждой задачи ее структура `thread_info` хранится в конце стека ядра этой задачи. Элемент структуры `thread_info` с именем `task` является указателем на структуру `task_struct` этой задачи.

Хранение дескриптора процесса

Система идентифицирует процессы с помощью уникального значения, которое называется *идентификатором процесса (process identification, PID)*. Идентификатор PID — это целое число, представленное с помощью скрытого типа `pid_t`[5], который обычно соответствует знаковому целому— `int`.

[5] Скрытый тип (opaque type) — это тип данных, физическое представление которого неизвестно или не существенно.

Однако, для обратной совместимости со старыми версиями ОС Unix и Linux максимальное значение этого параметра по умолчанию составляет всего лишь 32768 (что соответствует типу данных short int). Ядро хранит значение данного параметра в поле pid дескриптора процесса.

Это максимальное значение является важным, потому что оно определяет максимальное количество процессов, которые одновременно могут существовать в системе. Хотя значения 32768 и достаточно для офисного компьютера, для больших серверов может потребоваться значительно больше процессов. Чем меньше это значение, тем скорее нумерация процессов будет начинаться сначала, что приводит к нарушению полезного свойства: больший номер процесса соответствует процессу, который запустился позже. Если есть желание нарушить в системе обратную совместимость со старыми приложениями, то администратор может увеличить это максимальное значение во время работы системы с помощью записи его в файл /proc/sys/kernel/pid_max.

Обычно в ядре на задачи ссылаются непосредственно с помощью указателя на их структуры task_struct. И действительно, большая часть кода ядра, работающего с процессами, работает прямо со структурами task_struct. Следовательно, очень полезной возможностью было бы быстро находить дескриптор процесса, который выполняется в данный момент, что и делается с помощью макроса current. Этот макрос должен быть отдельно реализован для всех поддерживаемых аппаратных платформ. Для одних платформ указатель на структуру task_struct процесса, выполняющегося в данный момент, хранится в регистре процессора, что обеспечивает более эффективный доступ. Для других платформ, у которых доступно меньше регистров процессора, чтобы зря не тратить регистры, используется тот факт, что структура thread_info хранится в стеке ядра. При этом вычисляется положение структуры thread_info, а вслед за этим и адрес структуры task_struct процесса.

Для платформы x86 значение параметра current вычисляется путем маскирования 13 младших бит указателя стека для получения адреса структуры thread_info. Это может быть сделано с помощью функции current_thread_info (). Соответствующий код на языке ассемблера показан ниже.

```
movl $-8192, %eax
andl %esp, %eax
```

Окончательно значение параметра current получается путем разыменования значения поля task полученной структуры thread_info:

```
current_thread_info()->task;
```

Для контраста можно сравнить такой подход с используемым на платформе PowerPC (современный процессор на основе RISC-архитектуры фирмы IBM), для которого значение переменной current хранится в регистре процессора r2. На платформе PPC такой подход можно использовать, так как, в отличие от платформы x86, здесь регистры процессора доступны в изобилии. Так как доступ к дескриптору процесса — это очень частая и важная операция, разработчики ядра для платформы PPC сочли правильным пожертвовать одним регистром для этой цели.

Состояние процесса

Поле state дескриптора процесса описывает текущее состояние процесса (рис. 3-3). Каждый процесс в системе гарантированно находится в одном из пяти различных состояний.

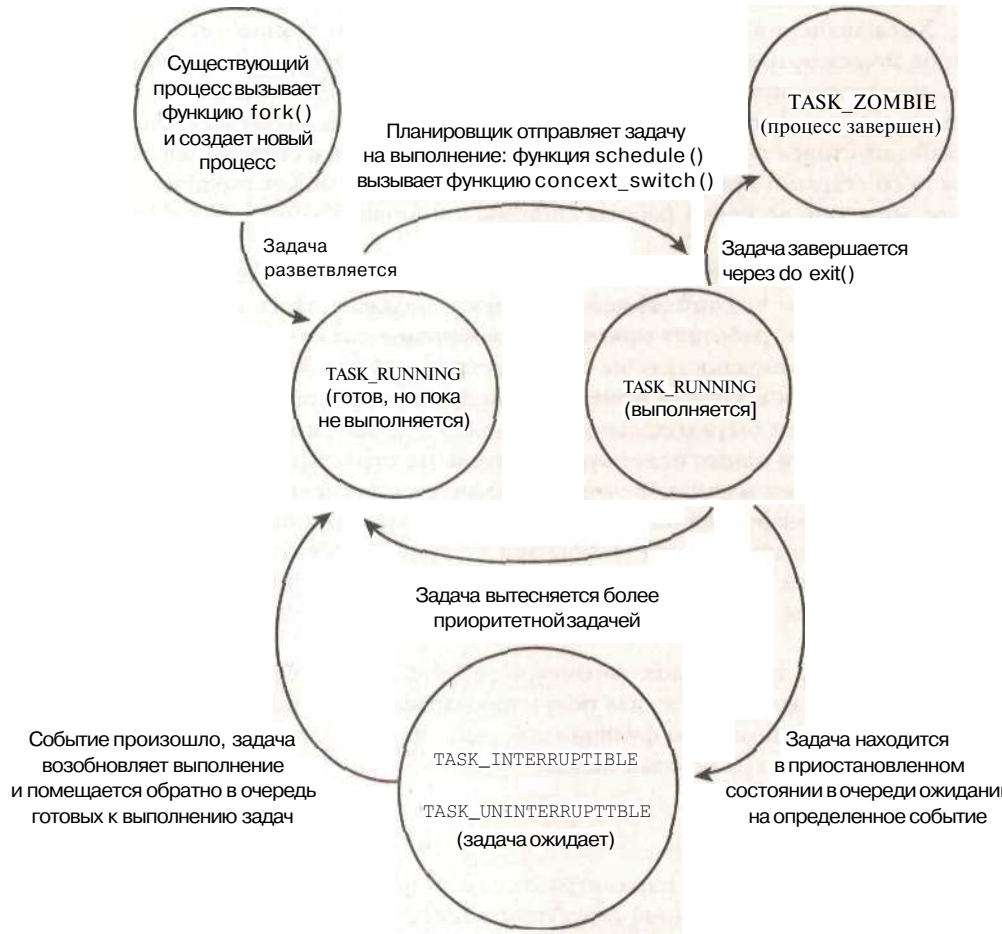


Рис. 3.3. Диаграмма состояний процесса

Эти состояния представляются значением одного из пяти возможных флагов, описанных ниже.

- TASK_RUNNING— процесс готов к выполнению (runnable). Иными словами, либо процесс выполняется в данный момент, либо находится в одной из очередей процессов, ожидающих на выполнение (эти очереди, runqueue, обсуждаются в главе 4. "Планирование выполнения процессов").
- TASK_INTERRUPTIBLE — процесс приостановлен (находится в состоянии ожидания, *sleeping),* т.е. заблокирован в ожидании выполнения некоторого

условия. Когда это условие выполнится, ядро переведет процесс в состояние TASK_RUNNING. Процесс также возобновляет выполнение *(wake up)* преждевременно при получении им сигнала.

- TASK_UNNTERRUPTIBLE - аналогично TASK_INTERRUPTIBLE, за исключением того, что процесс *не* возобновляет выполнение при получении сигнала. Используется в случае, когда процесс должен ожидать беспрерывно или когда ожидается, что некоторое событие может возникать достаточно часто. Так как задача в этом состоянии не отвечает на сигналы, TASK_UNINTERRUPTIBLE используется менее часто, чем TASK_INTERRUPTIBLE[6].
- TASK_ZOMBIE — процесс завершен, однако порождающий его процесс еще не вызвал системный вызов wait4 (). Дескриптор такого процесса должен оставаться доступным на случай, если родительскому процессу потребуется доступ к этому дескриптору. Когда родительский процесс вызывает функцию wait4 (), то такой дескриптор освобождается.
- TASK_STOPPED — выполнение процесса остановлено. Задача не выполняется и не имеет право выполняться. Такое может случиться, если задача получает какой-либо из сигналов SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU, а также если сигнал приходит в тот момент, когда процесс находится в состоянии отладки.

Манипулирование текущим состоянием процесса

Исполняемому коду ядра часто необходимо изменять состояние процесса. Наиболее предпочтительно для Этого использовать функцию

```
set_task state(task, state);
/* установить задание 'task' в состояние 'state' */
```

которая устанавливает указанное состояние для указанной задачи. Если применимо, то эта функция также пытается применить *барьер памяти (memory barrier),* чтобы гарантировать доступность установленного состояния для всех процессоров (необходимо только для SMP-систем). В других случаях это эквивалентно выражению:

```
task->state = state;
```

Вызов set current state (state) является синонимом к вызову set_task_ state(current, state).

Контекст процесса

Одна из наиболее важных частей процесса— это исполняемый программный код. Этот код считывается из *выполняемого файла (executable)* и выполняется в адресном пространстве процесса. Обычно выполнение программы осуществляется в *пространстве пользователя.* Когда программа выполняет системный вызов (см. главу 5, "Системные вызовы") или возникает исключительная ситуация, то программа входит в *пространство ядра.*

[6]Именно из-за этого появляются наводящие ужас "неубиваемые" процессы, для которых команда ps *(1)* показывает значение состояния, равное D, Так как процесс не отвечает на сигналы, ему нельзя послать сигнал *SIGKILL.* Более того, завершать такой процесс было бы неразумно, так как этот процесс, скорее всего, выполняет какую-либо важную операцию и может удерживать семафор.

С этого момента говорят, что ядро "выполняется от имени процесса" и делает это в *контексте процесса*. В контексте процесса макрос current является действительным[1]. При выходе из режима ядра процесс продолжает выполнение в пространстве пользователя, если в это время не появляется готовый к выполнению более приоритетный процесс. В таком случае активизируется планировщик, который выбирает для выполнения более приоритетный процесс.

Системные вызовы и обработчики исключительных ситуаций являются строго определенными интерфейсами ядра. Процесс может начать выполнение в пространстве ядра только посредством одного из этих интерфейсов — любые обращения к ядру возможны только через эти интерфейсы.

Дерево семейства процессов

В операционной системе Linux существует четкая иерархия процессов. Все процессы являются потомками процесса init, значение идентификатора PID для которого равно 1. Ядро запускает процесс init на последнем шаге процедуры загрузки системы. Процесс init, в свою очередь, читает системные *файлы сценариев начальной загрузки (initscripts)* и выполняет другие программы, что в конце концов завершает процедуру загрузки системы.

Каждый процесс в системе имеет всего один порождающий процесс. Кроме того, каждый процесс может иметь один или более порожденных процессов. Процессы, которые порождены одним и тем же родительским процессом, называются *родственными (siblings)*. Информация о взаимосвязи между процессами хранится в дескрипторе процесса. Каждая структура task_struct содержит указатель на структуру task_struct родительского процесса, который называется parent, эта структура также имеет список порожденных процессов, который называется children. Следовательно, если известен текущий процесс (current), то для него можно определить дескриптор родительского процесса с помощью выражения:

```
struct task_struct *task = current->parent;
```

Аналогично можно выполнить цикл по процессам, порожденным от текущего процесса, с помощью кода:

```
struct task_struct *task;
struct list_head *list;

list_for_each(list, scurrent->children) {
        task = list_entry(list, struct task_struct, sibling);
        /* переменная task теперь указывает на один из процессов,
           порожденных текущим процессом */
}
```

Дескриптор процесса init — это статически выделенная структура данных с именем inittask. Хороший пример использования связей между всеми процессами — это приведенный ниже код, который всегда выполняется успешно.

[1]Отличным от контекста процесса является контекст прерывания, описанный в главе 6, "Прерывания и обработка прерываний". В контексте прерывания система работает не от имени процесса, а выполняет обработчик прерывания. С обработчиком прерывании не связан ни один процесс, поэтому и контекст процесса отсутствует.

```
struct task_struct *task

for (task = current; task ! = $init_task; task = task->parent)

/* переменная task теперь указывает на процесс init */
```

Конечно, проходя по иерархии процессов, можно перейти от одного процесса системы к другому. Иногда, однако, желательно выполнить цикл по *всем* процессам системы. Такая задача решается очень просто, так как список задач — это двухсвязный список. Для того чтобы получить указатель на следующее задание из этого списка, имея действительный указатель на дескриптор какого-либо процесса, можно использовать показанный ниже код:

```
list_entry(task->tasks.next, struct task_struct, tasks)
```

Получение указателя на предыдущее задание работает аналогично.

```
list_entry (task->tasks.prev, struct task_struct, tasks)
```

Дна указанных выше выражения доступны также в виде макросов next_task (task) (получить следующую задачу), prev_task (task) (получить предыдущую задачу). Наконец, макрос for_each_process (task) позволяет выполнить цикл по всему списку задач. На каждом шаге цикла переменная task указывает на следующую задачу из списка:

```
struct task_struct *task;

for_each_process(task) {
        /* просто печатается имя команды и идентификатор PID
           для каждой задачи */
        printk("%s[%d]\n", task->comm, task->pid);
}
```

Следует заметить, что организация цикла по всем задачам системы, в которой выполняется много процессов, может быть достаточно дорогостоящей операцией. Для применения такого кода должны быть веские причины (и отсутствовать другие альтернативы).

Создание нового процесса

В операционной системе Unix создание процессов происходит уникальным образом. В большинстве операционных систем для создания процессов используется метод *порождения* процессов *(spawn).* При этом создается новый процесс в новом адресном пространстве, в которое считывается исполняемый файл, и после этого начинается исполнение процесса. В ОС Unix используется другой подход, а именно разбиение указанных выше операций на две функции: fork () и exec ()[8].

[8] Под exec() будем понимать любую функцию из семейства exec*(). В ядре реализован системный вызов execve(), на основе которого реализованы библиотечные функции execlp(), execle(), execv() и execvp().

В начале с помощью функции fork() создается порожденный процесс, который является копией текущего задания. Порожденный процесс отличается от родительского только значением идентификатора PID (который является уникальным в системе), значением параметра PPID (идентификатор PID родительского процесса, который устанавливается в значение PID порождающего процесса), некоторыми ресурсами, такими как ожидающие на обработку сигналы (которые не наследуются), а также статистикой использования ресурсов- Вторая функция — exec () — загружает исполняемый файл в адресное пространство процесса и начинает исполнять его. Комбинация функций fork () и exec () аналогична той одной функции создания процесса, которую предоставляет большинство операционных систем.

Копирование при записи

Традиционно при выполнении функции fork() делался дубликат всех ресурсов родительского процесса и передавался порожденному. Такой подход достаточно наивный и неэффективный. В операционной системе Linux вызов fork () реализован с использованием механизма *копирования при записи (copy-on-write)* страниц памяти. Технология копирования при записи (copy-on-write, *COW)* позволяет отложить или вообще предотвратить копирование данных. Вместо создания дубликата адресного пространства процесса родительский и порожденный процессы могут совместно использовать одну и ту же копию адресного пространства. Однако при этом данные помечаются особым образом, и если вдруг один из процессов начинает изменять данные, то создается дубликат данных, и каждый процесс получает уникальную копию данных. Следовательно, дубликаты ресурсов создаются только тогда, когда в эти ресурсы осуществляется запись, а до того момента они используются совместно в режиме только для чтения (read-only). Такая техника позволяет задержать копирование каждой страницы памяти до того момента, пока в эту страницу памяти не будет осуществляться запись. В случае, если в страницы памяти никогда не делается запись, как, например, при вызове функции exec () сразу после вызова fork (), то эти страницы никогда и не копируются. Единственные накладные расходы, которые вносит вызов функции fork (), — это копирование таблиц страниц родительского процесса и создание дескриптора порожденного процесса. Данная оптимизация предотвращает ненужное копирование большого количества данных (размер адресного пространства часто может быть более 10 Мбайт), так как процесс после разветвления в большинстве случаев сразу же начинает выполнять новый исполняемый образ. Эта оптимизация очень важна, потому чти идеология операционной системы Unix предусматривает быстрое выполнение процессов.

Функция fork ()

В операционной системе Linux функция fork () реализована через системный вызов clone () . Этот системный вызов может принимать в качестве аргументов набор флагов, определяющих, какие ресурсы должны быть общими (если вообще должны) у родительского и порожденного процессов. Далее в разделе "Реализация потоков в ядре Linux" об этих флагах рассказано более подробно. Библиотечные вызовы fork(),vfork() и cloned вызывают системную функцию clone () с соответствующими флагами. В свою очередь системный вызов clone () вызывает функцию ядра do_fork ().

Основную массу работы по разветвлению процесса выполняет функция do_fork (), которая определена в файле kernel/fork.c. Эта функция, в свою очередь, вызывает функцию copy_pracess () и запускает новый процесс на выполнение. Ниже описана та интересная работа, которую выполняет функция copy_process ().

- Вызывается функция dup_task_struct (), которая создает стек ядра, структуры thread_info и task_struct для нового процесса, причем все значения указанных структур данных идентичны для порождающего и порожденного процессов. На этом этапе дескрипторы родительского и порожденного процессов идентичны.
- Проверяется, не произойдет ли при создании нового процесса переполнение лимита на количество процессов для данного пользователя.
- Теперь необходимо сделать порожденный процесс отличным от родительского. При этом различные поля дескриптора порожденного процесса очищаются или устанавливаются в начальные значения. Большое количество данных дескриптора процесса является совместно используемым.
- Далее состояние порожденного процесса устанавливается в значение TASK_UNINTERRUPTIBLE, чтобы гарантировать, что порожденный процесс не будет выполняться.
- Из функции copy_process () вызывается функция copy_flags (), которая обновляет значение поля flags структуры task struct. При этом сбрасывается флаг PF_SUPERPRIV, который определяет, имеет ли процесс права суперпользователя. Флаг PF_FORKNOEXEC, который указывает на то, что процесс не вызвал функцию exec (), — устанавливается.
- Вызывается функция get_pid () , которая назначает новое значение идентификатора PID для новой задачи.
- В зависимости от значений флагов, переданных в функцию clone (), осуществляется копирование или совместное использование открытых файлов, информации о файловой системе, обработчиков сигналов, адресного пространства процесса и пространства имен *(namespace)*. Обычно эти ресурсы совместно используются потоками одного процесса. В противном случае они будут уникальными и будут копироваться на этом этапе.
- Происходит разделение оставшейся части кванта времени между родительским и порожденным процессами (это более подробно обсуждается в главе 4, "Планирование выполнения процессов").
- Наконец, происходит окончательная зачистка структур данных и возвращается указатель на новый порожденный процесс.

Далее происходит возврат в функцию do_fork () . Если возврат из функции copy_process () происходит успешно, то новый порожденный процесс возобновляет выполнение. Порожденный процесс намеренно запускается на выполнение раньше родительского[9].

[9] В действительности сейчас это работает не так, как хотелось бы, однако усилия прилагаются к тому, чтобы порожденный процесс запускался на выполнение первым.

В обычной ситуации, когда порожденный процесс сразу же вызывает функцию exec () , это позволяет избежать накладных расходов, связанных с тем, что если родительский процесс начинает выполняться первым, то он будет ожидать возможности записи в адресное пространство посредством механизма копирования при записи.

Функция vfork ()

Системный вызов vfork () позволяет получить тот же эффект, что и системный вызов fork (), за исключением того, что записи таблиц страниц родительского процесса не копируются. Вместо этого порожденный процесс запускается как отдельный поток в адресном пространстве родительского процесса и родительский процесс блокируется до того момента, пока порожденный процесс не вызовет функцию exec () или не завершится. Порожденному процессу *запрещена* запись в адресное пространство. Такая оптимизация была желанной в старые времена 3BSD, когда реализация системного вызова fork () не базировалась на технике копирования страниц памяти при записи. Сегодня, при использовании техники копирования страниц памяти при записи и запуске порожденного процесса перед родительским, единственное преимущество вызова vfork () — это отсутствие копирования таблиц страниц родительского процесса. Если когда-нибудь в операционной системе Linux будет реализовано копирование полей таблиц страниц при записи[10], то вообще не останется никаких преимуществ. Поскольку семантика функции vfork () достаточно ненадежна (что, например, будет, если вызов exec () завершится неудачно?), то было бы здорово, если бы системный вызов vfork () умер медленной и мучительной смертью. Вполне можно реализовать системный вызов vfork () через обычный вызов fork (), что действительно имело место в ядрах Linux до версии 2.2.

Сейчас системный вызов vfork () реализован через специальный флаг в системном вызове clone (), как показано ниже.

- При выполнении функции copy_process () поле vfork_done структуры task_struct устанавливается в значение NULL.
- При выполнении функции do_fvork (), если соответствующий флаг установлен, поле vfork_done устанавливается в ненулевое значение (начинает указывать на определенный адрес).
- После того как порожденный процесс в первый раз запущен, родительский процесс, вместо того чтобы возвратиться из функции copy_process () к выполнению, начинает ожидать, пока порожденный процесс не подаст ему сигнал через указатель vfork_done.
- При выполнении порожденным процессом функции mm_release () (которая вызывается, когда задание заканчивает работу со своим адресным пространством), если значение поля vfork_done не равно NULL, родительский процесс получает указанный выше сигнал.
- При возврате в функцию do_fork() родительский процесс возобновляет выполнение и выходит из этой функции.

[10] В действительности уже сейчас есть заплаты для добавления такой функции в ОС Linux. Хотя, скорее всего, возможность совместного использования таблиц страниц в ядрах серии 2.6 реализована не будет, такая возможность может появиться в будущих версиях.

Если все прошло так, как запланировано, то теперь порожденный процесс выполняется в новом адресном пространстве, а родительский процесс — в первоначальном адресном пространстве. Накладные расходы меньше, но реализация не очень привлекательна.

Реализация потоков в ядре Linux

Многопоточность — это популярная сегодня программная абстракция. Она обеспечивает выполнение нескольких потоков в совместно используемом адресном пространстве памяти. Потоки также могут совместно использовать открытые файлы и другие ресурсы. Многопоточность используется для *параллельного программирования (concurrent programming)*, что на многопроцессорных системах обеспечивает истинный *параллелизм*.

Реализация потоков в операционной системе Linux уникальна. Для ядра Linux не существует отдельной *концепции* потоков. В ядре Linux потоки реализованы так же, как и обычные процессы. В ОС Linux нет никакой особенной семантики для планирования выполнения потоков или каких-либо особенных структур данных для представления потоков. Поток— это просто процесс, который использует некоторые ресурсы совместно с другими процессами. Каждый поток имеет структуру `task_struct` и представляется для ядра обычным процессом (который совместно использует ресурсы, такие как адресное пространство, с другими процессами).

В этом смысле Linux отличается от других операционных систем, таких как Microsoft Windows или Sun Solaris, которые имеют *явные* средства поддержки потоков в ядре (в этих системах иногда потоки называются *процессами с быстрым переключением контекста, lightweight process)*. Название "процесс с быстрым переключением контекста" показывает разницу между философией Linux и других операционных систем. Для остальных операционных систем потоки— это абстракция, которая обеспечивает облегченные, более быстрые для исполнения сущности, чем обычные тяжелые процессы. Для операционной системы Linux потоки — это просто способ совместного использования ресурсов несколькими процессами (которые и так имеют достаточно малое время переключения контекста)[11].

Допустим, у нас есть процесс, состоящий из четырех потоков. В операционных системах с явной поддержкой потоков должен существовать дескриптор процесса, который далее указывает на четыре потока. Дескриптор процесса описывает совместно используемые ресурсы, такие как адресное пространство и открытые файлы. Потоки описываются ресурсами, которые принадлежат только им. В ОС Linux, наоборот, существует просто четыре процесса и, соответственно, четыре обычные структуры `task_struct`. Четыре процесса построены так, чтобы совместно использовать определенные ресурсы.

Потоки создаются так же, как и обычные задания, за исключением того, что в системный вызов `clone ()` передаются флаги с указанием, какие ресурсы должны использоваться совместно:

```
Clone (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

[11] Как пример можно привести тесты по измерению времени создания процессов (и даже потоков) и операционной системе Linux по сравнению с другими операционными системами. Результаты очень хорошие.

Результат выполнения показанного кода будет таким же, как и при выполнении обычного вызова fork (), за исключением того, что адресное пространство, ресурсы файловой системы, дескрипторы файлов и обработчики сигналов останутся общими. Другими словами, новая задача, так же как и родительский процесс, — обычные потоки. В отличие от этого, обычный вызов fork () может быть реализован следующим образом:

```
clone (SIGCHLD, 0);
```

а вызов vfork () в таком виде:

```
clone (CLONE_VFORK | CLONE_VM | SIGCHLD,   0);
```

Флаги, которые передаются в системный вызов clone () , помогают указать особенности поведения нового процесса и детализировать, какие ресурсы должны быть общими для родительского и порожденного процессов. В табл. 3.1 приведены флаги системного вызова clone () и их эффект.

Таблица 3 . 1 . Флаги системного вызова clone ()

Флаг	Описание
CLONE_FILES	Родительский и порожденный процессы совместно используют открытые файлы
CLONE_FS	Родительский и порожденный процессы совместно используют информацию о файловой системе
CLONE_IDLETASK	Установить значение PID в нуль (используется только для холостых (idle) задач)
CLONE_NEWNS	Создать новое пространство имен для порожденной задачи
CLONE_PARENT	Родительский процесс вызывающего процесса становится родительским и для порожденного
CLONE_PTRACE	Продолжить трассировку и для порожденного процесса
CLONE_SETTID	Возвратить значение идентификатора TID в пространство пользовател?
CLONE_SETTLS	Для порожденного процесса создать новую область локальных данных потока (thread local storage, TLS)
CLONE_SIGHAND	У порожденного и родительского процессов будут общие обработчики сигналов
CLONE_SYSVSEM	У родительского и порожденного процессов будет общая семантика обработки флага SEM_UNDO ДЛЯ семафоров System V
CLONE_THREAD	Родительский и порожденный процессы будут принадлежать одной группе потоков
CLONE_VFOK	Использовать v*fork ()*: родительский процесс будет находиться а приостановленном состоянии, пока порожденный процесс не возобновит его работу
CLONE_ONTRACED	Запретить родительскому процессу использование флага CLONE_PTRACE для порожденного процесса
CLONE_3T0P	Запустить процесс в состоянии TASK_STOPPED
CLONE_CHILD_CLEARTID	Очистить идентификатор TID для порожденного процесса
CLONE_CHILD_SETTID	Установить идентификатор TID для порожденного процесса
CLONE_PARENT_SETTID	Установить идентификатор TID для родительского процесса
CLONE_VM	У порожденного и родительского процессов будет общее адресное пространство

Потоки в пространстве ядра

Часто в ядре полезно выполнить некоторые операции в фоновом режиме. В ядре такая возможность реализована с помощью *потоков пространства ядра (kernel thread)* — обычных процессов, которые выполняются исключительно в пространстве ядра. Наиболее существенным отличием между потоками пространства ядра и обычными процессами является то, что потоки в пространстве ядра не имеют адресного пространства (значение указателя mm для них равно NULL). Эти потоки работают только в пространстве ядра, и их контекст не переключается в пространство пользователя. Тем не менее потоки в пространстве ядра планируются и вытесняются так же, как и обычные процессы.

В ядре Linux потоки пространства ядра выполняют определенные задания, наиболее часто используемые, — это *pdflush* и *ksoftirq*. Эти потоки создаются при загрузке системы другими потоками пространства ядра. В действительности поток в пространстве ядра может быть создан только другим потоком, работающим в пространстве ядра. Интерфейс для запуска нового потока в пространстве ядра из уже существующего потока следующий:

```
int kernel_thread(int (*fn) (void * ) , void * arg, unsigned long flags)
```

Новая задача создается с помощью обычного системного вызова clone () с соответствующими значениями флагов, указанными в параметре flags. При возврате из системного вызова родительский поток режима ядра завершается и возвращает указатель на структуру task_struct порожденного процесса. Порожденный процесс выполняет функцию, адрес которой указан в параметре fn, в качестве аргумента этой функции передается параметр arg. Для указания обычных флагов потоков пространства ядра существует флаг CLONE_KERNEL, который объединяет в себе флаги CLONE_FS, CLONE_FILES и CLONE_SIGHAND, так как большинство потоков пространства ядра должны указывать эти флаги в параметре flags.

Чаще всего поток пространства ядра продолжает выполнять свою функцию вечно (или, по крайней мере, до перегрузки системы, но когда она произойдет в случае ОС Linux- неизвестно). Функция потока обычно содержит замкнутый цикл, в котором поток пространства ядра по необходимости возобновляет выполнение, исполняет свои обязанности и снова переходит в приостановленное состояние.

В следующих главах более детально будут рассмотрены конкретные примеры потоков пространства ядра.

Завершение процесса

Как это ни грустно, но любой процесс в конечном итоге должен завершиться. Когда процесс завершается, ядро должно освободить ресурсы, занятые процессом, и оповестить процесс, который является родительским для завершившегося, о том, что его порожденный процесс, к сожалению, "умер".

Обычно уничтожение процесса происходит тогда, когда процесс вызывает системный вызов exit () явно или неявно при выходе из главной функции программы (компилятор языка С помещает вызов функции exit () после возврата из функции main ()). Процесс также может быть завершен непроизвольно. Это происходит, когда процесс получает сигнал или возникает исключительная ситуация, которую про-

цесс не может обработать или проигнорировать. Независимо от того, каким образом процесс завершается, основную массу работы выполняет функция doexec (), а именно указанные далее операции.

- Устанавливается флаг PF_EXITING в поле flags структуры task struct.
- Вызывается функция del_timer_sync (), чтобы удалить все таймеры ядра. После выхода из этой функции гарантируется, что нет никаких ожидающих таймеров и никакой обработчик таймера не выполняется.
- Если включена возможность учета системных ресурсов, занятых процессами (BSD process accounting), то вызывается функция acct_process () для записи информации об учете ресурсов, которые использовались процессом.
- Вызывается функция __exit_mm() для освобождения структуры mm_struct, занятой процессом. Если эта структура не используется больше ни одним процессом (другими словами, не является разделяемой), то она освобождается совсем.
- Вызывается функция exit_sem (). Если процесс находится в очереди ожидания на освобождение семафора подсистемы IPC, то в этой функции процесс удаляется из этой очереди.
- Вызываются функции __exit_files (), __exit_fs () , exit_namespace () и exit_signals () для уменьшения счетчика ссылок на объекты, которые отвечают файловым дескрипторам, данным по файловой системе, пространству имен и обработчикам сигналов соответственно. Если счетчик ссылок какого-либо объекта достигает значения, равного нулю, то соответствующий объект больше не используется никаким процессом и удаляется.
- Устанавливается код завершения задания, который хранится в поле exitcode структуры task struct. Значение этого кода передается как аргумент функции exit () или задается тем механизмом ядра, из-за которого процесс завершается.
- Вызывается функция exitnotify (), которая отправляет сигналы родительскому процессу завершающегося задания и назначает новый родительский процесс (reparent) для всех порожденных завершающимся заданием процессов, этим процессом становится или какой-либо один поток из группы потоков завершающегося процесса, или процесс init. Состояние завершающегося процесса устанавливается в значение TASK_ZOMBIE.
- Вызывается функция schedule () для переключения на новый процесс (см. главу 4, "Планирование выполнения процессов"). Поскольку процесс в состоянии TASK_ZOMBIE никогда не планируется на выполнение, этот код является последним, который выполняется завершающимся процессом.

Исходный код функции do_exit () описан в файле kernel/exit.c.

К этому моменту освобождены все объекты, занятые задачей (если они используются только этой задачей). Задача больше не может выполняться (действительно, у нее больше нет адресного пространства, в котором она может выполняться), а кроме того, состояние задачи — TASK_ZOMBIE. Единственные области памяти, которые теперь занимает процесс, — это стек режима ядра и слябовый объект, соответственно содержащие структуры thread_info и task_struct.

Задание завершено настолько, насколько остается возможность передать необходимую информацию родительскому процессу.

Удаление дескриптора процесса

После возврата из функции do_exit () дескриптор завершенного процесса все еще существует в системе, но процесс находится в состоянии TASK_ZOMBIE и не может выполняться. Как уже рассказывалось выше, это позволяет системе получить информацию о порожденном процессе после его завершения. Следовательно, завершение процесса и удаление его дескриптора происходят в разные моменты времени. После того как родительский процесс получил информацию о завершенном порожденном процессе, структура task_struct порожденного процесса освобождается.

Семейство функций wait () реализовано через единственный (и достаточно сложный) системный вызов wait4 (). Стандартное поведение этой функции — приостановить выполнение вызывающей задачи до тех пор, пока один из ее порожденных процессов не завершится. При этом возвращается идентификатор PID завершенного порожденного процесса. В дополнение к этому, в данную функцию передается указатель на область памяти, которая после возврата из функции будет содержать код завершения завершившегося порожденного процесса.

Когда приходит время окончательно освободить дескриптор процесса, вызывается функция release_task (), которая выполняет указанные ниже операции.

- Вызывается функция free_uid () для декремента счетчика ссылок на информацию о пользователе процесса. В системе Linux поддерживается кэш с информацией о каждом пользователе, в частности сколько процессов и открытых файлов имеет пользователь. Если счетчик ссылок достигает значения нуль, то пользователь больше не имеет запущенных процессов и открытых файлов, в результате кэш уничтожается.
- Вызывается функция unhash_process () для удаления процесса из хеш-таблицы идентификаторов процессов pidhash и удаления задачи из списка задач.
- Если задача была в состоянии трассировки *(ptrace),* то родительским для нее снова назначается первоначальный родительский процесс и задача удаляется из списка задач, которые находятся в состоянии трассировки (*pirate)* данным процессом.
- В конце концов вызывается функция put_task_struct () для освобождения страниц памяти, содержащих стек ядра процесса и структуру thread_info, а также освобождается слябовый кэш, содержащий структуру task_struct.

На данном этапе дескриптор процесса, а также все ресурсы, которые принадлежали только этому процессу, освобождены.

Дилемма "беспризорного" процесса

Если родительский процесс завершается до того, как завершаются вес его потомки, то должен существовать какой-нибудь механизм назначения нового родительского процесса для порожденных, иначе процессы, у которых нет родительского, навсегда останутся в состоянии зомби, что будет зря расходовать системную память. Решение этой проблемы было указано выше: новым родительским процессом становится или

какой-либо один поток из группы потоков завершившегося родительского процесса, или процесс init. При выполнении функции do_exit() вызывается функция notify_parent(), которая в свою очередь вызывает forget_original_parent() для осуществления переназначения родительского процесса (reparent), как показано ниже.

```
struct task_struct *p, *reaper = father;
struct list_head *list;

if (father->exit_signal != -1)
        reaper = prev_thread(reaper);
else
        reaper = child_reaper;

if (reaper == father)
        reaper = child_reaper;
```

Этот программный код присваивает переменной reaper указатель на другое задание в группе потоков данного процесса. Если в этой группе потоков нет другого задания, то переменной reaper присваивается значение переменной child_reaper, которая содержит указатель на процесс init. Теперь, когда найден подходящий родительский процесс, нужно найти все порожденные процессы и установить для них полученное значение родительского процесса, как показано ниже.

```
list_for_each(list,&father->children){
        p = list_entry(list, struct task_struct, sibling);
        reparent_thread(p, reaper, child_reaper);
}

list_for_each (list, sfather->ptrace__children) {
        p = list_entry(list, struct task:_struct, ptrace_list);
        reparent_thread(p, reaper, child_reaper);
}
```

В этом программном коде организован цикл по двум спискам: по списку порожденных процессов *child list* и по списку порожденных процессов, находящихся в состоянии трассировки другими процессами *ptraced child list.* Основная причина, по которой используется именно два списка, достаточно интересна (эта новая особенность появилась в ядрах серии 2.6). Когда задача находится в состоянии *ptrace,* для нее временно назначается родительским тот процесс, который осуществляет отладку (debugging). Когда завершается истинный родительский процесс для такого задания, то для такой дочерней задачи также нужно осуществить переназначение родительского процесса. В ядрах более ранних версий это приводило к необходимости *организации цикла по всем заданиям системы* для поиска порожденных процессов. Решение проблемы, как было указано выше, — это поддержка отдельного списка для порожденных процессов, которые находятся в состоянии трассировки, что уменьшает число операций поиска: происходит переход от поиска порожденных процессов по всему списку задач к поиску только по двум спискам с достаточно малым числом элементов.

Когда для процессов переназначение родительского процесса прошло успешно, больше нет риска, что какой-либо процесс навсегда останется в состоянии зомби.

Процесс Init периодически вызывает функцию wait () для всех своих порожденных процессов и, соответственно, удаляет все зомби-процессы, назначенные ему.

Резюме

В этой главе рассмотрена важная абстракция операционной системы — *процесс*. Здесь описаны общие свойства процессов, их назначение, а также представлено сравнение процессов и потоков. Кроме того, описывается, как операционная система Linux хранит и представляет информацию, которая относится к процессам (структуры task_struct и thread_info), как создаются процессы (вызовы clone () и fork ()), каким образом новые исполняемые образы загружаются в адресное пространство (семейство вызовов exec ()), иерархия процессов, каким образом родительский процесс собирает информацию о своих потомках (семейство функций wait ()) и как в конце концов процесс завершается (непроизвольно или с помощью вызова exit ()).

Процесс — это фундаментальная и ключевая абстракция, которая является основой всех современных операционных систем и, в конце концов, причиной, по которой вообще существуют операционные системы (чтобы выполнять программы).

В следующей главе рассказывается о планировании выполнения процессов — изящной и интересной функции ядра, благодаря которой ядро принимает решение, какие процессы должны выполняться, в какое время и в каком- порядке.

Планирование выполнения процессов

В предыдущей главе были рассмотрены процессы— абстракция операционной системы, связанная с активным программным кодом. В этой главе представлен планировщик процессов — код, который позволяет процессам выполняться.

Планировщик *(scheduler)* — это компонент ядра, который выбирает из всех процессов системы тот, который должен выполняться следующим. Таким образом, планировщик (или, как еще его называют, *планировщик выполнения процессов)* можно рассматривать как программный код, распределяющий конечные ресурсы процессорного времени между теми процессами операционной системы, которые могут выполняться. Планировщик является основой *многозадачных (multitasking)* операционных систем, таких как ОС Linux. Принимая решение о том, какой процесс должен выполняться следующим, планировщик несет ответственность за наилучшее использование ресурсов системы и создает впечатление того, что несколько процессов выполняются одновременно.

Идея, лежащая в основе планирования выполнения процессов, достаточно проста. При наличии готовых к выполнению процессов, для того чтобы лучше использовать процессорное время, необходимо, чтобы всегда выполнялся какой-нибудь процесс. Если в системе процессов больше, чем процессоров, то некоторые процессы будут выполняться не во все моменты времени. Эти процессы *готовы к выполнению (runnable).* Исходя из информации о наборе готовых к выполнению процессов, выбор того процесса, который должен выполняться в следующий момент времени, и есть то фундаментальное решение, которое принимает планировщик.

Многозадачные операционные системы— это те, которые могут выполнять попеременно или одновременно несколько процессов. На однопроцессорной машине такие системы создает иллюзию того, что несколько процессов выполняются одновременно. На многопроцессорной машине они позволяют процессам действительно выполняться параллельно на нескольких процессорах. На машинах любого типа эти системы позволяют процессам выполняться в фоновом режиме и не занимать процессорное время, если нет соответствующей работы. Такие задания, хотя и находятся в памяти, но *не готовы к выполнению.* Вместо этого данные процессы используют ядро, чтобы *блокироваться* до тех пор, пока не произойдет некоторое событие (ввод с клавиатуры, приход данных по сети, наступление некоторого момента времени в будущем и т.д.). Следовательно, ОС Linux может содержать 100 процессов в памяти, но только один из них будет в исполняемом состоянии.

Многозадачные (multitasking) операционные системы бывают двух видов: системы с *кооперативной (cooperative) многозадачностью и* системы с *вытесняющей (preemptive, преемптивной) многозадачностью.* Операционная система Linux, так же как и большинство вариантов ОС Unix и других современных операционных систем, обеспечивает вытесняющую многозадачность. В системе с вытесняющей многозадачностью решение о том, когда один процесс должен прекратить выполнение, а другой возобновить его, принимает планировщик. Событие, заключающееся в принудительном замораживании выполняющегося процесса, называется *вытеснением (preemption)* этого процесса. Период времени, в течение которого процесс выполняется перед тем, как будет вытеснен, известен заранее. Этот период называется *квантом времени (timesiice)* процесса. В действительности квант времени соответствует той *части* процессорного времени, которая выделяется процессу. С помощью управления величинами квантов времени процессов планировщик принимает также и глобальное решение о планировании работы всей системы. При этом, кроме всего прочего, предотвращается возможность монопольного использования ресурсов всей системы одним процессом. Как будет показано далее, величины квантов времени в операционной системе Linux рассчитываются динамически, что позволяет получить некоторые интересные преимущества.

В противоположность рассмотренному выше типу многозадачности, в системах с *кооперативной многозадачностью* процесс продолжает выполняться до тех пор, пока он добровольно не примет решение о прекращении выполнения. Событие, связанное с произвольным замораживанием выполняющегося процесса, называется *передачей управления (yielding).* У такого подхода очень много недостатков: планировщик не может принимать глобальные решения относительно того, сколько процессы должны выполняться; процесс может монополизировать процессор на большее время, чем это необходимо пользователю; "зависший" процесс, который никогда не передает управление системе, потенциально может привести к неработоспособности системы. К счастью, большинство операционных систем, разработанных за последнее десятилетие, предоставляют режим вытесняющей моногозадачности. Наиболее известным исключением является операционная система Mac OS версии 9 и более ранних версий. Конечно, операционная система Unix имеет вытесняющую многозадачность с момента своего создания.

При разработке ядер ОС Linux серии 2.5, планировщик ядра был полностью реконструирован. Новый тип планировщика часто называется *0(1)-плашировщиком (0(1) scheduler)* в связи с соответствующим масштабированием времени выполнения алгоритма планирования[1]. Этот планировщик позволяет преодолеть недостатки предыдущих версий планировщика ядра Linux и обеспечить расширенную функциональность, а также более высокие характеристики производительности. В этой главе будут рассмотрены основы работы планировщиков, как эти основы использованы в О(1)-планировщике, а также цели создания 0(1)-планировшика, его устройство, практическая реализация, алгоритмы работы и соответствующие системные вызовы.

[1]Обозначение О(1) — это пример обозначения "большого О". Практически, эта запись означает, что планировщик может выполнить все свои действии за постоянное время, независимо от объема входных данных. Полное объяснение того, что такое обозначение "большого О", приведено в приложении В, "Сложность алгоритмов".

Стратегия планирования

Стратегия (policy) планирования— это характеристики поведения планировщика, которые определяют, что и когда должно выполняться. Стратегия планирования определяет глобальный характер поведения системы и отвечает за оптимальное использование процессорного времени. Таким образом, это понятие очень важное.

Процессы, ограниченные скоростью ввода-вывода и скоростью процессора

Процессы можно классифицировать как те, которые *ограничены скоростью ввода-вывода (I/0-bound),* и те, которые *ограничены скоростью процессора (processor-bound)*. К первому типу относятся процессы, которые большую часть своего времени выполнения тратят на отправку запросов на ввод-вывод информации и на ожидание ответов на эти запросы. Следовательно, такие процессы часто готовы к выполнению, но могут выполняться только в течение короткого периода времени, так как в конце концов они блокируются в ожидании выполнения ввода-вывода (имеются в виду не только дисковые операции ввода-вывода, но и любой другой тип ввода-вывода информации, как, например, работа с клавиатурой).

Процессы, ограниченные скоростью процессора, наоборот, большую часть времени исполняют программный код. Такие процессы обычно выполняются до того момента, пока они не будут вытеснены, так как эти процессы не блокируются в ожидании на запросы ввода-вывода. Поскольку такие процессы не влияют на скорость ввода-вывода, то для обеспечения нормальной скорости реакции системы не требуется, чтобы они выполнялись часто. Стратегия планирования процессов, ограниченных скоростью процессора, поэтому предполагает, что такие процессы должны выполняться реже, но более продолжительный период времени. Конечно, оба эти класса процессов взаимно не исключают друг друга. Пример процесса, ограниченного скоростью процессора, — это выполнение бесконечного цикла.

Указанные классификации не являются взаимно исключающими. Процессы могут сочетать в себе оба типа поведения: сервер системы X Windows — это процесс, который одновременно интенсивно загружает процессор и интенсивно выполняет операции ввода-вывода. Некоторые процессы могут быть ограничены скоростью ввода-вывода, но время от времени начинают выполнять интенсивную процессорную работу. Хороший пример — текстовый процессор, который обычно ожидает нажатия клавиш, но время от времени может сильно загружать процессор, выполняя проверку орфографии.

Стратегия планирования операционной системы должна стремиться к удовлетворению двух несовместных условий: обеспечение высокой скорости реакции процессов (малого времени задержки, low latency) и высокой производительности (throughput). Для удовлетворения этим требованиям часто в планировщиках применяются сложные алгоритмы определения наиболее подходящего для выполнения процесса, которые дополнительно гарантируют, что все процессы, имеющие более низкий приоритет, также будут выполняться. В Unix-подобных операционных системах стратегия планирования направлена на то, чтобы процессы, ограниченные скоростью ввода-вывода, имели больший приоритет. Использование более высокого приоритета для процессов, ограниченных скоростью ввода-вывода, приводит к уве-

личению скорости реакции процессов, так как интерактивные программы обычно ограничиваются скоростью ввода-вывода. В операционной системе Linux для обеспечения хорошей скорости реакции интерактивных программ применяется оптимизация по времени оклика (обеспечение малого времени задержки), т.е. процессы, ограниченные скоростью ввода-вывода, имеют более высокий приоритет. Как будет видно далее, это реализуется таким образом, чтобы не пренебрегать и процессами, ограниченными скоростью процессора.

Приоритет процесса

Наиболее широко распространенным типом алгоритмов планирования является планирование *с управлением по приоритетам (priority-based).* Идея состоит в том, чтобы расположить процессы по порядку в соответствии с их важностью и необходимостью использования процессорного времени. Процессы с более высоким приоритетом будут выполняться раньше тех, которые имеют более низкий приоритет, в то время как процессы с одинаковым приоритетом планируются на выполнение циклически (по кругу, round-robin), т.е. периодически один за другим. В некоторых операционных системах, включая Linux, процессы с более высоким приоритетом получают также и более длительный квант времени. Процесс, который готов к выполнению, у которого еще не закончился квант времени и который имеет наибольший приоритет, будет выполняться всегда. Как операционная система, так и пользователь могут устанавливать значение приоритета процесса и таким образом влиять на работу планировщика системы.

В операционной системе Linux используется планировщик с *динамическим управлением по приоритетам (dynamic priority-based),* который основан на такой же идее. Основной принцип состоит в том, что вначале устанавливается некоторое начальное значение приоритета, а затем планировщик может динамически уменьшать или увеличивать это значение приоритета для выполнения своих задач. Например, ясно, что процесс, который тратит много времени на выполнение операций ввода-вывода, ограничен скоростью ввода-вывода. В операционной системе Linux такие процессы получают более высокое значение динамического приоритета. С другой стороны, процесс, который постоянно полностью использует свое значение кванта времени, — это процесс, ограниченный скоростью процессора. Такие процессы получают меньшее значение динамического приоритета.

В ядре Linux используется два различных диапазона приоритетов. Первый — это параметр *nice,* который может принимать значения в диапазоне от -20 до 19, по умолчанию значение этого параметра равно 0. Большее значение параметра *nice* соответствует меньшему значению приоритета — необходимо быть более тактичным к другим процессам системы *(nice*— англ. тактичный, хороший). Процессы с меньшим значением параметра *nice* (большим значением приоритета) выполняются раньше процессов с большим значением *niie* (меньшим приоритетом). Значение параметра *nice* позволяет также определить, насколько продолжительный квант времени получит процесс. Процесс со значением параметра *nice* равным -20 получит квант времени самой большой длительности, в то время как процесс со значением параметра *nice* равным 19 получит наименьшее значение кванта времени. Использование параметра *nice* — это стандартный способ указания приоритетов процессов для всех Unix-подобных операционных систем.

Второй диапазон значений приоритетов— это приоритеты реального времени (real-time priority), которые будут рассмотрены ниже. По умолчанию диапазон значений этого параметра лежит от 0 до 99. Все процессы реального времени имеют более высокий приоритет по сравнению с обычными процессами. В операционной системе Linux приоритеты реального времени реализованы в соответствии со стандартом POSIX. В большинстве современных Unix-систем они реализованы по аналогичной схеме.

Квант времени

Квант времени (timeslice[2]) — это численное значение, которое характеризует, как долго может выполняться задание до того момента, пока оно не будет вытеснено. Стратегия планирования должна устанавливать значение кванта времени, используемое по умолчанию, что является непростой задачей. Слишком большое значение кванта времени приведет к ухудшению интерактивной производительности системы— больше не будет впечатления, что процессы выполняются параллельно. Слишком малое значение кванта времени приведет к возрастанию накладных расходов на переключение между процессами, так как больший процент системного времени будет уходить на переключение с одного процесса с малым квантом времени на другой процесс с малым квантом времени. Более того, снова возникают противоречивые требования к процессам, ограниченным скоростью ввода-вывода и скоростью процессора. Процессам, ограниченным скоростью ввода-вывода, не требуется продолжительный квант времени, в то время как для процессов, ограниченных скоростью процессора, настоятельно требуется продолжительный квант времени, например, чтобы поддерживать кэши процессора в загруженном состоянии.

На основе этих аргументов можно сделать вывод, что *любое* большое значение кванта времени приведет к ухудшению интерактивной производительности. При реализации большинства операционных систем такой вывод принимается близко к сердцу и значение кванта времени, используемое по умолчанию, достаточно мало, например равно 20 мс. Однако в операционной системе Linux используется то преимущество, что процесс с самым высоким приоритетом всегда выполняется. Планировщик ядра Linux поднимает значение приоритета для интерактивных задач, что позволяет им выполняться более часто. Поэтому в ОС Linux планировщик использует достаточно большое значение кванта времени (рис 4.1). Более того, планировщик ядра Linux динамически определяет значение кванта времени процессов в зависимости от их приоритетов. Это позволяет процессам с более высоким приоритетом, которые считаются более важными, выполняться более часто и в течение большего периода времени. Использование динамического определения величины кванта времени и приоритетов позволяет обеспечить большую устойчивость и производительность планировщика.

Следует заметить, что процесс не обязательно должен использовать весь свой квант времени за один раз. Например, процесс, у которого квант времени равен 100 мс, не обязательно должен беспрерывно выполняться в течение 100 мс, рискуя потерять всю оставшуюся неистраченную часть кианта времени. Процесс может выполняться в течение пяти периодов длительностью по 20 мс каждый.

[2]Вместо термина timeslice (квант времени) иногда также используется quantum (квант) или processor slice. В ОС Linux применяется термин timeslice.

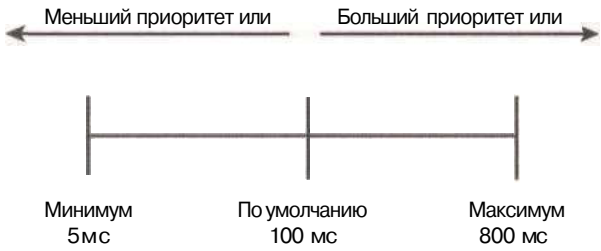


Рис. 4.1. Вычисление кванта времени процесса

Таким образом, интерактивные задачи также получают преимущество от использования продолжительного кванта времени, если даже вся продолжительность кванта времени не будет использована сразу, гарантируется, что такие процессы будут готовы к выполнению по возможности долго.

Когда истекает квант времени процесса, считается, что процесс потерял право выполняться. Процесс, у которого нет кванта времени, не имеет права выполняться до того момента, пока все другие процессы не используют свой квант времени. Когда это случится, то у всех процессов будет значение оставшегося кванта времени, равное нулю. В этот момент значения квантов времени для всех процессов пересчитываются. В планировщике ОС Linux используется интересный алгоритм для обработки ситуации, когда все процессы использовали свой квант времени. Этот алгоритм будет рассмотрен далее.

Вытеснение процесса

Как уже упоминалось, операционная система Linux использует *вытесняющую* многозадачность. Когда процесс переходит в состояние TASK_RUNNING, ядро проверяет значение приоритета этого процесса. Если это значение больше, чем приоритет процесса, который выполняется в данный момент, то активизируется планировщик, чтобы запустить новый процесс на выполнение (имеется в виду тот процесс, который только что стал готовым к выполнению). Дополнительно, когда квант времени процесса становится равным нулю, он вытесняется и планировщик готов к выбору нового процесса.

Стратегия планирования в действии

Рассмотрим систему с двумя готовыми к выполнению заданиями: программой для редактирования текстов и видеокодером. Программа для редактирования текстов ограничена скоростью ввода-вывода, потому что она тратит почти все свое время на ожидание ввода символов с клавиатуры пользователем (не имеет значение, с какой скоростью пользователь печатает, это не *те* скорости). Несмотря ни на что, при нажатии клавиши пользователь хочет, чтобы текстовый редактор отреагировал *сразу же.* В противоположность этому видеокодер ограничен скоростью процессора. Если не считать, что он время от времени считывает необработанные данные с диска и записывает результирующий видеоформат на диск, то кодер большую часть времени выполняет программу видеокодека для обработки данных, что легко загружает процессор на все 100%. Для этой программы нет строгих ограничений на время выпол-

нения: пользователю не важно, запустится она на полсекунды раньше или на полсекунды позже. Конечно, чем раньше она завершит работу, тем лучше.

В такой системе планировщик установит для текстового редактора больший приоритет и выделит более продолжительный квант времени, чем для видеокодера, так как текстовый редактор — интерактивная программа. Для текстового редактора продолжительности кванта времени хватит с избытком. Более того, поскольку текстовый редактор имеет больший приоритет, он может вытеснить процесс видеокодера при необходимости. Это гарантирует, что программа текстового редактора будет немедленно реагировать на нажатия клавиш. Однако это не причинит никакого вреда и видеокодеру, так как программа текстового редактора работает с перерывами, и во время перерывов видеокодер может монопольно использовать систему. Все это позволяет оптимизировать производительность для обоих приложений.

Алгоритм планирования

В предыдущих разделах была рассмотрена в самых общих чертах теория работы планировщика процессов в операционной системе Linux. Теперь, когда мы разобрались с основами, можно более глубоко погрузиться в то, как именно работает планировщик ОС Linux.

Программный код планировщика операционной системы Linux содержится в файле `kernel/sched.c`. Алгоритм планирования и соответствующий программный код были существенно переработаны в те времена, когда началась разработка ядер серии 2.5. Следовательно, программный код планировщика является полностью новым и отличается от планировщиков предыдущих версий. Новый планировщик разрабатывался для того, чтобы удовлетворять указанным ниже требованиям.

- Должен быть реализован полноценный O(1) -планировщик. Любой алгоритм, использованный в новом планировщике, должен завершать свою работу за постоянный период времени, независимо от числа выполняющихся процессов и других обстоятельств.
- Должна обеспечиваться хорошая масштабируемость для SMP-систем. Каждый процессор должен иметь свои индивидуальные элементы блокировок и свою индивидуальную очередь выполнения.
- Должна быть реализована улучшенная SMP-привязка (SMP affinity). Задания, для выполнения на каждом процессоре многопроцессорной системы, должны быть распределены правильным образом, и, по возможности, выполнение этих задач должно продолжаться на одном и том же процессоре. Осуществлять миграцию заданий с одного процессора на другой необходимо только для уменьшения дисбаланса между размерами очередей выполнения всех процессоров.
- Должна быть обеспечена хорошая производительность для интерактивных приложений. Даже при значительной загрузке система должна иметь хорошую реакцию и немедленно планировать выполнение интерактивных задач.
- Должна быть обеспечена равнодоступность ресурсов (fairness). Ни один процесс не должен ощущать нехватку квантов времени за допустимый период. Кроме того, ни один процесс не должен получить недопустимо большое значение кванта времени.

- Должна быть обеспечена оптимизация для наиболее распространенного случая, когда число готовых к выполнению процессов равно 1-2, но в то же время должно обеспечиваться хорошее масштабирование и на случай большого числа процессоров, на которых выполняется большое число процессов.

Новый планировщик позволяет удовлетворить всем этим требованиям.

Очереди выполнения

Основная структура данных планировщика — это *очередь выполнения (runqueue)*. Очередь выполнения определена в файле `kernel/sched.c`[3] в виде структуры `struct runqueue`. Она представляет собой список готовых к выполнению процессов для данного процессора.

Для каждого процессора определяется своя очередь выполнения. Каждый готовый к выполнению процесс может находиться в одной и только в одной очереди выполнения. Кроме этого, очередь выполнения содержит информацию, необходимую для планирования выполнения на данном процессоре. Следовательно, очередь выполнения — это действительно основная структура данных планировщика для каждого процессора. Рассмотрим нашу структуру, описание которой приведено ниже. Комментарии объясняют назначения каждого поля.

```
struct runqueue {
    spinlock_t lock; /* спин-блокировка для защиты этой очереди выполнения*/
    unsigned long nr_rinning; /* количество задач, готовых к выполнению */
    unsigned long nr_switches;     /* количество переключений контекста */
    unsigned long expired timestamp; /* время последнего обмена массивами*/
    unsigned long nr_uninterruptible; /* количество заданий в состоянии
                                         непрерываемого ожидания */
    unsigned long long timestamp last tick; /* последняя метка времени
                                               планировщика */
    struct task_struct *curr;   /* текущее задание, выполняемое на данном
                                   процессоре */
    struct task_struct *idle;   /* холостая задача данного процессора */
    struct mm_struct *prev_mm;  /* поле mm_struct последнего выполняемого
                                   задания */
    struct prio_array "active; /* указатель на активный массив приоритетов*/
    struct prio_array 'expired; /* указатель на истекший массив приоритетов*/
    struct prio_array arrays[2]; /* массивы приоритетов */
    struct task_3truct *migration_thread; /* миграционный поток для
                                             данного процессора */
    struct list_head migration_queue;     /* миграционная очередь для
                                             данного процессора */
    atomic_t nr_iowait;  /* количество заданий, ожидающих на ввод-вывод */
};
```

[3] Может возникнуть вопрос: почему используется файл *kernel/sched.c,* а не заголовочный файл *include/linux/sched.h?* Потому что желательно абстрагироваться от реализации кода планировщика и обеспечил, доступность для остального кода ядра только лишь некоторых интерфейсов.

Поскольку очередь выполнения — это основная структура данных планировщика, существует группа макросов, которые используются для доступа к определенным очередям выполнения. Макрос cpu_rq (processor) возвращает указатель на очередь выполнения, связанную с процессором, имеющим заданный номер. Аналогично макрос this_rq () возвращает указатель на очередь, связанную с текущим процессором. И наконец, макрос task_rq(task) возвращает указатель на очередь, в которой находится соответствующее задание.

Перед тем как производить манипуляции с очередью выполнения, ее необходимо заблокировать (блокировки подробно рассматриваются в главе 8, "Введение в синхронизацию выполнения кода ядра"). Так как очередь выполнения уникальна для каждого процессора, процессору редко необходимо блокировать очередь выполнения другого процессора (однако, как будет показано далее, такое все же иногда случается). Блокировка очереди выполнения позволяет запретить внесение любых изменений в структуру данных очереди, пока процессор, удерживающий блокировку, выполняет операции чтения или записи полей этой структуры. Наиболее часто встречается ситуация, когда необходимо заблокировать очередь выполнения, в которой выполняется текущее задание. В этом случае используются функции tapk_rq_lock () и task_rq_unlock(), как показано ниже.

```
struct runqueue *rq;
unsigned long flags;

rq = task_rq_lock(task, &flags);
/* здесь можно производить манипуляции с очередью выполнения */
task_rq_unlock(rq, &flags);
```

Альтернативными функциями выступают функция this_rq_lock (), которая позволяет заблокировать текущую очередь выполнения, и функция rq__unlock (struct runqueue *rq), позволяющая разблокировать указанную в аргументе очередь.

Для предотвращения взаимных блокировок код, который захватывает блокировки нескольких очередей, всегда должен захватывать блокировки в одном и том же порядке, а именно в порядке увеличения значения адреса памяти очереди (в главе 8, "Введение в синхронизацию выполнения кода ядра", приведено полное описание). Пример, как это осуществить, показан ниже.

```
/* для того, чтобы заблокировать ... */
if (rql < rq2) (
      spin_lock(s,rql->lock];
      spin_lock(Srq2->lock);
} else (
      spin_lock(Srq2->lock)   ;
      spin_lock(&rql->lock)
}

/* здесь можно манипулировать обеими очередями ... */

/• для того, чтобы разблокировать ... */
spin_unlock(brql->lock);
spin_unlock(&rq2->lock);
```

С помощью функций `double_rq_lock()` и `double_rq_unlock()` указанные шаги можно выполнить автоматически. При этом получаем следующее.

```
double_rq_lock(rq1, rq2);
/* здесь можно манипулировать обеими очередями ...*/
double_rq_unlock(rq1, rq2) ;
```

Рассмотрим небольшой пример, который показывает, почему важен порядок захвата блокировок. Вопрос взаимных блокировок обсуждается в главах 8, "Введение в синхронизацию выполнения кода ядра" и 9, "Средства синхронизации в ядре". Эта проблема касается не только очередей выполнения: вложенные блокировки должны всегда захватываться в одном и том же порядке. Спин-блокировки используются для предотвращения манипуляций с очередями выполнения несколькими задачами одновременно. Принцип работы этих блокировок аналогичен ключу, с помощью которого открывают дверь. Первое задание, которое подошло к двери, захватывает ключ, входит в дверь и закрывает ее с другой стороны. Если другое задание подходит к двери и определяет, что дверь закрыта (потому что за дверью находится первое задание), то оно должно остановиться и подождать, пока первое задание на выйдет и не возвратит ключ. Ожидание называется *спиннингом (вращением, spinning),* так как на самом деле задание постоянно выполняет цикл, периодически проверяя, не возвращен ли ключ. Теперь рассмотрим, что будет, если одно задание пытается сначала заблокировать первую очередь выполнения, а затем вторую, в то время как другое задание пытается сначала заблокировать вторую очередь, а затем — первую. Допустим, что первое задание успешно захватило блокировку первой очереди, в то время как второе задание захватило блокировку второй очереди. После этого первое задание пытается заблокировать вторую очередь, а второе задание — первую. Ни одно из заданий никогда не добьется успеха, так как другое задание уже захватило эту блокировку. Оба задания будут ожидать друг друга вечно. Так же как в тупике дороги создается блокировка движения, так и неправильный порядок захвата блокировок приводит к тому, что задания начинают ожидать друг друга вечно, и тоже возникает тупиковая ситуация, которая еще называется взаимоблокировкой. Если оба задания захватывают блокировки в одном и том же порядке, то рассмотренной ситуации произойти не может. В главах 8 и 9 представлено полное описание блокировок.

Массивы приоритетов

Каждая очередь выполнения содержит два *массива приоритетов (priority arrays):* активный и истекший. Массивы приоритетов определены в файле `kernel/sched.c` в виде описания `struct prio_array`. Массивы приоритетов — это структуры данных, которые обеспечивают 0(1)-планирование. Каждый массив приоритетов содержит для каждого значения приоритета одну очередь процессов, готовых к выполнению. Массив приоритетов также содержит *битовую маску приоритетов (priority bitmap),* используемую для эффективного поиска готового к выполнению задания, у которого значение приоритета является наибольшим в системе.

```
struct prio_array (
     int nr_active;                    /* количество заданий */
     unsigned long bitmap[BITMAP_SIZE]; /* битовая маска приоритетов */
     struct list head queue[MAX_PRIO];/* очереди приоритетов */
};
```

Константа MAX_PRIO — это количество уровней приоритета в системе. По умолчанию значение этой константы равно 140. Таким образом, для каждого значения приоритета выделена одна структура struct list_head. Константа BITMAP_SIZE — это размер массива переменных, каждый элемент которого имеет тип unsigned long. Каждый бит этого массива соответствует одному действительному значению приоритета. В случае 140 уровней приоритетов и при использовании 32-разрядных машинных слов, значение константы BITMAP_SIZE равно 5. Таким образом, поле bitmap — это массив из пяти элементов, который имеет длину 160 бит.

Все массивы приоритетов содержат поле bitmap, каждый бит этого поля соответствует одному значению приоритета в системе. В самом начале значения всех битов равны 0. Когда задача с определенным приоритетом становится готовой к выполнению (то есть значение статуса этой задачи становится равным TASK_RUNNING), соответствующий этому приоритету бит поля bitmap устанавливается в значение 1. Например, если задача с приоритетом, равным 7, готова к выполнению, то устанавливается бит номер 7. Нахождение задания с самым высоким приоритетом в системе сводится только лишь к нахождению самого первого установленного бита в битовой маске. Так как количество приоритетов неизменно, то время, которое необходимо затратить на эту операцию поиска, постоянно и не зависит от количества процессов, выполняющихся в системе. Более того, для каждой поддерживаемой аппаратной платформы в ОС Linux должен быть реализован быстрый алгоритм *поиска первого установленного бита (find first set)* для проведения быстрого поиска в битовой маске. Эта функция называется sched_find_first_bit (). Для многих аппаратных платформ существует машинная инструкция нахождения первого установленного бита в заданном машинном слове[4]. Для таких систем нахождение первого установленного бита является тривиальной операций и сводится к выполнению этой инструкции несколько раз.

Каждый массив приоритетов также содержит массив очередей, представленных структурами struct list_head. Этот массив называется queue. Каждому значению приоритета соответствует своя очередь. Очереди реализованы в виде связанных списков, и каждому значению приоритета соответствует список всех процессов системы, готовых к выполнению, имеющих это значение приоритета и находящихся в очереди выполнения данного процессора. Нахождение задания, которое будет выполняться следующим, является простой задачей и сводится к выбору следующего элемента из списка. Все задания с одинаковым приоритетом планируются на выполнение циклически.

Массив приоритетов также содержит счетчик nr_active, значение которого соответствует количеству готовых к выполнению заданий в данном массиве приоритетов.

Пересчет квантов времени

Во многих операционных системах (включая и более старые версии ОС Linux) используется прямой метод для пересчета значения кванта времени каждого задания, когда все эти значения достигают нуля.

[4] Для аппаратной платформы x86 используется инструкция *bsfl,* а для платформы PPC — инструкция *cntlzw.*

Обычно это реализуется с помощью цикла по всем задачам в системе, например, следующим образом.

```
for (каждого задания в системе) {
     пересчитать значение приоритета
     пересчитать значение кванта времени
}
```

Значение приоритета и другие атрибуты задачи используются для определения нового значения кванта времени. Такой подход имеет некоторые проблемы.

- Пересчет потенциально может занять много времени. Хуже того, время такого расчета масштабируется как O (n), где n — количество задач в системе.
- Во время пересчета должен быть использован какой-нибудь тип блокировки для защиты списка задач и отдельных дескрипторов процессов. В результате получается высокий уровень конфликтов при захвате блокировок.
- Отсутствие определенности в случайно возникающих пересчетах значений квантов времени является проблемой для программ реального времени.
- Откровенно говоря, это просто нехорошо (что является вполне оправданной причиной для каких-либо усовершенствований ядра Linux).

Новый планировщик ОС Linux позволяет избежать использования цикла пересчета приоритетов. Вместо этого в нем применяется *два* массива приоритетов для каждого процессора: *активный (active)* и *истекший (expired)*. Активный массив приоритетов содержит очередь, в которую включены все задания соответствующей очереди выполнения, для которых еще не иссяк квант времени. Истекший массив приоритетов содержит все задания соответствующей очереди, которые израсходовали свой квант времени. Когда значение кванта времени для какого-либо задания становится равным нулю, то перед тем, как поместить это задание в истекший массив приоритетов, для него вычисляется новое значение кванта времени. Пересчет значений кванта времени для всех процессов проводится с помощью перестановки активного и истекшего массивов местами. Так как на массивы ссылаются с помощью указателей, то переключение между ними будет выполняться так же быстро, как и перестановка двух указателей местами. Показанный ниже код выполняется в функции schedule ().

```
struct prio_array array = rq->active;

if (!array->nr_active) {
     rq->active = rq->expired;
     rq->expired = array;
}
```

Упомянутая перестановка и есть ключевым, моментом O(1)-планировщика. Вместо того чтобы все время пересчитывать значение приоритета и кванта времени для каждого процесса, O(1)-планировщик выполняет простую двухшаговую перестановку массивов. Такая реализация позволяет решить указанные выше проблемы.

Функция schedule ()

Все действия по выбору следующего задания на исполнение и переключение на выполнение этого задания реализованы в виде функции schedule (). Эта функция вызывается явно кодом ядра при переходе в приостановленное состояние (sleep), а также в случае когда какое-либо задание вытесняется. Функция schedule () выполняется независимо каждым процессором. Следовательно, каждый процессор самостоятельно принимает решение о том, какой процесс выполнять следующим.

Функция schedule () достаточно проста, учитывая характер тех действий, которые она выполняет. Следующий код позволяет определить задачу с наивысшим приоритетом.

```
struct task_struct *prev, *next;
struct list_head *queue;
struct prio_array *array;
int idx;

prev = current;
array = rq->active;
idx = sched_find_first_bit(array->bitmap);
queue = array->queue + idx;
next = list__entry(queue->next, struct task struct, run_ist);
```

Вначале осуществляется поиск в битовой маске активного массива приоритетов для нахождения номера самого первого установленного бита. Этот бит соответствует готовой к выполнению задаче с наивысшим приоритетом. Далее планировщик выбирает первое задание из списка заданий, которое соответствует найденному значению приоритета. Это и есть задача с наивысшим значением приоритета в системе, и эту задачу планировщик будет запускать на выполнение. Все рассмотренные операции показаны на рис. 4.2.

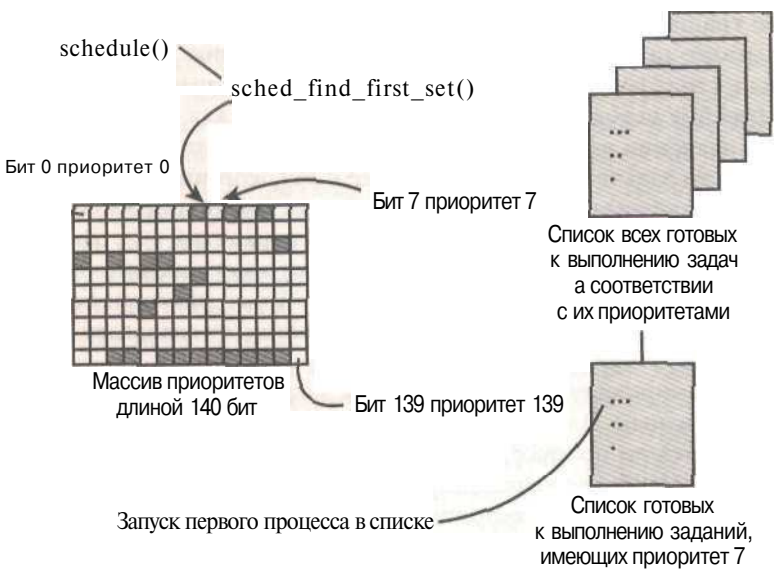


Рис. 4.2. Алгоритм работы O(1)-планировщка операционной системы Linux

Если полученные значения переменных prev и next не равны друг другу, то для выполнения выбирается новое задание (next). При этом для переключения с задания, на которое указывает переменная prev, на задание, соответствующее переменной next, вызывается функция context_switch (), зависящая от аппаратной платформы. Переключение контекста будет рассмотрено в одном из следующих разделов.

В рассмотренном коде следует обратить внимание на два важных момента. Во-первых, он очень простой и, следовательно, очень быстрый. Во-вторых, количество процессов в системе не влияет на время выполнения этого кода. Для нахождения наиболее подходящего для выполнения процесса не используются циклы. В действительности никакие факторы не влияют на время, за которое функция schedule () осуществляет поиск наиболее подходящего для выполнения задания. Время выполнения этой операции всегда постоянно.

Вычисление приоритетов и квантов времени

В начале этой главы было рассмотрено, как приоритет и квант времени используются для того, чтобы влиять на те решения, которые принимает планировщик. Кроме того, были рассмотрены процессы, ограниченные скоростью ввода-вывода и скоростью процессора, а также было описано, почему желательно поднимать приоритет интерактивных задач. Теперь давайте рассмотрим код, который реализует эти соображения.

Процессы имеют начальное значение приоритета, которое называется *nice*. Это значение может лежать в диапазоне от -20 до 19, по умолчанию используется значение 0. Значение 19 соответствует наиболее низкому приоритету, а значение -20 — наиболее высокому. Значение параметра *nice* хранится в поле static_prio структуры task_struct процесса. Это значение называется статическим приоритетом, потому что оно не изменяется планировщиком и остается таким, каким его указал пользователь. Планировщик свои решения основывает на динамическом приоритете, которое хранится в поле prio. Динамический приоритет вычисляется как функция статического приоритета и интерактивности задания.

Функция effective_prio () возвращает значение динамического приоритета задачи. Эта функция исходит из значения параметра *nice для* данной задачи и вычисляет для этого значения надбавку или штраф в диапазоне от -5 до 5, в зависимости от интерактивности задачи. Например, задание с высокой интерактивностью, которое имеет значение параметра *nice,* равное 10, может иметь динамический приоритет, равный 5. И наоборот, программа со значением параметра *nice,* равным 10, которая достаточно активно использует процессор, может иметь динамический приоритет, равный 12. Задачи, которые обладают умеренной интерактивностью, не получают ни надбавки, ни штрафа, и их динамический приоритет совпадает со значением параметра*nice*.

Конечно, планировщик по волшебству не может определить, какой процесс является интерактивным. Для определения необходима некоторая эвристика, которая отражает, является ли процесс ограниченным скоростью ввода-вывода или скоростью процессора. Наиболее выразительный показатель — сколько времени задача находится в приостановленном состоянии (sleep). Если задача проводит большую часть времени в приостановленном состоянии, то она ограничена вводом-выводом. Если задача больше времени находится в состоянии готовности к выполнению, чем в при-

остановленном состоянии, то эта задача не интерактивна. В экстремальных случаях, если задача большую часть времени находится в приостановленном состоянии, то она полностью ограничена скоростью ввода-вывода; если задача все время готова к выполнению, то она ограничена скоростью процессора.

Для реализации такой эвристики в ядре Linux предусмотрен изменяемый показатель того, как соотносится время, которое процесс проводит в приостановленном состоянии, со временем, которое процесс проводит в состоянии готовности к выполнению. Значение этого показателя хранится в поле sleep_avg структуры task_struct. Диапазон значений этого показателя лежит от нуля до значения MAX_SLEEP_AVG, которое по умолчанию равно 10 мс. Когда задача становится готовой к выполнению после приостановленного состояния, значение поля sleep_avg увеличивается на значение времени, которое процесс провел в приостановленном состоянии, пока значение sleep_avg не достигнет MAX_SLEEP_AVG. Когда задача выполняется, то в течение каждого импульса таймера (timer tick) значение этой переменной уменьшается, пока оно не достигнет значения 0.

Такой показатель является, на удивление, надежным. Он рассчитывается не только на основании того, как долго задача находится в приостановленном состоянии, но и на основании того, насколько мало задача выполняется. Таким образом, задача, которая проводит много времени в приостановленном состоянии и в то же время постоянно использует свой квант времени, не получит большой прибавки к приоритету: показатель работает не только для поощрения интерактивных задач, но и для наказания задач, ограниченных скоростью процессора. Этот показатель также устойчив по отношению к злоупотреблениям. Задача, которая получает повышенное Значение приоритета и большое значение кванта времени, быстро утратит свою надбавку к приоритету, если она постоянно выполняется и сильно загружает процессор. В конце концов, такой показатель обеспечивает малое время реакции. Только что созданный интерактивный процесс быстро достигнет высокого значения поля sleep_avg. Несмотря на все сказанное, надбавка и штраф применяются к значению параметра *nice,* так что пользователь также может влиять на работу системного планировщика путем изменения значения параметра *nice* процесса.

Расчет значения кванта времени, наоборот, более прост, так как значение динамического приоритета уже базируется на значении параметра *nice* и на интерактивности (эти показатели планировщик учитывает как наиболее важные). Поэтому продолжительность кванта времени может быть просто выражена через значение динамического приоритета. Когда создается новый процесс, порожденный и родительский процессы делят пополам оставшуюся часть кванта времени родительского процесса. Такой подход обеспечивает равнодоступность ресурсов и предотвращает возможность получения бесконечного значения кванта времени путем постоянного создания порожденных процессов. Однако после того, как квант времени задачи иссякает, это значение пересчитывается на основании динамического приоритета задачи. Функция task_timeslice () возвращает новое значение кванта времени для данного задания. Расчет просто сводится к масштабированию значения приоритета в диапазон значений квантов времени. Чем больше значение приоритета задачи, тем большей продолжительности квант времени получит задание в текущем цикле выполнения. Максимальное значение кванта времени равно MAX_TIMESLICE, которое по умолчанию равно 200 мс. Даже задания с самым низким приоритетом получают квант времени продолжительностью MIN_TIMESLICE, что соответствует 10 мс.

Задачи с приоритетом, используемым по умолчанию (значение параметра *nice,* равно О и отсутствует надбавка и штраф за интерактивность), получают квант времени продолжительностью 100 мс, как показано в табл. 4.1.

Таблица 4.1. Продолжительности квантов времени планировщика

Тип задания	Значение параметра *nice*	Продолжительность кванта времени
Вновь созданное	То же, что и у родительского процесса	Половина от родительского процесса
Минимальный приоритет	+19	5 мс (MIN_TIMESUCE)
Приоритет по умолчанию	0	100 мс (DEF_TIMESLICE)
Максимальный приоритет	-20	800 мс (MAX_TIMESLICE)

Для интерактивных задач планировщик оказывает дополнительную услугу: если задание достаточно интерактивно, то при исчерпании своего кванта времени оно будет помещено не в истекший массив приоритетов, а обратно в активный массив приоритетов. Следует вспомнить, что пересчет значений квантон времени производится путем перестановки активного и истекшего массивов приоритетов: активный массив становится истекшим, а истекший— активным. Такая процедура обеспечивает пересчет значений квантов времени, который масштабируется по времени как O(1). С другой стороны, это может привести к тому, что интерактивное задание станет готовым к выполнению, но не получит возможности выполняться, так как оно "застряло" в истекшем массиве. Помещение интерактивных заданий снова в активный массив позволяет избежать такой проблемы. Следует заметить, что это задание не будет выполняться сразу же, а будет запланировано на выполнение по кругу вместе с другими заданиями, которые имеют такой же приоритет. Данную логику реализует функция `scheduler_tick ()`, которая вызывается обработчиком прерываний таймера (обсуждается в главе 10, "Таймеры и управление временем"), как показано ниже.

```
struct task_struct *task = current;
struct runqueue *rq = this_rq();

if (!--task->time_slice) {
      if (!TASK_INTERACTIVE(task) || EXPIRED_STARVING(rq))
            enqueue_task(task, rq->expired);
      else
            enqueue_task(task, rq->active);
}
```

Показанный код уменьшает значение кванта времени процесса и проверяет, не стало ли это значение равным нулю. Если стало, то задание является истекшим и его необходимо поместить в один из массивов. Для этого код вначале проверяет интерактивность задания с помощью макроса TASK_INTERACTIVE (). Этот макрос на основании значения параметра *nice* рассчитывает, является ли задание "достаточно интерактивным". Чем меньше значение *nice* (чем выше приоритет), тем менее интерактивным должно быть задание. Задание со значением параметра *nice,* равным 19, никогда не может быть достаточно интерактивным для помещения обратно в актив-

ный массив. Наоборот, задание со значением *nice,* равным -20, должно очень сильно использовать процессор, чтобы его не поместили в активный массив. Задача со значением *nice,* используемым по умолчанию, т.е. равным нулю, должна быть достаточно интерактивной, чтобы быть помещенной обратно в активный массив, но это также отрабатывается достаточно четко. Следующий макрос, EXPIRED_STARVING (), проверяет, нет ли в истекшем массиве процессов, особенно *нуждающихся* в выполнении *(startving),* когда массивы не переключались в течение достаточно долгого времени. Если массивы давно не переключались, то обратное помещение задачи в активный массив еще больше задержит переключение, что приведет к тому, что задачи в истекшем массиве еще больше будут нуждаться в выполнении. Если это не так, то задача может быть помещена обратно в активный массив. В других случаях задача помещается в истекший массив, что встречается наиболее часто.

Переход в приостановленное состояние и возврат к выполнению

Приостановленное состояние задачи (состояние ожидания, заблокированное состояние, *sleeping, blocked)* представляет собой специальное состояние задачи, в котором задание не выполняется. Это является очень важным, так как в противном случае планировщик выбирал бы на выполнение задания, которые не "хотят" выполняться, или, хуже того, состояние ожидания должно было бы быть реализовано в виде цикла, занимающего время процессора. Задачи могут переходить в приостановленное состояние по нескольким причинам, но в любом случае— в ожидании наступления некоторого события. Событием может быть ожидание наступления некоторого момента времени, ожидание следующей порции данных при файловом вводе-ныводе или другое событие в аппаратном обеспечении. Задача также может переходить в приостановленное состояние непроизвольным образом, когда она пытается захватить семафор в режиме ядра (эта ситуация рассмотрена в главе 9, "Средства синхронизации в ядре"). Обычная причина перехода в приостановленное состояние — это выполнение операций файлового ввода-вывода, например задание вызывает функцию read () для файла, который необходимо считать с диска. Еще один пример— задача может ожидать на ввод данных с клавиатуры. В любом случае ядро ведет себя одинаково: задача помечает себя как находящуюся в приостановленном состоянии, помещает себя в очередь ожидания (wail queue), удаляет себя из очереди выполнения и вызывает функцию scheduled для выбора нового процесса на выполнение. Возврат к выполнению (wake up) происходит в обратном порядке: задача помечает себя как готовую к выполнению, удаляет себя из очереди ожидания и помепгает себя в очередь выполнения.

Как указывалось в предыдущей главе, с приостановленным состоянием связаны два значения поля состояния процесса: TASK_INTERRUPTIBLE и TASK_UNINTERRUPTIBLE. Они отличаются только тем, что в состоянии TASK_UNINTERRUPTIBLE задача игнорирует сигналы, в то время как задачи в состоянии TASK_INTERRUPTIBLE возвращаются к выполнению преждевременно и обрабатывают пришедший сигнал. Оба типа задач, находящихся в приостановленном состоянии, помещаются в очередь ожидания, ожидают наступления некоторого события и не готовы к выполнению.

Приостановленное состояние обрабатывается с помощью очередей ожидания (wait queue). Очередь ожидания— это просто список процессов, которые ожидают

наступления некоторого события. Очереди ожидания в ядре представляются с помощью типа данных wait_queue_head_t. Они могут быть созданы статически с помощью макроса DECLARE_WAIT_QUEUE_HEAD () или выделены динамически с последующей инициализацией с помощью функции init_waitqueue_head (). Процессы помещают себя в очередь ожидания и устанавливают себя в приостановленное состояние. Когда происходит событие, связанное с очередью ожидания, процессы, находящиеся в этой очереди, возвращаются к выполнению. Важно реализовать переход в приостановленное состояние и возврат к выполнению правильно, так чтобы избежать конкуренции за ресурсы (race).

Существуют простые интерфейсы для перехода в приостановленное состояние, и они широко используются. Однако использование этих интерфейсов может привести к состояниям конкуренции: возможен переход в приостановленное состояние *после* того, как соответствующее событие уже произошло. В таком случае задача может находиться в приостановленном состоянии неопределенное время. Поэтому рекомендуется следующий метод для перехода в приостановленное состояние в режиме ядра.

```
/* пусть q — это очередь ожидания (созданная в другом месте),
   где мы хотим находиться в приостановленном состоянии */
DECLARE_WAITQUEUE(wait, current);

add_wait_queue(q, &wait);
set_current_State(TASK_INTERRUPTIBLE); /* или TASK_UNINTERRUPTIBLE */
/* переменная condition характеризует наступление события,
   которого мы ожидаем */
while (!condition)
     schedule();
set_current_state(TASK_RUNNING);
remove_wait queue(q, &wait);
```

Опишем шаги, которые должна проделать задача для того, чтобы поместить себя в очередь ожидания.

- Создать элемент очереди ожидания с помощью макроса DECLARE_WAITQUEUE ().
- Добавить себя в очередь ожидания с помощью функции add wait_queue () . С помощью этой очереди ожидания процесс будет возвращен в состояние готовности к выполнению, когда условие, на выполнение которого ожидает процесс, будет выполнено. Конечно, для этого где-то в другом месте должен быть код, который вызывает функцию wake_up () для данной очереди, когда произойдет соответствующее событие.
- Изменить состояние процесса в значение TASK_INTERRUPTIBLE или TASK_ UNINTERRUPTIBLE.
- Проверить, не выполнилось ли ожидаемое условие. Если выполнилось, то больше нет необходимости переходить в приостановленное состояние. Если нет, то вызвать функцию schedule ().
- Когда задача становится готовой к выполнению, она снова проверяет выполнение ожидаемого условия. Если условие выполнено, то производится выход

из цикла. Если нет, то снова вызывается функция schedule () и повторяется проверка условия.

- Когда условие выполнено, задача может установить свое состояние в значение TASK_RUNNING и удалить себя из очереди ожидания с помощью функции remove_wait_queue().

Если условие выполнится перед тем, как задача переходит в приостановленное состояние, то цикл прервется и задача не перейдет в приостановленное состояние по ошибке. Следует заметить, что во время выполнения тела цикла код ядра часто может выполнять и другие задачи. Например, перед выполнением функции schedule () может возникнуть необходимость освободить некоторые блокировки и захватить их снова после возврата из этой функции; если процессу был доставлен сигнал, то необходимо возвратить значение -ERESTARTSYS; может возникнуть необходимость отреагировать на некоторые другие события.

Возврат к выполнению (wake up) производится с помощью функции wake_up (), которая возвращает все задачи, ожидающие в данной очереди, в состояние готовности к выполнению. Вначале вызывается функция try_to_wake_up () , которая устанавливает поле состояния задачи в значение TASK_RUNNING, далее вызывается функция activate_task () для добавления задачи в очередь выполнения и устанавливается флаг need_resched в ненулевое значение, если приоритет задачи, которая возвращается к выполнению, больше приоритета текущей задачи. Код, который отвечает за наступление некоторого события, обычно вызывает функцию wakeup () после того, как это событие произошло. Например, после того как данные прочитаны с жесткого диска, подсистема VFS вызывает функцию wake_up () для очереди ожидания, которая содержит все процессы, ожидающие поступления данных.

Важным может быть замечание о том, что переход в приостановленное состояние часто сопровождается ложными переходами к выполнению. Это возникает потому, что переход задачи в состояние выполнения не означает, что событие, которого ожидала задача, уже наступило: поэтому переход в приостановленное состояние должен всегда выполняться в цикле, который гарантирует, что условие, на которое ожидает задача, действительно выполнилось (рис. 4.3).

Балансировка нагрузки

Как уже рассказывалось ранее, планировщик операционной системы Linux реализует отдельные очереди выполнения и блокировки для каждого процессора в симметричной многопроцессорной системе. Это означает, что каждый процессор поддерживает свой список процессов и выполняет алгоритм планирования только для заданий из этого списка. Система планирования, таким образом, является уникальной для каждого процессора. Тогда каким же образом планировщик обеспечивает какую-либо глобальную стратегию планирования для многопроцессорных систем? Что будет, если нарушится балансировка очередей выполнения, скажем, в очереди выполнения одного процессора будет находиться пять процессов, а в очереди другого — всего один? Решение этой проблемы выполняется системой балансировки нагрузки, которая работает с целью гарантировать, что все очереди выполнения будут сбалансированными. Система балансировки нагрузки сравнивает очередь выполнения текущего процессора с другими очередями выполнения в системе.

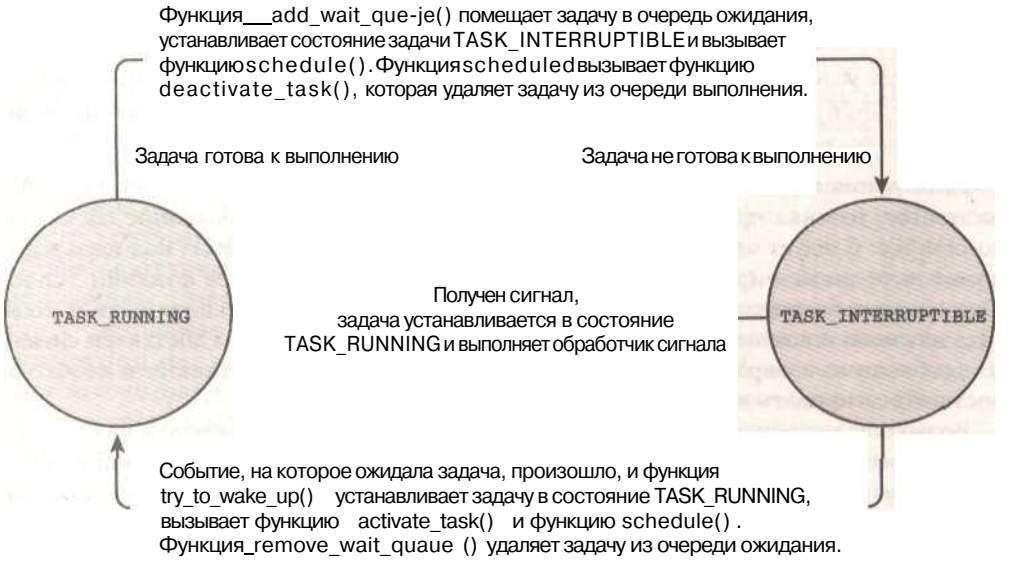


Рис. 4,3. Переход в приостановленное ***состояние*** *(sleeping) и возврат к выполнению (wake up)*

Если обнаруживается дисбаланс, то процессы из самой загруженной очереди выполнения *выталкиваются* в текущую очередь, В идеальном случае каждая очередь выполнения будет иметь одинаковое количество процессов. Такая ситуация, конечно, является высоким идеалом, к которому система балансировки может только приблизиться.

Система балансировки нагрузки реализована в файле kernel/sched.c в виде функции load_balance (). Эта функция вызывается в двух случаях. Она вызывается функцией schedule (), когда текущая очередь выполнения пуста. Она также вызывается по таймеру с периодом в 1 мс, когда система не загружена, и каждые 200 мс в другом случае. В однопроцессорной системе функция load_balance () не вызывается никогда, в действительности она даже не компилируется в исполняемый образ ядра, питому что в системе только одна очередь выполнения и никакой балансировки не нужно.

Функция балансировки нагрузки вызывается при заблокированной очереди выполнения текущего процессора, прерывания при этом также запрещены, чтобы защитить очередь выполнения от конкурирующего доступа. В том случае, когда функция load balance() вызывается из функции schedule (), цель ее вызова вполне ясна, потому что текущая очередь выполнения пуста и нахождение процессов в других очередях с последующим их проталкиванием в текущую очередь позволяет получить преимущества. Когда система балансировки нагрузки активизируется посредством таймера, то ее задача может быть не так очевидна. В данном случае это необходимо для устранения любого дисбаланса между очередями выполнения, чтобы поддерживать их в почти одинаковом состоянии, как показано на рис. 4.4.

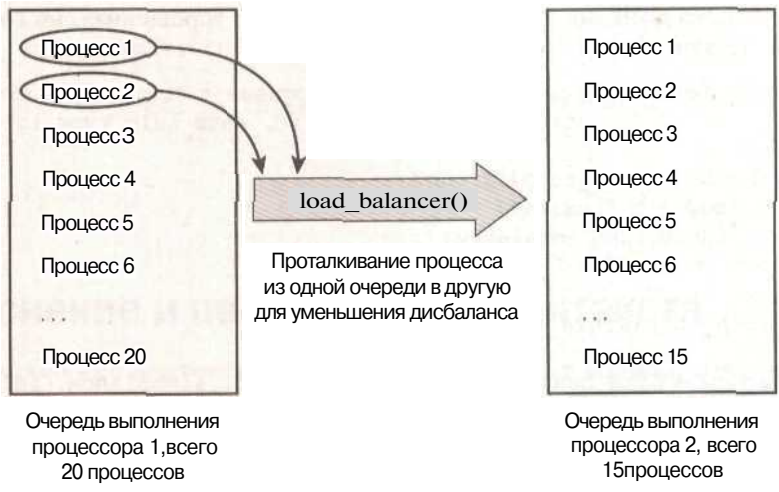


Рис. 4.4. Система балансировки нагрузки

Функция load_balance () и связанные с ней функции сравнительно большие и сложные, хотя шаги, которые они предпринимают, достаточно ясны.

- Функция load_balance () вызывает функцию find_busiest_queue () для определения наиболее загруженной очереди выполнения. Другими словами — очередь с наибольшим количеством процессов в ней. Если нет очереди выполнения, количество процессов в которой на 25% больше, чем в дайной очереди, то функция find_busiest_queue () возвращает значение NULL и происходит возврат из функции load_balance (). В другом случае возвращается указатель на самую загруженную очередь.
- Функция load_balance () принимает решение о том, из какого массива приоритетов самой загруженной очереди будут проталкиваться процессы. Истекший массив является более предпочтительным, так как содержащиеся в нем задачи не выполнялись достаточно долгое время и, скорее всего, не находятся в кэше процессора (т.е. не активны в кэше, not "cache hot"). Если истекший массив приоритетов пуст, то ничего не остается, как использовать активный массив.
- Функция load_balance () находит непустой список заданий, соответствующий самому высокому приоритету (с самым маленьким номером), так как важно более равномерно распределять задания с высоким приоритетом, чем с низким.
- Каждое задание с данным приоритетом анализируется для определения задания, которое не выполняется, не запрещено для миграции из-за процессорной привязки и не активно в кэше.Если найдена задача, которая удовлетворяет этому критерию, то вызывается функция pull_task () для проталкивания этой задачи из наиболее загруженной очереди в данную очередь.
- Пока очереди выполнения остаются разбалансированными, предыдущие два шага повторяются и необходимое количество заданий проталкивается из самой загруженной очереди выполнения в данную очередь выполнения. В конце концов, когда дисбаланс устранен, очередь выполнения разблокируется и происходит возврат из функции load_balance ().

Далее показана функция load_balance (), немного упрощенная, но содержащая все цажные детали.

```
static int load_balance(int this_cpu, runqueue_t *this_rq,
                    struct sched_doraain *sd, enum idle_type idle)
{
       struct sched_group *group;
       runqueue_t *busiest;
       unsigned long imbalance;
       int nr_moved;

       spin_lock(&this_rq->lock);

       group = find_busiest_group(sd, this_cpu, &imbalance, idle);
       if (!group)
              goto out_balanced;

       busiest = find_busiest_queue(group) ;
       if (!busiest)
              goto out_balanced;

       nr_moved = 0;
       if (busiest->nr_running > 1) {
              double_lock_balance(this_rq, busiest);
              nr_moved = move_tasks(this_rq, this_cpu, busiest,
                                    imbalance, sd, idle);
              spin_unlock(&busiest->lock);
       }
       spin_unlock(&this rq->lock);

       if (!nr_moved) {
              sd->nr_balance_failed++;

              if (unlikely(sd->nr_balance_failed > sd->cache_nice_tries+2)) {
                     int wake = 0;

                     spin_lock(abusiest->lock);
                     if (!busiest->active_balance) {
                            busiest->active_balance = 1;
                            busiest->push_cpu = this_cpu;
                            wake = 1;
                     }
                     spin_unlock(&busiest->lock);
                     if (wake)
                            wake_up_process(busiest->migration_thread);
                     sd->nr_balance_failed = sd->cache_nice_tries;
              )
       } else
              sd->nr_balance_failed = 0;
       sd->balance_interval = sd->min_interval;
       return nr_moved;
```

```
out_balanced:
        spin_unlock(&this_rq->lock);

        if (sd->balance_interval < sd->max_interval)
                sd->balance_interval *= 2;

        return 0;
}
```

Вытеснение и переключение контекста

Переключение контекста — это переключение от одной, готовой к выполнению задачи к другой. Это переключение производится с помощью функции context_switch(), определенной в файле kernel/sched.c. Данная функция вызывается функцией schedule (), когда новый процесс выбирается для выполнения. При этом выполняются следующие шаги.

- Вызывается функция switch_mm (), которая определена в файле include/asm/mmu_context.h и предназначена для переключения от виртуальной памяти старого процесса к виртуальной памяти нового процесса.
- Вызывается функция switch_to () , определенная в файле include/asm/system.h, для переключения от состояния процессора предыдущего процесса к состоянию процессора нового процесса. Эта процедура включает восстановление информации стека ядра и регистров процессора.

Ядро должно иметь информацию о том, когда вызывать функцию schedule (). Если эта функция будет вызываться только тогда, когда программный код вызывает ее явно, то пользовательские программы могут выполняться неопределенное время. Поэтому ядро поддерживает флаг need_resched для того, чтобы сигнализировать, необходимо ли вызывать функцию schedule () (табл. 4.2). Этот флаг устанавливается функцией schediiler_tick (), когда процесс истрачивает свой квант времени, и функцией try_to_wake_up (), когда процесс с приоритетом более высоким, чем у текущего процесса, возвращается к выполнению. Ядро проверяет значение этого флага, и если он установлен, то вызывается функция schedule () для переключения на новый процесс. Этот флаг является сообщением ядру о том, что планировщик должен быть активизирован по возможности раньше, потому что другой процесс должен начать выполнение.

Таблица 4.2. Функции для управления флагом need_resched

Функция	Назначение
set_tsk_need_resched (task)	Установить флаг need_resched для данного процесса
clear_tsk_need_resched (task)	Очистить флаг need_resched для данного процесса
need_resched()	Проверить значение флага need_resched для данного процесса. Возвращается значение true , если этот флаг установлен, и false , если не установлен

Во время переключения в пространство пользователи или при возврате из прерывания, значение флага need_resched проверяется. Если он установлен, то ядро активизирует планировщик перед тем, как продолжить работу.

Этот флаг не является глобальной переменной, так как обращение к дескриптору процесса получается более быстрым, чем обращение к глобальным данным (из-за скорости обращения к переменной current и потому, что соответствующие данные могут находиться в кэше). Исторически, этот флаг был глобальным в ядрах до серии 2.2. В ядрах серий 2.2 и 2.4 этот флаг принадлежал структуре task_struct и имел тип int. В серии ядер 2.6 этот флаг перемещен в один определенный бит специальной переменной флагов структуры thread info. Легко видеть, что разработчики ядра никогда не могут быть всем довольны.

Вытеснение пространства пользователя

Вытеснение пространства пользователя (user preemption) происходит в тот момент, когда ядро собирается возвратить управление режиму пользователя, при этом устанавливается флаг need_resched и, соответственно, активизируется планировщик. Когда ядро возвращает управление в пространство пользователя, то оно находится в безопасном и "спокойном" состоянии. Другими словами, если продолжение выполнения текущего задания является безопасным, то безопасным будет также и выбор нового задания для выполнения. Поэтому когда ядро готовится возвратить управление в режим пользователя или при возврате из прерывания или после системного вызова, происходит проверка флага need_resched. Если этот флаг установлен, то активизируется планировщик И выбирает новый, более подходящий процесс для исполнения. Как процедура возврата из прерывания, так и процедура возврата из системного вызова являются зависимыми от аппаратной платформы и обычно реализуются на языке ассемблера в файле entry.S (этот файл, кроме кода входа в режим ядра, также содержит и код выхода из режима ядра). Если коротко, то вытеснение пространства пользователя может произойти в следующих случаях.

- При возврате в пространство пользователя из системного вызова.
- При возврате в пространство пользователя из обработчика прерывания.

Вытеснение пространства ядра

Ядро операционной системы Linux, в отличие от ядер большинства вариантов ОС Unix, является полностью преемптивным (вытесняемым, preemptible). В непреемптивных ядрах код ядра выполняется до завершения. Иными словами, планировщик не может осуществить планирование для выполнения другого задания, пока какое-либо задание выполняется в пространстве ядра — код ядра планируется на выполнение кооперативно, а не посредством вытеснения. Код ядра выполняется до тех пор, пока он не завершится (возвратит управление в пространство пользователя) или пока явно не заблокируется. С появлением серии ядер 2.6, ядро Linux стало преемптивным: теперь есть возможность вытеснить задание в любой момент, конечно, пока ядро находится в состоянии, когда безопасно производить перепланирование выполнения.

В таком случае когда же безопасно производить перепланирование? Ядро способно вытеснить задание, работающее в пространстве ядра, когда это задание не удер-

живает блокировку. Иными словами, блокировки используются в качестве маркеров тех областей, в которые задание не может быть вытеснено. Ядро рассчитано на многопроцессорность (SMP-safe), поэтому если блокировка не удерживается, то код ядра является реентерабельным и его вытеснять безопасно.

Первое изменение, внесенное для поддержки вытеснения пространства ядра, — это введение счетчика преемптивности preempt_count в структуру thread_info каждого процесса. Значение этого счетчика вначале равно нулю и увеличивается на единицу при каждом захвате блокировки, а также уменьшается на единицу при каждом освобождении блокировки. Когда значение счетчика равно нулю— ядро является вытесняемым. При возврате из обработчика прерывания, если возврат выполняется в пространство ядра, ядро проверяет значения переменных need_resched и preempt_count. Если флаг need_resched установлен и значение счетчика preempt__count равно нулю, значит, более важное задание готово к выполнению и выполнять вытеснение безопасно. Далее активизируется планировщик. Если значение счетчика preempt_count не равно нулю, значит, удерживается захваченная блокировка и выполнять вытеснение не безопасно. В таком случае возврат из обработчика прерывания происходит в текущее выполняющееся задание. Когда освобождаются все блокировки, удерживаемые текущим заданием, значение счетчика preempt_count становится равным нулю. При этом код, осуществляющий освобождение блокировки, проверяет, не установлен ли флаг need_resched. Если установлен, то активизируется планировщик. Иногда коду ядра необходимо иметь возможность запрещать или разрешать вытеснение в режиме ядра, что будет рассмотрено в главе 9.

Вытеснение пространства ядра также может произойти явно, когда задача блокируется в режиме ядра или явно вызывается функция schedule () . Такая форма преемптивности ядра всегда поддерживалась, так как в этом случае нет необходимости в дополнительной логике, которая бы давала возможность убедиться, что вытеснение проводить безопасно. Предполагается, что если код явно вызывает функцию schedule (), то точно известно, что перепланирование производить безопасно.

Вытеснение пространства ядра может произойти в следующих случаях.

- При возврате из обработчика прерывания в пространство ядра.
- Когда код ядра снова становится преемптивным.
- Если задача, работающая в режиме ядра, явно вызывает функцию schedule ().
- Если задача, работающая в режиме ядра, переходит в приостановленное состояние, т.е. блокируется (что приводит к вызову функции schedule ()).

Режим реального времени

Операционная система Linux обеспечивает две стратегии планирования в режиме реального времени (real-lime): SCHED_FIFO и SCHED_RR. Стратегия планирования SCHED_OTHER является обычной стратегией планирования, т.е. стратегий планирования не в режиме реального времени. Стратегия SCHED_FIFO обеспечивает простой алгоритм планирования по идеологии "первым вошел — первым обслужен" (first-in first-out, FIFO) без квантов времени. Готовое к выполнению задание со стратегией планирования SCHED_FIFO всегда будет планироваться на выполнение перед всеми заданиями со стратегией планирования SCHED_OTHER. Когда задание со стратегией

SCHED_FIFO становится готовым к выполнению, то оно будет продолжать выполняться до тех пор, пока не заблокируется или пока явно не отдаст управление. Две или более задач с одинаковым приоритетом, имеющие стратегию планирования SCHED_FIFO, будут планироваться на выполнение по круговому алгоритму (round-robin). Если задание, имеющее стратегию планирования SCHED_FIFO, является готовым к выполнению, то все задачи с более низким приоритетом не могут выполняться до тех пор, пока это задание не завершится.

Стратегия SCHED_RR аналогична стратегии SCHED_FIFO, за исключением того, что процесс может выполняться только до тех пор, пока не израсходует предопределенный ему квант времени. Таким образом, стратегия SCHED_RR — это стратегия SCHED_FIFO с квантами времени, т.е. круговой алгоритм планирования (round-robin) реального времени. Когда истекает квант времени процесса со стратегией планирования SCHED_RR, то другие процессы с таким же приоритетом планируются по круговому алгоритму. Квант времени используется только для того, чтобы перепланировать выполнение заданий с таким же приоритетом. Так же как в случае стратегии SCHED_FIFO, процесс с более высоким приоритетом сразу же вытесняет процессы с более низким приоритетом, а процесс с более низким приоритетом никогда не сможет вытеснить процесс со стратегией планирования SCHED_RR, даже если у последнего истек квант времени.

Обе стратегии планирования реального времени используют статические приоритеты. Ядро не занимается расчетом значений динамических приоритетов для задач реального времени. Это означает, что процесс, работающий в режиме реального времени, *всегда* сможет вытеснить процесс с более низким значением приоритета.

Стратегии планирования реального времени в операционной системе Linux обеспечивают так называемый мягкий режим реального времени (soft real-time). Мягкий режим реального времени обозначает, что ядро пытается планировать выполнение пользовательских программ в границах допустимых временных сроков, но не всегда гарантирует выполнение этой задачи. В противоположность этому операционные системы с жестким режимом реального времени (hard real-time) всегда гарантируют выполнение всех требований по планированию выполнения процессов в заданных пределах. Операционная система Linux не может гарантировать возможности планирования задач реального времени. Тем не менее стратегия планирования ОС Linux гарантирует, что задачи реального времени будут выполняться всякий раз, когда они готовы к выполнению. Хотя в ОС Linux и отсутствуют средства, гарантирующие работу в жестком режиме реального времени, тем не менее производительность планировщика ОС Linux в режиме реального времени достаточно хорошая. Ядро серии 2.6 в состоянии удовлетворить очень жестким временным требованиям.

Приоритеты реального времени лежат в диапазоне от 1 до MAX_RT_PRIO минус 1, По умолчанию значение константы MAX_RT_PRIO равно 100, поэтому диапазон значений приоритетов реального времени по умолчанию составляет от 1 до 99. Это пространство приоритетов объединяется с пространством значений параметра *nice* для стратегии планирования SCHED_OTHER, которое соответствует диапазону приоритетов от значения MAX_RT_PRIO до значения (MAX_RT_PRIO+40). По умолчанию это означает, что диапазон значений параметра *nice* от -20 до +19 взаимно однозначно отображается в диапазон значений приоритетов от 100 до 139.

Системные вызовы для управления планировщиком

Операционная система Linux предоставляет семейство системных вызовов для управления параметрами планировщика. Эти системные вызовы позволяют манипулировать приоритетом процесса, стратегией планирования и процессорной привязкой, а также предоставляют механизм, с помощью которого можно явно *передать* процессор *(yield)* в использование другим заданиям.

Существуют различные книги, а также дружественные страницы системного руководства (man pages), которые предоставляют информацию об этих системных вызовах (реализованных в библиотеке С без особых интерфейсных оболочек, а прямым вызовом системной функции). В табл. 4.3 приведен список этих функций с кратким описанием. О том, как системные вызовы реализованы в ядре, рассказывается в главе 5, "Системные вызовы".

Таблица 4.3. Системные вызовы для управления планировщиком

Системный вызов	Описание
nice ()	Установить значение параметра nice
schedsetscheduler ()	Установить стратегию планирования
sched_getscheduler ()	Получить стратегию планирования
sched_setparam ()	Установить значение приоритета реального времени
sched_getparam ()	Получить значение приоритета реального времени
sched_get_priority_max ()	Получить максимальное значение приоритета реального времени
Eched_get_priority_min ()	Получить минимальное значение приоритета реального времени
sched_rr_get_interval ()	Получить продолжительность кванта времени
sched_setaffinity()	Установить процессорную привязку
sched_getaffinity ()	Получить процессорную привязку
sched_yield ()	Временно передать процессор другим заданиям

Системные вызовы, связанные с управлением стратегией и приоритетом

Системные вызовы sched_setscheduler () и sched_getcheduler () позволяют соответственно установить и получить значение стратегии планирования и приоритета реального времени для указанного процесса. Реализация этих функций, так же как и для большинства остальных системных вызовов, включает большое количество разнообразных проверок, инициализаций и очистку значений аргументов. Полезная работа включает в себя только чтение или запись полей policy и rt_priority структуры task_struct указанного процесса.

Системные вызовы sched_setparam () и sched_getparam () позволяют установить и получить значение приоритета реального времени для указанного процесса. Последняя функция просто возвращает значение поля rt_priority, инкапсулированное в специальную структуру sched_param. Вызовы sched_get_priority_max ()

и sched_get_priority__min () возвращают соответственно максимальное и минимальное значение приоритета реального времени для указанной стратегии планирования. Максимальное значение приоритета для стратегий планирования реального времени равно (MAX_USER_RT_PRIO-1), а минимальное значение - 1.

Для обычных задач функция nice () увеличивает значение статического приоритета вызывающего процесса на указанную в аргументе величину. Только пользователь root может указывать отрицательные значения, т.е. уменьшать значение параметра *nice* и соответственно увеличивать приоритет. Функция nice () вызывает функцию ядра set_user_nice (), которая устанавливает значение полей static_pria и prio структуры task_struct.

Системные вызовы управления процессорной привязкой

Планировщик ОС Linux может обеспечивать жесткую процессорную привязку (processor affinity). Хотя планировщик пытается обеспечивать мягкую или естественную привязку путем удержания процессов на одном и том же процессоре, он также позволяет пользователям сказать: "Эти задания должны выполняться только на указанных процессорах независимо ни от чего". Значение жесткой привязки хранится в виде битовой маски в поле cpus_allowed структуры task_struct. Эта битовая маска содержит один бит для каждого возможного процессора в системе. По умолчанию все биты установлены в значение 1, и поэтому процесс потенциально может выполняться на всех процессорах в системе. Пользователь с помощью функции sched_setaffinity () может указать другую битовую маску с любой комбинацией установленных битов. Аналогично функция sched_getaffinity () возвращает ткущее значение битовой маски cpus_allowed.

Ядро обеспечивает жесткую привязку очень простым способом. Во-первых, только что созданный процесс наследует маску привязки от родительского процесса. Поскольку родительский процесс выполняется на дозволенном процессоре, то и порожденный процесс также будет выполняться на дозволенном процессоре. Во-вторых, когда привязка процесса изменяется, ядро использует *миграционные потоки (migration threads)* для проталкивания задания на дозволенный пессор. Следовательно, процесс всегда выполняется только на том процессоре, которому соответствует установленный бит в поле cpus_allowed дескриптора процесса.

Передача процессорного времени

Операционная система Linux предоставляет системный вызов sched_yield () как механизм, благодаря которому процесс может явно передать процессор под управление другим ожидающим процессам. Этот вызов работает путем удаления процесса из активного массива приоритетов (где он в данный момент находится, потому что процесс выполняется) с последующим помещением этого процесса в истекший массив. Получаемый аффект состоит не только в том, что процесс вытесняется и становшея последним в списке заданий с соответствующим приоритетом, а также в том, что помещение процесса в истекший массив гарантирует, что этот процесс не будет выполняться некоторое время. Так как задачи реального времени никогда не могут быть помещены в истекший массив, они составляют специальный случай. Поэтому они только перемещаются в конец списка заданий с таким же значением приоритета (и не помещаются в истекший массив). В более ранних версиях

ОС. Linux семантика вызова sched_yield () была несколько иной. В лучшем случае задание только лишь перемещалось в конец списка заданий с данным приоритетом. Сегодня для пользовательских программ и даже для потоков пространства ядра должна быть полная уверенность в том, что действительно необходимо отказаться от использования процессора, перед тем как ввязывать функцию sched_yield ().

В коде ядра, для удобства, можно вызывать функцию yield () , которая проверяет, что состояние задачи равно TASK_RUNNING, а после этого вызывает функцию sched_yield (). Пользовательские программы должны использовать системный вызов sched_yield ().

В завершение о планировщике

Планировщик выполнения процессов является важной частью ядра, так как выполнение процессов (по крайней мере, для большинства из нас) — это основное использование компьютера. Тем не менее, удовлетворение всем требованиям, которые предъявляются к планировщику — не тривиальная задача. Большое количество готовых к выполнению процессов, требования масштабируемости, компромисс между производительностью и временем реакции, а также требования для различных типов загрузки системы приводят к тому, что тяжело найти алгоритм, который подходит для всех случаев. Несмотря на это, новый планировщик процессов ядра Linux приближается к тому, чтобы удовлетворить всем этим требованиям и обеспечить оптимальное решение для всех случаев, включая отличную масштабируемость и привлекательную реализацию.

Проблемы, которые остались, включают возможность точной настройки (или даже полную замену) алгоритма оценки степени интерактивности задания, который приносит много пользы, когда работает правильно, и приносит много неудобств, когда выполняет предсказания неверно. Работа над альтернативными реализациями продолжается. Когда-нибудь мы увидим новую реализацию в основном ядре.

Улучшение поведения планировщика для NUMA систем (систем с неоднородным доступом к памяти) становится все более актуальной задачей, так как количество машин на основе NUMA-платформ возрастает. Поддержка *доменов планирования (scheduler domain)* — абстракция, которая позволяет описать топологию процессов; она была включена в ядро 2.6 в одной из первых версий.

Эта глава посвящена теории планирования процессов, а также алгоритмам и специфической реализации планировщика ядра Linux. В следующей главе будет рассмотрен основной интерфейс, который предоставляется ядром для выполняющихся процессов, — системные вызовы.

5

Системные вызовы

Ядро операционной системы предоставляет набор интерфейсов, благодаря которым процессы, работающие в пространстве пользователя, могут взаимодействовать с системой. Эти интерфейсы предоставляют пользовательским программам доступ к аппаратному обеспечению и другим ресурсам операционной системы. Интерфейсы работают как посыльные между прикладными программами и ядром, при этом пользовательские программы выдвигают различные запросы, а ядро выполняет их (или приказывает убираться подальше). Тот факт, что такие интерфейсы существуют, а также то, что прикладные программы не имеют права непосредственно делать все, что им заблагорассудится, является ключевым моментом для обеспечения стабильности системы, а также позволяет избежать крупных беспорядков.

Системные вызовы являются прослойкой между аппаратурой и процессами, работающими в пространстве пользователя. Эта прослойка служит для трех главных целей. Во-первых, она обеспечивает абстрактный интерфейс между аппаратурой и пространством пользователя. Например, при записи или чтении данных из файла прикладным программам нет дела до типа жесткого диска, до среды, носителя информации, и даже до типа файловой системы, на которой находится файл. Во-вторых, системные вызовы гарантируют безопасность и стабильность системы. Так как ядро работает посредником между ресурсами системы и пространством пользователя, оно может принимать решение о предоставлении доступа в соответствии с правами пользователей и другими критериями. Например, это позволяет предотвратить возможность неправильного использования аппаратных ресурсов программами, воровство каких-либо ресурсов у других программ, а также возможность нанесения вреда системе. И наконец, один общий слой между пространством пользователя и остальной системой позволяет осуществить виртуальное представление процессов, как обсуждается в главе 3, "Управление процессами".

Если бы приложения имели свободный доступ ко всем ресурсам системы без помощи ядра, то было бы почти невозможно реализовать многозадачность и виртуальную память. В операционной системе Linux системные вызовы являются единственным средством, благодаря которому пользовательские программы могут связываться с ядром; они являются единственной законной точкой входа в ядро. Другие интерфейсы ядра, такие как файлы устройств или файлы на файловой системе `/proc`, в конечном счете сводятся к обращению через системные вызовы.

Интересно, что в ОС Linux реализовано значительно меньше системных вызовов, чем во многих других операционных системах[1].

В этой главе рассказывается о роли и реализации системных вызовов в операционной системе Linux.

API, POSIX и библиотека С

Обычно прикладные программы не разрабатываются с непосредственным использованием системных вызовов, при этом используются программные интерфейсы приложений (Application Programing Interface, API). Это является важным, так как в таком случае нет необходимости в корреляции между интерфейсами, которые используют приложения, и интерфейсами, которые предоставляет ядро. Различные API определяют набор программных интерфейсов, которые используются приложениями. Эти интерфейсы могут быть реализованы с помощью одного системного вызова, нескольких системных вызовов, а также вообще без использования системных вызовов. В действительности, может существовать один и тот же программный интерфейс приложений для различных операционных систем, в то время как реализация этих API может для разных ОС существенно отличаться.

Один из наиболее популярных программных интерфейсов приложений в мире Unix-подобных систем базируется на стандарте POSIX. Технически стандарт POSIX включает в себя набор стандартов IEEE[2], целью которого является обеспечение переносимого стандарта операционной системы, приблизительно базирующегося на ОС Unix. ОС Linux соответствует стандарту POSIX.

Стандарт POSIX является хорошим примером соотношения между интерфейсом API и системными вызовами. Для большинства Unix-подобных операционных систем вызовы интерфейса API, определенные в стандарте POSIX, сильно коррелируют с системными вызовами. Конечно, стандарт POSIX создавался для того, чтобы сделать те интерфейсы, которые предоставляли ранние версии ОС Unix, похожими между собой. С другой стороны, некоторые операционные системы, далекие от OS Unix, такие как Windows NT, предоставляют библиотеки, совместимые со стандартом POSIX.

Частично интерфейс к системным вызовам в операционной системе Linux, так же как и в большинстве Unix-систем, обеспечивается библиотекой функций на языке С. Библиотека С реализует главный программный интерфейс приложений для Unix-систем, что включает стандартную библиотеку языка программирования С и интерфейс системных вызовов. Библиотека С используется всеми программами, написанными на языке программирования С, а также, в связи со свойствами языка С, может быть легко использована для программ, написанных на других языках программирования.

---

[1] Дли аппаратной платформы x86 существует около 250 системных вызовов (для каждой аппаратной платформы разрешается определять свои уникальные системные вызовы). Хотя не для всех операционных систем опубликованы действительные системные вызовы, но по оценкам для некоторых операционных систем таких вызовов более тысячи.

[2] IEEE, eye-trple-E (Институт инженеров по электротехнике и радиоэлектронике, Institute of Electrical and Electronics Engineers) является бесприбыльной профессиональной ассоциацией, действующей во многих технических областях и отвечающей за многие важные стандарты, такие как стандарт POSIX. Больше информации доступно по адресу: http://wwn.ieee.org .

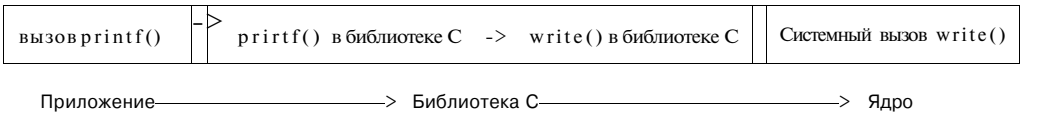


Рис. 5.1. Взаимоотношения между приложением, библиотекой C и ядром на примере вызова функции printf()

Дополнительно библиотека функций C также представляет большую часть API-стандарта POSIX.

С точки зрения прикладного программиста, системные вызовы не существенны: все, с чем работает программист, — это интерфейс API. С другой стороны, ядро имеет отношение только к системным вызовам: все, что делают библиотечные вызовы и пользовательские программы с системными вызовами, — это для ядра не существенно. Тем не менее с точки зрения ядра все-таки важно помнить о потенциальных возможностях использования системного вызова для того, чтобы по возможности поддерживать универсальность и гибкость системных вызовов.

Общий девиз для интерфейсов ОС Unix — это "предоставлять механизм, а не стратегию". Другими словами, системные вызовы существуют для того, чтобы обеспечить определенную функцию в наиболее абстрактном смысле. А то, каким образом используется эта функция, ядра не касается.

Вызовы syscall

Системные вызовы (часто называемые *syscall* в ОС Linux) обычно реализуются в виде вызова функции. Для них могут быть определены один или более аргументов (inputs), которые могут приводить к тем или иным побочным эффектам[3], например к записи данных в файл или к копированию некоторых данных в область памяти, на которую указывает переданный указатель. Системные вызовы также имеют возвращаемое значение типа long[4], которое указывает на успешность выполнения операции или на возникшие ошибки. Обычно, но не всегда, возвращение отрицательного значения указывает на то, что произошла ошибка. Возвращение нулевого значения обычно (но не всегда) указывает на успешность выполнения операции. Системные вызовы ОС Unix в случае ошибки записывают специальный код ошибки в глобальную переменную errno. Значение этой переменной может быть переведено в удобочитаемую форму с помощью библиотечной функции perror().

Системные вызовы, конечно, имеют определенное поведение. Например, системный вызов getpid() определен для того, чтобы возвращать целочисленное значение, равное значению идентификатора PID текущего процесса. Реализация этой функции в ядре очень проста.

[3]Следует обратить внимание на слово "могут". Хотя почти вге вызовы создают различные побочные эффекты (т.е. приводят к каким-либо иэменениям в состоянии системы), тем не менее небольшое количество вызовов, как, например, вызов *getpid()*, просто возвращают некоторые данные ядра.

[4]Тип *long* используется для совместимости с 64-разрядными платформами.

```
asmlinkage long sys_getpid(void)
{
        return current->tgid;
}
```

Следует заметить, что в определении ничего не говорится о способе реализации. Ядро должно обеспечить необходимую функциональность системного вызова, но реализация может быть абсолютно свободной, главное, чтобы результат был правильный. Конечно, рассматриваемый системный вызов в действительности является таким же простым, как и показано, и существует не так уж много различных вариантов для его реализации (на самом деле более простого метода не существует)[5].

Даже из такого примера можно сделать пару наблюдений, которые касаются системных вызовов. Во-первых, следует обратить внимание на модификатор asmlinkage в объявлении функции. Это волшебное слово дает компилятору информацию о том, что обращение к аргументам этой функции должно производиться только через стек. Для всех системных вызовов использование этого модификатора является обязательным. Во-вторых, следует обратить внимание, что системный вызов getpid () объявлен в ядре, как sys_getpid (). Это соглашение о присваивании имен используется для всех системных вызовов операционной системы Linux: системный вызов bar () должен быть реализован с помощью функции sys_bar ().

Номера системных вызовов

Каждому системному вызову операционной системы Linux присваивается *номер системного вызова (syscall number).* Этот уникальный номер используется для обращения к определенному системному вызову. Когда процесс выполняет системный вызов из пространства пользователя, процесс не обращается к системному вызову по имени.

Номер системного вызова является важным атрибутом. Однажды назначенный номер не должен меняться никогда, иначе это нарушит работу уже скомпилированных прикладных программ. Если системный вызов удаляется, то соответствующий номер не может использоваться повторно. В операционной системе Linux предусмотрен так называемый "не реализованный" ("not implemented") системный вызов — функция sys_ni_syscall (), которая не делает ничего, кроме того, что возвращает значение, равное -ENOSYS, — код ошибки, соответствующий неправильному системному вызову. Эта функция служит для "затыкания дыр" в случае такого редкого событии, как удаление системного вызова.

Ядро поддерживает список зарегистрированных системных вызовов в таблице системных вызовов. Эта таблица хранится в памяти, на которую указывает переменная sys_call_table. Данная таблица зависит от аппаратной платформы и обычно определяется в файле entry.S. В таблице системных вызовов каждому уникальному номеру системного вызова назначается существующая функция syscall.

[5] Может быть, интересно, почему вызов getpid *()* возвращает поле *tgid,* которое является идентификатором группы потоков (thread group ID)? Это делается потому, что дли обычных процессов значение параметра *TGID* равно значению параметра *PID.* При наличии нескольких потоков значение параметра *TGID* одинаково дли всек потоков одной группы. Такая реализация дает возможность различным потокам вызывать функцию *getpid ()* и получать одинаковое значение параметра *PID.*

Производительность системных вызовов

Системные вызовы в операционной системе Linux работают быстрее, чем во многих других операционных системах. Это отчасти связано с невероятно малым временем переключения контекста. Переход в режим ядра и выход из него являются хорошо отлаженным процессом и простым делом. Другой фактор — это простота как механизма обработки системных вызовов, так и самих системных вызовов.

Обработка системных вызовов

Приложения пользователя не могут непосредственно выполнять код ядра. Они не могут просто вызвать функцию, которая существует в пространстве ядра, так как ядро находится в защищенной области памяти. Если программы смогут непосредственно читать и писать в адресное пространство ядра, то безопасность системы "вылетит в трубу".

Пользовательские программы должны каким-либо образом сигнализировать ядру о том, что им необходимо выполнить системный вызов и что система должна переключиться в режим ядра, где системный вызов должен быть выполнен с помощью ядра, работающего от имени приложения.

Таким механизмом, который может подать сигнал ядру, является программное прерывание: создается исключительная ситуация (exception) и система переключается в режим ядра для выполнения обработчика этой исключительной ситуации. Обработчик исключительной ситуации в данном случае и является обработчиком системного вызова (system call handler). Для аппаратной платформы x8б это программное прерывание определено как машинная инструкция `int $0x80`. Она приводит в действие механизм переключения в режим ядра и выполнение вектора исключительной ситуации с номером 128, который является обработчиком системных вызовов. Обработчик системных вызовов— это функция с очень подходящим именем `system_call ()`. Данная функция зависима от аппаратной платформы и определена в файле `entry.S`[6]. В новых процессорах появилась такая новая функция, как *sysenter*. Эта функция обеспечивает более быстрый и специализированный способ входа в ядро для выполнения системного вызова, чем использование инструкции программного прерывания — `int`. Поддержка такой функции была быстро добавлена в ядро. Независимо от того, каким образом выполняется системный вызов, основным является то, что пространство пользователя вызывает исключительную ситуацию, или прерывание, чтобы вызвать переход в ядро.

Определение необходимого системного вызова

Простой переход в пространство ядра сам по себе не является достаточным, потому что существует много системных вызовов, каждый из которых осуществляет переход в режим ядра одинаковым образом. Поэтому ядру должен передаваться номер системного вызова.

---

[6] Большаячасть дальнейшего описания процесса обработки системных вызовов базируется на версии для аппаратной платформы x86. Но не стоит волноваться, для других аппаратных платформ это выполняется аналогичным образом.

Для аппаратной платформы x86 номер системного вызова сохраняется в регистре процессора eax перед тем, как вызывается программное прерывание. Обработчик системных вызовов после этого считывает это значение из регистра eax. Для других аппаратных платформ выполняется нечто аналогичное.

Функция system_call() проверяет правильность переданного номера системного вызова путем сравнения его со значением постоянной NR_syscalls. Если значение номера больше или равно значению NR_syscalls, то функция возвращает значение -ENOSYS. В противном случае вызывается соответствующий системный вызов следующим образом:

```
call *sys_call_table(,%eax,4)
```

Так как каждый элемент таблицы системных вызовов имеет длину 32 бит (4 байт), то ядро умножает данный номер системного вызова на 4 для получения нужной позиции в таблице системных вызовов (рис. 5.2).

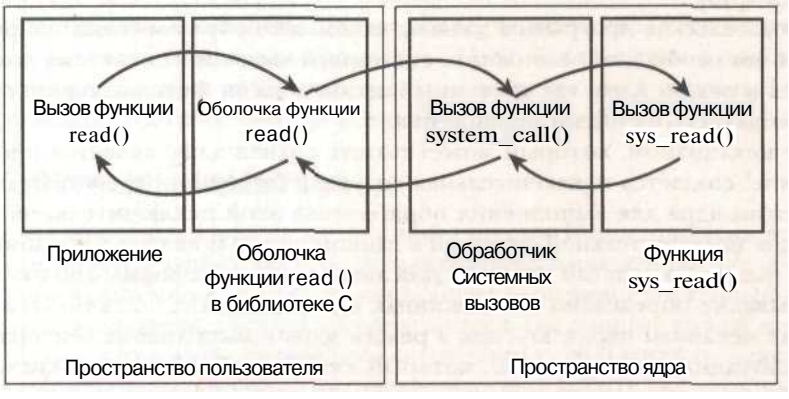


Рис. 5.2. Запуск обработчика системных вызовов и выполнение системного вызова

Передача параметров

В дополнение к номеру вызова, большинство системных вызовов требует передачи им одного или нескольких параметров. Во время перехвата исключительной ситуации пространство пользователя должно каким-либо образом передать ядру эти параметры. Самый простой способ осуществить такую передачу — это сделать по аналогии с передачей номера системной функции: параметры хранятся в регистрах процессора. Для аппаратной платформы x86 регистры ebx, ecx, edx, esi, edi содержат соответственно первые пять аргументов. В случае редких ситуаций с шестью или более аргументами, используется один регистр, который содержит указатель на память пространства пользователя, где хранятся все параметры.

Возвращаемое значение также передается в пространство пользователя через регистр. Для аппаратной платформа x86 оно хранится в регистре eax.

Реализация системных вызовов

Реализация системного вызова в ОС Linux не связана с поведением обработчика системных вызовов. Добавление нового системного вызова в операционной системе Linux является сравнительно простым делом. Тяжелая работа связана с разработкой и реализацией самого системного вызова. Регистрация его в ядре проста. Давайте рассмотрим шаги, которые необходимо предпринять, чтобы написать новый системный вызов в операционной системе Linux.

Первый шаг в реализации системного вызова — это определение его назначения, т.е. что он должен делать. Каждый системный вызов должен иметь только одно назначение. Мультиплексные системные вызовы (один системный вызов, который выполняет большой набор различных операций, в зависимости от значения флага, передаваемого в качестве аргумента) в операционной системе Linux использовать не рекомендуется. Для примера того, как *не надо делать,* можно обратиться к системной функции `ioctl()`.

Какие должны быть аргументы, возвращаемые значения и коды ошибок для новой системной функции? Системная функция должна иметь понятный и простой интерфейс, по возможности с меньшим количеством аргументов. Семантика и поведение системных функций — это очень важные вещи, они не должны меняться, потому что от них будет зависеть работа прикладных программ.

Важным является разработка интерфейса с прицелом на будущее. Не ограничены ли возможности функции без необходимости? Разрабатываемый системный вызов должен быть максимально общим. Не нужно полагать, что завтра он будет использоваться так же, как сегодня. *Назначение* системного вызова должно оставаться постоянным, но его *использование* может меняться. Является ли системный вызов переносимым? Не нужно делать допущений о возможном размере машинного слова или порядка следования байтов. В главе 19, "Переносимость", рассматриваются соответствующие вопросы. Нужно удостовериться, что никакие неверные допущения не будут мешать использованию системного вызова в будущем. Помните девиз Unix: "Обеспечивать механизм, а не стратегию".

При разработке системного вызова важно помнить, что переносимость и устойчивость необходимы не только сегодня, но и будут необходимы в будущем. Основные системные вызовы ОС Unix выдержали это испытание временем. Большинство из них такие же полезные и применимые сегодня, как и почти тридцать лет назад!

Проверка параметров

Системные вызовы должны тщательно проверять все свои параметры для того, чтобы убедиться, что их значения адекватны и законны. Системные вызовы выполняются в пространстве ядра, и если пользователь может передать неправильные значения ядру, то стабильность и безопасность системы могут пострадать.

Например, системные вызовы для файлового ввода-вывода данных должны проверить, является ли значение файлового дескриптора допустимым. Функции, связанные с управлением процессами, должны проверить, является ли значение переданного идентификатора PID допустимым. Каждый параметр должен проверяться не только на предмет допустимости и законности, но и на предмет правильности значения.

Одна из наиболее важных проверок— это проверка указателей, которые передает пользователь. Представьте, что процесс может передать любой указатель, даже тот, который указывает на область памяти, не имеющей прав чтения! Процесс может таким обманом заставить ядро скопировать данные, к которым процесс не имеет доступа, например данные, принадлежащие другому процессу. Перед тем как следовать указателю, переданному из пространства пользователя, система должна убедиться в следующем.

- Указатель указывает на область памяти в пространстве пользователя. Нельзя, чтобы процесс заставил ядро обратиться к памяти ядра от имени процесса.
- Указатель указывает на область памяти в адресном пространстве текущего процесса. Нельзя позволять, чтобы процесс заставил ядро читать данные других процессов.
- Для операций чтения есть права на чтение области памяти. Для операций записи есть права на запись области памяти. Нельзя, чтобы процессы смогли обойти ограничения на чтение и запись.

Ядро предоставляет две функции для выполнения необходимых проверок при копировании данных в пространство пользователя и из него. Следует помнить, что ядро никогда не должно слепо следовать за указателем в пространстве пользователя! Одна из этих двух функций должна использоваться всегда.

Для записи в пространство пользователя предоставляется функция copy_to_user (). Она принимает три параметра: адрес памяти назначения в пространстве пользователя; адрес памяти источника в пространстве ядра; и размер данных, которые необходимо скопировать, в байтах.

Для чтения из пространства пользователя используется функция copy_from_user (), которая аналогична функции copy_to_user (). Эта функция считывает данные, на которые указывает второй параметр, в область памяти, на которую указывает первый параметр, количество данных — третий параметр.

Обе эти функции возвращают количество байтов, которые они не смогли скопировать в случае ошибки. При успешном выполнении операции возвращается нуль. В случае такой ошибки стандартным является возвращение системным вызовом значения -EFAULT.

Давайте рассмотрим пример системного вызова, который использует функции copy_from_user () и copy_to_user () . Системный вызов silly_copy () является до крайности бесполезным. Он просто копирует данные из своего первого параметра во второй. Это очень не эффективно, так как используется дополнительное промежуточное копирование в пространство ядра безо всякой причины. Но зато это позволяет проиллюстрировать суть дела.

```
/*
* Системный вызов silly copy — крайне бесполезная функция,
* которая копирует len байтов иэ области памяти,
* на которую указывает параметр src, в область памяти,
* на которую указывает параметр dst, с использованием ядра
* безо всякой на то причины. Но это хороший пример!
*/
asmlinkage long sys_silly_copy(unsigned long *src,
        unsigned long *dst, unsigned long len)
}
```

```
        unsigned long buf;
        /* возвращаем ошибку, если размер машинного слова в ядре
           не совпадает с размером данных, переданных пользователем */
        if (len != sizeof(buf))
              return -EINVAL;
        /* копируем из src, который является адресом в пространстве
           пользователя, в buf */
        if (copy_from_user(&buf, src, len))
              return -EFAULT;
        /* копируем из buf в dst, который тоже является адресом
           в пространстве пользователя */
        if (copy_to_user(dst, &buf, len) )
              return -EFAULT;
  /* возвращаем количество скопированных данных */
  return len;
  }
```

Следует заметить, что обе функции, `copy_from_user()` и `copy_to_user()`, могут блокироваться. Это возникает, например, если страница памяти, содержащая данные пользователя, не находится в физической памяти, а в данный момент вытеснена на диск. В таком случае процесс будет находиться в приостановленном состоянии до тех пор, пока обработчик прерываний из-за отсутствия страниц (page fault handler) не возвратит страницу памяти в оперативную память из файла подкачки на диске.

Последняя проверка — это проверка на соответствие правам доступа. В старых версиях ядра Linux стандартом было использование функции `suser()` для системных вызовов, которые требуют прав пользователя root. Эта функция просто проверяла, запущен ли процесс от пользователя root. Сейчас эту функцию убрали и заменили более мелко структурированным набором системных "возможностей использования" (capabilities). В новых системах предоставляется возможность проверять специфические права доступа к специфическим ресурсам. Функция `capable()` с допустимым значением флага, определяющего тип прав, возвращает ненулевое значение, если пользователь обладает указанным правом, и нуль— в противном случае. Например, вызов `capable (CAP_SYS_NICE)` проверяет, имеет ли вызывающий процесс возможность модифицировать значение параметра *nice* других процессов. По умолчанию суперпользователь владеет всеми правами, а пользователь, не являющийся пользователем root, не имеет никаких дополнительных прав. Следующий пример системного вызова, который демонстрирует использование возможностей использования, тоже является практически бесполезным.

```
asmlinkage long sys_am_i_popular (void)
{
      /* Проверить, имеет ли право процесс использовать
         возможность CAP_SYS_NICE */
      if (!capable(CAP_SYS_NICE))
            return -EPERM;
      /* Возвратить нуль, чтобы обозначить успешное завершение */
      return 0;
      }
```

Список всех "возможностей использования" и прав, которые за ними закреплены, содержится в файле `<linux/capability.h>`.

Контекст системного вызова

Как уже обсуждалось в главе 3, "Управление процессами", при выполнении системного вызова ядро работает в контексте процесса. Указатель current указывает на текущее задание, которое и есть процессом, выполняющим системный вызов.

В контексте процесса ядро может переходит в приостановленное состояние (например, если системный вызов блокируется при вызове функции или явно вызывает функцию schedule ()), а также является полностью вытесняемым. Эти два момента важны. Возможность переходить в приостановленное состояние означает, что системный вызов может использовать большую часть функциональных возможностей ядра. Как будет видно из главы 6, "Прерывания и обработка прерываний", наличие возможности переходить в приостановленное состояние значительно упрощает программирование ядра[7]. Тот факт, что контекст процесса является вытесняемым, подразумевает, что, как и в пространстве пользователя, текущее задание может быть вытеснено другим заданием. Так как новое задание может выполнить тот же системный вызов, необходимо убедиться, что системные вызовы являются реентерабельными. Это очень похоже на требования, выдвигаемые для симметричной мультипроцессорной обработки. Способы защиты, которые обеспечивают реентерабельность, описаны в главе 8, "Введение в синхронизацию выполнения кода ядра", и в главе 9, "Средства синхронизации в ядре".

После завершение системного вызова управление передается обратно в функцию system_call (), которая в конце концов производит переключение в пространство пользователя, и далее выполнение пользовательского процесса продолжается.

Окончательные шаги регистрации системного вызова

После того как системный вызов написан, процедура его регистрации в качестве официального системного вызова тривиальна и состоит в следующем.

- Добавляется запись в конец таблицы системных вызовов. Это необходимо сделать для всех аппаратных платформ, которые поддерживают этот системный вызов (для большинства системных вызовов — это все возможные платформы). Положение системного вызова в таблице — это номер системного вызова, начиная с нуля. Например, десятая запись таблицы соответствует системному вызову с номером девять.
- Для всех поддерживаемых аппаратных платформ номер системной функции должен быть определен в файле include/linux/unistd.h.
- Системный вызов должен быть вкомпилирован в образ ядра (в противоположность компиляции в качестве загружаемого модуля[8]). Это просто соответствует размещению кода в каком-нибудь важном файле каталога kernel/.

---

[7] Обработчики прерываний не могут переходить в приостановленное состояние и, следовательно, более ограничены в своих действиях по сравнению с системными вызовами, которые работают в контексте процесса.

[8] Регистрации новых постоянных системных вызовов в ядре требует компиляции системного вызова в образ ядра. Тем не менее есть принципиальная возможность с помощью динамически загружаемого модуля ядра перехватить существующие системные вызовы и даже, ценой некоторых усилий, динамически зарегистрировать новые. — *Примеч. перев.*

Давайте более детально рассмотрим эти шаги на примере функции системного вызова, foo (). Вначале функция sys_foo () должна быть добавлена в таблицу системных вызовов. Для большинства аппаратных платформ таблица системных вызовов размещается в файле entry.S и выглядит примерно следующим образом.

```
ENTRY(sys_call_table)
        .long sys_restart_syscall     /* 0 */
        .long sys_exit
        .long sys_fork
        .long sys_read
        .long sys_write
        .long sys_open      /*      5      */
        ...
        .long sys_timer_delete
        .long sys_clock_settime
        .long sys_clock_gettime       /* 280 */
        .long sys_clock_getres
        .long sys_clock_nanosleep
```

Необходимо добавить новый системный вызов в конец этого списка:

```
.long sys_foo
```

Нашему системному вызову будет назначен следующий свободный номер, 283, котя мы этого явно и не указывали. Для каждой аппаратной платформы, которую мы будем поддерживать, системный вызов должен быть добавлен в таблицу системных вызовов соответствующей аппаратной платформы (нет необходимости получать номер системного вызова для каждой платформы). Обычно необходимо сделать системный вызов доступным для всех аппаратных платформ. Следует обратить внимание на договоренность указывать комментарии с номером системного вызова через каждые пять записей, что позволяет быстро найти, какой номер какому системному вызову соответствует.

Далее необходимо добавить номер системного вызова в заголовочный файл include/asm/unistd.h, который сейчас выглядит примерно так.

```
/*
* This file contains the system call numbers.
*/
#define __NR_restart_syscall    0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __HR_write 4
#define __NR_open 5
...
#define __NR_mq_unlink          278
#define __NR_mq_timedsend       279
#define __NR_mq_timedreceive    280
#define __NR_mq_notify          281
#define __NR_mq_getsetattr      282
```

В конец файла добавляется следующая строка.

```
#define __NR_foo 283
```

В конце концов необходимо реализовать сам системный вызов foo (). Так как системный вызов должен быть вкомпилирован в образ ядра во всех конфигурациях, мы его поместим в файл kernel/sys.c. Код необходимо размещать в наиболее подходящем файле. Например, если функция относится к планированию выполнения процессов, то ее необходимо помещать в файл sched.c.

```
/*
* sys_foo - всеми любимый системный вызов.
*
* Возвращает размер стека ядра процесса
*/
asmlinkage long sys_foo(void)
{
        return THREAD_SIZE;
}
```

Это все! Загрузите новое ядро. Теперь из пространства пользователя можно вызвать системную функцию foo ().

Доступ к системным вызовам из пространства пользователя

В большинстве случаев системные вызовы поддерживаются библиотекой функций языка С. Пользовательские приложения могут получать прототипы функций из стандартных заголовочных файлов и компоновать программы с библиотекой С для использования вашего системного вызова (или библиотечной функции, которая вызывает ваш системный вызов). Однако если вы только что написали системный вызов, то маловероятно, что библиотека glibc уже его поддерживает!

К счастью, ОС Linux предоставляет набор макросов-оболочек для доступа к системным вызовам. Они позволяют установить содержимое регистров и выполнить машинную инструкцию int 50x80. Эти макросы имеют имя syscalln (), где *n* — число от нуля до шести. Это число соответствует числу параметров, которые должны передаваться в системный вызов, так как макросу необходима информация о том, сколько ожидается параметров, и соответственно, нужно записать эти параметры в регистры процессора. Например, рассмотрим системный вызов open (), который определен следующим образом.

```
long open(const char "filename, int flags, int model
```

Макрос для вызова этой системной функции будет выглядеть так.

```
#define NR_open 5
_syscall3(long, NR_open, const char *, filename, int, flags, int, mode)
```

После этого приложение может просто вызывать функцию open ().

Каждый макрос принимает 2 + 2*n параметров. Первый параметр соответствует типу возвращаемого значения системного вызова. Второй параметр — имя системного вызова. После этого следуют тип и имя каждого параметра в том же поряд-

ке, что и у системного вызова. Постоянная NR_open, которая определена в файле <asm/unistd.h>, — это номер системного вызова. В функцию на языке программирования С такой вызов превращается с помощью вставок на языке ассемблера, которые выполняют рассмотренные в предыдущем разделе шаги. Значения аргументов помещаются в соответствующие регистры, и выполняется программное прерывание, которое перехватывается в режиме ядра. Вставка данного макроса в приложение — это все, что необходимо для выполнения системного вызова open ().

Напишем макрос, который позволяет вызвать нашу замечательную системную функцию, и соответствующий код, который позволяет этот вызов протестировать.

```
#define__NR_foo 283
__syscall0(long, foo)

int main ()
{
        long stack_size;

        stack_size = foo () ;
        printf ("Размер стека ядра равен %ld\n", stack_size);

        return 0;
}
```

Почему не нужно создавать системные вызовы

Новый системный вызов легко реализовать, тем не менее это необходимо делать только тогда, когда ничего другого не остается. Часто, для того чтобы обеспечить новый системный вызов, существуют более подходящие варианты. Давайте рассмотрим некоторые "за" и "против" и возможные варианты.

Для создания нового интерфейса в виде системного вызова могут быть следующие "за".

- Системные вызовы просто реализовать и легко использовать.
- Производительность системных вызовов в операционной системе Linux очень высока.

Возможные "против".

- Необходимо получить номер системного вызова, который должен быть официально назначен в период работы над разрабатываемыми сериями ядер.
- После того как системный вызов включен в стабильную серию ядра, он становится "высеченным в камне". Интерфейс не должен меняться, чтобы не нарушить совместимости с прикладными пользовательскими программами.
- Для каждой аппаратной платформы необходимо регистрировать отдельный системный вызов и осуществлять его поддержку.
- Для простого обмена информацией системный вызов — это "стрельба из пушки по воробьям".

Возможные варианты.

- Реализовать файл устройства и использовать функции read () и write () для этого устройства, а также использовать функцию ioctl () для манипуляции специфическими параметрами или для получения специфической информации.
- Некоторые интерфейсы, например семафоры, могут быть представлены через дескрипторы файлов. Управлять этими устройствами также можно по аналогии с файлами.
- Добавить информационный файл в соответствующем месте файловой системы sysfs.

Для большого числа интерфейсов, системные вызовы— это правильный выбор. В операционной системе Linux пытаются избегать простого добавления системного нызова для поддержки каждой повой, вдруг появляющейся абстракции. В результате получился удивительно четкий уровень системных вызовов, который принес очень мало разочарований и привел к малому числу не рекомендованных к использованию и устаревших (deprecated) интерфейсов (т.е. таких, которые больше не используются или не поддерживаются).

Малая частота добавления новых системных вызовов свидетельствует о том, что Linux— это стабильная операционная система с полным набором функций. Очень немного системных вызовов было добавлено во время разработки серий ядер 2.3 и 2.5. Большая часть из новых системных вызовов предназначена для улучшения производительности.

В заключение о системных вызовах

В этой главе было рассмотрено, что такое системные вызовы и как они соотносятся с вызовами библиотечных функций и интерфейсом прикладных программ (API). После этого было описано, как системные вызовы реализованы в ядре Linux, а также была представлена последовательность событий для выполнения системного вызова: программное прерывание ядра, передача номера системного вызова и аргументов системного вызова, выполнение соответствующей функции системного вызова и возврат результатов работы в пространстно пользователя.

Далее было рассказано, как добавить новый системный вызов, и был приведен простой пример использования системного вызова из пространства пользователя. Весь процесс является достаточно простым! Из простоты создания системного вызова следует, что основная работа по добавлению нового системного вызова сводится к реализации функции системного вызова. В оставшейся части книги рассмотрены основные принципы, а также интерфейсы, которые необходимо использовать при создании хорошо работающих, оптимальных и безопасных системных вызовов.

В конце главы были рассмотрены "за" и "против" относительно реализации системных вызовов и представлен краткий список возможных вариантов добавления новых системных вызовов.

6

Прерывания и обработка прерываний

Управление аппаратными устройствами, которые подключены к вычислительной машине, — это одна из самых ответственных функций ядра. Частью этой работы является необходимость взаимодействия с отдельными устройствами машины. Поскольку процессоры обычно работают во много раз быстрее, чем аппаратура, с которой они должны взаимодействовать, то для ядра получается неэффективным отправлять запросы и тратить время, ожидая ответы от потенциально более медленного оборудования. Учитывая небольшую скорость отклика оборудования, ядро должно иметь возможность оставлять на время работу с оборудованием и выполнять другие действия, пока аппаратное устройство не закончит обработку запроса. Одно из возможных решений этой проблемы — периодический *опрос* оборудования (*polling*). Ядро периодически может проверять состояние аппаратного устройства системы и соответственным образом реагировать. Однако такой подход вносит дополнительные накладные расходы, потому что, независимо от того, готов ответ от аппаратного устройства или оно еще выполняет запрос, все равно осуществляется постоянный систематический опрос состояния устройства через постоянные интервалы времени. Лучшим решением является обеспечение механизма, который позволяет подавать ядру сигнал о необходимости уделить внимание оборудованию. Такой механизм называется прерыванием (interrupt).

Прерывания

Прерывания позволяют аппаратным устройствам взаимодействовать с процессором. Например, при наборе на клавиатуре контроллер клавиатуры (или другое устройство, которое обслуживает клавиатуру) генерирует прерывание, чтобы объявить операционной системе о том, что произошли нажатия клавиш. Прерывания — это специальные электрические сигналы, которые аппаратные устройства посылают процессору. Процессор получает прерывание и дает сигнал операционной системе о том, что ОС может обработать новые данные. Аппаратные устройства генерируют прерывания асинхронно по отношению к тактовому генератору процессора — прерывания могут возникать непредсказуемо, в любой момент времени. Следовательно, работа ядра может быть прервана в любой момент для того, чтобы обработать прерывания.

Физически прерывания производятся электрическими сигналами, которые создаются устройствами и направляются на входные контакты микросхемы контроллера прерываний. Контроллер прерываний в свою очередь отправляет сигнал процессору. Процессор выполняет детектирование сигнала и прерывает выполнение работы для того, чтобы обработать прерывание. После этого процессор извещает операционную систему о том, что произошло прерывание и операционная система может соответствующим образом это прерывание обработать.

Различные устройства связаны со своими прерываниями с помощью уникальных числовых значений, соответствующих каждому прерыванию. Отсюда следует, что прерывания, поступившие от клавиатуры, отличаются от прерываний, поступивших от жесткого диска. Это позволяет операционной системе различать прерывания и иметь информацию о том, какое аппаратное устройство произвело данное прерывание. Поэтому операционная система может обслуживать каждое прерывание с помощью своего уникального обработчика.

Идентификаторы, соответствующие прерываниям, часто называются линиями запросов на прерывание (interrupt request lines, IRQ lines). Обычно это некоторые числа. Например, для платформы PC значение IRQ, равное 0, — это прерывание таймера, а IRQ, равное 1, — прерывание клавиатуры. Однако не все номера прерываний жестко определены. Прерывания, связанные с устройствами шины PCI, например, назначаются динамически. Другие платформы, которые не поддерживают стандарт PCI, имеют аналогичные функции динамического назначения номеров прерываний. Основная идея состоит в том, что определенные прерывания связаны с определенными устройствами, и у ядра есть вся эта информация. Аппаратное обеспечение, чтобы привлечь внимание ядра, генерирует прерывание вроде *"Эй! Было новое нажатие клавиши! Его необходимо обработать!.*

Исключительные ситуации

Исключительные ситуации (exceptions) часто рассматриваются вместе с прерываниями. В отличие от прерываний, они возникают синхронно с тактовым генератором процессора. И действительно, их часто называют синхронными прерываниями. Исключительные ситуации генерируются процессором при выполнении машинных инструкций как реакция на ошибку программы (например, деление на нуль) или как реакция на аварийную ситуацию, которая может быть обработана ядром (например, прерывание из-за отсутствия страницы, page fault). Так как большинство аппаратных платформ обрабатывают исключительные ситуации аналогично обработке прерываний, то инфраструктуры ядра, для обоих видов обработки, также аналогичны. Большая часть материала, посвященная обработке прерываний (асинхронных, которые генерируются аппаратными устройствами), также относится и к исключительным ситуациям (синхронным, которые генерируются самим процессором).

С одним типом исключительной ситуации мы уже встречались в предыдущей главе. Там было рассказано, как для аппаратной платформы x86 реализованы системные вызовы на основе программных прерываний. При этом генерируется исключительная ситуация, которая заставляет переключиться в режим ядра и в конечном итоге приводит к выполнению определенного обработчика системного вызова. Прерывания работают аналогичным образом, за исключением того, что прерывания генерируются не программным, а аппаратным обеспечением.

Обработчики прерываний

Функция, которую выполняет ядро в ответ на определенное прерывание, называется *обработчиком прерывания (interrupt handler)* или *подпрограммой обслуживания прерывания (interrupt service routine)*. Каждому устройству, которое генерирует прерывания, соответствует свой обработчик прерывания. Например, одна функция обрабатывает прерывание от системного таймера, а другая — прерывания, сгенерированные клавиатурой. Обработчик прерывания для какого-либо устройства является частью *драйвера* этого устройства — кода ядра, который управляет устройством.

В операционной системе Linux обработчики прерываний — это обычные функции, написанные на языке программирования C. Они должны соответствовать определенному прототипу, чтобы ядро могло стандартным образом принимать информацию об обработчике, а в остальном— это обычные функции. Единственное, что отличает обработчики прерываний от других функций ядра, — это то, что они вызываются ядром в ответ на прерывание и выполняются в специальном контексте, именуемом *контекстом прерывания* (interrupt context), который будет рассмотрен далее.

Так как прерывание может возникнуть в любой момент времени, то, соответственно, и обработчик прерывания может быть вызван в любой момент времени. Крайне важно, чтобы обработчик прерывания выполнялся очень быстро и возобновлял управление прерванного кода по возможности быстро. Поэтому, хотя для аппаратного обеспечения и важно, чтобы прерывание обслуживалось немедленно, для остальной системы важно, чтобы обработчик прерывания выполнялся в течение максимально короткого промежутка времени. Минимально возможная работа, которую должен сделать обработчик прерывания, — это отправить подтверждение устройству, что прерывание получено. Однако обычно обработчики прерываний должны выполнить большее количество работы. Например, рассмотрим обработчик прерывания сетевого устройства. Вместе с отправкой подтверждения аппаратному обеспечению, обработчик прерывания должен скопировать сетевые пакеты из аппаратного устройства в память системы, обработать их, отправить соответствующему стеку протоколов или соответствующей программе. Очевидно, что для этого требуется много работы.

Верхняя и нижняя половины

Ясно, что два указанных требования о том, что обработчик прерывания должен выполняться быстро и, в дополнение к этому, выполнять много работы, являются противоречивыми. В связи с конфликтными требованиями, обработчик прерываний разбивается на две части, или половины. Обработчик прерывания является *верхней половиной (top half)*— он выполняется сразу после приема прерывания и выполняет работу, критичную к задержкам во времени, такую как отправка подтверждения о получении прерывания или сброс аппаратного устройства. Работа, которую можно выполнить позже, откладывается до выполнения *нижней* (или основной) *половины (bottom half)*. Нижняя половина обрабатывается позже, в более удобное время, когда все прерывания разрешены. Достаточно часто нижняя половина выполняется сразу же после возврата из обработчика прерывания.

Операционная система предоставляет различные механизмы для реализации обработки нижних половин, которые обсуждаются в главе 7, "Обработка нижних половин и отложенные действия".

Рассмотрим пример разделения обработчика прерывания на верхнюю и нижнюю половины на основе старой доброй сетевой платы. Когда сетевой интерфейсный адаптер получает входящие из сети пакеты, он должен уведомить ядро о том, что доступны новые данные. Это необходимо сделать немедленно, чтобы получить оптимальную пропускную способность и время задержки при передаче информации по сети. Поэтому немедленно генерируется прерывание: *"Эй, ядро! Есть свежие пакеты!.* Ядро отвечает выполнением зарегистрированного обработчика прерывания от сетевого адаптера.

Обработчик прерывания выполняется, аппаратному обеспечению направляется подтверждение, пакеты копируются в основную память, и после этого сетевой адаптер готов к получению новых пакетов. Эта задача является важной, критичной ко времени выполнения и специфической для каждого типа аппаратного обеспечения. Остальная часть обработки сетевых пакетов выполняется позже — нижней половиной обработчика прерывания. В этой главе мы рассмотрим обработку верхних половин, а в следующей — нижних.

Регистрация обработчика прерывания

Ответственность за обработчики прерываний лежит на драйверах устройств, которые управляют определенным типом аппаратного обеспечения. С каждым устройством связан драйвер, и если устройство использует прерывания (а большинство использует), то драйвер должен выполнить регистрацию обработчика прерывания.

Драйвер может зарегистрировать обработчик прерывания для обработки заданной линии с помощью следующей функции.

```
/* request_irq: выделить заданную линию прерывания */
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int, void *, struct pt_regs *),
                unsigned long irqflags,
                const char * devname,
                void *dev_id);
```

Первый параметр, `irq`, указывает назначаемый номер прерывания. Для некоторых устройств, таких как, например, обычные устройства персонального компьютера, таймер и клавиатура, это значение, как правило, жестко закреплено. Для большинства других устройств это значение определяется путем проверки (probing) или другим динамическим способом.

Второй параметр, `handler`, — это указатель на функцию обработчика прерывания, которая обслуживает данное прерывание. Эта функция вызывается, когда в операционную систему приходит прерывание. Следует обратить внимание на специфический прототип функции-обработчика. Она принимает три параметра и возвращает значение типа `irqreturn_t`. Ниже в этой главе мы более подробно обсудим эту функцию.

Третий параметр, `irqflags`, может быть равным нулю или содержать битовую маску с одним или несколькими следующими флагами.

- SA_INTERRUPT. Этот флаг указывает, что данный обработчик прерывания — это *быстрый обработчик прерывания*. Исторически так сложилось, что операционная система Linux различает быстрые и медленные обработчики прерываний. Предполагается, что быстрые обработчики выполняются быстро, но потенциально очень часто, поэтому поведение обработчика прерывания изменяется, чтобы обеспечить максимально возможную скорость выполнения. Сегодня существует только одно отличие: при выполнении быстрого обработчика прерываний запрещаются *все* прерывания на локальном процессоре. Это позволяет быстрому обработчику завершится быстро, и другие прерывания никак этому не пометают. По умолчанию (если этот флаг не установлен) разрешены все прерывания, кроме тех, которые маскированы на всех процессорах и обработчики которых в данный момент выполняются. Для всех прерываний, кроме прерываний таймера, нет необходимости устанавливать этот флаг.

- SA_SAMPLE_RANDOM. Этот флаг указывает, что прерывания, сгенерированные данным устройством, должны вносить вклад в пул энтропии ядра. Пул энтропии ядра обеспечивает генерацию истинно случайных чисел на основе различных случайных событий. Если этот флаг указан, то моменты времени, когда приходят прерывания, будут введены в пул энтропии. Этот флаг нельзя устанавливать, если устройство генерирует прерывания в предсказуемые моменты времени (как, например, системный таймер) или на устройство может повлиять внешний злоумышленник (как, например, сетевое устройство). С другой стороны, большинство устройств генерируют прерывания в непредсказуемые моменты времени и поэтому являются хорошим источником энтропии. Для более подробного описания пула энтропии ядра см.. приложение Б, "Генератор случайных чисел ядра".

- SA_SHIRQ. Этот флаг указывает, что номер прерывания может совместно использоваться несколькими обработчиками прерываний (shared). Каждый обработчик, который регистрируется на одну и ту же линию прерывания, должен указывать этот флаг. В противном случае для каждой линии может существовать только один обработчик прерывания. Более подробная информация о совместно используемых обработчиках прерываний приведена в следующем разделе.

Четвертый параметр, devname, — это ASCII-строка, которая описывает, какое устройство связано с прерыванием. Например, для прерывания клавиатуры персонального компьютера это значение равно "keyboard". Текстовые имена устройств применяются для взаимодействия с пользователями с помощью интерфейсов /proc/irq и /proc/interrupts, которые вскоре будут рассмотрены.

Пятый параметр, devid, в основном, применяется для совместно используемых линий запросов на прерывания. Когда обработчик прерывания освобождается (описано ниже), параметр dev_id обеспечивает уникальный идентификатор (cookie), который позволяет удалять только необходимый обработчик линии прерывания. Без этого параметра было бы невозможно ядру определить, какой обработчик данной линии прерывания следует удалить. Если линия запроса на прерывание не является совместно используемой, то можно в качестве этого параметра указывать NULL, если же номер прерывания является совместно используемым, то необходимо указывать уникальный идентификатор (cookie) (если устройство не подключено к тине ISA, то, скорее всего, оно поддерживает совместно используемые номера прерываний).

Этот параметр также передается обработчику прерывания при каждом вызове. Обычная практика — это передача указателя на структуру устройства (контекст устройства), так как этот параметр является уникальным, и, кроме того, в обработчике прерывания может быть полезным иметь указатель на эту структуру.

В случае успеха функция request_irq () возвращает нуль. Возврат ненулевого значения указывает на то, что произошла ошибка и указанный обработчик прерывания не был зарегистрирован. Наиболее часто встречающийся код ошибки — это значение -EBUSY, что указывает на то, что данная линия запроса на прерывание уже занята (и или при текущем вызове, или при первом вызове не был указан флаг SA_SHIRQ).

Следует обратить внимание, что функция request_irq () может переходить в состояние ожидания (sleep) и, соответственно, не может вызываться из контекста прерывания, или в других ситуациях, когда код не может блокироваться. Распространенной ошибкой является мнение, что функцию request_irq() можно безопасно вызывать в случаях, когда нельзя переходить в состояние ожидания. Это происходит отчасти от того, что действительно сразу непонятно, почему функция request_irq() должна чего-то ожидать. Дело в том. что при регистрации происходит добавление информации о линии прерывания в каталоге /proc/irq. Функция proc_mkdir () используется для создания новых элементов на файловой системе procfs. Эта функция вызывает функцию proc_create () для создания новых элементов файловой системы procfs, которая в свою очередь вызывает функцию kmalloc () для выделения памяти. Как будет показано в главе 11, "Управление памятью", функция kmalloc () может переходить в состояние ожидания. Вот так вот!

Для регистрации линии прерывания и инсталляции обработчика в коде драйвера можно использовать следующий вызов.

```
if (request_irq(irqn, my_interrupt, SA_SHIRQ, "my_device", dev)){
        printk(KERN_ERR "my_device: cannot register IRQ %d\n", irqn);
        return -EIO;
}
```

В этом примере параметр irqn — это запрошенный номер линии запроса на прерывание, параметр my_interrupt — это обработчик этой линии прерывания, линия запроса на прерывание может быть совместно используемой, имя устройства — "my_device", dev — значение параметра dev_id. В случае ошибки код печатает сообщение, что произошла ошибка, и возвращается из выполняющейся функции. Если функция регистрации возвращает нулевое значение, то обработчик прерывания инсталлирован успешно. С этого момента обработчик прерывания будет вызываться в ответ на приходящие прерывания. Важно произвести инициализацию оборудования и регистрацию обработчика прерывания в правильной последовательности, чтобы предотвратить возможность вызова обработчика до того момента, пока оборудование не инициализировано.

Освобождение обработчика прерывания

Для освобождения линии прерывания необходимо вызвать функцию

```
void free_irq(unsigned int irq, void *dev_id)
```

Если указанная линия не является совместно используемой, то эта функция удаляет обработчик и запрещает линию прерывания. Если линия запроса на прерывание является совместно используемой, то удаляется обработчик, соответствующий параметру dev_id. Линия запроса на прерывание также запрещается, когда удаляется последний обработчик. Теперь понятно, почему важно передавать уникальное значение параметра dev_id. При использовании совместно используемых прерываний требуется уникальный идентификатор для того, чтобы отличать друг от друга различные обработчики, связанные с одним номером прерывания, и позволить функции free_irq() удалять правильный обработчик. В любом случае, если параметр devoid не равен значению NULL, то он должен соответствовать тому обработчику, который удаляется.

Вызов функции free_irq() должен производиться из контекста процесса.

Таблица 6.1. Список функций управления регистрацией прерываний

Функция	Описание
request_irq ()	Зарегистрировать заданный обработчик прерывания для заданной линии прерывания
free_irq ()	Освободить указанный обработчик прерывания. Если с линией прерывания больше не связан ни один обработчик, то запретить указанную линию прерывания

Написание обработчика прерывания

Следующее описание является типичным для обработчика прерывания.

```
static irqreturn_t intr_handler (int irq, void *dev_id, struct pt_regs *regs)
```

Заметим, что оно должно соответствовать аргументу, который передается в функцию request_irq (). Первый параметр, irq, — это численное значение номера прерывания, которое обслуживается обработчиком. Сейчас этот параметр практически не используется, кроме разве что при печати сообщений. Для версий ядра, меньших 2.0, не было параметра dev_id, поэтому параметр irq использовался, чтобы различать устройства, которые обслуживаются одним драйвером, и поэтому используют один и тот же обработчик прерываний (как пример можно рассмотреть компьютер с несколькими контроллерами жесткого диска одного типа).

Второй параметр, dev_id, — это указатель, равный значению, которое было передано в функцию request_irq () при регистрации обработчика прерывания. Если значение этого параметра является уникальным, что необходимо для поддержки совместно используемых прерываний, то его можно использовать как идентификатор для того, чтобы отличать друг от друга различные устройства, которые потенциально могут использовать один обработчик. В связи с тем, что структура (контекст) устройства (device structure) является как уникальной, так и, возможно, полезной при использовании в обработчике, обычно в качестве параметра dev_id передают указатель на эту структуру.

Последний параметр, regs, — это указатель на структуру, содержащую значения регистров процессора и состояние процессора, которые были сохранены перед началом обслуживания прерывания. Этот параметр используется редко, в основном

для отладки. Сейчас разработчики начинают склоняться к мысли, что этот параметр нужно убрать. В существующих обработчиках прерываний он используется мало, и если его убрать, то не будет больших разочарований.

Возвращаемое значение обработчиков прерываний имеет специальный тип irqreturn_t. Обработчик может возвращать два специальных значения: IRQ_NONE или IRQ_HANDLED. Первое значение возвращается, если обработчик прерывания обнаружил, что устройство, которое он обслуживает, не является источником прерывания. Второе значение возвращается, если обработчик вызван правильно и устройство, которое он обслуживает, является источником прерывания. Кроме этого, может быть использован макрос IRQ_RETVAL *(x)*. Если значение параметра x не равно нулю, то макрос возвращает значение IRQ_HANDLED, иначе возвращается значение, равное IRQ_NONE. Эти специальные значения позволяют дать ядру информацию о том, генерирует ли устройство паразитные (необрабатываемые) прерывания. Если все обработчики прерывания, которые обслуживают данную линию, возвращают значение IRQ_NONE, то ядро может обнаружить проблему. Заметим, что этот странный тип возвращаемого значения, irqreturn_t, просто соответствует типу int. Подстановка типа используется для того, чтобы обеспечить совместимость с более ранними версиями ядра, у которых не было подобной функции. До серии ядер 2.6 обработчик прерывания имел возвращаемое значение типа void. В коде новых драйверов можно применить переопределение типа typedef irqreturn_t в тип void и драйверы могут работать с ядрами серии 2.4 без дальнейшей модификации.

Обработчик прерываний может помечаться как static, так как он никогда не вызывается непосредственно в других файлах кода.

Роль обработчика прерывания зависит только от устройства и тех причин, по которым это устройство генерирует прерывания. Минимально, обработчик прерывания должен отправить устройству подтверждение о том, что прерывание получено. Для более сложных устройств необходимо дополнительно отправить и принять данные, а также выполнить другую более сложную работу. Как уже упоминалось, сложная работа должна по возможности выполняться обработчиком нижней половины прерывания, которые будут рассмотрены в следующей главе.

Реентерабельность и обработчики прерываний

Обработчики прерываний в операционной системе Linux не обязаны быть реентерабельными. Когда выполняется некоторый обработчик прерывания, соответствующая линия запроса на прерывание маскируется на всех процессорах, что предотвращает возможность приема запроса на прерывание с этой пинии. Обычно все остальные прерывания в этот момент разрешены, поэтому другие прерывания могут обслуживаться, тогда как текущая линия всегда является запрещенной. Следовательно, никакой обработчик прерываний никогда не вызывается параллельно самому себе для обработки вложенных запросов на прерывание. Это позволяет значительно упростить написание обработчиков прерываний.

Совместно используемые обработчики

Совместно используемые (shared) обработчики выполняются практически так же, как и не совместно используемые. Существует, однако, три главных отличия.

- Флат SA_SHIRQ должен быть установлен в параметре flags при вызове функции request_irq ().

- Аргумент dev_id этой же функции должен быть уникальным для каждого зарегистрированного обработчика. Достаточным является передача указателя на структуру, которая описывает устройство. Обычно так и поступают, поскольку структура контекста устройства является уникальной для каждого устройства, и, кроме того, данные этой структуры потенциально могут быть полезными при выполнении обработчика. Для совместно используемого обработчика нельзя присваивать параметру dev_id значение NULL!
- Обработчик прерывания должен иметь возможность распознать, сгенерировано ли прерывание тем устройством, которое обслуживается этим обработчиком. Для этого требуется как поддержка аппаратного обеспечения, так и наличие соответствующей логики в обработчике прерывания. Если аппаратное устройство не имеет необходимых функций, то не будет никакой возможности в обработчике прерывания определить, какое из устройств на совместно используемой линии является источником прерывания.

Все драйверы, которые рассчитаны на совместно используемую линию прерывания, должны удовлетворять указанным выше требованиям. Если хотя бы одно из устройств, которые совместно используют линию прерывания, не делает это корректно, то все остальные устройства также не смогут совместно использовать линию. Если функция request_irq () вызывается с указанием флага SH_SHIRQ, то этот вызов будет успешным только в том случае, если для данной линии прерывания еще нет зарегистрированных обработчиков или все обработчики для данной линии зарегистрированы с указанием флага SH_SHIRQ. Заметим, что для ядер серии 2.6, в отличие от более ранних серий, можно "смешивать" совместно используемые обработчики с различными значениями флага SA_INTERRUPT.

Когда ядро получает прерывание, то оно последовательно вызывает все обработчики, зарегистрированные для данной линии. Поэтому важно, чтобы обработчик прерывания был в состоянии определить, какое устройство является источником этого прерывания. Обработчик должен быстро завершиться, если соответствующее ему устройство не генерировало это прерывание. Такое условие требует, чтобы аппаратное устройство имело регистр состояния (status register) или другой аналогичный механизм, которым обработчик может воспользоваться для проверки. На самом деле большинство устройств действительно имеют данную функцию.

Настоящий обработчик прерывания

Давайте рассмотрим настоящий обработчик прерывания, который используется в драйвере устройства RTC (real-time clock, часы реального времени), находящегося в файле drivers/char/rtc.c. Устройство RTC есть во многих вычислительных системах, включая персональные компьютеры (PC). Это отдельное от системного таймера устройство, которое используется для установки системных часов, для подачи сигналов таймера (alarm) или для реализации генераторов периодических сигналов (periodic timer). Установка системных часов обычно производится путем записи значений в специальный регистр или диапазон адресов (номеров портов) ввода-вывода (I/O range). Подача сигналов таймера или генератор периодических сигналов обычно реализуются через прерывания. Прерывание эквивалентно некоторому сигналу таймера: оно генерируется, когда истекает период времени сигнального таймера. При загрузке драйвера устройства RTC вызывается функция rtc init () для ини-

циализации драйвера. Одна из ее обязанностей — это регистрация обработчика прерывания. Делается это следующим образом.

```
if (request_irq(RTC_IRQ, rtc_interrupt, SA_INTERRUPT, "rtc", NULL) {
        printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
        return -EIO;
}
```

Из данного примера видно, что номер линии прерывания — это константа RTC_IRQ, значение которой определяется отдельно для каждой аппаратной платформы с помощью препроцессора. Например, для персональных компьютеров это значение всегда соответствует IRQ 8. Второй параметр— это обработчик прерывания, rtc interrupt, при выполнении которого запрещены все прерывания в связи с указанием флага SA_INTERRUPT. Из четвертого параметра можно заключить, что драйвер будет иметь имя "rtc". Так как наше устройство не может использовать линию прерывания совместно с другими устройствами и обработчик прерывания не используется для каких-либо других целей, в качестве параметра dev_id передается значение NULL.

И наконец, собственно сам обработчик прерывания.

```
/*
* Очень маленький обработчик прерывания. Он выполняется с
* установленным флагом SA_INTERRUPT, однако существует
* возможность конфликта с выполнением функции set_rtc_mmss ()
*  (обработчик прерывания rtc и обработчик прерывания системного
* таймера могут выполняться одновременно на двух разных
* процессорах). Следовательно, необходимо сериализировать доступ
* к микросхеме с помощью спин-блокировки rtc_lock, что должно
* быть сделано для всех аппаратных платформ в коде работы с
* таймером.  (Тело функции set_rtc_mmss() ищите в файлах
*  ./arch/XXXX/kernel/time.c)
*/
static irqreturn_t rtc_interrupt(int irq, void *dev_id, struct pt_regs *regs)

        /*
        * Прерывание может оказаться прерыванием таймера, прерыванием
        *  завершения обновления или периодическим прерыванием.
        * Состояние (причина) прерывания хранится в самом
        * младшем байте, а общее количество прерывание — в оставшейся
        *  части переменной rtc_irq_data
        */

        spin_lock (&rtc_lock);

        rtc_irq_data += 0x100;
        rtc_irq_data &= ~0xff;
        rtc_irq_data |= (CMOS_READ(RTC_INTR_FLAGS) & 0xF0);

        if (rtc_status & RTC_TIMER_ON)
              rnod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);
```

```
    spin_unlock(&rtc_lock);

    /*
    * Теперь выполним остальные действия
/*
    spin_lock(&rtc_task_lock);
    if (rtc_callback)
         rtc_callback->func(rtc_callback->private_data);
    spin_unlock(&rtc_task_lock);
    wake_up_interruptible(&rtc_wait);

    kill_fasync(&rtc_async_queue, SIGIO, POLL_IN);

    return IRQ_HANDLED;

    }
```

Эта функция вызывается всякий раз, когда система получает прерывание от устройства RTC. Прежде всего, следует обратить внимание на вызовы функций работы со спин-блокировками: первая группа вызовов гарантирует, что к переменной rtc_irq_data не будет конкурентных обращений другими процессами на SMP-машине, а вторая — защищает в аналогичной ситуации параметры структуры rtc_callback. Блокировки обсуждаются в главе 9, "Средства синхронизации в ядре".

Переменная rtc_irq data содержит информацию об устройстпе RTC и обновляется с помощью функции modtimer (). О таймерах рассказывается в главе 10, "Таймеры и управление временем".

Последняя часть кода, окруженная спин-блокировками, выполняет функцию обратного вызова (callback), которая может быть установлена извне. Драйвер RTC позволяет устанавливать функцию обратного вызова, которая может быть зарегистрирована пользователем и будет исполняться при каждом прерывании, приходящем от устройства RTC.

В конце функция обработки прерывания возвращает значение IRQ_HANDLED, чтобы указать, что прерывание от данного устройства обработано правильно. Так как этот обработчик прерывания не поддерживает совместное использование линий прерывания и не существует механизма, посредством которого обработчик прерываний RTC может обнаружить вложенные запросы на прерывание, то этот обработчик всегда возвращает значение IRQ_HANDLED.

Контекст прерывания

При выполнении обработчика прерывания или обработчика нижней половины, ядро находится в *контексте прерывания*. Вспомним, что контекст процесса — это режим, в котором работает ядро, выполняя работу от имени процесса, например выполнение системного пызова или потока пространства ядра. В контексте процесса макрос current возвращает указатель на соответствующее задание. Более того, поскольку в контексте процесса процесс связан с ядром, то контекст процесса может переходить в состояние ожидания или использовать функции планировщика каким-либо другим способом..

В противоположность только что рассмотренному, контекст прерывания не связан ни с одним процессом. Макрос `current` в контексте прерывания является незаконным (хотя он и указывает на процесс, выполнение которого было прервано). Так как нет процесса, то контекст прерывания не может переходить в состояние ожидания (sleep) — действительно, каким образом можно перепланировать его выполнение? Поэтому некоторые функции ядра не могут быть вызваны из контекста прерывания. Если функция может переводить процесс в состояние ожидания, то ее нельзя вызывать в обработчике прерывания, что ограничивает набор функций, которые можно использовать в обработчиках прерываний.

Контекст прерывания является критичным ко времени исполнения, так как обработчик прерывания прерывает выполнение некоторого программного кода. Код же самого обработчика должен быть простой и быстрый. Использование циклов проверки состояния чего-либо (busy loop) крайне нежелательно. Это очень важный момент. Всегда следует помнить, что обработчик прерывания прерывает работу некоторого кода (возможно, даже обработчика другой линии запроса на прерывание!). В связи со своей асинхронной природой обработчики прерыпаний должны быть как можно более быстрыми и простыми. Максимально возможную часть работы необходимо изъять из обработчика прерывания и переложить на обработчик нижней половины, который выполняется в более подходящее время.

Возможность установить стек контекста прерывания является конфигурируемой. Исторически, обработчик прерывания не имеет своего стека. Вместо этого он должен был использовать стек ядра прерванного процесса[1]. Стек ядра имеет размер две страницы памяти, что обычно соответствует 8 Кбайт для 32-разрядных аппаратных платформ и 16 Кбайт для 64-разрядных платформ. Так как в таком случае обработчики прерываний совместно используют стек, то они должны быть очень экономными в отношении того, что они в этом стеке выделяют. Конечно, стек ядра изначально является ограниченным, поэтому любой код ядра должен принимать это во внимание.

В ранних версиях ядер серии 2.6 была введена возможность ограничить размер стека ядра от двух до одной страницы памяти, что равно 4 Кбайт на 32-разрядных аппаратных платформах. Это уменьшает затраты памяти, потому что раньше каждый процесс требовал две страницы памяти ядра, которая не может быть вытеснена на диск. Чтобы иметь возможность работать со стеком уменьшенного размера, каждому обработчику прерывания выделяется свой стек, отдельный для каждого процессора. Этот стек называется *стеком прерывания*. Хотя общий размер стека прерывания и равен половине от первоначально размера совместно используемого стека, тем не менее в результате выходит, что суммарный размер стека получается большим, потому что на каждый стек прерывания выделяется целая страница памяти.

Обработчик прерывания не должен зависеть от того, какие настройки стека используются и чему равен размер стека ядра. Всегда необходимо использовать минимально возможное количество памяти в стеке.

[1] Какой-нибудь процесс выполняется всегда. Если не выполняется никакой процесс, то выполняется холостая задача (idle task).

Реализация системы обработки прерываний

Возможно, не вызовет удивления, что реализация системы обработки прерываний в операционной системе Linux очень сильно зависит от аппаратной платформы. Она зависит от типа процессора, типа контроллера прерываний, особенностей аппаратной платформы и устройства самой вычислительной машины.

На рис. 6.1 показана диаграмма пути, который проходит запрос на прерывание в аппаратном обеспечении и в ядре.

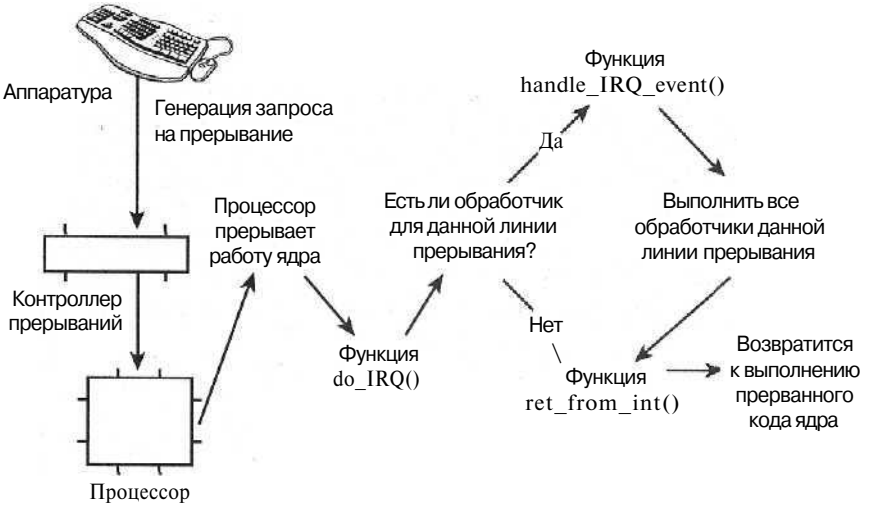


Рис. 6.1. Прохождение запроса на прерывание в аппаратном обеспечении и в ядре

Устройство инициирует прерывание путем отправки электрического сигнала контроллеру прерывания по аппаратной шине. Если соответствующая линия запроса на прерывание не запрещена (линия может быть в данный момент времени замаскирована), то контроллер прерываний отправляет прерывание процессору. Для большинства аппаратных платформ это осуществляется путем подачи сигнала на специальный вывод процессора. Если прерывания не запрещены в процессоре (может случиться, что они запрещены), то процессор немедленно прекращает ту работу, которую он выполнял, запрещает систему прерываний, осуществляет переход на специальный предопределенный адрес памяти и начинает выполнять программный код, который находится по этому адресу. Этот предопределенный адрес памяти устанавливается ядром и является точкой входа в обработчики прерываний.

Прохождение прерывания в ядре начинается из жестко определенной точки входа, так же как и в случае системных вызовов. Для каждой линии прерывания существует своя уникальная точка, куда переходит процессор. Именно этим способом ядро получает информацию о номере IRQ приходящего прерывания. В точке входа сначала в стеке ядра сохраняется значение номера прерывания и значения всех регистров процессора (которые соответствуют прерванному заданию). После этого ядро вызывает функцию do_IRQ (). Далее, начиная с этого момента, почти весь код обработки прерываний написан на языке программирования С, хотя несмотря на это код все же остается зависимым от аппаратной платформы.

Функция do_IRQ() определена следующим образом.

```
unsigned int do_IRQ(struct pt_regs regs)
```

Так как соглашение о вызовах функций в языке C предусматривает сохранение аргументов функций в вершине стека, то структура ptregs содержит первоначальные значения всех регистров процессора, которые были сохранены ассемблерной подпрограммой в точке входа. Так как значение номера прерывания также сохраняется, то функция do_IRQ () может это значение восстановить. Для аппаратной платформы x86 код будет следующим.

```
int irq = regs.orig_eax & 0xff;
```

После вычисления значения номера линии прерывания, функция do_IRQ() отправляет уведомление о получении прерывания и запрещает доставку прерываний с данной линии. Для обычных машин платформы PC, эти действия выполняются с помощью функции mask_and_ack_8295A (), которую вызывает функция do_IRQ (). Далее функция do_IRQ () выполняет проверку, что для данной линии прерывания зарегистрирован правильный обработчик прерывания, что этот обработчик разрешен и что он не выполняется в данный момент. Если все эти условия выполнены, то вызывается функция handle_IRQ_event (), которая выполняет установленные для данной линии обработчики прерывания. Для аппаратной платформы x86 функция handle_IRQ_event () имеет следующий вид.

```
int handle_IRQ_event (unsigned int irq, struct pt_regs *regs,
struct irqaction *action)
{
        int status = 1;
        if (!(action->flags & SA_INTERRUPT))
              local_irq_enable  ();
        do {
              status != action->flags;
              action->chandler(irq, action->dev_id, regs);
              action = action->next;
        } while (action);
        if (status & SA_SAMPLE_RANDOM)
              add_interrupt_randomness (irq);
        local_irq_disable();
        return status;
}
```

Так как процессор запретил прерывания, они снова разрешаются, если не указан флаг SA_INTERRUPT при регистрации обработчика. Вспомним, что флаг SA_INTERRUPT указывает, что обработчик должен выполняться при всех запрещенных прерываниях. Далее в цикле вызываются все потенциальные обработчики прерываний. Если эта линия не является совместно используемой, то цикл заканчивается после первой итерации. В противном случае вызываются все обработчики. После этого вызывается функция add_interrupt_randomness (), если при регистрации указан флаг SA_SAMPLE_RANDOM. Данная функция использует временные характеристики прерывания, чтобы сгенерировать значение энтропии для генератора случайных чисел. В приложении Б, "Генератор случайных чисел ядра", приведена более подробная информация о генераторе случайных чисел ядра.

В конце прерывания снова запрещаются (для функции do_IRQ () требуется, чтобы прерывания были запрещены). Функция do_IRQ () производит очистку стека и возврат к первоначальной точке входа, откуда осуществляется переход к функции ret_from_intr().

Функция ret_from_intr (), так же как и код входа, написана на языке ассемблера. Эта функция проверяет, есть ли ожидающий запрос на перепланирование выполнения процессов (следует вспомнить главу 4, "Планирование выполнения процессов", и флаг need_resched). Если есть запрос на перепланирование и ядро должно передать управление в пространство пользователя (т.е. прерывание прервало работу пользовательского процесса), то вызывается функция schedule () . Если возврат производится в пространство ядра (т.е. прерывание прервало работу кода ядра), то функция schedule () вызывается, только если значение счетчика preempt_count равно нулю (в противном случае небезопасно производить вытеснение кода ядра), После возврата из функции schedule () или если нет никакой ожидающей работы, восстанавливаются первоначальные значения регистров процессора и ядро продолжает работу там, где оно было прервано.

Для платформы x86, подпрограммы, написанные на языке ассемблера, находятся в файле arch/i386/kernel/entry.S, а соответствующие функции на языке C — в файле arch/i386/kernel/irq.c. Для других поддерживаемых аппаратных платформ имеются аналогичные файлы.

Инперфейс /proc/interrupts

Файловая система *procfs*— это виртуальная файловая система, которая существует только в памяти ядра и обычно монтируется на каталог /proc. Чтение или запись файлов на файловой системе procfs приводит к вызовам функций ядра, которые имитируют чтение или запись обычных файлов. Важный пример— это файл /proc/interrupts, который содержит статистику, связанную с прерываниями в системе, Ниже приведен пример вывода из этого файла на однопроцессорном персональном компьютере.

```
           CPU0
  0:    3602371  XT-PIC     timer
  1:       3048  XT-PIC     i8042
  2:          0  XT-PIC     cascade
  4:    2689466  XT-PIC     uhci-hed, ethO
  5:          0  XT-PIC     EMU10K1
 12:      85077  XT-PIC     uhei-hcd
 15:      24571  XT-PIC     aic7xxx
NMI:          0
LOC:    3602236
ERR:          0
```

Первая колонка содержит названия линий прерывания. В показанной системе присутствуют линии прерываний с номерами 0-2, 4, 5, 12 и 15. Линии, для которых не инсталлирован обработчик, не показываются. Вторая колонка — это количество запросов на прерывания с данным номером. В действительности такая колонка является отдельной для каждого процессора, но в данной машине только один процессор.

Как легко видеть, обработчик прерываний таймера получил 3.602.371[2] запрос на прерывание, в то время как обработчик прерываний звукового адаптера (EMU10K1) не получил ни одного прерывания (это говорит о том, что он не использовался с того момента, как машина была загружена). Третья колонка— это контроллер прерываний, который обслуживает данное прерывание. Значение XT-PIC соответствует программируемому контроллеру прерываний PC (PC programmable interrupt controller). Для систем с устройством I/O APIC для большинства прерываний в качестве контроллера прерываний будет указано значение IO-APIC-level или IO-APIC-edge. И наконец, последняя колонка— это устройство, которое связано с прерыванием. Имя устройства указывается в параметре dev_name при вызове функции request_irq () , как обсуждалось ранее. Если прерывание используется совместно, как в случае прерывания номер 4 в этом примере, то перечисляются все устройства, зарегистрированные на данной линии прерывания.

Для любопытствующих, код, связанный с файловой системой procfs, находится в файле fs/proc. Функция, которая обеспечивает работу интерфейса /proc/interrupts, называется show_interrupts () и является зависимой от аппаратной платформы.

Управление прерываниями

В ядре Linux реализовано семейство интерфейсов для управления состояниями прерываний в машине. Эти интерфейсы позволяют запрещать прерывания для текущего процессора или маскировать линию прерывания для всей машины. Эти функции очень сильно зависят от аппаратной платформы и находятся в файлах <asm/system. h> и <asm/irq. h>. В табл. 6.2 приведен полный список этих интерфейсов.

Причины, по которым необходимо управлять системой обработки прерываний, в основном, сводятся к необходимости обеспечения синхронизации. Путем запрещения прерываний можно гарантировать, что обработчик прерывания не вытеснит текущий исполняемый код. Более того, запрещение прерываний также запрещает и вытеснение кода ядра. Однако ни запрещение доставки прерываний, ни запрещение преемптивности ядра не дают никакой защиты от конкурентного обращения других процессоров. Так как операционная система Linux поддерживает многопроцессорные системы, в большинстве случаев код ядра должен захватить некоторую блокировку, чтобы предотвратить доступ другого процессора к совместно используемым данным. Эти блокировки обычно захватываются в комбинации с запрещением прерываний на текущем процессоре. Блокировка предоставляет защиту от доступа другого процессора, а запрещение прерываний обеспечивает защиту от конкурентного доступа из возможного обработчика прерывания. В главах 8 и 9 обсуждаются различные аспекты проблем синхронизации и решения этих проблем.

Тем не менее понимание интерфейсов ядра для управления прерываниями является важным.

[2] После прочтения главы 10, "Таймеры и управление временем", можно ли сказать, сколько времени (в единицах HZ) машина работала без перегрузки исходя из числа прерываний таймера?

Запрещение и разрешение прерываний

Для локального запрещения прерываний на текущем процессоре (и *только* на текущем процессоре) и последующего разрешения можно использовать следующий код.

```
local_irq_disable();
/* прерывания запрещены .. */
local_irq_enable();
```

Эти функции обычно реализуются в виде одной инструкции на языке ассемблера (что, конечно, зависит от аппаратной платформы). Для платформы x86 функция `local_irq_disable ()` — это просто машинная инструкция `cli`, а функция `local_irq_enable ()` — просто инструкция `sti`. Для хакеров, не знакомых с платформой x86, `sti` и `cli` — это ассемблерные вызовы, которые соответственно позволяют установить *(set)* или очистить *(clear)* флаг разрешения прерываний *(allow interrupt flag)*. Другими словами, они разрешают или запрещают доставку прерываний на вызвавшем их процессоре.

Функция `local_irq_disable ()` является опасной в случае, когда *перед* ее вызовом прерывания уже были запрещены. При этом соответствующий ей вызов функции `local_irq_enable ()` разрешит прерывания независимо от того, были они запрещены первоначально (до вызова `local_irq_disable ()`) или нет. Для того чтобы избежать такой ситуации, необходим механизм, который позволяет восстанавливать состояние системы обработки прерывании в первоначальное значение. Это требование имеет общий характер, потому что некоторый участок кода ядра в одном случае может выполняться при разрешенных прерываниях, а в другом случае— при запрещенных, в зависимости от последовательности вызовов функций. Например, пусть показанный фрагмент кода является частью функции. Эта функция вызывается двумя другими функциями, и в первом случае перед вызовом прерывания запрещаются, а во втором — нет. Так как при увеличении объема кода ядра становится сложно отслеживать все возможные варианты вызова функции, значительно безопаснее сохранять состояние системы прерываний перед тем, как запрещать прерывания. Вместо разрешения прерываний просто восстанавливается первоначальное состояние системы обработки прерываний следующим образом.

```
unsigned long flags;

local_irq_save(flags);
/* прерывания запрещены . . */
local_irq_restore(flags);
/*состояние системы прерываний восстановлено в первоначальное значение ..*/
```

Нужно заметить, что эти функции являются макросами, поэтому передача параметра `flags` выглядит как передача по значению. Этот параметр содержит зависящие от аппаратной платформы данные, которые в свою очередь содержат состояние системы прерываний. Так как, по крайней мере для одной аппаратной платформы (SPARC), в этой неременной хранится информация о стеке, то параметр `flags` *нельзя* передавать в другие функции (другими словами, он должен оставаться в одном и том же стековом фрейме). По этой причине вызовы функций сохранения и восстановления должны выполняться в теле одной функции.

Все описанные функции могут вызываться как из обработчика прерываний, так и из контекста процесса.

Больше нет глобального вызова cli ()

Ранее ядро предоставляло функцию, с помощью которой можно было запретить прерывания на всех процессорах системы. Более того, если какой-либо процессор вызывал эту функцию, то он должен был ждать, пока прерывания не будут разрешены. Эта функция называлась `cli()`, а соответствующая ей разрешающая функция — `sti();` очень "x86-центрично" (хотя и было доступно для всех аппаратных платформ). Эти интерфейсы были изъяты во время разработки ядер серии 2.5, и, следовательно, все операции по синхронизации обработки прерываний должны использовать комбинацию функций по управлению локальными прерываниями и функций работы со спин-блокировками (обсуждаются в главе 9, "Средства синхронизации в ядре"). Это означает, что код, который ранее должен был всего лишь глобально запретить прерывания для того, чтобы получить монопольный доступ к совместно используемым данным, теперь должен выполнить несколько больше работы.

Ранее разработчики драйверов могли считать, что если в их обработчике прерывания и в любом другом коде, который имеет доступ к совместно используемым данным, вызывается функция `cli()`, то это позволяет получить монопольный доступ. Функция `cli()` позволяла гарантировать, что ни один из обработчиков прерываний (в том числе и другие экземпляры текущего обработчика) не выполняется. Более того, если любой другой процессор входит в участок кода, защищенный с помощью функции `cli()`, то он не продолжит работу, пока первый процессор не выйдет из участка кода, защищенного с помощью функции `cli()`, т.е. не вызовет функцию `sti()`.

Изъятие глобальной функции `cli()` имеет несколько преимуществ. Во-первых, это подталкивает разработчиков драйверов к использованию настоящих блокировок. Специальные блокировки на уровне мелких структурных единиц работают быстрее, чем глобальные блокировки, к которым относится и функция `cli()`. Во-вторых, это упрощает значительную часть кода и позволяет удалить большой объем кода. В результате система обработки прерываний стала проще и понятнее.

Запрещение определенной линии прерывания

В предыдущем разделе были рассмотрены функции, которые позволяют запретить доставку всех прерываний на определенном процессоре. В некоторых случаях полезным может оказаться запрещение *определенной* линии прерывания во *всей* Системе. Это называется *маскированием* линии прерывания. Например, может потребоваться запрещение доставки прерывания от некоторого устройства перед манипуляциями с его состоянием. Для этой цели операционная система Linux предоставляет четыре интерфейса.

```
void disable_irq(unsigned int irq);
void disable_irq_nosync(unsigned int irq);
void enable_irq(unsigned int irq);
void synchronize_irq(unsigned int irq);
```

Первые две функции позволяют запретить указанную линию прерывания в контроллере прерываний. Это запрещает доставку данного прерывания *всем* процессорам в системе. Кроме того, функция `disable_irq()` не возвращается до тех пор, пока все обработчики прерываний, которые в данный момент выполняются, не закончат работу. Таким образом гарантируется не только то, что прерыпания с данной

линии не будут доставляться, но и то, что все выполняющиеся обработчики закончили работу. Функция disable_irq_nosync () не имеет последнего свойства.

Функция synchronize_irq () будет ожидать, пока не завершится указанный обработчик, если он, конечно, выполняется.

Вызовы этих функций должны быть сгруппированы, т.е. каждому вызову функции disable_irq () или disable_irq_nosync () должен соответствовать вызов функции enable_irq (). Только после последнего вызова функции enable_irq () линия запроса на прерывание будет снова разрешена. Например, если функция disable_irq () последовательно вызвана два раза, то линия запроса на прерывание не будет разрешена, пока функция enable_irq () тоже не будет вызвана два раза.

Эти три функции могут быть вызваны из контекста прерывания и из контекста процесса и не приводят к переходу в приостановленное состояние (sleep). При вызове из контекста прерывания следует быть осторожным! Например, нельзя разрешать линию прерывания во время выполнения обработчика прерывания (вспомним, что линия запроса на прерывание обработчика, который в данный момент выполняется, является замаскированной).

Было бы также плохим тоном запрещать линию прерывания, которая совместно используется несколькими обработчиками. Запрещение линии прерывания запрещает доставку прерываний для *всех* устройств, которые используют эту линию. Поэтому в драйверах новых устройств не рекомендуется использовать эти интерфейсы[3]. Так как устройства PCI должны согласно спецификации поддерживать совместное использование линий прерываний, они вообще не должны использовать эти интерфейсы. Поэтому функция disable_irq () и дружественные ей обычно используются для устаревших устройств, таких как параллельный порт персонального компьютера.

Состояние системы обработки прерываний

Часто необходимо знать состояние системы обработки прерываний (например, прерывания запрещены или разрешены, выполняется ли текущий код в контексте прерывания или в контексте процесса).

Макрос irq_disabled (), который определен в файле <asm/system.h>, возвращает ненулевое значение, если обработка прерываний на локальном процессоре запрещена. В противном случае возвращается нуль. Два следующих макроса позволяют определить контекст, в котором в данный момент выполняется ядро.

```
in_interrupt()
in_irq()
```

Наиболее полезный из них — это первый макрос. Он возвращает ненулевое значение, если ядро выполняется в контексте прерывания. Это включает выполнение как обработчика прерывания, так и обработчика нижней половины. Макрос in_irq() возвращает ненулевое значение, только если ядро выполняет обработчик прерывания.

[3] Многие старые устройства, в частности устройства ISA, не предоставляют возможности определить, являются ли они источником прерывания. Из-за этого линии прерывания для ISA-устройств часто не могут быть совместно используемыми. Поскольку спецификация шины PCI требует обязательной поддержки совместно используемых прерываний, современные устройства PCI поддерживают совместное использование прерываний. В современных компьютерах практически все линии прерываний могут быть совместно используемыми.

Наиболее часто необходимо проверять, выполняется ли код в контексте процесса, т.е. необходимо проверить, что код выполняется *не* в контексте прерывания. Это требуется достаточно часто, когда коду необходимо выполнять что-то, что может быть выполнено только из контекста процесса, например переход в приостановленное состояние. Если макрос `in_interrupt()` возвращает нулевое значение, то ядро выполняется в контексте процесса.

Таблица 6.2. Список функций управления прерываниями

Функция	Описание
`local_irq_disable()`	Запретить доставку прерываний на локальном процессоре
`local_irq_enable()`	Разрешить доставку прерываний на локальном процессоре
`local_irq_save(unsigned long flags)`	Сохранить текущее состояние системы обработки прерываний на локальном процессоре и запретить прерывания
`local_irq_restore(unsigned long flags)`	Восстановить указанное состояние системы прерываний на локальном процессоре
`disable_irq(unsigned int irq)`	Запретить указанную линию прерывания с гарантией, что после возврата из этой функции не выполняется ни один обработчик данной линии
`disable_irq_nosync(unsigned int irq)`	Запретить указанную линию прерывания
`enable_irq(unsigned int irq)`	Разрешить указанную линию прерываний
`irqs_disabled()`	Возвратить ненулевое значение, если запрещена доставка прерываний на локальном процессоре, в противном случае возвращается нуль
`in_interrupt()`	Возвратить ненулевое значение, если выполнение производится в контексте прерывания, и нуль — если в контексте процесса
`in_irq()`	Возвратить ненулевое значение, если выполнение производится в контексте прерывания, и нуль — в противном случае

Не нужно прерывать, мы почти закончили!

В этой главе были рассмотрены прерывания, аппаратные ресурсы, которые используются устройствами для подачи асинхронных сигналов процессору. Прерывания используются аппаратным обеспечением, чтобы *прервать* работу операционной системы.

Большинство современного аппаратного обеспечения использует прерывания, чтобы взаимодействовать с операционной системой. Драйвер устройства, который управляет некоторым оборудованием, должен зарегистрировать обработчик прерывания, чтобы отвечать на эти прерывания и обрабатывать их. Работа, которая выполняется обработчиками прерываний, включает отправку подтверждения устройству о получении прерывания, инициализацию аппаратного устройства, копирование дан-

ных из памяти устройства в память системы и, наоборот, обработку аппаратных запросов и отправку ответов на них.

Ядро предоставляет интерфейсы для регистрации и освобождения обработчиков прерываний, запрещения прерываний, маскирования линий прерываний и проверки состояния системы прерываний. В табл. 6.2 приведен обзор некоторых из этих функций.

Так как прерывания прерывают выполнение другого кода (кода процессов, кода ядра и другие обработчики прерываний), то они должны выполняться быстро. Тем не менее часто приходится выполнять много работы. Для достижения компромисса между большим количеством работы и необходимостью быстрого выполнения обработка прерывания делится на две половины. Верхняя половина — собственно обработчик прерывания — рассматривается в этой главе. Теперь давайте рассмотрим нижнюю половину процесса обработки прерывания.

7

Обработка нижних половин и отложенные действия

В предыдущей главе были рассмотрены обработчики прерываний — механизм ядра, который позволяет решать задачи, связанные с аппаратными прерываниями. Конечно, обработчики прерываний очень полезны и являются необходимой частью ядра. Однако, в связи с некоторыми ограничениями, они представляют собой лишь часть процесса обработки прерываний. Эти ограничения включают следующие моменты.

- Обработчики прерываний выполняются асинхронно и потенциально могут прерывать выполнение другого важного кода (даже другие обработчики прерываний). Поэтому обработчики прерываний должны выполняться как можно быстрее.
- Обработчики прерываний выполняются в лучшем случае при запрещенной обрабатываемой линии прерывания и в худшем случае (когда установлен флаг SA_INTERRUPT) — при всех запрещенных линиях запросов на прерывания. И снова они должны выполняться как можно быстрее.
- Обработчики прерываний очень критичны ко времени выполнения, так как они имеют дело с аппаратным обеспечением.
- Обработчики прерываний не выполняются в контексте процесса, поэтому они не могут блокироваться.

Теперь должно быть очевидным, что обработчики прерываний являются только частью полного решения проблемы обработки аппаратных прерываний. Конечно, необходим быстрый, простой и асинхронный обработчик, позволяющий немедленно отвечать на запросы оборудования и выполнять критичную ко времени выполнения работу. Эту функцию обработчики прерываний выполняют хорошо, но другая, менее критичная ко времени выполнения работа должна быть отложена до того момента, когда прерывания будут разрешены.

Для этого обработка прерывания делится на две части или *половины.* Первая часть обработчика прерывания *(top half, верхняя половина)* выполняется асинхронно

и немедленно в ответ на аппаратное прерывание так, как это обсуждалось в предыдущей главе. В этой главе мы рассмотрим вторую часть процесса обработки прерываний — *нижние половины (bottom half)*.

Нижние половины

Задача обработки нижних половин — это выполнить всю связанную с прерываниями работу, которую не выполнил обработчик прерывания. В идеальной ситуации — это почти вся работа, так как необходимо, чтобы обработчик прерывания выполнил по возможности меньшую часть работы (т.е. выполнился максимально быстро) и побыстрее возвратил управление.

Тем не менее обработчик прерывания должен выполнить некоторые действия. Например, почти всегда обработчик прерывания должен отправить устройству уведомление, что прерывание получено. Он также должен произвести копирование некоторых данных из аппаратного устройства. Эта работа чувствительна ко времени выполнения, поэтому есть смысл выполнить ее в самом обработчике прерывания.

Практически все остальные действия будет правильным выполнить в обработчике нижней половины. Например, если в верхней половине было произведено копирование данных из аппаратного устройства в память, то в обработчике нижней половины, конечно, имеет смысл эти данные обработать. К сожалению, не существует твердых правил, позволяющих определить, какую работу где нужно выполнять, — право решения остается за автором драйвера. Хотя ни одно из решений этой задачи не может быть *неправильным,* решение легко может оказаться *неоптимальным.* Следует помнить, что обработчики прерываний выполняются асинхронно при запрещенной, по крайней мере, текущей линии запроса на прерывание. Минимизация этой задержки является важной. Хотя нет строгих правил по поводу того, как делить работу между обработчиками верхней и нижней половин, все же можно привести несколько полезных советов.

- Если работа критична ко времени выполнения, то ее необходимо выполнять в обработчике прерыпания.
- Если работа связана с аппаратным обеспечением, то ее следует выполнить в обработчике прерывания.
- Если для выполнения работы необходимо гарантировать, что другое прерывание (обычно с тем же номером) не прервет обработчик, то работу нужно выполнить в обработчике прерывания.
- Для всего остального работу стоит выполнять в обработчике нижней половины.

При написании собственного драйвера устройства есть смысл посмотреть на обработчики прерываний и соответствующие им обработчики нижних половин других драйверов устройств — это может помочь. Принимая решение о разделении работы между обработчиками верхней и нижней половины, следует спросить себя: "Что *должно* быть в обработчике верхней половины, а что *может* быть в обработчике нижней половины". В общем случае, чем быстрее выполняется обработчик прерывания, тем лучше.

Когда нужно использовать нижние половины

Часто не просто понять, зачем нужно откладывать работу и когда именно ее нужно откладывать. Вам необходимо ограничить количество работы, которая выполняется в обработчике прерывания, потому что обработчик прерывания выполняется при запрещенной текущей линии прерывания. Хуже того, обработчики, зарегистрированные с указанием флага SA_INTERRUPT, выполняются при *всех* запрещенных линиях прерываний на локальном процессоре (плюс текущая линия прерывания запрещена глобально). Минимизировать время, в течение которого прерывания запрещены, важно для уменьшения времени реакции и увеличения производительности системы. Если к этому добавить, что обработчики прерываний выполняются асинхронно по отношению к другому коду, и даже по отношению к другим обработчикам прерываний, то становится ясно, что нужно минимизировать время выполнения обработчика прерывания. Решение — отложить некоторую часть работы на более поздний срок.

Но что имеется в виду под более поздним сроком? Важно понять, что *позже* означает *не сейчас*. Основной момент в обработке нижних половин — это не отложить работу до *определенного* момента времени в будущем, а отложить работу до некоторого *неопределенного* момента времени в будущем, когда система будет не так загружена и все прерывания снова будут разрешены.

Не только в операционной системе Linux, но и в других операционных системах обработка аппаратных прерываний разделяется на две части. Верхняя половина выполняется быстро, когда все или некоторые прерывания запрещены. Нижняя половина (если она реализована) пыполняется позже, когда все прерывания разрешены. Это решение позволяет поддерживать малое время реакции системы, благодаря тому что работа при запрещенных прерываниях выполняется в течение возможно малого периода времени.

Многообразие нижних половин

В отличие от обработчиков верхних половин, которые могут быть реализованы только в самих обработчиках прерываний, для реализации обработчиков нижних половин существует несколько механизмов. Эти механизмы представляют собой различные интерфейсы и подсистемы, которые позволяют пользователю реализовать обработку нижних половин. В предыдущей главе мы рассмотрели единственный существующий механизм реализации обработчиков прерываний, а в этой главе рассмотрим несколько методов реализации обработчиков нижних половин. На самом деле за историю операционной системы Linux существовало много механизмов обработки нижних половин. Иногда сбивает с толку то, что эти механизмы имеют очень схожие или очень неудачные названия. Для того чтобы придумывать названия механизмам обработки нижних половин, необходимы "специальные программисты".

В этой главе мы рассмотрим принципы работы и реализацию механизмов обработки нижних половин, которые существуют в ядрах операционной системы Linux серии 2.6. Также будет рассмотрено, как использовать эти механизмы в коде ядра, который вы можете написать. Старые и давно изъятые из употребления механизмы обработки нижних половин представляют собой историческую ценность, поэтому, где это важно, о них также будет рассказано.

В самом начале своего существования операционная система Linux предоставляла единственный механизм для обработки нижних половин, который так и назывался "нижние половины" ("bottom half"). Это название было понятно, так как существовало только одно средство для выполнения отложенной обработки. Соответствующая инфраструктура называлась "BH" и мы ее так дальше и будем назвать, чтобы избежать путаницы с общим термином "bottom half (нижняя половина). Интерфейс BH был очень простым, как и большинство вещей в те старые добрые времена. Он предоставлял статический список из 32 обработчиков нижних половин. Обработчик верхней половины должен был отметить какой из обработчиков нижних половин должен выполняться путем установки соответствующего бита в 32-разрядном целом числе. Выполнение каждого обработчика BH синхронизировалось глобально, т.е. никакие два обработчика не могли выполняться одновременно, даже на разных процессорах. Такой механизм был простым в использовании, хотя и не гибким; простым в реализации, хотя представлял собой узкое место в плане производительности.

Позже разработчики ядра предложили механизм *очередей заданий* (*task queue)* — одновременно как средство выполнения отложенной обработки и как замена для механизма BH. В ядре определялось семейство очередей. Каждая очередь содержала связанный список функций, которые должны были выполнять соответствующие действия. Функции, стоящие в очереди, выполнялись в определенные моменты времени, в зависимости от того, в какой очереди они находились. Драйверы могли регистрировать собственные обработчики нижних половин в соответствующих очередях. Этот механизм работал достаточно хорошо, но он был не настолько гибким, чтобы полностью заменить интерфейс BH. Кроме того, он был достаточно "тяжеловесным" для обеспечения высокой производительности критичных к этому систем, таких как сетевая подсистема.

Во время разработки серии ядер 2.3 разработчики ядра предложили механизм *отложенных прерываний*[1] *(softirq)* и механизм *тасклетов (tasklet)*.

За исключением решения проблемы совместимости с существующими драйверами, механизмы отложенных прерываний и тасклетов были в состоянии полностью заменить интерфейс BH[2].

Отложенные прерывания - это набор из 32 статически определенных обработчиков нижних половин, которые могут одновременно выполняться на разных процессорах, даже два обработчика одного типа могут выполняться параллельно. Тасклеты — это гибкие, динамически создаваемые обработчики нижних половин, которые являются надстройкой над механизмом отложенных прерываний и имеют ужасное название, смущающее всех[3].

---

[1] Термин softirq часто переводится, как "программное прерывание", однако, чтобы не вносить путаницу с синхронными программными прерываниями (исключительными ситуациями) в этом контексте используется термин "отложенное прерывание". *(Прим. ред.)*

[2] Всвязи с глобальным синхронизмом выполнения обработчиков BH друг с другом, не так просто было их конвертировать для использования механизмов отложенных прерываний и тасклетов. Однако в ядрах серии 2.5 это наконец-то получилось сделать.

[3] Онине имеют ничего общего с понятием task (задача). Их следует понимать как простые в использовании отложенные прерывания (softirq).

Два различных тасклета могут выполняться параллельно на разных процессорах, но при этом два тасклета одного типа не могут выполняться одновременно. Таким образом, тасклеты — это хороший компромисс между производительностью и простотой использования. В большинстве случаев для обработки нижних половин достаточно использования таскелетов. Обработчики отложенных прерываний являются полезными, когда критична производительность, например, для сетевой подсистемы. Использование механизма отложенных прерываний требует осторожности, потому что два обработчика одного и того же отложенного прерывания могут выполняться одновременно. В дополнение к этому, отложенные прерывания должны быть зарегистрированы статически на этапе компиляции. Тасклеты, наоборот, могут быть зарегистрированы динамически.

Еще больше запутывает ситуацию то, что некоторые люди говорят о всех обработчиках нижних половин как о программных прерываниях, или отложенных прерываниях (software interrupt, или softirq). Другими словами, они называют механизм отложенных прерываний и в общем обработку нижних половин программными прерываниями. На таких людей лучше не обращать внимания, они из той же категории, что и те, которые придумали название "BH" и тасклет.

Во время разработки ядер серии 2.5 механизм BH был в конце концов выброшен, потому что все пользователи этого механизма конвертировали свой код для использования других интерфейсов обработки нижних половин. В дополнение к этому, интерфейс очередей заданий был заменен на новый интерфейс очередей отложенных действий (work queue). Очереди отложенных действий— это простой и в то же время полезный механизм, позволяющий поставить некоторое действие в очередь для выполнения в контексте процесса в более поздний момент времени.

Следовательно, сегодня ядро серии 2.6 предоставляет три механизма обработки нижних половин в ядре: отложенные прерывания, тасклеты и очереди отложенных действий. В ядре также использовались интерфейсы BH и очередей заданий, но сегодня от них осталась только светлая память.

Таймеры ядра

Еще один механизм выполнения отложенной работы — это таймеры ядра. В отличие от механизмов, рассмотренных в этой главе, таймеры позволяют отсрочить работу на указанный интервал времени. Инструменты, описанные в этой главе, могут быть полезны для откладывания работы *с текущего момента до какого-нибудь момента времени* в будущем. Таймеры используются для откладывания работы до того момента, пока не пройдет указанный период времени.

Поэтому таймеры имеют другое назначение, чем механизмы, описанные в данной главе. Более полное обсуждение таймеров ядра будет приведено в главе 10, "Таймеры и управление временем".

Путаница с нижними половинами

Некоторая путаница с обработчиками нижних половин имеет место, но на самом деле — это только проблема названий. Давайте снова вернемся к этому вопросу.

Термин "нижняя половина" ("bottom half") — это общий термин, который касается операционных систем и связан с тем, что некоторая часть процесса обработки прерывания откладывается на будущее. В операционной системе Linux сейчас этот термин означает то же самое. Все механизмы ядра, которые предназначены для отложенной обработки, являются обработчиками нижних половин.

Некоторые люди также называют обработчики нижних половин программными прерываниями иди "softirq", но они просто пытаются досадить остальным.

Термин "Bottom Half" также соответствует названию самого первого механизма выполнения отложенных действий п операционной системе Linux. Этот механизм еще называется "BH", поэтому далее так будем называть именно этот механизм, а термин "нижняя половина" ("bottom half") будет касаться общего названия. Механизм BH был некоторое время назад выключен из употребления и полностью изъят в ядрах серии 2.5.

Сейчас есть три метода для назначения отложенных операций: механизм отложенных прерываний (softirq), механизм тасклетов и механизм очередей отложенных действий. Тасклеты построены на основе механизма softirq, а очереди отложенных действий имеют полностью отличную реализацию. В табл. 7.1 показана история обработчиков нижних половин.

Таблица 7.1. Состояние обработчиков нижних половин

Механизм обработчиков	Состояние
BH	Изъято в серии 2.5
Очереди заданий	Изъято в серии 2.5
Отложенные прерывания	Доступно начиная с серии 2.3
Тасклеты	Доступно начиная с серии 2.3
Очереди отложенных действий	Доступно начиная с серии 2.3

Давайте продолжим рассмотрение каждого из механизмов в отдельности, пользуясь этой устойчивой путаницей в названиях.

Механизм отложенных прерываний (softirq)

Обсуждение существующих методов обработки нижних половин начнем с механизма softirq. Обработчики на основе механизма отложенных прерываний используются редко. Тасклеты — это более часто используемая форма обработчика нижних половин. Поскольку тасклеты построены на основе механизма softirq, с механизма softirq и стоит начать. Код, который касается обработчиков отложенных прерываний, описан в файле `kernel/softirq.c`.

Реализация отложенных прерываний

Отложенные прерывания определяются статически во время компиляции. В отличие от тасклетов, нельзя динамически создать или освободить отложенное прерывание. Отложенные прерывания представлены с помощью структур `softirq_action`, определенных в файле `<linux/interrupt.h>` в следующем виде.

```
/*
* структура, представляющая одно отложенное прерывание
*/

struct softirq_action
{
```

```
void (*action)(struct softirq_action * ) ;
      /* функция, которая должна выполниться */
void *data;      /* данные для передачи в функцию */
};
```

Массив из 32 экземпляров этой структуры определен в файле kernel/softirq.c в следующем виде.

```
static struct softirq_action softirq_vec[32];
```

Каждое зарегистрированное отложенное прерывание соответствует одному элементу этого массива. Следовательно, имеется возможность создать 32 обработчика softirq. Заметим, что это количество фиксированно. Максимальное число обработчиков softirq не может быть динамически изменено. В текущей версии ядра из 32 элементов используется только шесть[4].

Обработчик softirq

Прототип обработчика отложенного прерывания, поля action, выглядит следующим образом.

```
void softirq_handler(struct  softirg_action  *)
```

Когда ядро выполняет обработчик отложенного прерывания, то функция action вызывается С указателем на соответствующую структуру softirq_action в качестве аргумента. Например, если переменная rny_softirq содержит указатель на элемент массива softirq_vec, то ядро вызовет функцию-обработчик соответствующего отложенного прерывания в следующем виде.

```
my_softirq->action(my_softirq)
```

Может быть, несколько удивляет, что ядро передает в обработчик указатель на всю структуру, а не только на поле data. Этот прием позволяет в будущем вводить дополнительные поля в структуру без необходимости внесения изменений в существующие обработчики. Обработчик может получить доступ к значению ноля data простым разыменованием указателя на структуру и чтением ее поля data.

Обработчик одного отложенного прерывания никогда не вытесняет другой обработчик softirq. Б действительности, единственное событие, которое может вытеснить обработчик softirq, — это аппаратное прерывание. Однако на другом процессоре одновременно *с* обработчиком отложенного прерывания может выполняться другой (и даже этот же) обработчик отложенного прерывания.

Выполнение отложенных прерываний

Зарегистрированное отложенное прерывание должно быть отмечено для того, чтобы его можно было выполнить. Это называется *генерацией отложенного прерывания (rise softirq)*. Обычно обработчик аппаратного прерывания перед возвратом отмечает свои обработчики отложенных прерываний. Затем в подходящий момент времени отложенное прерывание выполняется. Ожидающие выполнения обработчики отложенных прерываний проверяются и выполняются в следующих ситуациях.

[4] Большинство драйверов использует для обработки своих нижних половин механизм тасклетов, Тасклеты построены на механизме softirq, как это будет покапано ниже.

- После обработки аппаратного прерывания.
- В контексте потока пространства ядра ksoftirqd.
- В любом коде ядра, который явно проверяет и выполняет ожидающие обработчики отложенных прерываний, как, например, это делает сетевая подсистема.

Независимо от того, каким способом выполняется отложенное прерывание, его выполнение осуществляется в функции do_softirq(). Эта функция по-настоящему проста. Если есть ожидающие отложенные прерывания, то функция do_softirq() в цикле проверяет их все и вызывает ожидающие обработчики. Давайте рассмотрим упрощенный вариант наиболее важной части функции do_softirq().

```
u32 pending = softirq_pending(cpu);

if (pending) {
        struct softirq_action *h = softirq_vec;
        softirq_pending(cpu) = 0;
        do {
              if (pending & 1)
              h->action (h);
              h++;
              pending »= 1;
        } while (pending);
}
```

Этот фрагмент кода является сердцем обработчика отложенных прерываний. Он проверяет и выполняет все ожидающие отложенные прерывания.

- Присваивает локальной переменной pending значение, возвращаемое макросом softirq_pending(). Это значение — 32-х разрядная битовая маска ожидающих на выполнение отложенных прерываний. Если установлен бит с номером n, то отложенное прерывание с этим номером ожидает на выполнение.
- Когда значение битовой маски отложенных прерываний сохранено, оригинальная битовая маска очищается[5].
- Переменной h присваивается указатель на первый элемент массива softirq_vec.
- Если первый бит маски, которая хранится в переменной pending, установлен, то вызывается функция h->action(h).
- Указатель h увеличивается на единицу, и теперь он указывает на второй элемент массива softirq_vec.
- Осуществляется логический сдвиг битовой маски, хранящейся в переменной pending, вправо на один разряд. Эта операция отбрасывает самый младший бит и сдвигает все оставшиеся биты на одну позицию вправо. Следовательно, второй бит теперь стал первым и т.д.

[5] На самом деле эта операция выполняется при всех запрещенных на локальном процессоре прерываниях, что не показано в упрощенной версии. Если бы прерывания не были запрещены, то в период времени между сохранением и очисткой маски могло бы быть сгенерировано новое отложенное прерывание (которое бы ожидало на выполнение), что привело бы к неверной очистке бита соответствующего отложенного прерывания.

- Указатель h теперь указывает на второй элемент массива, а в битовой маске — второй бит стал первым. Теперь необходимо повторить все ранее проделанные шаги.
- Последовательное повторение производится до тех пор, пока битовая маска не станет равной нулю. В этот момент больше нет ожидающих отложенных прерываний, и наша работа выполнена. Заметим, что такой проверки достаточно для того, чтобы гарантировать, что указатель h всегда указывает на законную запись в массиве softirq_vec, так как битовая маска pending имеет 32 бит и цикл не может выполниться больше 32 раз.

Использование отложенных прерываний

Отложенные прерывания зарезервированы для наиболее важных и критичных ко времени выполнения обработчиков нижних половин в системе. Сейчас только две подсистемы — подсистема SCSI и сетевая подсистема — напрямую используют механизм softirq. В дополнение к этому, таймеры ядра и тасклеты построены на базе отложенных прерываний. Если есть желание добавить новое отложенное прерывание, то стоит себя спросить, почему будет недостаточно использования тасклетов. Тасклеты могут создаваться динамически, а также их легче использовать в связи с более простыми требованиями к блокировкам. Кроме того, их производительность все еще остается очень хорошей. Тем не менее для задач, критичных ко времени выполнения, которые способны сами обеспечивать эффективные блокировки, использование механизма softirq — будет правильным решением.

Назначение индексов

Отложенные прерывания должны объявляться на этапе компиляции с помощью соответствующего перечисления (enum) в файле <linux/interrupt.h>. Ядро использует указанный в перечислении индекс, который начинается с нуля, как значение относительного приоритета отложенных прерываний. Отложенные прерывания с меньшим номером выполняются раньше отложенных прерываний с большим номером.

Создание нового отложенного прерывания состоит в добавлении новой записи в этот перечень (enum). Однако нужно не просто добавить новую строчку в конец списка, как в других местах. Вместо этого нужно вставить строчку в соответствии с приоритетом, который дается этому прерыванию. Исторически, HI_SOFTIRQ— имеет наибольший приоритет, а TASKLET_SOFTIRQ — наименьший. Новая запись, скорее всего, должна быть где-то ниже записей для сетевых устройств и выше записи для TASKLET_SOFTIRQ. В табл. 7.2 показан список всех типов отложенных прерываний.

Таблица 7.2. Список отложенных прерываний

Отложенное прерывание	Приоритет	Описание
HI_SOFTIRQ	0	Высокоприоритетные тасклеты
TIMER_SOFTIRQ	1	Обработчик нижних половин таймеров
NET_TX_S0FTIRQ	2	Отправка сетевых пакетов
NET_RX_SOFTIRQ	3	Прием сетевых пакетов
SCSI_SOFTIRQ	4	Обработчик нижних половин подсистемы SCSI
TASKLET_SOFTIRQ	5	Тасклеты

Регистрация обработчика

Далее во время выполнения должен быть зарегистрирован обработчик отложенного прерывания с помощью вызова open_softirq() , который принимает три параметра: индекс отложенного прерывания, функция-обработчик и значение поля data. Для сетевой подсистемы это делается, например, следующим образом.

```
open_softirq(NET_TX_SOFTIRQ,  net_tx_action, NULL);
open_softirq(NET_RX_SOFTIRQ,  net_rx_action, NULL);
```

Обработчик отложенного прерывания выполняется при разрешенных прерываниях и не может переходить в состояние ожидания (sleep). Во время выполнения обработчика отложенные прерывания на данном процессоре запрещаются. Однако на другом процессоре обработчики отложенных прерываний могут выполняться. На самом деле, если вдруг генерируется отложенное прерывание в тот момент, когда выполняется его обработчик, то такой же обработчик может быть запущен на другом процессоре одновременно с первым обработчиком. Это означает, что любые совместно используемые данные, которые используются в обработчике отложенного прерывания, и даже глобальные данные, которые используются только в самом обработчике, должны соответствующим образом блокироваться (как показано в следующих двух разделах). Это очень важный момент, и именно по этой причине использование тасклетов обычно предпочтительнее. Простое предотвращение конкурентного выполнения обработчиков — это далеко не идеал. Если обработчик отложенного прерывания просто захватит блокировку, которая предотвращает его выполнение параллельно самому себе, то для использования отложенных прерыпаний не остается почти никакого смысла. Следовательно, большинство обработчиков отложенных прерываний используют данные, уникальные для каждого процессора (и следовательно, не требующие блокировок), или какие-нибудь другие ухищрения, чтобы избежать явного использования блокировок и обеспечить отличную масштабируемость.

Главная причина использования отложенных прерываний — масштабируемость. Если нет необходимости масштабироваться на бесконечное количество процессоров, то лучше использовать механизм тасклетов. Тасклеты — это отложенные прерывания, для которых обработчик не может выполняться параллельно на нескольких процессорах.

Генерация отложенных прерываний

После того как обработчик добавлен в перечень и зарегистрирован с помощью вызова open_softirq() , он готов выполняться. Для того чтобы отметить его как ожидающего исполнения и, соответственно, чтобы он выполнился при следующем вызове функции do_softirq(), необходимо вызвать функцию raise_softirq(). Например, сетевая подсистема должна вызвать эту функцию в следующем виде.

```
raise_softirq(NET_TX_SOFTIRQ);
```

Этот вызов сгенерирует отложенное прерывание с индексом NET_TX_SOFTIRQ. Соответствующий обработчик net_tx_action () будет вызван при следующем выполнении программных прерываний ядром. Эта функция запрещает аппаратные прерывания перед тем, как сгенерировать отложенное прерывание, а затем восстанавливает их в первоначальное состояние. Если аппаратные прерывания в данный

момент запрещены, то для небольшой оптимизации можно воспользоваться функцией raise_softirq_irqoff (), как показано в следующем примере.

```
/*
* прерывания должны быть запрещены!
*/
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

Наиболее часто отложенные прерывания генерируются из обработчиков аппаратных прерываний. В этом случае обработчик аппаратного прерывания выполняет всю основную работу, которая касается аппаратного обеспечения, генерирует отложенное прерывание и завершается. После завершения обработки аппаратных прерываний ядро вызывает функцию do_softirq(). Обработчик отложенного прерывания выполняется и подхватывает работу с того места, где обработчик аппаратного прерывания ее отложил. В таком примере раскрывается смысл названий "верхняя половина" и "нижняя половина".

Тасклеты

Тасклеты — это механизм обработки нижних половин, построенный на основе механизма отложенных прерываний. Как уже отмечалось, они не имеют ничего общего с заданиями (task). Тасклеты по своей природе и принципу работы очень похожи на отложенные прерывания. Тем не менее они имеют более простой интерфейс и упрощенные правила блокировок.

Решение о том, стоит ли использовать тасклеты, принять достаточно просто: в большинстве случаев необходимо использовать тасклеты. Как было показано в предыдущем разделе, примеры использования отложенных прерываний можно посчитать на пальцах одной руки. Отложенные прерывания необходимо использовать только в случае, когда необходима очень большая частота выполнений и интенсивно используется многопоточная обработка. Тасклеты используются в очень большом количестве случаев — они работают достаточно хорошо и их очень просто использовать.

Реализация тасклетов

Так как тасклеты реализованы на основе отложенных прерываний, они тоже являются *отложенными прерываниями* (softirq). Как уже рассказывалось, тасклеты представлены двумя типами отложенных прерываний: HI_SOFTIRQ и TASKLET_SOFTIRQ. Единственная разница между ними в том, что тасклеты типа HI_SOFTIRQ выполняются всегда раньше тасклетов типа TASKLET_SOFTIRQ.

Структуры тасклетов

Тасклеты представлены с помощью структуры tasklet_struct. Каждый экземпляр структуры представляет собой уникальный тасклет. Эта структура определена в заголовочном файле <linux/interrupt.h> в следующем виде.

```
struct tasklet_struct {
        struct tasklet_struct *next;   /* указатель на следующий
                                          тасклет в списке */
        unsigned long state;           /* состояние тасклета */

                                }
```

```
atomic_t count;                /* счетчик ссылок */
void (*func) (unsigned long);  /* функция-обработчик тасклета*/
unsigned long data; /* аргумент функции-обработчика тасклета */
);
```

Поле func — это функция-обработчик тасклета (эквивалент поля action для структуры, представляющей отложенное прерывание), которая получает поле data в качестве единственного аргумента при вызове.

Поле state может принимать одно из следующих значений: нуль, TASKLET_STATE_SCHED или TASLET_STATE_RUN. Значение TASKLET_STATE_SCHED указывает на то, что тасклет запланирован на выполнение, а значение TASLET_STATE_RUN — что тасклет выполняется. Для оптимизации значение TASLET_STATE RUN может использоваться только на многопроцессорной машине, так как на однопроцессорной машине и без этого точно известно, выполняется ли тасклет (действительно, ведь код, который выполняется, либо принадлежит тасклету, либо нет).

Поле count используется как счетчик ссылок на тасклет. Если это значение не равно нулю, то тасклет запрещен и не может выполняться; если оно равно нулю, то тасклет разрешен и может выполняться в случае, когда он помечен как ожидающий выполнения.

Планирование тасклетов на выполнение

Запланированные (scheduled) на выполнение тасклеты (эквивалент сгенерированных отложенных прерываний)[6] хранятся в двух структурах, определенных для каждого процессора: структуре tasklet_vec (для обычных тасклетов) и структуре tasklet_hi_vec (для высокоприоритетных тасклетов). Каждая из этих структур — это связанный список структур tasklet_struct. Каждый экземпляр структуры tasklet_struct представляет собой отдельный тасклет.

Тасклеты могут быть запланированы на выполнение с помощью функций tasklet_schedule () и tasklet_hi_schedule (), которые принимают единственный аргумент— указатель на структуру тасклета— tasklet_struct. Эти функции очень похожи (отличие состоит в том, что одна использует отложенное прерывание с номером TASKLET_SOFTIRQ, а другая — с номером HI_SOFTIRQ). К написанию и использованию тасклетов мы вернемся в следующем разделе. А сейчас рассмотрим детали реализации функции tasklet_hi_schedule (), которые состоят в следующем.

- Проверяется, не установлено ли поле state в значение TASKLET_STATE_SCHED. Если установлено, то тасклет уже запланирован на выполнение и функция может возвратить управление.
- Сохраняется состояние системы прерываний и запрещаются прерывания на локальном процессоре. Это гарантирует, что ничто на данном процессоре не будет мешать выполнению этого кода.
- Добавляется тасклет, который планируется на выполнение, в начало связанного списка структуры tasklet_vec или tasklet_hi_vec, которые уникальны для каждого процессора в системе.

[6] Этоеще один пример плохой терминологии. Почему отложенные прерывания (softirq) генерируются (rise), а тасклеты (lasklet) планируются (schedule)? Кто знает? Оба термина означают, что обработчики нижних половин помечаются как ожидающие на выполнение и в скором времени будут выполнены.

- Генерируется отложенное прерывание с номером TASKLET_SOFTIRQ или HI_SOFTIRQ, чтобы в ближайшее время данный тасклет выполнился при вызове функции do_softirq().
- Устанавливается состояние системы прерываний в первоначальное значение и возвращается управление.

При первой же удобной возможности функция do_softirq() выполнится, как это обсуждалось в предыдущем разделе. Поскольку большинство тасклетов помечаются как готовые к выполнению в обработчиках прерываний, то, скорее всего, функция do_softirq () вызывается сразу же, как только возвратится последний обработчик прерывания. Так как отложенные прерывания с номерами TASKLET_SOFTIRQ или HI_SOFTIRQ к этому моменту уже сгенерированы, то функция do_softirq () выполняет соответствующие обработчики. Эти обработчики, а также функции tasklet_action () и tasklet_hi_action () являются сердцем механизма обработки тасклетов- Давайте рассмотрим, что они делают.

- Запрещаются прерывания, и получается весь список tasklet_vec или tasklet_hi_vec для текущего процессора.
- Список текущего процессора очищается путем присваивания значения нуль указателю на него.
- Разрешаются прерывания (нет необходимости восстанавливать состояние системы прерываний в первоначальное значение, так как этот код может выполняться только в обработчике отложенного прерывания, который вызывается только при разрешенных прерываниях).
- Организовывается цикл по всем тасклетам в полученном списке.
- Если данная машина является многопроцессорной, то нужно проверить не выполняется ли текущий тасклет на другом процессоре, то есть проверить не установлен ли флаг TASLET_STATE_RUN. Если тасклет уже выполняется, то его необходимо пропустить и перейти к следующему тасклету в списке (вспомним, что только один тасклет данного типа может выполняться в любой момент времени).
- Если тасклет не выполняется, то нужно установить флаг TASLET_STATE_RUN, чтобы другой процессор не мог выполнить этот тасклет.
- Проверяется значение поля count на равенство нулю, чтобы убедиться, что тасклет не запрещен. Если тасклет запрещен (поле count не равно нулю), то нужно перейти к следующему тасклету, который ожидает на выполнение.
- Теперь можно быть уверенным, что тасклет нигде не выполняется, нигде не будет выполняться (так как он помечен как выполняющийся на данном процессоре) и что значение поля count равно нулю. Необходимо выполнить обработчик тасклета. После того как тасклет выполнился, следует очистить флаг TASLET_STATE_RUN и поле state.
- Повторить описанный алгоритм для следующего тасклета, пока не останется ни одного тасклета, ожидающего выполнения.

Реализация тасклетов проста, но в то же время очень остроумна. Как видно, все тасклеты реализованы на базе двух отложенных прерываний TASKLET_SOFTIRQ и HI_SOFTIRQ. Когда тасклет запланирован на выполнение, ядро генерирует одно из этих двух отложенных прерываний. Отложенные прерывания, в свою очередь, обрабатываются специальными функциями, которые выполняют все запланированные на выполнение тасклеты. Эти специальные функции гарантируют, что только один тасклет данного типа выполняется в любой момент времени (но тасклеты разных типов могут выполняться одновременно). Вся эта сложность спрятана за простым и ясным интерфейсом.

Использование тасклетов

В большинстве случаев тасклеты — это самый предпочтительный механизм, с помощью которого следует реализовать обработчики нижних половин для обычных аппаратных устройств. Тасклеты можно создавать динамически, их просто использовать, и они сравнительно быстро работают.

Объявление тасклетов

Тасклеты можно создавать статически и динамически. Какой вариант лучше выбрать, зависит от того, как необходимо (или желательно) пользователю обращаться к тасклету: прямо или через указатель. Для статического создания тасклета (и соответственно, обеспечения прямого доступа к нему) необходимо использовать один из двух следующих макросов, которые определены в файле <linux/interrupts,h>;

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры struct_tasklet_struct с указанным именем (name). Когда тасклет запланирован на выполнение, то вызывается функция func, которой передается аргумент data. Различие между этими макросами состоит в значении счетчика ссылок на тасклет (поле count). Первый макрос создает тасклет, у которого значение поля count равно нулю, и, соответственно, этот тасклет разрешен. Второй макрос создает тасклет и устанавливает для него значение поля count, равное единице, и, соответственно, этот тасклет будет запрещен. Можно привести следующий пример.

```
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);

Эта строка эквивалентна следующей декларации.

struct tasklet_struct rny_tasklet = { NULL, 0, ATOMIC_INIT(0),
        tasklet_handler, dev);
```

В данном примере создается тасклет с именем my_tasklet, который разрешен для выполнения. Функция tasklet_handler будет обработчиком этого тасклета. Значение параметра dev передается в функцию-обработчик при вызове данной функции.

Для инициализации тасклета, на который указывает заданный указатель struct tasklet_struct* t — косвенная ссылка на динамически созданную ранее структуру, необходимо использовать следующий вызов.

```
tasklet_init(t, tasklet_handler, dev); /* динамически, а не статически*/
```

Написание собственной функции-обработчика тасклета

Функция-обработчик тасклета должна соответствовать правильному прототипу.

```
void tasklet_handler(unsigned long data)
```

Так же как и в случае отложенных прерываний, тасклет не может переходить в состояние ожидания (блокироваться). Это означает, что в тасклетах нельзя использовать семафоры или другие функции, которые могут блокироваться. Тасклеты также выполняются при всех разрешенных прерываниях, поэтому необходимо принять все меры предосторожности (например, может понадобиться запретить прерывания и захватить блокировку), если тасклет имеет совместно используемые данные с обработчиком прерывания. В отличие от отложенных прерываний, ни один тасклет не выполняется параллельно самому себе, хотя два разных тасклета могут выполняться на разных процессорах параллельно. Если тасклет совместно использует данные с обработчиком прерывания или другим тасклетом, то необходимо использовать соответствующие блокировки (см. глапу 8, "Введение в синхронизацию выполнения кода ядра" и главу 9, "Средства синхронизации в ядре").

Планирование тасклета на выполнение

Для того чтобы запланировать тасклет на выполнение, должна быть вызвана функция `tasklet_schedule ()`, которой в качестве аргумента передается указатель на соответствующий экземпляр структуры `tasklet_struct`.

```
tasklet_schedule(&my_tasklet) ; /* отметить, что тасклет my_tasklet
                                   ожидает на выполнение */
```

После того как тасклет запланирован на выполнение, он выполняется один раз в некоторый момент времени в ближайшем будущем. Если тасклет, который запланирован на выполнение, будет запланирован еще раз до того, как он выполнится, то он также выполнится всего один раз. Если тасклет уже выполняется, скажем, на другом процессоре, то будет запланирован снова и снова выполнится. Для оптимизации тасклет всегда выполняется на том процессоре, который его запланировал на выполнение, что дает надежду на лучшее использование кэша процессора.

Указанный тасклет может быть запрещен с помощью вызова функции `tasklet_disable ()`. Если тасклет в данный момент времени выполняется, то эта функция не возвратит управление, пока тасклет не закончит выполняться. Как альтернативу можно использовать функцию `tasklet_disable_nosync ()`, которая запрещает указанный тасклет, но возвращается сразу и не ждет, пока тасклет завершит выполнение. Это обычно небезопасно, так как в данном случае нельзя гарантировать, что тасклет не закончил выполнение. Вызов функции `tasklet_enable ()` разрешает тасклет. Эта функция также должна быть вызвана для того, чтобы можно было использовать тасклет, созданный с помощью макроса DECLARE_TASKLET_DISABLED (), как показано в следующем примере.

```
tasklet_disable(&my_tasklet); /* тасклет теперь запрещен */
/*Мы можем делать все, что угодно, зная, что тасклет не может выполняться. */
tasklet_enable(&my_tasklet); /* теперь тасклет разрешен */
```

Из очереди тасклетов, ожидающих на выполнение, тасклет может быть удален с помощью функции `tasklet_kill ()`. Эта функция получает указатель на соответ-

ствующую структуру tasklet_struct в качестве единственного аргумента. Удаление запланированного на выполнение тасклета из очереди очень полезно в случае, когда используются тасклеты, которые сами себя планируют на выполнение. Эта функция сначала ожидает, пока тасклет не закончит выполнение, а потом удаляет его из очереди. Однако это, конечно, не может предотвратить возможности, что другой код запланирует этот же тасклет на выполнение. Так как данная функция может переходить в состояние ожидания, то ее нельзя вызывать из контекста прерывания.

Демон ksoftirqd

Обработка отложенных прерываний (softirq) и, соответственно, тасклетов может осуществляться с помощью набора потоков пространства ядра (по одному потоку на каждый процессор). Потоки пространства ядра помогают обрабатывать отложенные прерывания, когда система перегружена большим количеством отложенных прерываний.

Как уже упоминалось, ядро обрабатывает отложенные прерывания в нескольких местах, наиболее часто это происходит после возврата из обработчика прерывания. Отложенные прерывания могут генерироваться с очень большими частотами (как, например, в случае интенсивного сетевого трафика). Хуже того, функции-обработчики отложенных прерываний могут самостоятельно возобновлять свое выполнение (реактивизировать себя). Иными словами, во время выполнения функции-обработчики отложенных прерываний могут генерировать свое отложенное прерывание для того, чтобы выполниться снова (на самом деле, сетевая подсистема именно так и делает). Возможность больших частот генерации отложенных прерываний в сочетании с их возможностью активизировать самих себя может привести к тому, что программы, работающие в пространстве пользователя, будут страдать от недостатка процессорного времени. В свою очередь, не своевременная обработка отложенных прерываний также не допустима. Возникает дилемма, которая требует решения, но ни одно из двух очевидных решений не является подходящим. Давайте рассмотрим оба этих очевидных решения.

Первое решение — это немедленная обработка всех отложенных прерываний, как только они приходят, а также обработка всех ожидающих отложенных прерываний перед возвратом из обработчика. Это решение гарантирует, что все отложенные прерывания будут обрабатываться немедленно и в то же время, что более важно, что все вновь активизированные отложенные прерывания также будут немедленно обработаны. Проблема возникает в системах, которые работают при большой загрузке и в которых возникает большое количество отложенных прерываний, которые постоянно сами себя активизируют. Ядро может постоянно обслуживать отложенные прерывания без возможности выполнять что-либо еще. Заданиями пространства пользователя пренебрегают, а выполняются только лишь обработчики прерываний и отложенные прерывания, в результате пользователи системы начинают нервничать. Подобный подход может хорошо работать, если система не находится под очень большой нагрузкой. Если же система испытывает хотя бы умеренную нагрузку, вызванную обработкой прерываний, то такое решение не применимо. Пространство пользователя не должно продолжительно страдать из-за нехватки процессорного времени.

Второе решение — это вообще *не обрабатывать* реактивизированные отложенные прерывания. После возврата из очередного обработчика прерывания ядро просто

просматривает список всех ожидающих на выполнение отложенных прерываний и выполняет их как обычно. Если какое-то отложенное прерывание реактивизирует себя, то оно не будет выполняться до того времени, пока ядро *в следующий раз* снова не приступит к обработке отложенных прерываний. Однако такое, скорее всего, произойдет, только когда поступит следующее аппаратное прерывание, что может быть равносильно ожиданию в течение длительного промежутка времени, пока новое (или вновь активизированное) отложенное прерывание будет выполнено. В таком решении плохо то, что на не загруженной системе выгодно обрабатывать отложенные прерывания сразу же. К сожалению, описанный подход не учитывает то, какие процессы могут выполняться, а какие нет. Следовательно, данный метод хотя и предотвращает нехватку процессорного времени для задач пространства пользователя, но создает нехватку ресурсов для отложенных прерываний, и к тому же такой подход не выгоден для систем, работающих при малых нагрузках.

Необходим какой-нибудь компромисс. Решение, которое реализовано в ядре, — *не обрабатывать* немедленно вновь активизированные отложенные прерывания. Вместо этого, если сильно возрастает количество отложенных прерываний, ядро возвращает к выполнению (wake up) семейство потоков пространства ядра, чтобы они справились с нагрузкой. Данные потоки ядра работают с самым минимально возможным приоритетом (значение параметра nice равно 19). Это гарантирует, что они не будут выполняться вместо чего-то более важного. Но они в конце концов тоже когда-нибудь обязательно выполняются. Это предотвращает ситуацию нехватки процессорных ресурсов для пользовательских программ. С другой стороны, это также гарантирует, что даже в случае большого количества отложенных прерываний они все в конце концов будут выполнены. И наконец, такое решение гарантирует, что в случае незагруженной системы отложенные прерывания также обрабатываются достаточно быстро (потому что соответствующие потоки пространства ядра будут запланированы на выполнение немедленно).

Для каждого процессора существует свой поток. Каждый поток имеет имя в виде `ksoftirqd/n`, где *n* — номер процессора. Так в двухпроцессорной системе будут запущены два потока с именами `ksoftiqd/0` и `ksoftirqd/1`. То, что на каждом процессоре выполняется свой поток, гарантирует, что если в системе есть свободный процессор, то он всегда будет *в* состоянии выполнять отложенные прерывания. После того как потоки запущены, они выполняют замкнутый цикл, похожий на следующий.

```
for (;;) {
       set_task_state(current, TASK_INTERRUPTIBLE);
       add_wait_queue(&cwq->more_work, &wait);

       if  (list_empty(&cwq->worklist))
              schedule();
       else
              set_task_state(current, TASK_RUNNING);
       remove_wait_queue(&cwq->more_work, &wait);

       if  (!list_empty(&cwq->worklist))
              run workqueue(cwq);
}
```

Если есть отложенные прерывания, ожидающие на обработку (что определяет вызов функции softirq_pending ()), то поток ядра ksoftirqd вызывает функцию do_softirq () , которая эти прерывания обрабатывает. Заметим, что это делается периодически, чтобы обработать также вновь активизированные отложенные прерывания. После каждой итерации при необходимости вызывается функция schedule (), чтобы дать возможность выполняться более важным процессам. После того как вся обработка выполнена, поток ядра устанавливает свое состояние в значение TASK_INTERRUPTIBLE и активизирует планировщик для выбора нового готового к выполнению процесса.

Поток обработки отложенных прерываний вновь возвращается в состояние готовности к выполнению, когда функция do_softirq () определяет, что отложенное прерывание реактивизировало себя.

Старый механизм BH

Хотя старый интерфейс BH, к счастью, уже отсутствует в ядрах серии 2.6, тем не менее им пользовались очень *долгое* время — с первых версий ядра. Учитывая, что этому интерфейсу удалось продержаться очень долго, он, конечно, представляет собой историческую ценность и заслуживает большего, чем просто беглого рассмотрения. Этот раздел никаким образом не касается ядер серии 2.6, но значение истории переоценить трудно.

Интерфейс BH очень древний, и это заметно. Каждый обработчик BH должен быть определен статически, и количество этих обработчиков ограничено максимальным значением 32. Так как все обработчики BH должны быть определены на этапе компиляции, загружаемые модули ядра не могли напрямую использовать интерфейс BH. Тем не менее можно было встраивать функции в уже существующие обработчики BH. Со временем необходимость статического объявления и максимальное количество обработчиков нижних половин, равное 32, стали надоедать.

Все обработчики BH выполнялись строго последовательно — никакие два обработчика BH, даже разных типов, не могли выполняться параллельно. Это позволяло обеспечить простую синхронизацию, но все же для получения высокой производительности при многопроцессорной обработке это было не очень хорошо. Драйверы, которые использовали интерфейс BH, очень плохо масштабировались на несколько процессоров. Например, страдала сетевая подсистема.

В остальном, за исключением указанных ограничений, механизм BH был похож на механизм тасклетов. На самом деле, в ядрах серии 2.4 механизм BH был реализован па основе тасклетов. Максимальное количество обработчиков нижних половин, равное 32, обеспечивалось значениями констант, определенных в заголовочном файле <linux/interrupt.h>. Для того чтобы отметить обработчик BH как ожидающий на выполнение, необходимо было вызвать функцию mark_bh () с передачей номера обработчика BH в качестве параметра. В ядрах серии 2.4 при этом планировался на выполнение тасклет BH, который выполнялся с помощью обработчика bh_action (). До серии ядер 2.4 механизм BH существовал самостоятельно, как сейчас механизм отложенных прерываний.

В связи с недостатками этого типа обработчиков нижних половин, разработчики ядра предложили механизм очередей заданий (task queue), чтобы заменить механизм нижних половин. Очереди заданий так и не смогли справиться с этой задачей, хотя и завоевали расположение большого количества пользователей. При разработке серии

ядер 2.3 были предложены механизмы отложенных прерываний (softirq) и механизм тасклетов (lasklet), для того чтобы положить конец механизму BH. Механизм BH при этом был реализован на основе механизма тасклетов. К сожалению, достаточно сложно переносить обработчики нижних половин с использования интерфейса BH на использование механизм тасклетов или отложенных прерываний, в связи с тем что у новых интерфейсов нет свойства строгой последовательности выполнения[7].

Однако при разработке ядер серии 2.5 необходимую конвертацию все же сделали, когда таймеры ядра и подсистему SCSI (единственные оставшиеся системы, которые использовали механизм BH) наконец-то перевели на использование отложенных прерываний. И в завершение, разработчики ядра совсем убрали интерфейс BH. Скатертью дорога тебе, интерфейс BH!

Очереди отложенных действий

Очереди отложенных действий (work queue) — это еще один способ реализации отложенных операций, который отличается от рассмотренных ранее. Очереди действий позволяют откладывать некоторые операции для последующего выполнения потоком пространства ядра — отложенные действия всегда выполняются в контексте процесса. Поэтому код, выполнение которого отложено с помощью постановки в очередь отложенных действий, получает все преимущества, которыми обладает код, выполняющийся в контексте процесса. Наиболее важное свойство — это то, что выполнение очередей действий управляется планировщиком процессов и, соответственно, выполняющийся код может переходить в состояние ожидания (sleep).

Обычно принять решение о том, что необходимо использовать: очереди отложенных действий или отложенные прерывания/тасклеты, достаточно просто. Если отложенным действиям необходимо переходить в состояние ожидания, то следует использовать очереди действий. Если же отложенные операции не могут переходить в состояние ожидания, то воспользуйтесь тасклетами или отложенными прерываниями. Обычно альтернатива использованию очередей отложенных действий — это создание новых потоков пространства ядра. Поскольку при введении новых потоков пространства ядра разработчики ядра обычно хмурят брови (а у некоторых народов это означает смертельную обиду), настоятельно рекомендуется использовать очереди отложенных действий. Их действительно *очень* просто использовать.

Если для обработки нижних половин необходимо использовать нечто, что планируется на выполнение планировщиком процессов, то воспользуйтесь очередями отложенных действий. Это единственный механизм обработки нижних половин, который всегда выполняется в контексте процесса, и, соответственно, единственный механизм, с помощью которого обработчики нижних половин могут переходить в состояние ожидания. Это означает, что они полезны в ситуациях, когда необходимо выделять много памяти, захватывать семафор или выполнять блочные операции ввода-вывода. Если для выполнения отложенных операций нет необходимости использовать поток ядра, то стоит подумать об использовании тасклетов.

[7] Отсутствие строгой последовательности выполнения хорошо сказывается на производительности, но приводит к усложнению программирования. Конвертация обработчиков BH в обработчики тасклетов, например, требует тщательного осмысления того, безопасно ли выполнять этот же код в то же самое время другим тасклетом? Однако после того как конвертация выполнена, это окупается повышением производительности.

Реализация очередей отложенных действий

В своей наиболее общей форме подсистема очередей отложенных действий — это интерфейс для создания потоков пространства ядра, которые выполняют некоторые действия, где-то поставленные в очередь. Эти потоки ядра называются *рабочими потоками (worker threads)*. Очереди действий позволяют драйверам создавать специальные рабочие потоки ядра для того, чтобы выполнять отложенные действия. Кроме того, подсистема очередей действий содержит рабочие потоки ядра, которые работают по умолчанию. Поэтому в своей общей форме очереди отложенных действий — это простой интерфейс пользователя для откладывания работы, которая будет выполнена потоком ядра.

Рабочие потоки, которые выполняются по умолчанию, называются `events/n`, где *n*—номер процессора. Для каждого процессора выполняется один такой поток. Например, в однопроцессорной системе выполняется один поток `events/0`. Б двухпроцессорной системе добавляется еще один поток— `events/1`. Рабочие потоки, которые выполняются по умолчанию, обрабатывают отложенные действия, которые приходят из разных мест. Многие драйверы, которые работают в режиме ядра, откладывают обработку своих нижних половин с помощью потоков, работающих по умолчанию. Если для драйвера или подсистемы нет строгой необходимости в создании своего собственного потока ядра, то использование потоков, работающих по умолчанию, более предпочтительно.

Тем не менее ничто не запрещает коду ядра создавать собственные потоки. Это может понадобиться, если в рабочем потоке выполняется большое количество вычислительных операций. Операции, критичные к процессорным ресурсам или к высокой производительности, могут получить преимущества от использования отдельного выделенного потока. Это также уменьшает нагрузку на потоки, работающие по умолчанию, и предотвращает нехватку ресурсов для остальных отложенных действий.

Структуры данных для представления потоков

Рабочие потоки представлены с помощью следующей структуры `workqueue_struct`.

```
/*
* Внешне видимая абстракция для представления очередей отложенных
  действий представляет собой массив очередей для каждого процессора:
*/
struct workqueue_struct {
        struct cpu_workqueue_struct cpu_wq[NR_CPUS] ;
        const char* name;

        struct list_head list;
        };
```

Эта структура содержит массив структур `struct cpu_workqueue_struct`, по одному экземпляру на каждый возможный процессор в системе. Так как рабочий поток существует для каждого процессора в системе, то для каждого рабочего потока, работающего на каждом процессоре машины, существует такая структура.

Структура cpu_workqueue_struct определена в файле kernel/workqueue.c и является основной. Эта структура показана ниже.

```
/*
* Очередь отложенных действий, связанная с процессором:
*/
struct cpu_workqueue_struct {
        spinlock_t lock; /* Очередь для защиты данной структуры */

        long rernove_sequence;   /* последний добавленный элемент
                                    (следующий для запуска ) */
        long insert sequence;   /* следующий элемент для добавления */
        struct list_head worklist;     /* список действий */
        wait_queue_head_t more_work;
        wait_queue_head_t work_done;

        struct workqueue_struct *wq;   /* соответствующая структура
                                          workqueue_struct */
        task_t *thread; /* соответствующий поток */

        int run_depth;   /* глубина рекурсии функции run_workqueue() */
};
```

Заметим, что каждый *тип* рабочих потоков имеет одну, связанную с этим типом структуру workqueue_struct. Внутри этой структуры имеется по одному экземпляру структуры cpu_workqueue_struct для каждого рабочего потока и, следовательно, для каждого процессора в системе, так как существует только один рабочий поток каждого типа на каждом процессоре.

Структуры для представления действий

Все рабочие потоки реализованы как обычные потоки пространства ядра, которые выполняют функцию worker_thread (). После начальной инициализации эта функция входит в бесконечный цикл и переходит в состояние ожидания. Когда какие-либо действия ставятся в очередь, поток возвращается к выполнению и выполняет эти действия. Когда в очереди не остается работы, которую нужно выполнять, поток снова возвращается в состояние ожидания. Каждое действие представлено с помощью структуры work_struct, определенной в файле <linux/workqueue.h>. Эта структура показана ниже.

```
struct work_struct {
        unsigned long pending; /* ожидает ли это действие на выполнение? */
        struct list_head entry; /* связанный список всех действий */
        void (*func)(void * ) ;   /* функция-обработчик */
        void *data;      /* аргумент функции-обработчика */
        void *wq_data;   /* для внутреннего использования */
        struct timer_list timer; /* таймер, который используется для
                        очередей отложенных действий с задержками */
};
```

Эти структуры объединены в связанный список, по одному списку на каждый тип очереди для каждого процессора. Например, для каждого процессора существует список отложенных действий, которые выполняются потоками, работающими по умолчанию. Когда рабочий поток возвращается к выполнению, он начинает выполнять все действия, которые находятся в его списке. После завершения работы рабочий поток удаляет соответствующие структуры work_struct из списка. Когда список становится пустым, поток переходит в состояние ожидания.

Давайте рассмотрим упрощенную основную часть функции worker_thread () .

```
for (;;) {
        set_task_state(current, TASK_INTERRUPTIBLE);
        add_wait_queue(&cwq->more_work, &wait);
        if (list_empty(&cwq->worklist))
              schedule();
        else
        set_task_state(current, TASK_RUNNING);
        remove_wait_queue (&cwq->more_work, &wait);
        if (! list_empty (&cwq->worklist))
              run_workqueue(cwq);
}
```

Эта функция выполняет следующие действия в бесконечном цикле.

- Поток переводит себя в состояние ожидания (флаг состояния устанавливается в значение TASK_INTERRUPTIBLE), и текущий поток добавляется в очередь ожидания.
- Если связанный список действий пуст, то поток вызывает функцию schedule () и переходит в состояние ожидания.
- Если список не пуст, то поток не переходит в состояние ожидания. Вместо этого он устанавливает свое состояние в значение TASK_RUNNING и удаляет себя из очереди ожидания.
- Если список не пустой, то вызывается функция run_workqueue () для выполнения отложенных действий.

Функция run_workqueue()

Функция run_workqueue () в свою очередь выполняет сами отложенные действия, как показано ниже.

```
while (!list_empty(&cwq->worklist)) {
       struct work_struct *work;
       void (*f) (void * ) ;
       void *data;

       work = list_entry(cwq->worklist.next, struct work_struct, entry);
       f = work->func;
       data = work->data;
       list_del_init(cwq->worklist.next);
       clear_bit(0, &work->pending);
       f(data);
}
```

Эта функция просматривает в цикле все элементы списка отложенных действий и выполняет для каждого элемента функцию, на которую указывает поле func соответствующей структуры workqueue_struct. Последовательность действий следующая.

- Если список не пустой, получить следующий элемент списка.
- Получить указатель на функцию (поле func), которую необходимо вызвать, и аргумент этой функции (поле data).
- Удалить полученный элемент из списка и обнулить бит ожидания в структуре элемента.
- Вызвать полученную функцию.
- Повторить указанные действия.

Извините, если не понятно

Взаимоотношения между различными, рассмотренными в этом разделе структурами достаточно запутанные. На рис. 7.1 показана диаграмма, которая эти взаимоотношения поясняет.

На самом верхнем уровне находятся рабочие потоки. Может существовать несколько типов рабочих потоков. Для каждого типа рабочих потоков существует один рабочий поток для каждого процессора. Различные части ядра при необходимости могут создавать рабочие потоки. По умолчанию выполняются только рабочие потоки *events* (события). Каждый рабочий поток представлен с помощью структуры cpu_workqueue_struct. Структура workqueue_struct представляет все рабочие потоки одного типа.

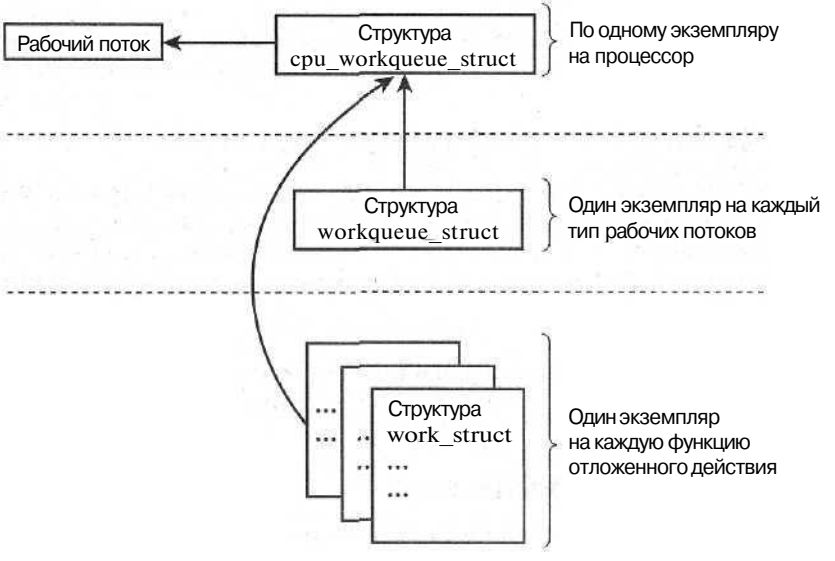


Рис. 7.1. Соотношения между отложенными действиями, очередями, действий и рабочими потоками

Например, давайте будем считать, что в дополнение к обычному типу рабочих потоков *events* был создан еще один тип рабочих потоков — *falcon*. Также имеется в распоряжении четырехпроцессорный компьютер. Следовательно, выполняется четыре потока типа *events* (соответственно, определено четыре экземпляра структуры cpu_workqueue_struct) и четыре потока типа *falcon* (для которых тоже определены другие четыре экземпляра структуры cpu_workqueue_struct). Для потоков типа *events* определен один экземпляр структуры workqueue_struct, а для потоков типа *falcon* — другой экземпляр этой структуры.

На самом нижнем уровне находятся отложенные действия. Драйвер создает отложенное действие, которой должно выполниться позже. Действия представлены структурами work_struct. Кроме других полей, эта структура содержит указатель на функцию, которая должна обработать отложенное действие. Отложенное действие отправляется на выполнение *определенному* потоку. Соответствующий поток переводится в состояние выполнения и выполняет отложенную работу.

Большинство драйверов использует существующие по умолчанию рабочие потоки, которые называются *events*. Они просты в реализации и в использовании. Однако в некоторых, более серьезных ситуациях необходимо создавать новые специальные рабочие потоки. Например, драйвер файловой системы XFS создает два новых типа рабочих потоков.

Использование очередей отложенных действий

Использовать очереди действий просто. Сначала мы рассмотрим рабочие потоки, используемые по умолчанию, — *events,* а затем опишем создание новых типов рабочих потоков.

Создание отложенных действий

Первый этап — это создание самого действия, которое должно быть отложено. Для создания статической структуры на этапе компиляции необходимо использовать следующий макрос.

```
DECLARE_WORK(name, void (*func) (void *), void *data);
```

Это выражение создает структуру works_truct с именем name, с функцией-обработчиком func и аргументом функции-обработчика data.

Во время выполнения отложенное действие можно создать с помощью передачи указателя на структуру, используя следующий макрос.

```
INIT_WORK(struct work_struct *work, void (*func)(void *),void *data);
```

Этот макрос динамически инициализирует отложенное действие, на структуру которого указывает указатель work, устанавливая функцию-обработчик func и аргумент data.

Обработчик отложенного действия

Прототип обработчика отложенного действия имеет следующий вид.

```
void work_handler (void *data)
```

Рабочий поток выполняет эту функцию, и, следовательно, эта функция выполняется в контексте процесса. По умолчанию при этом вес прерывания разрешены и никакие захваченные блокировки не удерживаются. Ели это необходимо, то функция может переходить в состояние ожидания. Следует заметить, что несмотря на то, что обработчики отложенных действий и выполняются в контексте процесса, эти обработчики не могут переходить в пространство пользователя, так как у потоков пространства ядра нет адресного пространства пользователя. Ядро может обращаться в пространство пользователя, только когда оно выполняется от имени пользовательского процесса, который имеет адресное пространство пользователя, отображенное на память, как, например, в случае выполнения системного вызова.

Блокировки между очередями отложенных действий и другими частями ядра осуществляются также, как и в случае любого другого кода, работающего в контексте процесса. Это позволяет сделать написание обработчиков отложенных действий достаточно простым. В следующих двух главах это раскрывается более детально.

Планирование действий на выполнение

Теперь, когда отложенное действие создано, его нужно запланировать на выполнение. Для того чтобы поставить обработчик данного действия в очередь на выполнение потоками *events,* которые работают по умолчанию, необходимо просто вызвать следующую функцию.

```
schedule_work(&work);
```

Действие планируется на выполнение немедленно и будет выполнено, как только рабочий поток events, работающий на данном процессоре, перейдет в состояние выполнения.

Иногда необходимо, чтобы действие было выполнено не немедленно, а с некоторой задержкой. В этом случае работа может быть запланирована на выполнение в некоторый момент времени в будущем. Для этого используется следующая функция.

```
schedule_delayed_work (&work, delay);
```

В этом случае действие, представленное структурой work_struct, с адресом &work, не будет выполнено, пока не пройдет хотя бы заданное в параметре delay количество импульсов таймера. О том, как использовать импульсы таймера для измерения времени, рассказывается в главе 10, "Таймеры и управление временем".

Ожидание завершения действий

Действия, поставленные в очередь, выполняются, когда рабочий поток возвращается к выполнению. Иногда нужно гарантировать, что, перед тем как двигаться дальше, заданный пакет отложенных действий завершен. Это особенно важно для загружаемых модулей, которые, вероятно, должны вызывать эту функцию, перед выгрузкой. В других местах также может быть необходимо гарантировать, что нет ожидающих на выполнение действий, для предотвращения состояния конкуренции.

Для этого есть следующая функция, которая позволяет ждать, пока очередь действий *events* не будет очищена.

```
void flush_scheduled_work(void);
```

Данная функция ожидает, пока все действия в очереди действий *events* не будут выполнены. В ожидании завершения всех заданий очереди, эта функция переводит вызывающий процесс в состояние ожидания. Поэтому ее можно вызывать только из контекста процесса.

Заметим, что эта функция не отменяет никаких отложенных действий с задержками. Любые действия, которые запланированы на выполнение с помощью функции schedule_delayed_work () и задержки которых еще не закончены, — не очищаются с помощью функций flush_scheduled_work (). Для отмены отложенных действий с задержками следует использовать функцию

```
int cancel_delayed_work(struct work_struct *work);
```

Эта функция отменяет отложенное действие, которое связано с данной структурой work_struct, если оно запланировано.

Создание новых очередей отложенных действий

Если для поставленных целей недостаточно очереди отложенных действий, которая используется по умолчанию, то можно создать новую очередь действий и соответствующие рабочие потоки. Так как при этом создается по одному потоку на каждый процессор, то новые очереди действий необходимо создавать, только если необходима большая производительность за счет выделенного набора потоков.

Новая очередь действий и связанные с ней рабочие потоки создаются с помощью простого вызова функции.

```
struct workqueue_struct *create_workqueue(const char *name);
```

Параметр name используется для того, чтобы присваивать имена потокам ядра. Например, очередь events, которая используется по умолчанию, создается с помощью следующего вызова.

```
struct workqueue_struct *keventd_wq=create_workqueue("events");
```

При этом также создаются все рабочие потоки (по одному на каждый процессор), которые подготавливаются к выполнению работы.

Создание отложенных действий выполняется одинаково, независимо от тина очереди. После того как действия созданы, могут быть использованы функции, аналогичные функциям schedule_work() и schedule_delayed_work(), которые отличаются тем, что работают с заданной очередью действий, а не с очередью, используемой по умолчанию.

```
int queue_work struct workqueue_struct *wq, struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *wq,
                       struct wesrk_struct *work, unsigned long delay);
```

И наконец, ожидание завершения действий в заданной очереди может быть выполнено с помощью функции

```
flush_workqueue(struct workqueue_struct *wq);
```

Эта функция работает по аналогии с функцией flush_scheduled_work (), как описывалось ранее, за исключением того, что она ожидает, пока заданная очередь не станет пустой.

Старый механизм очередей заданий

Так же как и в случае интерфейса BH, который дал начало интерфейсам отложенных прерываний (softirq) и тасклетов (tasklet), интерфейс очередей действий возник благодаря недостаткам интерфейса очередей заданий (task queue). Интерфейс очередей заданий (который еще называют просто *tq),* так же как и тасклеты, не имеет ничего общего с заданиями (task), в смысле с процессами[8]. Все подсистемы, которые использовали механизм очередей заданий, были разбиты на две группы еще во времена разработки серии ядер 2.5. Первая группа была переведена на использование тасклетов, а вторая— продолжала использовать интерфейс очередей заданий. Все, что осталось от интерфейса очередей заданий, перешло в интерфейс очередей отложенных действий. Краткое рассмотрение очередей заданий, которым пользовались в течение некоторого времени, — это хорошее упражнение по истории.

Интерфейс очередей заданий позволял определять набор очередей. Очереди имели имена, такие как scheduler queue (очередь планировщика), immediate queue (немедленная очередь) или timer queue (очередь таймера). Каждая очередь выполнялась в определенных местах в ядре. Поток пространства ядра *keventd* выполнял работу, связанную с очередью планировщика. Эта очередь была предшественником интерфейса очередей отложенных действий. Очередь таймера выполнялась при каждом импульсе системного таймера, а немедленная очередь выполнялась в нескольких местах, чтобы гарантировать "немедленное" выполнение. Были также и другие очереди. Кроме того, можно было динамически создавать новые очереди.

Все это может показаться полезным, но на практике интерфейс очередей заданий приносил только неприятности. Все очереди были, по сути, оторваны от действительности. Единственной ценной очередью оказалась очередь планировщика, которая предоставляла единственную возможность, чтобы выполнять отложенные действия в контексте процесса.

Еще одним преимуществом механизма очередей заданий была простота интерфейса. Несмотря на большое количество очередей и разнообразие правил, по которым они выполнялись, интерфейс был максимально прост. Все остальное, что касается очередей заданий, необходимо было убрать.

Различные использования очередей заданий были заменены другими механизмами обработки нижних половин; большинство — тасклетами. Осталось только то, что касалось очереди планировщика. В конце концов, код демона keventd был обобщен в отличный механизм очередей действий, который мы имеем сегодня, а очереди заданий были полностью удалены из ядра.

Какие обработчики нижних половин необходимо использовать

Решение о том, какой из механизмов обработки нижних половин следует использовать, является важным. В современных ядрах серии 2.6 есть три варианта выбора: отложенные прерывания (softirq), тасклеты (tasklet) и очереди отложенных действий (work queue). Тасклеты построены на основе отложенных прерываний, и поэтому

[8] Названиядля механизмов обработки нижних половин, очевидно, выбираются из соображений конспирации, чтобы сбивать с толку молодых и неопытных разработчиков ядра.

эти два механизма похожи. Механизм очередей действий полностью от них отличается, он построен на базе потоков пространства ядра.

Благодаря своей реализации, отложенные прерывания обеспечивают наибольший параллелизм. Это требует от обработчиков отложенных прерываний применения дополнительных мер для того, чтобы гарантировать безопасный доступ к совместно используемым данным, так как два или более экземпляров одного и того же отложенного прерывания могут выполняться параллельно на разных процессорах. Если код уже очень хорошо распараллелен для многопоточного выполнения, как, например, сетевая подсистема, которая использует данные, связанные с процессорами, то использование отложенных прерываний— это хороший выбор. Они, конечно, представляют собой наиболее быстрый механизм для критичных ко времени или частоте выполнения задач. Тасклеты имеет больший смысл использовать для кода, который не очень хорошо распараллелен для многопоточности. Они имеют более простой интерфейс, и поскольку тасклеты одного типа не могут выполняться параллельно, то их легко программировать. Тасклеты — это фактически отложенные прерывания, которые не могут выполняться параллельно. Разработчики драйверов всегда должны использовать тасклеты, а не отложенные прерывания, кроме, конечно, случаев, когда они готовы связываться с такими вещами, как переменные, связанные с процессорами (per-CPU data), или другими хитростями, чтобы гарантировать безопасное параллельное выполнение отложенных прерываний на разных процессорах.

Если отложенные операции требуют выполнения в контексте процесса, то из трех возможных вариантов остается единственный выбор — это очереди действий. Если выполнение в контексте процесса не является обязательным, в частности, если нет необходимости переходить в состояние ожидания (sleep), то использование отложенных прерываний или тасклетов, скорее всего, подойдет больше. Очереди действий вносят наибольшие накладные расходы, так как они используют потоки ядра и, соответственно, переключение контекста. Нельзя сказать, что они не эффективны, но в свете тех тысяч прерываний в секунду, что сетевая подсистема может обеспечить, использование других механизмов может иметь больший смысл. Хотя для большинства ситуаций очередей действий также бывает достаточно.

В плане простоты использования пальму первенства получают очереди действий. Использование очереди *events,* которая существует по умолчанию, — это просто детская игра. Далее идут тасклеты, которые тоже имеют простой интерфейс. Последними стоят отложенные прерывания, которые должны быть определены статически.

В табл. 7.3 приведено сравнение различных механизмов обработки нижних половин.

Таблица 7.3. Сравнение механизмов обработки нижних половин

Механизм обработки нижних половин	Контекст выполнения	Сериализация
Отложенные прерывания (softirq)	Прерывание	Отсутствует
Тасклеты (tasklet)	Прерывание	По отношению к тасклету такого же типа
Очереди отложенных действий (work queue)	Процесс	Отсутствует (планируется на выполнение как контекст процесса)

Если коротко, то разработчики обычных драйверов имеют всего два варианта выбора. Необходимо ли использовать возможности планировщика, чтобы выполнять отложенные действия, т.е. необходимо ли переходить в состояние ожидания по какой-либо причине? Если да, то единственный вариант — очереди отложенных действий. В противном случае предпочтительно использовать тасклеты. Только если важна масштабируемость, то стоит обратиться к отложенным прерываниям.

Блокировки между обработчиками нижних половин

Мы еще не касались вопросов, связанных с блокировками. Этой теме посвящены следующие две главы. Тем не менее очень важно понимать, что решающим моментом при обработке нижних половин является защита данных общего доступа от конкурентных изменений, даже на однопроцессорной машине. Следует помнить, что обработчик нижней половины прерывания потенциально может выполняться в любой момент времени. Может потребоваться вернуться к текущему разделу, после прочтения следующих двух глав, если вы далеки от вопросов, связанных с блокировками.

Одно из преимуществ использования тасклетов состоит в том, что они всегда выполняются последовательно по отношению к себе: один и тот же такслет никогда не будет выполняться параллельно себе даже на двух разных процессорах. Это означает, что нет необходимости заботиться о проблемах, связанных с конкурентным выполнением тасклетов одного типа. Конкурентное выполнение тасклетов нескольких разных типов (в случае, если они совместно используют одни данные) требует применения блокировок.

Так как отложенные прерывания не обеспечивают строгой последовательности выполнения (даже два обработчика одного и того же отложенного прерывания могут выполняться параллельно), то все совместно используемые данные требуют соответствующих блокировок.

Если из контекста процесса необходимо обращаться к данным, которые используются как контекстом процесса, так и обработчиком нижней половины, то необходимо запретить обработку нижних половин и захватить блокировку перед тем, как начинать работу с данными. Это позволяет гарантировать защиту совместно используемых данных как на локальном процессоре, так и на разных процессорах SMP системы, а также предотвратить взаимоблокировки.

Если имеются данные, которые могут совместно использоваться в контексте прерывания и в обработчике нижней половины, то необходимо запретить прерывания и захватить блокировку перед тем, как обращаться к этим данным. Именно эти две операции позволяют предотвратить взаимоблокировку и обеспечить защиту для SMP-систем.

Все совместно используемые данные, к которым необходимо обращаться из очередей действий, также требуют применения блокировок. Проблема блокировок в этом случае ничем не отличается от блокировок обычного кода ядра, так как очереди действий всегда выполняются в контексте процесса.

В главе 8 будут рассмотрены хитрости, связанные с блокировками. В главе 9 будут описаны базовые элементы ядра, которые позволяют осуществлять блокировки.

Далее в этом разделе рассказывается о том, как защитить данные, которые используются обработчиками нижних половин.

Запрещение обработки нижних половин

Обычно только одного запрещения обработки нижних половин недостаточно. Наиболее часто, чтобы полностью защитить совместно используемые данные, необходимо захватить блокировку *и* запретить обработку нижних половин. Методы, которые позволяют это сделать и которые обычно используются при разработке драйверов, будут рассмотрены в главе 9. Однако при разработке самого кода ядра иногда необходимо запретить только обработку нижних половин.

Для того чтобы запретить обработку всех типов нижних половин (всех отложенных прерываний и, соответственно, тасклетов), необходимо вызвать функцию local_bh_disable (). Для разрешения обработки нижних половин необходимо вызвать функцию local_bh_enable (). Да, у этих функций "неправильные" названия. Никто не потрудился переименовать эти функции, когда интерфейс BH уступил место интерфейсу отложенных прерываний. В табл. 7.4 приведены сведения об этих функциях.

Таблица 7.4. Список функций управления обработкой нижних половин

Функция	Описание
void local_bh_disable ()	Запретить обработку всех отложенных прерываний (softirq) и тасклетов (tasklet) на локальном процессоре
void local_bh_enable ()	Разрешить обработку всех отложенных прерываний (softirq) и тасклетов (tasklet) на локальном процессоре

Вызовы этих функций могут быть вложенными — при этом только последний вызов функции local_bh_enable () разрешает обработку нижних половин. Например, при первом вызове функции local_bh_disable () запрещается выполнение отложенных прерываний на текущем процессоре. Если функция local_bh_disable () вызывается еще три раза, то выполнение отложенных прерываний будет запрещено. Их выполнение не будет разрешено до тех пор, пока функция local_bh_enable () не будет вызвана четыре раза.

Такая функциональность реализована с помощью счетчика preerapt_count, который поддерживается для каждого задания (интересно, что этот же счетчик используется и для вытеснения процессов в режиме ядра)[11]. Когда значение этого счетчика достигает нуля, то можно начать обработку нижних половин. Так как при вызове функции local_bh_enable () обработка нижних половин запрещена, то эта функция также проверяет наличие ожидающих на обработку нижних половин и выполняет их.

[11] На самом деле этот счетчик используется как системой обработки прерываний, так и системой обработки нижних половин. Наличие одного счетчика для задания позволяет в операционной системе Linux реализоиать атомарность заданий. Как показала практика, такой подход очень полезен для нахождения ошибок, например, связанных с тем, что задание переходит в состояние ожидания в то время, когда выполняет атомарные операции (sleeping-while-atomic bug).

Для каждой поддерживаемой аппаратной платформы имеются спои функции, которые обычно реализуются через сложные макросы, описанные в файле <asrn/softirq.h>. Для любопытных ниже приведены соответствующие реализации на языке программирования С.

```
/*
* запрещение обработки нижних половин путем увеличения значения
  счетчика preempt_count
*/
void local_bh_disable(void)
{
        struct thread_info *t = current_thread_info();

        t->preempt_count += SOFTIRQ_OFFSET;
}



/*
* уменьшение значения счетчика preempt_count "автоматически" разрешает
* обработку нижних половин, если значение счетчика равно нулю
*
* опционально запускает все обработчики нижних половин,
  которые ожидают на обработку
*/
void local_bh_enable(void)
{
        struct thread_info *t = current_thread_info();

        t->preempt_count -= SOFTIRQ_OFFSET;

        /*
        * равно ли значение переменной preempt_count нулю и ожидают ли
          на обработку какие-либо обработчики нижних половин?
        * если да, то запустить их
        /
        if (unlikely(!t->preempt_count &&
          softirq_pending (smp_processor_id())))
              do_softirq();
}
```

Эти функции не запрещают выполнения очередей действий. Так как очереди действий выполняются в контексте процесса, нет никаких проблем с асинхронным выполнением и нет необходимости запрещать их. Поскольку отложенные прерывания и тасклеты могут "возникать" асинхронно (например, при возвращении из обработчика аппаратного прерывания), то ядру может потребоваться запрещать их. В случае использования очередей отложенных действий защита совместно используемых данных осуществляется так же, как и при работе в контексте процесса. Детали рассмотрены в главах 8 и 9.

Внизу обработки нижних половин

В этой главе были рассмотрены три механизма, которые используются для реализации отложенных действий в ядре Linux, — отложенные прерывания (softirq), тасклеты (tasklet) и очереди отложенных действий (work queue). Было показано, как эти механизмы работают и как они реализованы. Также обсуждались основные моменты, связанные с использованием этих механизмов в собственном программном коде, и было показано, какие у них неподходящие названия. Для того чтобы восстановить историческую справедливость, мы также рассмотрели те механизмы обработки нижних половин, которые существовали в предыдущих версиях ядра Linux: механизмы BH и task queue.

Очень часто в главе поднимались вопросы, связанные с синхронизацией и параллельным выполнением, потому что эти моменты имеют прямое отношение к обработке нижних половин. В эту главу специально был включен раздел, который касается запрещения обработки нижних половин для защиты от конкурентного доступа, Теперь настало время углубиться в эти моменты с головой. В следующей главе будут рассмотрены особенности синхронизации и параллельного выполнения кода в ядре: основные понятия и соответствующие проблемы. Далее будут рассмотрены интерфейсы, которые позволяют осуществлять синхронизацию в ядре и решать указанные проблемы. Вооруженные следующими двумя главами, вы сможете покорить мир.

8

Введение в синхронизацию выполнения кода ядра

В приложениях, рассчитанных на работу с совместно используемой памятью (shared memory), необходимо позаботиться о том, чтобы совместно используемые ресурсы были защищены от конкурентного доступа. Ядро — не исключение. Совместно используемые ресурсы требуют защиты от конкурентного доступа в связи с тем, что несколько потоков выполнения[1] могут одновременно манипулировать одними и теми же данными: эти потоки могут переписывать изменения, сделанные другими потоками, а также обращаться к данным, которые находятся в несогласованном (противоречивом, неконсистентном) состоянии. Конкурентный доступ к совместно используемым данным — это хороший способ получить нестабильность системы, причины которой, как показывает опыт, впоследствии очень сложно обнаружить и исправить. В связи с этим важно при разработке сразу сделать все правильно.

Осуществить необходимую защиу совместно используемых ресурсов может оказаться трудной задачей. Много лет назад, когда операционная система Linux не поддерживала симметричную многопроцессорную обработку, предотвратить конкурентный доступ к данным было просто. Так как поддерживался только один процессор, то единственная возможность конкурентного доступа к данным возникала при получении прерывания или когда выполнение кода ядра явно перепланировалось, давая возможность выполняться другому заданию. Да, раньше жить было проще.

Эти дни закончились. Поддержка симметричной многопроцессорности была введена в ядрах серии 2.0, и с тех пор эта поддержка постоянно совершенствуется. Поддержка мультипроцессорности предполагает, что код ядра может одновременно выполняться на двух или более процессорах. Следовательно, без специальной защиты части кода ядра, которые выполняются на двух разных процессорах, принципиально могут обратиться к совместно используемым данным в один и тот же момент

[1] Термин поток выполнения подразумевает любой выполняющийся код. Это включает, например, задание в ядре, обработчик прерывания или поток пространства ядра. В этой главе поток выполнения сокращенно называется просто поток. Следует помнить, что этот термин подразумевает любой выполняющийся код.

времени. Начиная с серии ядер 2.6 ядро операционной системы Linux является преемптивным (вытесняемым). Это подразумевает, что (при отсутствии необходимой защиты) планировщик может вытеснить код ядра в любой момент времени и запустить на выполнение другое задание. Сегодня есть много сценариев, благодаря которым может возникнуть конкурентный доступ к данным в ядре, и все эти варианты требуют защиты данных.

В этой главе рассматриваются проблемы, связанные с параллельным выполнением кода и синхронизацией выполнения кода в ядре операционной системы. В следующей главе детально рассмотрены механизмы и интерфейсы, которые предоставляет ядро операционной системы Linux для решения проблем синхронизации и предотвращения состояния конкуренции за ресурс (race condition, состояние "гонок").

Критические участки и состояние конкуренции за ресурсы

Ветки кода, которые получают доступ к совместно используемыми данным и манипулируют ими, называются *критическими участками* (critical region). Обычно небезопасно нескольким потокам выполнения одновременно обращаться к одному и тому же ресурсу. Для предотвращения конкурентного доступа во время выполнения критических участков программист, т.е. Вы, должен гарантировать, что код выполняется *атомарно* - без перерывов, так если бы весь критический участок был одной неделимой машинной инструкцией. Если два потока выполнения одновременно находятся в критическом участке, то это— ошибка в программе. Если такое вдруг случается, то такая ситуация называется *состоянием, конкуренции за ресурс* (состояние "гонок", race condition). Название связано с тем, что потоки как бы соревнуются друг с другом за доступ к ресурсу. Следует обратить внимание на то, насколько редко такая ситуация может возникать, — поэтому обнаружение состояний конкуренции за ресурсы при отладке программ часто очень сложная задача, потому что подобную ситуацию очень трудно воспроизвести. Обеспечение гарантии того, что конкуренции не будет и, следовательно, что состояний конкуренции за ресурсы возникнуть не может, называется *синхронизацией*.

Зачем нужна защита

Для лучшего понимания того, к чему может привести состояние конкуренции, давайте рассмотрим примеры повсеместно встречающихся критических участков.

В качестве первого примера рассмотрим ситуацию из реальной жизни; банкомат (который еще называют ATM, Automated Teller Machine, или кэш-машиной).

Одно из наиболее часто встречающихся действий, которые приходится выполнять с помощью банкомата— это снятие денег с персонального банковского счета физического лица. Человек подходит к банкомату, вставляет карточку, вводит PIN-код, проходит аутентификацию, выбирает пункт меню Снятие наличных, вводит необходимую сумму, нажимает OK, забирает деньги и отправляет их автору этой книги.

После того как пользователь ввел необходимую сумму, банкомат должен проверить, что такая сумма действительно есть на счету. Если такие деньги есть, то необ-

ходимо вычесть снимаемую сумму из общего количества доступных денег. Код, который выполняет эту операцию, может выглядеть следующим образом.

```
int total = get_total_from_account(); / *общее количество денег на счету*/
int withdrawal = get_withdrawal_amount(); /* количество денег,
                                              которые хотят снять */

/* проверить, есть ли у пользователя деньги на счету */
if (total < withdrawal)
      error("У Вас нет таких денег!")

/* Да, у пользователя достаточно денег: вычесть снимаемую сумму из
   общего количества денег на счету */
total -= withdrawal;
update_total_funds(total);

/* Выдать пользователю деньги */
spit_out_money(withdrawal);
```

Теперь представим, что в тот же самый момент времени со счета этого же пользователя снимается еще одна сумма денег. Не имеет значения, каким образом пыполняется снятие второй суммы. Например, или супруг пользователя снимает деньги с другого банкомата, или кто-то переводит со счета деньги электронным платежом, или банк снимает со счета в качестве платы за что-то (как это обычно любят делать банки), или происходит что-либо еще.

Обе системы, которые снимают деньги со счета, выполняют код, аналогичный только что рассмотренному: проверяется, что снятие денег возможно, после этого вычисляется новая сумма денег на счету и, наконец, деньги снимаются физически. Теперь рассмотрим некоторые численные значения. Допустим, что первая снимаемая сумма равна $100, а вторая — $10, например, за то, что пользователь зашел в банк (что не приветствуется: необходимо использовать банкомат, так как в банках людей не хотят видеть). Допустим также, что у пользователя на счету есть сумма, равная $105. Очевидно, что одна из этих двух транзакций не может завершиться успешно без получения минусов на счету.

Можно ожидать, что получится что-нибудь вроде следующего: первой завершится транзакция по снятию платы за вход в банк. Десять долларов — это меньше чем $105, поэтому, если от $105 отнять $10, на счету останется $95, а $10 заработает банк. Далее начнет пыполняться снятие денег через банкомат, но оно завершится неудачно, так как $95 — это меньше чем $100.

Тем не менее жизнь может оказаться значительно интереснее, чем ожидалось. Допустим, что две указанные выше транзакции начинаются почти в один и тот же момент времени. Обе транзакции убеждаются, что на счету достаточно денег: $105 — это больше $100 и больше $10. После этого процесс снятия денег с банкомата вычтет $100 из $105 и получится $5. В это же время процесс снятия платы за вход сделает то же самое и вычтет $10 из $105, и получится $95. Далее процесс снятия денег обновит состояние счета пользователя: на счету окажется сумма $5. В конце транзакция снятия платы за вход также обновит состояние счета, и на счету окажется $95. Получаем деньги в подарок!

Ясно, что финансовые учреждения считают своим долгом гарантировать, чтобы такой ситуации не могло возникнуть никогда. Необходимо блокировать счет во время выполнения некоторых операций, чтобы гарантировать атомарность транзакций по отношению к другим транзакциям. Такие транзакции должны полностью выполняться не прерываясь или не выполняться совсем.

Общая переменная

Теперь рассмотрим пример, связанный с компьютерами. Пусть у нас есть очень простой совместно используемый ресурс: одна глобальная целочисленная переменная и очень простой критический участок — операция инкремента значения этой переменной:

```
i++
```

Это выражение можно перевести в инструкции процессора следующим образом.

```
Загрузить текущее значение переменной i из памяти в регистр.
Добавить единицу к значению, которое находится в регистре.
Записать новое значение переменной i обратно в память.
```

Теперь предположим, что есть два потока, которые одновременно выполняют этот критический участок, и начальное значение переменной i равно 7. Результат выполнения будет примерно следующим (каждая строка соответствует одному интервалу времени).

Поток 1	**Поток 2**
получить значение i из памяти (7)	
увеличить i на 1 <7->8)	
записать значение i в память (8)	
	получить значение i из памяти (8)
	увеличить i на 1 (8->9)
	записать значение i в память (9)

Как и ожидалось, значение переменной ±, равное 7, было увеличено на единицу два раза и стало равно 9. Однако возможен и другой вариант.

Поток 1	**Поток 2**
получить значение i из памяти (7)	
	получить значение i из памяти (7)
увеличить i на 1 (7->8)	
	увеличить i на 1 (7->8)
записать значение i в память (8)	
	записать значение i в память (8)

Если оба потока выполнения прочитают первоначальное значение переменной i перед тем, как оно было увеличено на 1, то оба потока увеличат его на единицу и запишут в память одно и то же значение. В результате переменная i будет содержать значение 8, тогда как она должна содержать значение 9. Это один из самых простых примеров критических участков. К счастью, решение этой проблемы простое — необходимо просто обеспечить возможность выполнения всех рассмотренных операций за один неделимый шаг. Для большинства процессоров есть машинная инструкция, которая позволяет атомарно считать данные из памяти, увеличить их значение на 1

и записать обратно в память, выделенную для переменной. Использование такой инструкции позволяет решить проблему. Возможно только два варианта правильного выполнения этого кода — следующий.

Поток 1	**Поток 2**
увеличить i на 1 (7->8)	
	увеличить i на 1 (8->9)

Или таким образом.

Поток 1	**Поток 2**
	увеличить i на 1 (7->8)
увеличить i на 1 (8->9)	

Две атомарные операции никогда не могут перекрываться. Процессор на физическом уровне гарантирует это. Использование такой инструкции решает проблему. Ядро предоставляет несколько интерфейсов, которые позволяют реализовать атомарные операции. Эти интерфейсы будут рассмотрены в следующей главе.

Блокировки

Теперь давайте рассмотрим более сложный пример конкуренции за ресурсы, который требует более сложного решения. Допустим, что у нас есть очередь запросов, которые должны быть обработаны. Как реализована очередь — не существенно, но мы будем считать, что это — связанный список, в котором каждый узел соответствует одному запросу. Очередью управляют две функции: одна— добавляет новый запрос в конец очереди, а другая — извлекает запрос из головы очереди и делает с ним нечто полезное. Различные части ядра вызывают обе эти функции, поэтому запросы могут постоянно поступать, удаляться и обрабатываться. Все манипуляции очередью запросов, конечно, требуют нескольких инструкций. Если один из потоков пытается считывать данные из очереди, а другой поток находится в средине процесса манипуляции очередью, то считывающий поток обнаружит, что очередь находится в несогласованном состоянии. Легко понять, что при конкурентном обращении к очереди может произойти разрушение структуры данных. Часто ресурс общего доступа — это сложная структура данных, и в результате состояния конкуренции возникает разрушение этой структуры.

Вначале кажется, что описанная ситуация не имеет простого решения. Как можно предотвратить чтение очереди на одном процессоре в тот момент, когда другой процессор обновляет ее? Вполне логично аппаратно реализовать простые инструкции, такие как атомарные арифметические операции или операции сравнения, тем не менее было бы смешно аппаратно реализовывать критические участки неопределенного размера, как в приведенном примере. Все что нужно — это предоставить метод, который позволяет отметить начало и конец; критического участка, и предотвратить или *заблокировать* (lock) доступ к этому участку, пока другой поток выполняет его.

Блокировки (lock) предоставляют такой механизм. Он работает почти так же, как и дверной замок. Представим, что комната, которая находится за дверью, — это критический участок. Внутри комнаты в любой момент времени может присутствовать только один поток выполнения. Когда поток входит в комнату, он запирает за собой дверь. Когда поток заканчивает манипуляции с совместно используемыми данными, он выходит из комнаты, отпирая дверь перед выходом. Если другой поток подхо-

дит к двери, когда она заперта, то он должен ждать, пока поток, который находится внутри комнаты, не отопрет дверь и не выйдет. Потоки удерживают блокировки, а блокировки защищают данные.

В приведенном выше примере очереди запросов для защиты очереди может использоваться одна блокировка. Как только необходимо добавить запрос в очередь, поток должен вначале захватить блокировку. Затем он может безопасно добавить запрос в очередь и после этого освободить блокировку. Если потоку необходимо извлечь запрос из очереди, он тоже должен захватить блокировку. После этого он может прочитать запрос и удалить его из очереди. В конце поток освобождает блокировку. Любому другому потоку для доступа к очереди также необходимо аналогичным образом захватывать блокировку. Так как захваченная блокировка может удерживаться только одним потоком в любой момент времени, то только один поток может производить манипуляции с очередью в любой момент времени. Блокировка позволяет предотвратить конкуренцию и защитить очередь от состояния конкуренции за ресурс.

Код, которому необходимо получить доступ к очереди, должен захватить соответствующую блокировку. Если неожиданно появляется другой поток выполнения, то это позволяет предотвратить конкуренцию.

Поток 1	**Поток 2**
Попытаться заблокировать очередь	Попытаться заблокировать очередь
успешно: блокировка захвачена	неудачно: ожидаем...
	ожидаем...
обратиться к очереди...	ожидаем...
разблокировать очередь	успешно: блокировка захвачена
	обратиться к очереди.,.
	разблокировать очередь
	...

Заметим, что блокировки бывают *необязательными* (рекомендуемыми, advisory) и *обязательными* (навязываемыми, voluntary). Блокировки— это чисто программные конструкции, преимуществами которых должны пользоваться программисты. Никто не запрещает писать код, который манипулирует нашей воображаемой очередью без использования блокировок. Однако такая практика в конечном итоге приведет к состоянию конкуренции за ресурс и разрушению данных.

Блокировки бывают различных "форм" и "размеров". В операционной системе Linux реализовано несколько различных механизмов блокировок. Наиболее существенная разница между ними — это поведение кода в условиях, когда блокировка захватываетя (конфликт при захвате блокировки, contended lock). Для некоторых типов блокировок, задания просто ожидают освобождения блокировки, постоянно выполняя проверку освобождения в замкнутом цикле (busy wait[2]), в то время как другие тины блокировок переводят задание в состояние ожидания до тех пор, пока блокировка не освободится.

[2] Иными словами, "вращаются" (spin) в замкнутом цикле, ожидая на освобождение блокировки.

В следующей главе рассказывается о том, как ведут себя различные типы блокировок в операционной системе Linux, и об интерфейсах взаимодействия с этими блокировками.

Проницательный читатель в этом месте должен воскликнуть: "Блокировки не решают проблемы, они просто сужают набор всех возможных критических участков до кода захвата и освобождения блокировок. Тем не менее, здесь потенциально может возникать состояние конкуренции за ресурсы, хотя и с меньшими последствиями!" К счастью, блокировки реализованы на основе атомарных операций, которые гарантируют, что состояние конкуренции за ресурсы не возникнет. С помощью одной машинной инструкции выполняется проверка захвачен ли ключ, и, если нет, то этот ключ захватывается. То, как это делается, очень сильно зависит от аппаратной платформы, но почти для всех процессоров определяется машинная инструкция *test-and-set* (проверить и установить), которая позволяет проверить значение целочисленной переменной и присвоить этой переменной указанное число, если се значение равно нулю. Значение нуль соответствует незахваченной блокировке.

Откуда берется параллелизм

При работе в пространстве пользователя необходимость синхронизации возникает из того факта, что программы выполняются преемптивно, т.е. могут быть вытеснены другой программой по воле планировщика. Поскольку процесс может быть вытеснен в любой момент и другой процесс может быть запущен планировщиком для выполнения на этом же процессоре, появляется возможность того, что процесс может быть вытеснен независящим от него образом во время выполнения критического участка. Если новый, запланированный на выполнение процесс входит в тот же критический участок (скажем, если оба процесса — потоки одной программы, которые могут обращаться к общей памяти), то может возникнуть состояние конкуренции за ресурс. Аналогичная проблема может возникнуть даже в однопоточной программе при использовании сигналов, так как сигналы приходят асинхронно. Такой тип параллелизма, когда два события происходят не одновременно, а накладываются друг на друга, так вроде они происходят в один момент времени, называется *псевдопараллелизмом*(pseudo-concurrency).

На машине с симметричной многопроцессорностью два процесса могут действительно выполнять критические участки в один и тот же момент времени. Это называется *истинным, параллелизмом* (true concurrency). Хотя причины и семантика истинного и псевдопараллелизма разные, они могут приводить к совершенно одинаковым состояниям конкуренции и требуют аналогичных средств защиты. В ядре причины параллельного выполнения кода следующие.

- *Прерывания.* Прерывания могут возникать асинхронно, практически в любой момент времени, прерывая код, который выполняется в данный момент.
- *Отлаженные прерывания и тасклеты.* Ядро может выполнять обработчики softirq и тасклеты практически в любой момент времени и прерывать код, который выполняется в данный момент времени.
- *Преемптивность* ядра. Так как ядро является вытесняемым, то одно задание, которое работает в режиме ядра, может вытеснить другое задание, тоже работающее в пространстве ядра.

- *Переход в состояние ожидания и синхронизация с пространством пользователя.* Задание, работающее в пространстве ядра, может переходить в состояние ожидания, что вызывает активизацию планировщика и выполнение нового процесса.
- *Симметричная многопроцессорность.* Два или больше процессоров могут выполнять код в один и тот же момент времени.

Важно, что разработчики ядра поняли все причины и подготовились к возможным случаям параллелизма. Если прерывание возникает во время выполнения кода, который работает с некоторым ресурсом, и обработчик прерывания тоже обращается к этому же ресурсу, то это является ошибкой. Аналогично ошибкой является и то, что код ядра вытесняется в тот момент, когда он обращается к совместно используемому ресурсу. Переход в состояние ожидания во время выполнения критического участка в ядре открывает большой простор для состояний конкуренции за ресурсы. И наконец, два процессора никогда не должны одновременно обращаться к совместно используемым данным. Когда ясно, какие данные требуют защиты, то уже нетрудно применить соответствующие блокировки, чтобы обеспечить всем безопасность. Сложнее идентифицировать возможные условия возникновения таких ситуаций и определить, что для предотвращения конкуренции необходима та или иная форма защиты. Давайте еще раз пройдем через этот момент, потому что он очень важен. Применить блокировки в коде для того, чтобы защитить совместно используемые данные, — это не тяжело, особенно если это делается на самых ранних этапах разработки кода. Сложность состоит в том, чтобы найти эти самые совместно используемые данные и эти самые критические участки. Именно поэтому требование аккуратного использования блокировок с самого начала разработки кода — а не когда-нибудь потом — имеет первостепенную важность. Постфактум очень сложно отследить, что необходимо блокировать, и правильно внести изменения в существующий код. Результаты подобной разработки обычно не очень хорошие. Мораль — *всегда* нужно аккуратно учитывать необходимость применения блокировок с самого начала процесса разработки кода.

Код, который безопасно выполнять параллельно с обработчиком прерывания, называется *безопасным при прерываниях* (iterrupt-safe). Код, который содержит защиту от конкурентного доступа к ресурсам при симметричной многопроцессорной обработке, называется *безопасным при SMP-обработке* (SMP-safe). Код, который имеет защиту от конкурентного доступа к ресурсам при вытеснении кода ядра, называется *безопасным при вытеснения*[3] (preempt-safe). Механизмы, которые во всех этих случаях используются для обеспечения синхронизации и защиты от состояний конкуренции, будут рассмотрены в следующей главе.

Что требует защиты

Жизненно важно определить, какие данные требуют защиты. Так как любой код, который может выполняться параллельно, может потребовать защиты. Вероятно, легче определить, какие данные *не требуют* защиты, и работать дальше, отталкиваясь от этого. Очевидно, что все данные, которые доступны только одному потоку выполнения, не требуют защиты, поскольку только этот поток может обращаться к

[3] Дальше будет показано, что, за некоторыми исключениями, код, который безопасен при SMP-обработке, также безопасен и при вытеснениях.

этим данным. Например, локальные переменные, которые выделяются в автоматической памяти (и те, которые находятся в динамически выделяемой памяти, если их адреса хранятся только в стеке), не требуют никаких блокировок, так как они существуют только в стеке выполняющегося потока. Точно так же данные, к которым обращается только одно задание, не требуют применения блокировок (так как один поток может выполняться только на одном процессоре в любой момент времени).

Что же тогда *требует* применения блокировок? Это — большинство глобальных структур данных ядра. Есть хорошее эмпирическое правило: если, кроме одного, еще и другой поток может обращаться к данным, то эти данные требуют применения какого-либо типа блокировок. Если что-то видно кому-то еще — блокируйте его. Помните, что блокировать необходимо *данные,* а не *код.*

Параметры КОНФИГУРАЦИИ ядра: SMP или UP

Так как ядро операционной системы Linux может быть сконфигурировано на этапе компиляции, имеет смысл "подогнать" ядро под данный тип машины. Важной функцией ядра является поддержка симметричной многопроцессорной обработки (SMP), которая включается с помощью параметра конфигурации ядра CONFIG_SMP. На однопроцессорной {uniprocessor, UP) машине исчезают многие проблемы, связанные с блокировками, и, следовательно, если параметр CONFIG_SMP не установлен, то код, в котором нет необходимости, не компилируется в исполняемый образ ядра. Например, это позволяет на однопроцессорной машине отказаться от накладных расходов, связанных со спин-блокировками. Аналогичный прием используется для параметра CONFIG_PREEMPT (параметр ядра, который указывает, будет ли ядро вытесняемым). Такое решение является отличным проектным решение, поскольку позволяет использовать общий четкий исходный код, а различные механизмы блокировок используются при необходимости. Различные комбинации параметров CONFIG_SMP И CONFIG_PREEMPT на различных аппаратных платформах позволяют компилировать в ядро различные механизмы блокировок.

При написании кода необходимо обеспечить все возможные варианты защиты для всех возможных случаев жизни и всех возможных сценариев, которые будут рассмотрены.

При написании кода ядра следует задать себе следующие вопросы.

- Являются ли данные глобальными? Может ли другой поток выполнения, кроме текущего, обращаться к этим данным?
- Являются ли данные совместно используемыми из контекста процесса и из контекста прерывания? Используют ли их совместно два обработчика прерываний?
- Если процесс во время доступа к данным будет вытеснен, может ли новый процесс, который запланирован на выполнение, обращаться к этим же данным?
- Может ли текущий процесс перейти в состояние ожидания (заблокироваться) на какой-либо операции? Если да, то в каком состоянии он оставляет все совместно используемые данные?
- Что запрещает освободить память, в которой находятся данные?
- Что произойдет, если эта же функция будет вызвана на другом процессоре?
- Как все это учесть?

Если коротко, то почти все глобальные данные требуют применения тех или других методов синхронизации, которые будут рассмотрены в следующей главе.

Взаимоблокировки

Взаимоблокировка (тупиковая ситуация, deadlock) — это состояние, при котором каждый поток ожидает на освобождение одного из ресурсов, а все ресурсы при этом захвачены. Потоки будут ожидать друг друга, и они никогда не смогут освободить захваченные ресурсы. Поэтому ни один из потоков не сможет продолжать выполнение, что означает наличие взаимоблокировки.

Хорошая аналогия— это перекресток, на котором стоят четыре машины, которые подъехали с четырех разных сторон. Каждая машина ожидает, пока не уедут остальные машины, и ни одна из машин не сможет уехать; в результате получается тупиковая ситуация.

Самый простой пример взаимоблокировки— это *самоблокировка*[4] (self-deadlock). Если поток выполнения пытается захватить ту блокировку, которую он уже удерживает, то ему необходимо дождаться, пока блокировка не будет освобождена. Но поток никогда не освободит блокировку, потому что он ожидает на ее захват, и это приводит к тупиковой ситуации.

```
захватить блокировку
захватить блокировку еще раз
ждать, пока блокировка не будет освобождена
...
```

Аналогично рассмотрим n потоков и n блокировок. Если каждый поток удерживает блокировку, на которую ожидает другой поток, то все потоки будут заблокированы до тех пор, пока не освободятся те блокировки, на освобождение которых ожидают потоки. Наиболее часто встречающийся пример — это два потока и две блокировки, что часто называется *взаимоблокировка типа ABBA* (ABBA deadlock).

Поток 1	**Поток 2**
захватить блокировку A	захватить блокировку B
попытка захватить блокировку B	попытка захватить блокировку A
ожидание освобождения блокировки B	ожидание освобождения блокировки A

Оба потока будут ожидать друг друга, и ни один из потоков никогда не освободит первоначально захваченной блокировки, поэтому ни одна из блокировок не будет освобождена. Такая тупиковая ситуация еще называется *deadly embrace* (буквально. смертельные объятия).

Важно не допустить появление взаимоблокировок. Хотя сложно проверить готовый код на наличие взаимоблокировок, можно написать код, который не содержит взаимоблокировок. Такую возможность дает соблюдение нескольких простых правил.

- Жизненно важным является порядок захвата блокировок. Вложенные блокировки всегда должны захватываться в одном и том же порядке. Это предотвращает взаимоблокировку нескольких потоков (deadly embrace). Порядок захвата блокировок необходимо документировать, чтобы другие тоже могли его соблюдать.

[4] Б некоторых ядрах такой тип тупиковой ситуации предотвращается с помощью рекурсивных блокировок, которые позволяют одному потоку выполнения захватывать блокировку несколько раз. В операционной системе Linux, к счастью, таких блокировок нет. И это считается хорошим тоном. Хотя рекурсивные блокировки позволяют избежать проблемы самоблокировок, они приводят к небрежному использованию блокировок.

- Необходимо предотвращать зависания. Следует спросить себя: *"Всегда ли этот код сможет завершиться'?"'.* Если не выполнится какое-либо условие, то не будет ли что-то ожидать вечно?
- Не захватывать одну и ту же блокировку дважды.
- Сложность в схеме блокировок — верный путь к тупиковым ситуациям, поэтому при разработке необходимо стремиться к простоте.

Первый пункт важный и наименее сложный для выполнения. Если две или более блокировок захватываются в одном месте, то они *всегда* должны захватываться в строго определенном порядке. Допустим, у нас есть три блокировки cat, dog и fox, которые используются для защиты данных с такими же именами. И еще допустим, что у нас есть функция, которая должна работать с этими тремя структурами данных одновременно— например, может копировать данные между ними. Б любом случае, для того чтобы гарантировать безопасность доступа, эти структуры данных необходимо защищать блокировками. Если одна функция захватывает эти блокировки в следующем порядке: cat, dog и в конце fox, то *любая* другая функция должна захватывать эти блокировки (или только некоторые из них) в том же порядке. Например, если захватывать сначала блокировку fox, а потом блокировку dog, то это потенциальная возможность взаимоблокировки (а значит, ошибки в работе), потому что блокировка dog всегда должна захватываться перед блокировкой fox. И еще раз рассмотрим пример, как может возникнуть взаимоблокировка.

Поток 1	**Поток 2**
захватить блокировку cat	захватить блокировку fox
захватить блокировку dog	попытка захватить блокировку dog
попытка захватить блокировку fox	ожидание освобождения блокировки dog
ожидание освобождения блокировки fox	—

Поток 1 ожидает освобождения блокировки fox, которую удерживает поток 2, а поток 2 в это время ожидает освобождения блокировки dog, которую удерживает поток 1. Ни один из потоков никогда не освободит своих блокировок, и, соответственно, оба потока будут ждать вечно — возникает тупиковая ситуация. Если оба потока всегда захватывают блокировки в одном и том же порядке, то подобной тупиковой ситуации возникнуть не может.

Если несколько процедур захвата блокировок вложены друг в друга, то должен быть принят определенный порядок захвата. Хорошая практика — всегда использовать комментарий сразу перед объявлением блокировки, который указывает на порядок захвата. Использовать что-нибудь вроде следующего будет хорошей идеей.

```
/*
* cat_lock - всегда захватывать перед блокировкой dog
* (и всегда захватывать блокировку dog перед блокировкой fox)
*/
```

Следует заметить, что порядок *освобождения* блокировок не влияет на возможность появления взаимоблокировок, хотя освобождать блокировки в обратном порядке по отношению к их захвату — это хорошая привычка.

Очень важно предотвращать взаимоблокировки. В ядре Linux есть некоторые отладочные возможности, которые позволяют обнаруживать взаимоблокировки при выполнении кода ядра. Эти возможности будут рассмотрены в следующем разделе.

Конфликт при захвате блокировки и масштабируемость

Термин *"конфликт при захвате блокировки"* (lock contention, или просто contention) используется для описания блокировки, которая в данный момент захвачена и на освобождение которой ожидают другие потоки. Блокировки с *высоким уровнем конфликтов* (highly contended) — это те, на освобождение которых всегда ожидает много потоков. Так как задача блокировок — это сериализация доступа к ресурсу, то не вызовет большого удивления тот факт, что блокировки снижают производительность системы. Блокировка с высоким уровнем конфликтов может стать узким местом в системе, быстро уменьшая производительность. Конечно, блокировки необходимы для того, чтобы предотвратить "развал" системы, поэтому решение проблемы высокого уровня конфликтов при блокировках также должно обеспечивать необходимую защиту от состояний конкуренции за ресурсы.

Масштабируемость (scalability) — это мера того, насколько система может быть расширена. В случае операционных систем, когда говорят о масштабируемости, подразумевают большее количество процессов, большее количество процессоров, больший объем памяти. О маштабируемости можно говорить в приложении практически к любому компоненту компьютера, который можно охарактеризовать количественным параметром. В идеале удвоение количества процессоров должно приводить к удвоению процессорной производительности системы. Однако на практике, конечно, такого не бывает никогда.

Масштабируемость операционной системы Linux на большее количество процессоров возросла поразительным образом с того времени, когда поддержка многопроцессорной обработки была встроена в ядра серии 2.0. В те дни, когда поддержка многопроцессорности в операционной системе Linux только появилась, лишь одно задание могло выполняться в режиме ядра в любой момент времени. В ядрах серии 2.2 это ограничение было снято, так как механизмы блокировок стали более мелкоструктурными. В серии 2.4 и выше блокировки стали еще более мелкоструктурными. Сегодня в ядрах серии 2.6 блокировки имеют очень мелкую гранулярность, а масштабируемость получается очень хорошей.

Структурность (гранулярность, granularity) блокировки— это описание объемов тех данных, которые защищаются блокировкой, например все структуры данных одной подсистемы. С другой стороны, блокировка на уровне очень мелких структурных единиц (fine grained), используется для защиты очень маленького объема данных, например одного поля структуры. В реальных ситуациях большинство блокировок попадают между этими крайностями, они используются не для защиты целой подсистемы, но и не для защиты одного поля, а возможно для защиты отдельного экземпляра структуры. Большинство блокировок начинали использоваться на уровне крупных структурных единиц (cource graine), а потом их стали разделять на более мелкие структурные уровни, как только конфликты при захвате этих блокировок становились проблемой.

Один из примеров перевода блокировок на более мелкий структурный уровень — это блокировки очередей выполнения планировщика (runqueue), которые рассмотрены в главе 4, "Планирование выполнения процессов". В ядрах серии 2.4 и более ранних планировщик имел всего одну очередь выполнения (вспомним, что очередь

выполнения— это список готовых к выполнению процессов). В серии 2.6 был предложен O(1)-планировтчик, в котором для каждого процессора используется своя очередь пыполнения, каждая очередь имеет свою блокировку. Соответствующие блокировки развились из одной глобальной блокировки в несколько отдельных блокировок для каждой очереди, а использование блокировок развилось из глобального блокирования в использование блокировок на отдельных процессорах. Эта оптимизация очень существенна, так как на больших машинах блокировка очереди выполнения имеет очень высокий уровень конфликтов при захвате, что приводит к сериализации планирования выполнения процессов. Иными словами, код планировщика выполнял только один процессор системы в любой момент времени, а остальные процессоры — ждали.

В общем такое повышение масштабируемости — это очень хорошая вещь, которая позволяет повысить производительность операционной системы Linux на больших и более мощных системах. Чрезмерное увлечение "ростом" масштабируемости может привести к снижению производительности на небольших многопроцессорных и однопроцессорных машинах, потому что для небольших машин не требуются такие мелкоструктурные блокировки, а им приходится иметь дело с большей сложностью и с большими накладными расходами. Рассмотрим связанный список. Первоначальная схема блокировки обеспечивает одну блокировку на весь список. Со временем эта одна блокировка может стать узким местом на очень большой многопроцессорной машине, на которой очень часто обращаются к связанному списку. Для решения проблемы одна блокировка может быть разбита на большое количество блокировок — одна блокировка на один элемент списка. Для каждого элемента списка, который необходимо прочитать или записать, необходимо захватывать уникальную блокировку этого элемента. Теперь конфликт при захвате блокировки будет только в случае, когда несколько процессоров обращаются к одному элементу списка. Что делать, если все равно есть высокий уровень конфликтов? Может быть, необходимо использовать блокировку для каждого поля элемента списка? (Ответ: НЕТ.) Если серьезно, даже когда очень мелко структурированные блокировки хорошо работают на очень больших SMP-машинах, то как они будут работать на двухпроцессорных машинах? Накладные расходы, связанные с дополнительными блокировками, будут напрасными, если на двухпроцессорной машине нет существенных конфликтов при работе с блокировками.

Тем не менее масштабируемость— это важный фактор. Важно с самого начала разрабатывать схему блокировок для обеспечения хорошей масштабируемости. Блокировки на уровне крупных структурных единиц могут стать узким местом даже на машинах с небольшим количеством процессоров. Между крупноструктурными и мелкоструктурными блокировками очень тонкая грань. Слишком крупноструктурные блокировки приводят к большому уровню конфликтов, а слишком мелкоструктурные — к напрасным накладным расходам, если уровень конфликтов при захвате блокировок не очень высокий. Оба варианта эквивалентны плохой производительности.

Необходимо начинать с простого и переходить к сложному только при необходимости. Простота — это ключевой момент.

Блокировки в вашем коде

Обеспечение безопасности кода при SMP-обработке — это не то, что можно откладывать на потом. Правильная синхронизация, блокировки без тупиковых ситуаций, масштабируемость и ясность кода- все это следует учитывать при разработке с самого начала и до самого конца. При написании кода ядра, будь то новый системный вызов или переписывание драйвера устройства, необходимо, прежде всего, позаботиться об обеспечении защиты данных от конкурентного доступа.

Обеспечение достаточной защиты для любого случая — SMP, вытеснение кода ядра и так далее — в результате приведет к гарантии того, что все данные будут защищены на любой машине и в любой конфигуратщи. В следующей главе будет рассказано о том, как это осуществить.

Теперь, когда мы хорошо подкованы в теории параллелизма, синхронизации и блокировок, давайте углубимся в то, какие существуют конкретные инструменты, предоставляемые ядром Linux, для того чтобы гарантировать отсутствие состояний конкуренции и тупиковых ситуаций в коде.

9

Средства синхронизации в ядре

В предыдущей главе обсуждались источники и решения проблем, связанных с конкуренцией за ресурсы. К счастью, в ядре Linux реализовано большое семейство средств синхронизации. В этой главе обсуждаются эти средства, интерфейсы к ним, а также особенности их работы и использования. Эти решения позволяют разработчикам писать код, в котором отсутствуют состояния конкуренции за ресурсы.

Атомарные операции

Атомарные операции (atomic operations) предоставляют инструкции, которые выполняются *атомарно,* — т.е. не прерываясь. Так же как и атом вначале считался неделимой частицей, атомарные операции являются неделимыми инструкциями. Например, как было показано в предыдущей главе, операция атомарного инкремента позволяет считывать из памяти и увеличивать на единицу значение переменной за один неделимый и непрерывный шаг. В отличие от состояния конкуренции за ресурс, которая обсуждалась в предыдущей главе, результат выполнения такой операции всегда один и тот же, например, как показано в следующем примере (допустим, что значение переменной i вначале равно 7).

Поток 1	**Поток 2**
инкремент i (7->8)	
	инкремент i (8->9)

Результирующее значение 9— правильное. Параллельное выполнение двух атомарных операций с одной и той же переменной невозможно никогда. Таким образом, для такой операции инкремента состояние конкуренции за ресурс возникнуть не может.

Ядро предоставляет два набора интерфейсов для выполнения атомарных операций: один — для работы с целыми числами, а другой — для работы с отдельными битами. Эти интерфейсы реализованы для всех аппаратных платформ, которые поддерживаются операционной системой Linux. Большинство аппаратных платформ поддерживают атомарные операции или непосредственно, или путем блокировки

шины доступа к памяти при выполнении одной операции (что в свою очередь гарантирует, что другая операция не может выполниться параллельно). Это как-то позволяет справиться с проблемой в случае аппаратных платформ, таких как SPARC, которые не поддерживают базовых машинных инструкций для выполнения атомарных операций.

Целочисленные атомарные операции

Средства выполнения атомарных операций с целыми числами работают с типом данных `atomict`. Вместо того, чтобы использовать функции, которые работают непосредственно с типом данных `int` языка С, по ряду причин используется специальный тип данных. Во-первых, функции, которые выполняют атомарные операции, принимают только аргументы типа `atomict`, это гарантирует, что атомарные операции выполняются только с данными этого специального типа. В то же время это также гарантирует, что данные этого типа не смогут передаваться в другие функции, которые не выполняют атомарных операций. Действительно, ничего хорошего не будет от таких атомарных операций, которые иногда атомарные, а иногда— нет. Следующий момент— использование типа `atomic_t` позволяет гарантировать, что компилятор (по ошибке, но для повышения эффективности) не будет оптимизировать операции обращения к атомарным переменным. Важно, чтобы атомарные операции получали правильное значение адреса переменной в памяти, а не адреса временных копий. И наконец, за типом `atomict` скрываются различия между реализациями для различных аппаратных платформ.

Кроме того, что тип `atomic_t` — это 32-разрядное целое число на всех машинах,- которые поддерживаются операционной системой Linux, при разработке кода необходимо учитывать, что максимальный диапазон значений переменной этого типа не может быть больше 24 бит. Это связано с аппаратной платформой SPARC, для которой используется несколько странная реализация атомарных операций: в младшие 8 бит 32-разрядного целого числа типа `int` встроена блокировка, как показано па рис. 9.1.

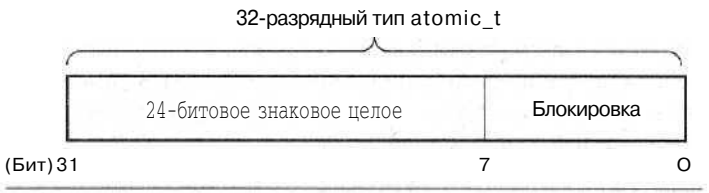


***Рис.** 9.1, Структура 32-битового типа atomic_t для аппаратной платформы SPARC в старой реализации*

Блокировка используется для предотвращения параллельного доступа к переменной атомарного типа, так как для аппаратной платформы SPARC отсутствует соответствующая поддержка на уровне машинных инструкций. Следовательно, на машинах SPARC могут быть использованы только 24 бит. Хотя код, который рассчитан на использование полного 32-битового диапазона значений, будет работать и на машинах других типов, он может приводить к странным и коварным ошибкам на машинах типа SPARC, и так делать не нужно. В последнее время умные хакеры додумались, как для аппаратной платформы SPARC обеспечить тип `atomic_t`, который позволя-

ет хранить полноценное 32-разрядное целое число, и указанного ограничения больше не существует. Тем не менее старая 24-битовая реализация все еще используется в старом коде для аппаратной платформы SPARC, и этот код все еще имеется в файле <asm/atomic.h> для этой аппаратной платформы.

Объявления всего, что необходимо для использования целочисленных атомарных операций, находятся в заголовочном файле <asm/atomic.h>. Для некоторых аппаратных платформ существуют дополнительные средства, которые уникальны только для этой платформы, но для всех аппаратных платформ существует минимальный набор операций, которые используются в ядре повсюду. При написании кода ядра необходимо гарантировать, что соответствующие операции доступны и правильно реализованы для всех аппаратных платформ.

Объявление переменных типа atomic_t производится обычным образом. При необходимости можно установить заданное значение этой переменной.

```
. atomic_t u; /* определение переменной и */
  atomic_t v = ATOMIC_INIT(0); /* определение переменной v и
  инициализация ее в значение нуль */
```

Выполнять операции так же просто.

```
atomic_set(&v, 4); /* v = 4 (атомарно) */
atomic_add(2, &v); /*v = v + 2 = 6 (атомарно) */
atomic_inc(&v); /*v = v + 1 = 7 (атомарно) */
```

Если необходимо конвертировать тип atomic_t в тип int, то нужно использовать функцию atomic_read().

```
printk("%d\n", atomic_read(&v)); /* будет напечатано "7" */
```

Наиболее частое использование атомарных целочисленных операций — это инкремент счетчиков. Защищать один счетчик с помощью сложной системы блокировок — это глупо, поэтому разработчики используют вызовы atomic_int() и atomic_dec(), которые значительно быстрее. Еще одно использование атомарных целочисленных операций — это атомарное выполнение операции с проверкой результата. Наиболее распространенный пример — это атомарные декремент и проверка результата, с помощью функции

```
int atomic_dec_and_test(atomic_t *v)
```

Эта функция уменьшает на единицу значение заданной переменной атомарного типа. Если результат выполнения операции равен нулю, то возвращается значение true, иначе возвращается false. Полный список всех атомарных операций с целыми числами (т.е. тех, которые доступны для всех аппаратных платформ) приведен в табл. 9.1. Все операции, которые реализованы для определенной аппаратной платформы, приведены в файле <asm/atomic.h>.

Обычно атомарные операции реализованы как функции с подстановкой тела и встраиваемыми инструкциями на языке ассемблера (разработчики ядра любят inline). В случае если какая-либо из функций обладает внутренней атомарностью, то обычно она выполняется в виде макроса. Например, для большинства нормальных аппаратных платформ считывание одного машинного слова данных — это атомарная операция. Операция считывания всегда возвращает машинное слово в непротиворечивом состоянии или перед операцией записи, или после нее, но во

время операции записи чтение не может быть выполнено никогда.. Следовательно, функция atomic_read() обычно реализуется как макрос, который возвращает целочисленное значение переменной типа atomic_t.

Таблица 9.1. Полный список всех атомарных операций с целыми числами

Атомарная целочисленная операция	Описание
ATOMIC_INIT(int i)	Объявление и инициализация в значение i переменной типа atomic_t
int atomic_ read(atomic_t *y)	Атомарное считывание значения целочисленной переменной v
void atomic_set (atomic_t *v, int i)	Атомарно установить переменную v в значение i
void atomic_add (int i, atomic_t *v)	Атомарно прибавить значение i к переменной v
void atomic_sub(int i, atomic_t *v)	Атомарно вычесть значение 1 из переменной v
void atomic_inc(atomic_t *v)	Атомарно прибавить единицу к переменной v
void atomic_dec(atomic_t *v)	Атомарно вычесть единицу из переменной v
int atomic_sub_and_test(int i, atomic_t *v)	Атомарно вычесть значение i из переменной v и возвратить true, если результат равен нулю, и false в противном случае
int atomic_add_negative(int i, atomic_t *v)	Атомарно прибавить значение i к переменной v и возвратить true, если результат операции меньше нуля, иначе возвратить false
int atomic_dec_and_test (atomic_t *v)	Атомарно вычесть единицу из переменной v и возвратить true, если результат операции равен нулю, иначе возвратить false
int atomic_inc_and_test(atomic_t *v)	Атомарно прибавить единицу к переменной v и возвратить true, если результат операции равен нулю, иначе возвратить false

Атомарность и порядок выполнения

От атомарных операций чтения перейдем к различиям между атомарностью и порядком выполнения. Как уже рассказывалось, операции чтения одного машинного слова всегда выполняются атомарно. Эти операции никогда не перекрываются операциями записи того же машинного слова. Иными словами, операция чтения данных всегда возвращает машинное слово в консистентном состоянии: иногда возвращается значение, которое было до записи, а иногда— то, которое стало после записи, но никогда не возвращается значение, которое было во время записи. Например, если целочисленное значение вначале было равно 42, а потом стало 365, то операция чтения всегда вернет значение 42 или 365, но никогда не смешанное значение. Это называется*атомарностью.*

Иногда бывает так, что вашему коду необходимо нечто большее, например операция чтения всегда выполняется перед ожидающей операцией записи. Это называется не атомарностью, а *порядком выполнения (ordering).* Атомарность гарантирует, что инструкции выполняются не прерываясь и что они либо выполняются полностью, либо не выполняются совсем. Порядок выполнения же гарантирует, что две или более инструкций, даже если они выполняются разными потоками или разными процессами, всегда выполняются в нужном порядке.

Атомарные операции, которые обсуждаются в этом разделе, гарантируют только атомарность. Порядок выполнения гарантируется с помощью операций *барьеров (barrier),* которые будут рассмотрены дальше в текущей главе.

В любом коде использование атомарных операций, где это возможно, более предпочтительно по сравнению со сложными механизмами блокировок. Для большинства аппаратных платформ одна или две атомарные операции приводят к меньшим накладным затратам и к более эффективному использованию процессорного кэша, чем в случае более сложных методов синхронизации. Как и в случае любого кода, который чувствителен к производительности, всегда разумным будет протестировать несколько вариантов.

Битовые атомарные операции

В дополнение к атомарным операциям с целыми числами, ядро также предоставляет семейство функций, которые позволяют работать на уровне отдельных битов. Не удивительно, что эти операции зависят от аппаратной платформы и определены в файле <asm/bitops.h>.

Тем не менее может вызвать удивление то, что функции, которые реализуют битовые операции, работают с обычными адресами памяти. Аргументами функций являются указатель и номер бита. Бит 0 — это наименее значащий бит числа, которое находится по указанному адресу. На 32-разрядных машинах бит 31 — это наиболее значащий бит, а бит 0 — наименее значащий бит машинного слова. Нет ограничений на значение номера бита, которое передается в функцию, хотя большинство пользователей работают с машинными словами и номерами битов от 0 до 31 (или до 63 для 64-битовых машин).

Так как функции работают с обычными указателями, то в этом случае нет аналога типу atomic_t, который используется для операций с целыми числами. Вместо этого можно использовать указатель на любые данные. Рассмотрим следующий пример.

```
unsigned long word = 0;

set_bit(0,&word);     /* атомарно устанавливается бит 0 */
set_bit(1, &word);      /* атомарно устанавливается бит 1  */
printk("%ul\n", word);   /* будет напечатано "3" */
clear_bit(1, &word);     /* атомарно очищается бит 1 */
change_bit(0,  &word);  /* атомарно изменяется значение бита 1,
                           теперь он очищен */

/* атомарно устанавливается бит нуль и возвращается предыдущее
   значение этого бита (нуль) */
if (test_and_set_bit(0, &word)) {
        /* условие никогда не выполнится ... */
        }
```

Список стандартных атомарных битовых операций приведен в табл. 9.2.

Для удобства работы также предоставляются неатомарные версии всех битовых операций. Эти операции работают так же, как и их атомарные аналоги, но они не гарантируют атомарности выполнения операций, и имена этих функций начинаются с двух символов подчеркивания. Например, неатомарная форма функции test_bit() будет иметь имя __test_bit(). Если нет необходимости в том, чтобы операции были атомарными, например, когда данные уже защищены с помощью блокировки, неатомарные операции могут выполняться быстрее.

Таблица 9.2. Список стандартных атомарных битовых операций

Атомарная битовая операция	Описание
`void set_bit(int nr, void *addr)`	Атомарно установить `nr`-й бит в области памяти, которая начинается с адреса `addr`
`void clear_bit(int nr, void *addr)`	Атомарно очистить `nr`-й бит в области памяти, которая начинается с адреса `addr`
`void change_bit(int nr, void *addr)`	Атомарно изменить значение `nr`-го бита в области памяти, которая начинается с адреса `addr`, на инвертированное
`int test_and_set_bit(int nr, void *addr)`	Атомарно установить значение `nr`-го бита в области памяти, которая начинается с адреса `addr`, и возвратить предыдущее значение этого бита
`int test_and_clear_bit(int nr, void *addr)`	Атомарно очистить значение `nr`-го бита в области памяти, которая начинается с адреса `addr`, и возвратить предыдущее значение этого бита
`int test_and_change_bit(int nr, void *addr)`	Атомарно изменить значение `nr`-го бита в области памяти, которая начинается с адреса `addr`, на инвертированное и возвратить предыдущее значение этого бита
`int test_bit(int nr, void *addr)`	Атомарно возвратить значение `nr`-го бита в области памяти, которая начинается с адреса `addr`

Откуда берутся неатомарные битовые операции

На первый взгляд, такое понятие, как неатомарная битовая операция, вообще не имеет смысла. Задействован только один бит, и здесь не может быть никакого нарушения целостности. Одна из операций всегда завершится успешно, что еще нужно? Да, *порядок выполнения* может быть важным, но *атомарность-то* тут при чем? В конце концов, если значение бита равно тому, которое устанавливается хотя бы одной из операций, то все хорошо, не так ли?

Давайте вспомним, что такое атомарность? Атомарность означает, что операция или завершается полностью, не прерываясь, или не выполняется вообще. Следовательно, если выполняется две атомарные битовые операции, то предполагается, что они обе должны выполниться. Понятно, что значение бита должно быть правильным (и равным тому значению, которое устанавливается с помощью последней операции, как рассказано в конце предыдущего параграфа). Более того, если другие битовые операции тоже выполняются успешно, то в некоторые моменты времени значение бита должно соответствовать тому, которое устанавливается этими промежуточными операциями.

Допустим, выполняются две атомарные битовые операции: первоначальная установка бита, а затем очистка бита. Без атомарности этот бит может быть очищен, но *никогда* не установлен. Операция установки может начаться одновременно с операцией очистки и не выполниться совсем. Операция очистки бита может завершиться успешно, и бит будет очищен, как и предполагалось. В случае атомарных операций, установка бита выполнится на самом деле. Будет существовать момент времени, в который операция считывания покажет, что бит установлен, после этого выполнится операция очистки и значение бита станет равным нулю.

Иногда может требоваться именно такое поведение, особенно если критичен порядок выполнения.

Ядро также предоставляет функции, которые позволяют найти номер первого установленного (или не установленного) бита, в области памяти, которая начинается с адреса addr:

```
int find_first_bit(unsigned long *addr, unsigned int size)
int find_first_zero_bit(unsigned long *addr, unsigned int size)
```

Обе функции в качестве первого аргумента принимают указатель на область памяти и в качестве второго аргумента — количество битов, по которым будет производиться поиск. Эти функции возвращают номер первого установленного или не установленного бита соответственно. Если код производит поиск в одном машинном слове, то оптимальным решением будет использовать функции __ffs() и _ffz(), которые в качестве единственного параметра принимают машинное слово, где будет производиться поиск.

В отличие от атомарных операций с целыми числами, при написании кода обычно нет возможности выбора, использовать или не использовать рассмотренные битовые операции, они являются единственными переносимыми средствами, которые позволяют установить или очистить определенный бит. Вопрос лишь в том, какие разновидности этих операций использовать — атомарные или неатомарные. Если код по своей сути является защищенным от состояний конкуренции за ресурсы, то можно использовать неатомарные операции, которые могут выполняться быстрее для определенных аппаратных платформ.

Спин-блокировки

Было бы очень хорошо, если бы все критические участки были такие же простые, как инкремент или декремент переменной, однако в жизни все более серьезно. В реальной жизни критические участки могут включать в себя несколько вызовов функций. Например, очень часто данные необходимо извлечь из одной структуры, затем отформатировать, произвести анализ этих данных и добавить результат в другую структуру. Весь этот набор операций должен выполняться атомарно. Никакой другой код не должен иметь возможности читать ни одну из структур данных до того, как данные этих структур будут полностью обновлены. Так как ясно, что простые атомарные операции не могут обеспечить необходимую защиту, то используется более сложный метод защиты — блокировки (lock).

Наиболее часто используемый тип блокировки в ядре Linux - это *спин-блокировки (spin lock)*. Спин-блокировка — это блокировка, которую может удерживать не более чем один поток выполнения. Если поток выполнения пытается захватить блокировку, которая находится в *состоянии конфликта (contended),* т.е. уже захвачена, поток начинает выполнять постоянную циклическую проверку (busy loop) — *"вращаться" (spin),* ожидая на освобождение блокировки. Если блокировка не находится в состоянии конфликта при захвате, то поток может сразу же захватить блокировку и продолжить выполнение. Циклическая проверка предотвращает ситуацию, в которой более одного потока одновременно может находиться в критическом участке. Следует заметить, что одна и та же блокировка может использоваться в нескольких разных местах кода, и при этом всегда будет гарантирована защита и синхронизация при доступе, например, к какой-нибудь структуре данных.

Тот факт, что спин-блокировка, которая находится в состоянии конфликта, заставляет потоки, ожидающие на освобождение этой блокировки, выполнять замкнутый цикл (и, соответственно, тратить процессорное время), является важным. *Неразумно* удерживать спин-блокировку в течение длительного времени. По своей сути спин-блокировка — это быстрая блокировка, которая должна захватываться на короткое время одним потоком. Альтернативным является поведение, когда при попытке захватить блокировку, которая находится в состоянии конфликта, поток переводится в состояние ожидания и возвращается к выполнению, когда блокировка освобождается. В этом случае процессор может начать выполнение другого кода. Такое поведение вносит некоторые накладные затраты, основные из которых — это два переключения контекста. Вначале переключение на новый поток, а затем обратное переключение на заблокированный поток. Поэтому разумным будет использовать спин-блокировку, когда время удержания этой блокировки меньше длительности двух переключений контекста. Так как у большинства людей есть более интересные занятия, чем измерение времени переключения контекста, то необходимо стараться удерживать блокировки по возможности в течение максимально короткого периода времени[1]. В следующем разделе будут описаны *семафоры (semaphore)* — механизм блокировок, который позволяет переводить потоки, ожидающие на освобождение блокировки, в состояние ожидания, вместо того чтобы периодически проверять, не освободилась ли блокировка, находящаяся в состоянии конфликта.

Спин-блокировки япляются зависимыми от аппаратной платформы и реализованы на языке ассемблера. Зависимый от аппаратной платформы код определен в заголовочном файле <asm/spinlock.h>. Интерфейс пользователя определен в файле <linux/spinlock.h>. Рассмотрим пример использования спин-блокировок.

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock(&mr_lock);

/* критический участок ... */

spin_unlock(&mr_lock);
```

Б любой момент времени блокировка может удерживаться не более чем одним потоком выполнения. Следовательно, только одному потоку позволено войти в критический участок в данный момент времени. Это позволяет организовать защиту от состояний конкуренции на многопроцессорной машине. Заметим, что на однопроцессорной машине блокировки не компилируются в исполняемый код, и, соответственно, их просто не существует. Блокиропки играют роль маркеров, чтобы запрещать и разрешать вытеснение кода (преемптивпость) в режиме ядра. Если преемптивность ядра отключена, то блокировки совсем не компилируются.

[1] Сейчас это требование становится еще более важным, так как ядро является преемптивным. Время, в течение которого удерживаются блокировки, эквивалентно времени задержки (латентпости) системного планировщика.

Внимание: спин-блокировки не рекурсивны!

В отличие от реализаций в других операционных системах, спин-блокировки в операционной системе Linux не рекурсивны. Это означает, что если поток пытается захватить блокировку, которую он уже удерживает, то этот поток начнет периодическую проверку, ожидая, пока он сам не освободит блокировку. Но поскольку поток будет периодически проверять, не освободилась ли блокировка, он никогда не сможет ее освободить, и возникнет тупиковая ситуация (самоблокировка). Нужно быть внимательными!

Спин-блокировки могут использоваться в обработчиках прерываний (семафоры не могут использоваться, поскольку они переводят процесс в состояние ожидания). Если блокировка используется в обработчике прерывания, то перед тем, как захватить эту блокировку (в другом месте - не в обработчике прерывания), необходимо запретить все локальные прерывания (запросы на прерывания на данном процессоре). В противном случае может возникнуть такая ситуация, что обработчик прерывания прерывает выполнение кода ядра, Который уже удерживает данную блокировку, и обработчик прерывания также пытается захватить эту же блокировку. Обработчик прерывания постоянно проверяет (spin), не освободилась ли блокировка. С другой стороны, код ядра, который удерживает блокировку, не будет выполняться, пока обработчик прерывания не закончит выполнение. Это пример взаимоблокировки (двойной захват), который обсуждался в предыдущей главе. Следует заметить, что прерывания необходимо запрещать только на *текущем* процессоре. Если прерывание возникает на другом процессоре (по отношению к коду ядра, захватившего блокировку) и обработчик будет ожидать на освобождение блокировки, то это не приведет к тому, что код ядра, который захватил блокировку, не сможет никогда ее освободить.

Ядро предоставляет интерфейс, который удобным способом позволяет запретить прерывания и захватить блокировку. Использовать его можно следующим образом.

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

unsigned long flags;

spin_lock_irqsave(&mr_lock,  flags);

/* критический участок ... */

spin_unlock_irqre_store(&rnr_lock,  flags);
```

Подпрограмма spin_lock_irqsave () сохраняет текущее состояние системы прерываний, запрещает прерывания и захватывает указанную блокировку. Функция spin_unlock_irqrestore (), наоборот, освобождает указанную блокировку и восстанавливает предыдущее состояние системы прерываний. Таким образом, если прерывания были запрещены, показанный код не разрешит их по ошибке. Заметим, что переменная flags передается по значению. Это потому, что указанные функции частично выполнены в виде макросов.

На однопроцессорной машине показанный пример только лишь запретит прерывания, чтобы предотвратить доступ обработчика прерывания к совместно используемым данным, а механизм блокировок скомпилирован не будет. Функции захвата и освобождения блокировки также соответственно запрещают и разрешают преемптивность ядра.

Что необходимо блокировать

Важно, чтобы каждая блокировка была четко связана с тем, что она блокирует. Еще более важно — это защищать *данные,* а не *код.* Несмотря на то что во всех примерах этой главы рассматриваются критические участки, в основе этих критических участков лежат данные, которые требуют защиты, а никак не код. Если блокировки просто блокируют участки кода, то такой код труднопонимаем и подвержен состояниям гонок. Необходимо ассоциировать данные с соответствующими блокировками. Например, структура `struct foo` блокируется с помощью блокировки `foo_lock`. С данной блокировкой также необходимо ассоциировать некоторые данные. Если к некоторым данным осуществляется доступ, то необходимо гарантировать, что этот доступ будет безопасным. Наиболее часто это означает, что перед тем, как осуществить манипуляции с данными, необходимо захватить соответствующую блокировку и освободить эту блокировку нужно после завершения манипуляций.

Если точно известно, что прерывания разрешены, то нет необходимости восстанавливать предыдущее состояние системы прерываний. Можно просто разрешить прерывания при освобождении блокировки. В этом случае оптимальным будет использование функций `spin_lock_irq()` и `spin_unlock_irq()`.

```
spinlock_t mr_lock = SPIN_LOCK_UNLOCKED;

spin_lock_irq(&mr_lock);

/* критический участок ... */

spin_unlock_irq(&mr_lock);
```

Для любого участка кода очень сложно гарантиропать, что прерывания всегда разрешены. В связи с этим не рекомендуется использовать функцию `spinlock_irq ()`. Если стоит вопрос об использовании этих функций, то лучше быть точно уверенным, что прерывания запрещены, а не огорчаться, когда найдете, что прерывания разрешены не там, где нужно.

Отладка спин-блокировок

Параметр конфигурации ядра `CONFIG_DEBUG_SPINLOCK` включает несколько отладочных проверок в коде спин-блокировок. Например, с этим параметром код спин-блокировок будет проверять использование неинициализированных спин-блокировок и освобождение блокировок, которые не были захваченными. При тестировании кода всегда необходимо включать отладку спин-блокировок.

Другие средства работы со спин-блокировками

Функция `spin_lock_init ()` используется для инициализации спин-блокировок, которые были созданы динамически (переменная типа `spinlock_t`, к которой нет прямого доступа, а есть только указатель на нее).

Функция `spin_try_lock ()` производит попытку захватить указанную спин-блокировку. Если блокировка находится в состоянии конфликта, то, вместо циклической проверки и ожидания на освобождение блокировки, эта функция возвращает ненулевое значение. Если блокировка была захвачена успешно, то функция возвращает нуль. Аналогично функция `spin_is_locked ()` возвращает ненулевое значение, если

блокировка в данный момент захвачена. В противном случае возвращается нуль. Эта функция никогда не захватывает блокировку[2].

В табл. 9.3 приведен полный список функций работы со спин-блокировками.

Таблица 9.3. Список функций работы со спин-блокировками

Функция	Описание
spin_iock()	Захватить указанную блокировку
spin_lock_irq()	Запретить прерывания на локальном процессоре и захватить указанную блокировку
spin_lock_irqsave()	Сохранить текущее состояние системы прерываний, запретить прерывания на локальном процессоре и захватить указанную блокировку
spin_unlock()	Освободить указанную блокировку
spin_unlock_irq()	Освободить указанную блокировку и разрешить прерывания на локальном процессоре
spin_unlock_irqrestore()	Освободить указанную блокировку и восстановить состояние системы прерываний на локальном процессоре в указанное первоначальное значение
spin_lock_init()	Инициализировать объект типа spinlock_t в заданной области памяти
spin_trylock()	Выполнить попытку захвата указанной блокировки и в случае неудачи возвратить ненулевое значение
spin_is_locked()	Возвратить ненулевое значение, если указанная блокировка в данный момент захвачена, и нулевое значение в противном случае

Спин-блокировки и обработчики нижних половин

Как было указано в главе 7, "Обработка нижних половин и отложенные действия", при использовании блокировок в работе с обработчиками нижних половин необходимо принимать некоторые меры предосторожности. Функция spin_lock_bh() позволяет захватить указанную блокировку и запретить все обработчики нижних половин. Функция spin_unlock_bh() выполняет обратные действия.

Обработчик нижних половин может вытеснять код, который выполняется в контексте процесса, поэтому, если данные совместно используются обработчиком нижней половины и контекстом процесса, в контексте процесса эти данные необходимо защищать путем запрещения обработки нижних половин и захвата блокировки. Аналогично, поскольку обработчик прерывания может вытеснить обработчик нижней половины, необходимо запрещать прерывания и захватывать блокировку.

Вспомним, что два тасклета (tasklet) одного типа не могут выполняться параллельно. Поэтому нет необходимости защищать данные, которые используются только тасклетами одного типа.

[2] Использование этих функций может привести к тому, что код становится "грязным". Нет необходимости часто проверять значение спин-блокировок - код или всегда должен захватывать блокировку, или вызываться только, если блокировка захвачена. Однако существуют некоторые ситуации, когда такие функции логично использовать, поэтому эти интерфейсы и предоставляются.

Если данные используются тасклетами разных типов, то необходимо использовать обычную спин-блокировку перед тем, как обращаться к таким данным в обработчике нижней половины. В этом случае нет необходимости запрещать обработку нижних половин, так как тасклет никогда не вытесняет другой тасклет, выполняющийся на том же процессоре.

В случае отложенных прерываний (softirq), независимо от того, это отложенные прерывания одного типа или разных, данные, совместно используемые обработчиками отложенных прерываний, необходимо защищать с помощью блокировки. Вспомним, что обработчики отложенных прерываний, даже одного типа, могут выполняться одновременно на разных процессорах системы. Обработчик отложенного прерывания никогда не вытесняет другие обработчики отложенных прерываний, которые выполняются на одном процессоре с ним, поэтому запрещать обработку нижних половин в этом случае не нужно.

Спин-блокировки чтения-записи

Иногда в соответствии с целью использования блокировок их можпо разделить два типа — блокировки чтения (reader lock) и блокировки записи (writer lock). Рассмотрим некоторый список, который может обновляться и в котором может выполняться поиск. Когда список обновляется (в него осуществляется запись), никакой другой код не может параллельно осуществлять запись *или* чтение этого списка. Запись означает исключительный доступ. С другой стороны, если в списке выполняется поиск (чтение информации), важно только, чтобы никто другой не выполнял записи в список. Работа со списком заданий в системе (как обсуждалось в главе 3, "Управление процессами") аналогична только что описанной ситуации. Не удивительно, что список заданий в системе защищен с помощью спин-блокировки чтения-записи (reader-writer spin lock).

Если работа со структурой данных может быть четко разделена на этапы чтения/записи, как в только что рассмотренном случае, то имеет смысл использовать механизмы блокировок с аналогичной семантикой. Для таких ситуаций операционная система Linux предоставляет спин-блокировки чтения-записи. Спин-блокировки чтения-записи обеспечивают два варианта блокировки. Один или больше потоков выполнения, которые одновременно выполняют операции считывания, могут удерживать такую блокировку. Блокировка на запись, наоборот, может удерживаться в любой момент времени только одним потоком, осуществляющим запись, и никаких параллельных считываний не разрешается. Блокировки чтения-записи иногда также называются соответственно shared/exclusive (общая/исключающая) или concurrent/exclusive (параллельная/исключающая).

Инициализировать блокировку для чтения-записи можно с помощью следующего программного кода.

```
rwlock_t mr_rwlock = RW_LOCK_UNLOCKED;

Следующий код осуществляет считывание.

read_lock(&mr_rwlock);
/* критический участок (только для считывания) ... */
read unlock(&mr_rwlock);
```

И наконец, показанный ниже код осуществляет запись.

```
write_lock(&mr_rwlock);
/* критический участок (чтение и запись) ... */
write_unlock{&mr_rwlock);
```

Обычно считывание и запись информации осуществляются в разных участках кода, как это показано в данном примере.

Заметим, что блокировку, захваченную для чтения, нельзя "повышать" до блокировки, захваченной для записи. В следующем коде

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

возникнет самоблокировка, так как при захвате блокировки на запись будет выполняться периодическая проверка, пока все потоки, которые захватили блокировку для чтения, ее не освободят; это касается и текущего потока. Если в каком-либо месте будет необходима запись, то нужно сразу же захватывать блокировку для записи. Если в вашем коде нет четкого разделения на код, который осуществляет считывание, и код, который осуществляет запись, то нет необходимости использовать блокировки чтения-записи. В таком случае оптимальным будет использование обычных спин-блокировок.

Несколько потоков чтения безопасно могут удерживать одну и ту же блокировку чтения-записи. На самом деле один поток также может безопасно рекурсивно захватывать одну и ту же блокировку для чтения. Это позволяет выполнить полезную и часто используемую оптимизацию. Если в обработчиках прерываний осуществляется только чтение и не выполняется запись, то можно "смешивать" использование блокировок с запрещением прерываний и без запрещения. Для защиты данных при чтении можно использовать функцию `read_lock()` вместо `read_lock_irqsave()`. При обращении к данным для записи все равно необходимо запрещать прерывания, например использовать функцию `write_lock_irqsave()`, так как в обработчике прерывания может возникнуть взаимоблокировка в связи с ожиданием захвата блокировки на чтение при захваченной блокировке на запись. В табл. 9.4 показан полный список средств работы с блокировками чтения-записи.

Еще один факт, который необходимо принимать во внимание при работе с блокировками чтения-записи в операционной системе Linux, — это то, что блокировка на чтение всегда имеет большее преимущество по сравнению с блокировкой на запись. Если блокировка захвачена на чтение и поток записи ожидает на ее освобождение, то все потоки, которые будут пытаться захватить блокировку на чтение, будут добиваться успеха. Поток записи, который периодически проверяет на освобождение блокировки, не сможет захватить блокировку, пока все потоки чтения эту блокировку не освободят. Поэтому большое количество потоков чтения будет приводить к "подвисанию" ожидающих потоков записи. Это важное обстоятельство всегда нужно помнить при разработке схемы блокировок.

Спин-блокировки обеспечивают очень быстрые и простые блокировки. Выполнение постоянных проверок в цикле является оптимальным, когда блокировки захватываются на очень короткое время и код не может переходить в состояние ожидания (например, в обработчиках прерываний). Если же период времени ожидания на освобождение блокировки может быть большим или имеется потенциальная возможность перехода в состояние ожидания при захваченной блокировке, то задача может быть решена с помощью семафоров.

Таблица 9.4. Список функций работы со спин-блокировками чтения-записи

Функция	Описание
read_lock ()	Захватить указанную блокировку на чтение
read_lock_irq()	Запретить прерывания на локальном процессоре и захватить указанную блокировку на чтение
read_lock_irqsave()	Сохранить состояние системы прерываний на текущем процессоре, запретить прерывания на локальном процессоре и захватить указанную блокировку на чтение
read_unlock()	Освободить указанную блокировку, захваченную для чтения
read_unlock_irq ()	Освободить указанную блокировку, захваченную для чтения, и разрешить прерывания на локальном процессоре
read_unlock_irqrestore ()	Освободить указанную блокировку, захваченную для чтения, и восстановить состояние системы прерываний в указанное значение
write_lock()	Захватить заданную блокировку на запись
write_lock_irq()	Запретить прерывания на локальном процессоре и захватить указанную блокировку на запись
write_lock_irqsave ()	Сохранить состояние системы прерываний на текущем процессоре, запретить прерывания на локальном процессоре и захватить указанную блокировку на запись
write_unlock ()	Освободить указанную блокировку, захваченную для записи
write_unlock_irq ()	Освободить указанную блокировку, захваченную для записи, и разрешить прерывания на локальном процессоре
write_unlock_irqrestore()	Освободить указанную блокировку, захваченную для записи, и восстановить состояние системы прерываний в указанное значение
write_trylock()	Выполнить попытку захватить заданную блокировку на запись и в случае неудачи возвратить ненулевое значение
rw_lock_init()	Инициализировать объект типа rwlock_t в заданной области памяти
rw is locked ()	Возвратить ненулевое значение, если указанная блокировка захвачена, иначе возвратить нуль

Семафоры

В операционной системе Linux семафоры (semaphore) — это блокировки, которые переводят процессы в состояние ожидания. Когда задание пытается захватить семафор, который уже удерживается, семафор помещает это задание в очередь ожидания (wait queue) и переводит это задание в состояние ожидания (sleep). Когда процессы[3], которые удерживают семафор, освобождают блокировку, одно из заданий очереди ожидания возвращается к выполнению и может захватить семафор.

Давайте снова возвратимся к аналогии двери и ключа. Когда человек, который хочет открыть дверь, подходит к двери, он может захватить ключ и войти в комнату. Отличие состоит в том, что произойдет, если к двери подойдет другой человек. В этом случае он записывает свое имя в список на двери и ложится поспать.

[3] Как будет показано дальше, несколько процессов могут при необходимости одновременно удерживать один семафор.

Когда человек, который находился внутри комнаты, выходит из нее, он проверяет список на двери. Если в списке есть чье-либо имя, то он выбирает первое из имен списка, дает соответствующему человеку пинка, будит его и позволяет войти в комнату. Таким образом, ключ (т.е. семафор) позволяет только одному человеку (т.е. потоку) находиться в комнате (т.е. в критическом разделе) в один момент времени. Если комната занята, то, вместо того чтобы периодически проверять, человек записывает свое имя в список (т.е. в очередь ожидания) и засыпает (т.е. блокируется в очереди ожидания и переходит в приостановленное состояние), что позволяет процессору выполнять некоторый другой код. Такой режим работы позволяет лучше использовать процессор, чем в случае спин-блокировок, так как при этом не тратится процессорное время на выполнение периодических проверок в цикле. Тем не менее использование семафоров, по сравнению со спин-блокировками, связано со значительно большими накладными расходами.

Из такого поведения семафоров, связанного с переводом процессов в состояние ожидания, можно сделать следующие интересные заключения.

- Так как задания, которые конфликтуют при захвате блокировки, переводятся в состояние ожидания и в этом состоянии ждут, пока блокировка не будет освобождена, семафоры хорошо подходят для блокировок, которые могут удерживаться в течение длительного времени.
- С другой стороны, семафоры не оптимальны для блокировок, которые удерживаются в течение очень короткого периода времени, так как накладные затраты на перевод процессов в состояние ожидания могут "перевесить" время, в течение которого удерживается блокировка.
- Так как поток выполнения во время конфликта при захвате блокировки находится в состоянии ожидания, то семафоры можно захватывать только в контексте процесса. Контекст прерывания планировщиком не управляется.
- При удержании семафора (хотя разработчик может и не очень хотеть этого) процесс может переходить в состояние ожидания. Это не может привести к тупиковой ситуации, когда другой процесс попытается захватить блокировку (он просто переходит в состояние ожидания, что в конце концов дает возможность выполняться первому процессу).
- При захвате семафора нельзя удерживать спин-блокировку, поскольку процесс может переходить в состояние ожидания, ожидая на освобождение семафора, а при удержании спин-блокировки в состояние ожидания переходить нельзя.

Эти факты подчеркивают особенности использования семафоров по сравнению со спин-блокировками. В большинстве случаев выбор решения, использовать или не использовать семафоры, прост. Если код должен переходить в состояние ожидания, что очень часто возникает при необходимости синхронизации с пространством пользователя, семафоры — единственное решение. Использовать семафоры всегда проще, даже если в них нет строгой необходимости, так как они предоставляют гибкость, связанную с переходом процессов в состояние ожидания. Если же возникает необходимость выбора между семафорами и спид-блокировками, то решение должно основываться на времени удержания блокировки. В идеале, все блокировки необходимо удерживать, по возможности, в течение наиболее короткого времени. При использовании семафоров, однако, допустимы более длительные периоды удержания

блокировок. В дополнение ко всему, семафоры не запрещают преемптивность ядра, и, следовательно, код, который удерживает семафор, может быть вытеснен. Это означает, что семафоры не оказывают вредного влияния на задержки (латентность) планировщика.

Последняя полезная функция семафоров — это то, что они позволяют иметь любое количество потоков, которые одновременно удерживают семафор. В то время как спин-блокировки позволяют удерживать блокировку только одному заданию в любой момент времени, количество заданий, которым разрешено одновременно удерживать семафор, может быть задано при декларации семафора. Это значение называется *счетчиком использования (usage count)* или просто *счетчиком (count)*. Наиболее часто встречается ситуация, когда разрешенное количество потоков, которые одновременно могут удерживать семафор, равно одному, как и для спин-блокировок. В таком случае счетчик использования равен единице и семафоры называются *бинарными семафорами (binary semaphore)* (потому что он может удерживаться только одним заданием или совсем никем не удерживаться) или *взаимоисключающими блокировками* (*mutex, мьютекс)* (потому что он гарантирует взаимоисключающий доступ —mutual exclusion). Кроме того, счетчику при инициализации может быть присвоено значение, большее единицы. В этом случае семафор называется *счетным семафором (counting semaphore, семафор-счетчик),* и он допускает количество потоков, которые одновременно удерживают блокировку, не большее чем значение счетчика использования. Семафоры-счетчики не используются для обеспечения взаимоисключающего доступа, так как они позволяют нескольким потокам выполнения одновременно находиться в критическом участке. Вместо этого они используются для установки лимитов в определенном коде. В ядре они используются мало. Если вы используете семафор, то, скорее всего, вы используете взаимоисключающую блокировку (семафор со счетчиком, равным единице).

Семафоры были формализованы Эдсгером Вайбом Дейкстрой[4] (Edsger Wybe Dijkstra) в 1968 году как обобщенный механизм блокировок. Семафор поддерживает две атомарные операции P () и V (), название которых происходит от голландских слов *Proben* (тестировать) и *Verhogen* (выполнить инкремент). Позже эти операции начали называть down () и up () соответственно.

В операционной системе Linux они имеют такое же название. Операция down () используется для того, чтобы захватить семафор путем уменьшения его счетчика на единицу. Если значение этого счетчика больше или равно нулю, то блокировка захватывается успешно и задание может входить в критический участок. Если значение счетчика меньше нуля, то задание помещается в очередь ожидания и процессор переходит к выполнению каких-либо других операций. Об использовании этой функции говорят в форме глагола— семафор *опускается (down)* для того, чтобы его захватить. Метод up () используется для того, чтобы освободить семафор после завершения выполнения критического участка. Эту операцию называют *поднятием (upping)* семафора.

[4] Доктор Дейкстра (1930-2002 г.) один из самых талантливых ученых за всю (конечно, не очень долгую) историю существования вычислительной техники как области науки. Его многочисленные труды включают работы но проектированию операционных систем и по теории алгоритмов, сюда же входит концепция семафоров. Он родился в городе Роттердам, Нидерланды, и преподавал в университете штата Техас в течение 15 лет. Тем не менее, он был бы не очень доволен большим количеством директив *GOTO* в ядре Linux.

Последний метод используется для инкремента значения счетчика. Если очередь ожидания семафора не пуста, то одно из заданий этой очереди возвращается к выполнению и захватывает семафор.

Создание и инициализация семафоров

Реализация семафоров зависит от аппаратной платформы и определена в файле <asm/semaphore.h>. Структура `struct semaphore` представляет объекты типа семафор. Статическое определение семафоров выполняется следующим образом.

```
static DECLARE_SEMAPHORE_GENERIC(name, count);
```

где name — имя переменной семефора, а `count` — счетчик семафора. Более короткая запись для создания взаимоисключающей блокировки (mutex), которая используются наиболее часто, имеет следующий вид.

```
static DECLARE_MUTEX(name);
```

где name — это снова имя переменной типа семафор. Чаще всего семафоры создаются динамически, как часть больших структур данных. В таком случае для инициализации семафора, который создается динамически и на который есть только непрямая ссылка через указатель, необходимо использовать функцию

```
sema_init(sem, count);
```

где sem— это указатель, а `count` — счетчик использования семафора. Аналогично для инициализации динамически создаваемой взаимоисключающей блокировки можно использовать функцию

```
init_MUTEX(sem);
```

Неизвестно, почему слово "mutex" в имени функции init_MUTEX() выделено большими буквами и почему слово "init" идет перед ним, в то время как имя функции `sema_init ()` таких особенностей не имеет. Тем не менее ясно, что это выглядит не логично, и я приношу свои извинения за это несоответствие. Надеюсь, что после прочтения главы 7 ни у кого уже не будет вызывать удивление то, какие имена придумывают символам ядра.

Использование семафоров

Функция `down_interruptible ()` выполняет попытку захватить данный семафор. Если эта попытка неудачна, то задание переводится в состояние ожидания с флагом TASK_INTERRUPTIBLE. Из материала главы 3 следует вспомнить, что такое состояние процесса означает, что задание может быть возвращено к выполнению с помощью сигнала и что такая возможность обычно очень ценная. Если сигнал приходит в тот момент, когда задание ожидает на освобождение семафора, то задание возвращается к выполнению, а функция `down_interruptible ()` возвращает значение -EINTR. Альтернативой рассмотренной функции выступает функция down (), которая переводит задание в состояние ожидания с флагом TASK_UNINTERRUPTIBLE. В большинстве случаев это нежелательно, так как процесс, который ожидает на освобождение семафора, не будет отвечать на сигналы. Поэтому функция `down_interruptible ()` используется значительно более широко, чем функция down (). Да, имена этих функций, конечно, далеки от идеала.

Функция down_trylock () используется для неблокирующего захвата указанного семафора. Если семафор уже захвачен, то функция немедленно возвращает ненулевое значение. В случае успеха по захвату блокировки возвращается нулевое значение и захватывается блокировка.

Для освобождения захваченного семафора необходимо вызвать функцию up (). Рассмотрим следующий пример.

```
/* объявление и описание семафора с именем mr_sem и первоначальным
   значением счетчика, равным 1 */
static DECLARE_MUTEX(mr_sem);

if (down_interruptible(&mr_sem))
        /* получен сигнал и семафор не захвачен */

/* критический участок ... */

/* освободить семафор */
up(&mr_sem);
```

Полный список функций работы с семафорами приведен в табл. 9.5.

Таблица 9.5. Список функций работы с семафорами

Функция	Описание
sema_init(struct semaphore *, int)	Инициализация динамически созданного семафора и установка для него указанного значения счетчика использования
init_MUTEX(struct semaphore *)	Инициализация динамически созданного семафора и установка его счетчика использования в значение 1
init_MUTEX_LOCKED (struct semaphore *)	Инициализация динамически созданного семафора и установка его счетчика использования в значение 0 (т.е. семафор изначально заблокирован)
down_interruptible(struct semaphore *)	Выполнить попытку захватить семафор и перейти в прерываемое состояние ожидания, если семафор находится в состоянии конфликта при захвате (contended)
down(struct semaphore *)	Выполнить попытку захватить семафор и перейти в непрерываемое состояние ожидания, если семафор находится в состоянии конфликта при захвате (contended)
down_trylock(struct semaphore *)	Выполнить попытку захватить семафор и немедленно возвратить ненулевое значение, если семафор находится в состоянии конфликта при захвате (contended)
up(struct semaphore *)	Освободить указанный семафор и возвратить к выполнению ожидающее задание, если такое есть

Семафоры чтения-записи

Семафоры, так же как и спин-блокировки, могут быть типа чтения-записи. Ситуации, в которых предпочтительнее использовать семафоры чтения-записи такие же как и в случае использования спин-блокировок чтения-записи.

Семафоры чтения-записи представляются с помощью структуры `struct rw_semaphore`, которая определена в файле <asm/rwsem.h>. Статически определенный семафор чтения-записи может быть создан с помощью функции

```
static DECLARE_RWSEM(name);
```

где name — это имя нового семафора.

Семафоры чтения-записи, которые создаются динамически, могут быть инициализированы с помощью следующей функции.

```
init_rwsem(struct rw_semaphore *sem)
```

Все семафоры чтения-записи являются взаимоисключающими (mutex), т.е. их счетчик использования равен единице. Любое количество потоков чтения может одновременно удерживать блокировку чтения, если при этом нет ни одного потока записи. И наоборот, только один поток записи может удерживать блокировку, захваченную на запись, если нет ни одного потока чтения. Все семафоры чтения-записи используют непрерываемое состояние ожидания, поэтому существует только одна версия функции down (). Рассмотрим следующий пример.

```
static DECLARE_RWSEM(mr_rwsem);

/* попытка захватить семафор для чтения */
down_read(&mr_rwsem);

/* критический участок (только чтение) .. */

/* освобождаем семафор */
up_read(&rar_rwsem);
/* ... */

/* попытка захватить семафор на запись */
down_write(&mr_rwsem);

/* освобождаем семафор */
/* критический участок (чтение и запись) ... */
up write(&mr rwsem);
```

Для семафоров есть реализации функций down_read_trylock () и down_write_trylock (). Каждая из них принимает один параметр — указатель на семафор чтения-записи. Обе функции возвращают ненулевое значение, если блокировка захвачена успешно, и нуль, если блокировка находится в состоянии конфликта. Следует быть внимательными — поведение этих функций противоположно поведению аналогичных функций для обычных семафоров, причем без всякой на то причины!

Семафоры чтения-записи имеют уникальную функцию, аналога которой нет для спин-блокировок чтения-записи. Это функция downgradewriter (), которая авто-

матичсски превращает блокировку, захваченную на запись, в блокировку, захваченную на чтение.

Семафоры чтения-записи, так же как и спин-блокировки аналогичного типа, должны использоваться, только если есть четкое разделение между участками кода, которые осуществляют чтение, и участками кода, которые осуществляют запись. Использование механизмов блокировок чтения-записи приводит к дополнительным затратам, поэтому их стоит использовать, только если код можно четко разделить на участки чтения и записи.

Сравнение спин-блокировок и семафоров

Понимание того, когда использовать спин-блокировки, а когда семафоры является важным для написания оптимального кода. Однако во многих случаях выбирать очень просто. В контексте прерывания могут использоваться только спин-блокировки, и только семафор может удерживаться процессом, который находится в состоянии ожидания. В табл. 9.6 показан обзор требований того, какой тип блокировок использовать.

Таблица 9.6. Что следует использовать: семафоры или спин-блокировки

Требование	Рекомендуемый тип блокировки
Блокировка с малыми накладными затратами (low overhead)	Спин-блокировки более предпочтительны
Малое время удержания блокировки	Спин-блокировки более предпочтительны
Длительное время удержания блокировки	Семафоры более предпочтительны
Необходимо использовать блокировку в контексте прерывания	Необходима спин-блокировка
Необходимо переходить в состояние ожидания (steep) при захваченной блокировке	Необходимо использовать семафоры

Условные переменные

Условные переменные (conditional variable, completion variable) — простое средство синхронизации между двумя заданиями, которые работают в режиме ядра, когда необходимо, чтобы одно задание послало сигнал другому о том, что произошло некоторое событие. При этом одно задание ожидает *на* условной переменной, пока другое задание не выполнит некоторую работу. Когда другое задание завершит выполнение своей работы, оно использует условную переменную для того, чтобы возвратить к выполнению все ожидающие на ней задания. Если это кажется похожим на работу семафора, то именно так оно и есть, идея та же. В действительности, условные переменные просто обеспечивают простое решение проблемы, для которой в других ситуациях используются семафоры. Например, в системном вызове vfork() условная переменная используется для возврата к выполнению родительского процесса при завершении порожденного.

Условные переменные представляются с помощью структуры `struct completion`, которая определена в файле `<linux/completion.h>`.

Статически условная переменная может быть создана с помощью макроса

```
DECLARE_COMPLETI0N(mr_comp);
```

Динамически созданная условная переменная может быть инициализирована с помощью функции `init_completion ()`.

Задание, которое должно ожидать на условной переменной, вызывает функцию `wait_for_completion ()`. После того как наступило ожидаемое событие, вызов функции `complete ()` посылает сигнал заданию, которое ожидает на условной переменной, и это задание возвращается к выполнению. В табл. 9.7 приведены методы работы с условными переменными.

Таблица. 9.7. Методы работы с условными переменными

Метод	Описание
`init_completion(struct completion *)`	Инициализация динамически созданной условной переменной в заданной области памяти
`wait_for_completion(struct completion *)`	Ожидание сигнала на указанной условной переменной
`complete(struct completion*)`	Отправка сигнала всем ожидающим заданиям и возвращение их к выполнению

Для примеров использования условных переменных смотрите файлы `kernel/sched.c` и `kernel/fork.c`. Наиболее часто используются условные переменные, которые создаются динамически, как часть структур данных. Код ядра, который ожидает па инициализацию структуры данных, вызывает функцию `wait_for_completion()`. Когда инициализация закончена, ожидающие задания возвращаются к выполнению с помощью вызова функции `complete()`.

BLK: Большая блокировка ядра

Добро пожаловать к "рыжему пасынку" ядра. Большая блокировка ядра (Big Kernel Lock, BKL) — это глобальная спин-блокировка, которая была создана специально для того, чтобы облегчить переход от первоначальной реализации SMP n операционной системе Linux к мелкоструктурным блокировкам. Блокировка BKL имеет следующие интересные свойства.

- Во время удержания BKL можно переходить в состояние ожидания. Блокировка автоматически освобождается, когда задание переходит в состояние ожидания, и снова захватывается, когда задание планируется на выполнение. Конечно, это не означает, что *безопасно* переходить в состояние ожидания при удержании BKL, просто это *можно* делать и это не приведет к взаимоблокировке.
- Блокировка BKL рекурсивна. Один процесс может захватывать эту блокировку несколько раз подряд, и это не приведет к самоблокировке, как в случае обычных спин-блокировок.
- Блокировка BKL может использоваться только в контексте процесса.
- Блокировка BKL — это от лукавого.

Рассмотренные свойства дали возможность упростить переход от ядер серии 2.0 к серии 2.2. Когда в ядро 2-0 была введена поддержка SMP, только одно задание могло выполняться в режиме ядра в любой момент времени (конечно, сейчас ядро распараллелено очень хорошо— пройден огромный путь). Целью создания ядра серии 2.2 было обеспечение возможности параллельного выполнения кода ядра на нескольких процессорах. Блокировка BKL была введена для того, чтобы упростить переход к мелкоструктурным блокировкам. В те времена она оказала большую помощь, а сегодня она приводит к ухудшению масштабируемости[5].

Использовать блокировку BKL не рекомендуется. На самом деле, новый код никогда не должен использовать BKL. Однако эта блокировка все еще достаточно интенсивно используется в некоторых частях ядра. Поэтому важно понимать особенности большой блокировки ядра и интерфейса к ней. Блокировка BKL ведет себя, как обычная спин-блокировка, за исключением тех особенностей, которые были рассмотрены выше. Функция `lock_kernel ()` позволяет захватить блокировку, а функция `unlock_kernel ()` — освободить блокировку. Каждый поток выполнения может рекурсивно захватывать эту блокировку, но после этого необходимо столько же раз вызвать функцию `unlock_kernel ()`. При последнем вызове функции освобождения блокировки блокировка будет освобождена. Функция `kernellocked ()` возвращает ненулевое значение, если блокировка в данный момент захвачена, в противном случае возвращается нуль. Эти интерфейсы определены в файле <linux/smp_lock.h>. Рассмотрим простой пример использования этой блокировки.

```
lock_kernel();

/*
* Критический раздел, который синхронизирован со всеми пользователями
* блокировки BKL...
* Заметим, что здесь можно безопасно переходить в состояние ожидания
* и блокировка будет прозрачным образом освобождаться.
* После перепланирования блокировка будет прозрачным образом снова
* захватываться.
* Это гарантирует, что не возникнет состояния взаимоблокировки,
* но все-таки лучше не переходить в состояние ожидания,
* если необходимо гарантировать защиту данных!
*/

unlock_kernel();
```

Когда эта блокировка захвачена, происходит запрещение преемптивности. Для ядер, скомпилированных под однопроцессорную машину, код BKL на самом деле не выполняет никаких блокировок. В табл. 9.8 приведен полный список функций работы с BKL.

[5] Хотя, может быть, она и не такая страшная, какой ее иногда пытаются представить, вес же некоторые люди считают ее "воплощением дьявола" в ядре.

Таблица 9.8. Функции работы с большой блокировкой ядра

Функция	Описание
lock_kernel()	Захватить блокировку BKL
unlock_kernel()	Освободить блокировку BKL
kernel_locked()	Возвратить ненулевое значение, если блокировка захвачена, и нуль- в противном случае

Одна из самых главных проблем, связанных с большой блокировкой ядра, — как определить, что защищается с помощью данной блокировки. Часто блокировка BKL ассоциируется с кодом (например, она "синхронизирует вызовы функции foo () "), а не с данными ("защита структуры foo "). Это приводит к тому, что заменить BKL обычными сиин-блокировками бывает сложно, потому что нелегко определить, что же все-таки необходимо блокировать. На самом деле, подобная замена еще более сложна, так как необходимо учитывать все взаимоотношения между всеми участками кода, которые используют эту блокировку.

Секвентные блокировки

Секвентная блокировка (seq lock) — это новый тип блокировки, который появился в ядрах серии 2.6. Эти блокировки предоставляют очень простой механизм чтения и записи совместно используемых данных. Работа таких блокировок основана на счетчике последовательности событий. Перед записью рассматриваемых данных захватывается спин-блокировка, и значение счетчика увеличивается на единицу. После записи данных значение счетчика снова увеличивается на единицу, и спин-блокировка освобождается, давая возможность записи другим потокам. Перед чтением и после чтения данных проверяется значение счетчика. Если два полученных значения одинаковы, то во время чтения данных новый акт записи не начинался, Если к тому же оба эти значения четные, то к моменту начала чтения акт записи был закончен (при захвате блокировки на запись значение счетчика становится нечетным, а перед освобождением — снова четным, так как изначальное значение счетчика равно нулю).

Определение секвентной блокировки можно записать следующим образом.

```
seqlock_t mr_seq_lock = SEQLOCK_UNLOCKED;
```

Участок кода, который осуществляет запись, может выглядеть следующим образом.

```
write_seqlock(&mr_seq_lock);
/* блокировка захвачена на запись ... */
write_sequnlock(&mr_seq_lock);
```

Это выглядит, как работа с обычной спин-блокировкой. Необычность появляется в коде чтения, который несколько отличается от ранее рассмотренных.

```
unsigned long seq;
do {
seq = read_seqbegin(&mr_seq_lock);
/* здесь нужно читать данные ... */
} while (read_seqretry(&mr_seq_lock, seq));
```

Секвентные блокировки полезны для обеспечения очень быстрого доступа к данным в случае, когда применяется много потоков чтения и мало потоков записи. Кроме того, при использовании этого типа блокировок потоки записи получают более высокий приоритет перед потоками чтения. Блокировка записи всегда будет успешно захвачена, если нет других потоков записи. Потоки чтения никак не влияют на захват блокировки записи, в противоположность тому, что имеет место для спин-блокировок и семафоров чтения-записи. Более того, потоки, которые ожидают на запись, будут вызывать постоянные повторения цикла чтения (как в показанном примере) до тех пор, пока не останется ни одного потока, удерживающего блокировку записи во время чтения данных.

Средства запрещения преемптивности

Так как ядро является вытесняемым, процесс, работающий в режиме ядра, может прекратить выполнение в любой момент, чтобы позволить выполняться более высокоприоритетному процессу. Это означает, что новое задание может начать выполняться в том же критическом участке, в котором выполнялось вытесненное задание. Для того чтобы предотвратить такую возможность, код, который отвечает за преемптивность ядра, использует спин-блокировки в качестве маркеров, чтобы отмечать участки "непреемптивности". Если спин-блокировка захвачена, то ядро является невытесняемым. Так как проблемы, связанные с параллелизмом, в случае SMP и преемптивного ядра одинаковы, то, если ядро уже является безопасным для SMP-обработки, такое простое дополнение позволяет также сделать ядро безопасным и при вытеснении.

Будем надеяться, что это действительно так. На самом деле возникают некоторые ситуации, в которых нет необходимости использовать спин-блокировки, но нужно запрещать преемптивность ядра. Наиболее часто ситуация такого рода возникает из-за данных, привязанных к определенным процессорам (per-processor data). Если используются данные, уникальные для каждого процессора, то может быть необязательным защищать их с помощью спин-блокировок, потому что только один процессор может получать доступ к этим данным. Если никакая спин-блокировка не захвачена и ядро является преемптивным, то появляется возможность доступа к тем же переменным для вновь запланированного задания, как показано в следующем примере.

```
задание A манипулирует переменной foo
задание A вытесняется
задание B планируется на выполнение
задание B манипулирует переменной foo
задание B завершается
задание A планируется на выполнение
задание A манипулирует переменной foo
```

Следовательно, даже для однопроцессорного компьютера к некоторой переменной может псевдопараллельно обращаться несколько процессов. В обычной ситуации для такой переменной требуется спин-блокировка (для защиты при истинном параллелизме на многопроцессорной машине). Если эта переменная связана с одним процессором, то для нее не требуется блокировка.

Для решения указанной проблемы преемптивность ядра можно запретить с помощью функции preempt_disable () . Этот вызов может быть вложенным, т.е. функцию можно вызывать много раз подряд. Для каждого такого вызова требуется соответствующий вызов функции preerapt_enable (). Последний вызов функции preemptenable() разрешает преемптивность, как показано в следующем примере.

```
preempt_disable();
/* преемптивность запрещена ... */
preempt_enable();
```

Счетчик преемптивности текущего процесса содержит значение, равное количеству захваченных этим процессом блокировок плюс количество вызовов функции preempt_disable(). Если значение этого счетчика равно нулю, то ядро является вытесняемым. Если значение этого счетчика больше или равно единице, то ядро не вытесняемое. Данный счетчик невероятно полезен для отладки атомарных операций совместно с переходами в состояние ожидания. Функция preempt_count () возвращает значение данного счетчика. В табл. 9.9 показан полный список функций управления преемптивностыо.

Таблица 9.9. Функции управления преемптивностыо ядра

Функция	Описание
preempt_disable()	Запретить вытеснение кода ядра
preempt_enable()	Разрешить вытеснение кода ядра
preempt_enable_no_resched()	Разрешить вытеснение кода ядра, но не перепланировать выполнение процесса
preempt_count()	Возвратить значение счетчика преемптивности

Более полное решение задачи работы с данными, связанными с определенным процессором, — это получение номера процессора (который используется в качестве индекса для доступа к данным, связанным с определенным процессором) с помощью функции get_cpu (). Эта функция запрещает преемптивность ядра перед тем, как возвратить номер текущего процессора.

```
int cpu = get_cpu();

/* работаем с данными, связанными с текущим процессором ... */

/* работа закончена, снова разрешаем вытеснение кода ядра */
put_cpu();
```

Барьеры и порядок выполнения

В случае, когда необходимо иметь дело с синхронизацией между разными процессорами или разными аппаратными устройствами, иногда возникает требование, чтобы чтение памяти (load) или запись в память (save) выполнялись в том же порядке, как это указано в исходном программном коде. При работе с аппаратными устройствами часто необходимо, чтобы некоторая указанная операция чтения была выполнена перед другими операциями чтения или записи. В дополнение к этому, на

симметричной многопроцессорной системе может оказаться необходимым, чтобы операции записи выполнялись строго в том порядке, как это указано в исходном программном коде (обычно для того, чтобы гарантировать, что последовательные операции чтения получают данные в том же порядке). Эти проблемы усложняются тем, что как компилятор, так и процессор могут менять порядок операций чтения и записи[6] для повышения производительности. К счастью, все процессоры, которые переопределяют порядок операций чтения или записи предоставляют машинные инструкции, которые требуют выполнения операций чтения-записи памяти в указанном порядке. Также существует возможность дать инструкцию компилятору, что нельзя изменять порядок выполнения операций при переходе через определенную точку программы. Эти инструкции называются *барьерами (barrier)*.

Рассмотрим следующий код.

```
a = 1;
b = 2;
```

На некоторых процессорах запись нового значения в область памяти, занимаемую переменной b, может выполниться до того, как будет записано новое значение в область памяти переменной a. Компилятор может выполнить такую перестановку статически и внести в файл объектного кода, что значение переменной b должно быть установлено перед переменной a. Процессор может изменить порядок выполнения динамически путем предварительной выборки и планирования выполнения внешне вроде бы независимых инструкций для повышения производительности. В большинстве случаев такая перестановка операций будет оптимальной, так как между переменными a и b нет никакой зависимости. Тем не менее иногда программисту все-таки виднее.

Хотя в предыдущем примере и может быть изменен порядок выполнения, ни процессор, ни компилятор никогда не будут менять порядок выполнения следующего кода, где переменные a и b являются глобальными.

```
a = 1;
b = a;
```

Это происходит потому, что в последнем случае четко видно зависимость между переменными a и b. Однако ни компилятор, ни процессор не имеют никакой информации о коде, который выполняется в других контекстах. Часто важно, чтобы результаты записи в память "виделись" в нужном порядке другим кодом, который выполняется за пределами нашей досягаемости. Такая ситуация часто имеет место при работе с аппаратными устройствами, а также возникает на многопроцессорных машинах.

Функция rmb () позволяет установить барьер чтения памяти (read memory barrier). Она гарантирует, что никакие операции чтения памяти, которые выполняются перед вызовом функции rmb (), не будут переставлены местами с операциями, которые выполняются после этого вызова. Иными словами, все операции чтения, которые указаны до этого вызова, будут выполнены перед этим вызовом, а все операции чтения, которые указаны после этого вызова никогда не будут выполняться перед ним.

---

[6] Процессоры Intel x86 никогда не переопределяют порядок операций записи, Т.е. выполняют запись всегда в указанном порядке. Тем не менее другие процессоры могут нести себя и по-другому,

Функция wmb () позволяет установить барьер записи памяти (write barrier). Она работает так же, как и функция rmb (), но не с операциями чтения, а с операциями записи — гарантируется, что операции записи, которые находятся по разные стороны барьера, никогда не будут переставлены местами друг с другом.

Функция mb () позволяет создать барьер на чтение и запись. Никакие операции чтения и записи, которые указаны по разные стороны вызова функции rab {) , не будут переставлены местами друг с другом. Эта функция предоставляется пользователю, так как существует машинная инструкция (часто та же инструкция, что используется вызовом rmb ()), которая позволяет установить барьер на чтение и запись.

Вариант функции rmb() - read_barrier_depends() - обеспечивает создание барьера чтения, но *только для тех операций чтения, от которых зависят следующие, за ними операции чтения.* Гарантируется, что все операции чтения, которые указаны перед барьером выполнятся перед теми операциями чтения, которые находятся после барьера и зависят от операций чтения, идущих перед барьером. Все понятно? В общем, эта функция позволяет создать барьер чтения, так же как и функция rmb (), но этот барьер будет установлен только для некоторых операций чтения — тех, которые зависят друг от друга.

Для некоторых аппаратных платформ функция read_barrier_depends () выполняется значительно быстрее, чем функция rmb (), так как для этих платформ функция read_barrier_depends () просто не нужна и вместо нее выполняется инструкция noop (нет операции).

Рассмотрим пример использования функций mb () и rmb (). Первоначальное значение переменной a равно 1, а переменной b равно 2-

Поток 1	**Поток 2**
`a=3;`	
`mb();`	
`b=4;`	`c=b;`
	`rmb();`
	`d=a;`

Без использования барьеров памяти для некоторых процессоров возможна ситуация, в которой после выполнения этих фрагментов кода переменной с присвоится *новое,* значение переменной b, в то время как переменной d присвоится *старое* значение переменной a. Например, переменная c может стать равной 4 (что мы и хотим), а переменная d может остаться равной 1 (чего мы не хотим). Использование функции mb () позволяет гарантировать, что переменные a и b записываются в указанном порядке, а функция rmb () гарантирует, что чтение переменных b и a будет выполнено в указанном порядке.

Такое изменение порядка выполнения операций может возникнуть из-за того, что современные процессоры обрабатывают и передают на выполнение инструкции в измененном порядке для того, чтобы оптимизировать использование конвейеров. Это может привести к тому, что инструкции чтения переменных b и a выполнятся не в том порядке. Функции rmb () и wmb () соответствуют инструкциям, которые заставляют процессор выполнить все незаконченные операции чтения и записи перед тем, как продолжить работу далее.

Рассмотрим простой пример случая, когда можно использовать функцию read_ barrier_depends () вместо функции rmb(). В этом примере изначально переменная a равна 1, b - 2, а p - &b.

Поток 1	**Поток 2**
`a=3;`	
`mb();`	
`p=&a;`	`pp=p;`
	`read_barrier_depends();`
	`b=*pp;`

Снова без использования барьеров памяти появляется возможность того, что переменной b будет присвоено значение *pp до того, как переменной pp будет присвоено значение переменной p. Функция read_barrier_depends () обеспечивает достаточный барьер, так как считывание значения *pp зависит от считывания переменной p. Здесь также будет достаточно использовать функцию rmb (), но поскольку операции чтения зависимы между собой, то можно использовать потенциально более быструю функцию read_barrier_depends (). Заметим, что в обоих случаях требуется использовать функцию mb () для того, чтобы гарантировать необходимый порядок выполнения операций чтения-записи в потоке 1.

Макросы smp_rmb () , smp_wmb (), smp_mb() и smpread_barrier_depends () позволяют выполнить полезную оптимизацию. Для SMP-ядра они определены как обычные барьеры памяти, а для ядра, рассчитанного на однопроцессорную машину, — только как барьер компилятора. Эти SMP-варианты барьеров можно использовать, когда ограничения на порядок выполнения операций являются специфичными для SMP-систем.

Функция barrier () предотвращает возможность оптимизации компилятором операций считывания и записи данных, если эти операции находятся по разные стороны от вызова данной функции (т.е. запрещает изменение порядка операций). Компилятор не изменяет порядок операций записи и считывания в случаях, когда это может повлиять на правильность выполнения кода, написанного на языке С, или на существующие зависимости между данными. Однако у компилятора нет информации о событиях, которые могут произойти вне текущего контекста. Например, компилятор не может иметь информацию о прерываниях, в контексте которых может выполняться считывание данных, которые в данный момент записываются. Например, по этой причине может оказаться необходимым гарантировать, что операция записи выполнится перед операцией считывания. Указанные ранее барьеры памяти работают и как барьеры компилятора, но барьер компилятора значительно быстрее, чем барьер памяти (практически не влияет на производительность). Использование барьера компилятора на практике является опциональным, так как он просто предотвращает *возможность* того, что компилятор что-либо изменит.

В табл. 9.10 приведен полный список функций установки барьеров памяти и компилятора, которые доступны для разных аппаратных платформ, поддерживаемых ядром Linux.

Следует заметить, что эффекты установки барьеров могут быть разными для разных аппаратных платформ. Например, если машина не изменяет порядок операций записи (как в случае набора микросхем Intel x86), то функция wmb () не выполняет никаких действий. Можно использовать соответствующий барьер памяти для самой плохой ситуации (т.е. для процессора с самым плохим порядком выполнения), и ваш код будет скомпилирован оптимально для вашей аппаратной платформы.

Таблица 9.10. Средства установки барьеров компилятора и памяти

Барьер	Описание
`rmb()`	Предотвращает изменение порядка выполнения операций чтения данных из памяти при переходе через барьер
`read_barrier_depends()`	Предотвращает изменение порядка выполнения операций чтения данных из памяти при переходе через барьер, но только для операций чтения, которые зависимы друг от друга
`wtmb()`	Предотвращает изменение порядка выполнения операций записи данных в память при переходе через барьер
`mb()`	Предотвращает изменение порядка выполнения операций чтения и записи данных при переходе через барьер
`smp_rmb()`	Для SMP-ядер эквивалентно функции `rmb()` , а для ядер, рассчитанных на однопроцессорные машины, эквивалентно функции `barrier()`
`smp_read_barrier_depends()`	Для SMP-ядер эквивалентно функции `read_barrier_depends()`. а для ядер, рассчитанных на однопроцессорные машины, эквивалентно функции `barrier()`
`srap_wmb()`	Для SMP-ядер эквивалентно функции `wmb()`, а для ядер, рассчитанных на однопроцессорные машины, эквивалентно функции `barrier()`
`smp_mb()`	Для SMP-ядер эквивалентно функции `mb()`, а для ядер, рассчитанных на однопроцессорные машины, эквивалентно функции `barrier()`
`barrier()`	Предотвращает оптимизации компилятора по чтению и записи данных при переходе через барьер

10

Таймеры и управление временем

Отслеживание хода времени очень важно для ядра. Большое количество функций, которые выполняет ядро, управляются временем (time driven), в отличие от тех функций, которые выполянются по событиям[1] (event driven). Некоторые из этих функций выполняются периодически, как, например, балансировка очередей выполнения планировщика или обновление содержимого экрана. Такие функции вызываются в соответствии с постоянным планом, например 100 раз в секунду. Другие функции, такие как отложенные дисковые операции ввода-выпода, ядро планирует на выполнение в некоторый относительный момент времени в будущем. Например, ядро может запланировать работу на выполнение в момент времени, который наступит позже текущего на 500 миллисекунд. Наконец, ядро должно вычислять время работы системы (uptime), а также текущую дату и время.

Следует обратить внимание на разницу между относительным и абсолютным временем. Планирование выполнения некоторой работы через 5 секунд в будущем не требует учета *абсолютного* времени, а только *относительного* (например, через пять секунд от текущего момента времени). В рассмотренной ситуации расчет текущей даты и времени требует от ядра не только учета хода времени, но и абсолютного измерения времени. Обе концепции являются важными для управления временем.

Также следует обратить внимание па отличия между событиями, которые возникают периодически, и событиями, которые ядро планирует на выполнение в некоторый фиксированный момент времени в будущем. События, которые возникают периодически, скажем каждые 10 миллисекунд, управляются *системным, таймером*. Системный таймер — это программируемое аппаратное устройство, которое генерирует аппаратное прерывание с фиксированной частотой. Обработчик этого прерывания, который называется *прерыванием таймера (timer interrupt),* обновляет значение системного времени и выполняет периодические действия. Системный таймер и его прерывание являются важными для работы операцтонной системы Linux, и в текущей главе им уделяется главное внимание.

---

[1] Если быть точными, то функции, которые управляются временем, также управляются и событиями. В этом случае событие соответствует ходу времени. В этой главе будут рассмотрены, в основном, события, управляемыми временем, так как они встречаются очень часто и являются важными для ядра.

Кроме того, в этой главе будут рассмотрены *динамические таймеры (dynamic timers)* — средства, позволяющие планировать события, которые выполняются один раз, после того как истек некоторый интервал времени. Например, драйвер накопителя на гибких магнитных дисках использует таймер, чтобы остановить двигатель дисковода, если дисковод неактивен в течение некоторого периода времени. В ядре можно динамически создавать и ликвидировать таймеры. В данной главе рассказывается о реализации динамических таймеров, а также об интерфейсе, который доступен для использования в программном коде.

Информация о времени в ядре

Концепция времени для компьютера является несколько неопределенной. В действительности, для того чтобы получать информацию о времени и управлять системным временем, ядро должно взаимодействовать с системным аппаратным обеспечением. Аппаратное обеспечение предоставляет системный таймер, который используется ядром для измерения времени. Системный таймер работает от электронного эталона времени, такого как цифровые электронные часы или тактовый генератор процессора. Интервал времени системного таймера периодически истекает (еще говорят таймер *срабатывает — hitting, popping)* с определенной запрограммированной частотой. Эта частота называется *частотой импульсов таймера, (tick rate)*. Когда срабатывает системный таймер, он генерирует прерывание, которое ядро обрабатывает с помощью специального обработчика прерывания.

Так как в ядре есть информация о запрограммированной частоте следования импульсов таймера, ядро может вычислить интервал времени между двумя успешными прерываниями таймера. Этот интервал называется *временной отметкой* или *импульсом таймера (tick)* и в секундах равен *единице, деленной на частоту импульсов.* Как будет показано дальше, именно таким способом ядро отслеживает абсолютное время (wall time) и время работы системы (uptime). Абсолютное время — это фактическое время дня, которое наиболее важно для пользовательских приложений. Ядро отслеживает это время просто потому, что оно контролирует прерывание таймера. В ядре есть семейство системных вызовов, которое позволяет пользовательским приложениям получать информацию о дате и времени дня. Это необходимо, так как многие программы должны иметь информацию о ходе времени. Разница между двумя значениями времени работы системы — "сейчас" и "позже" — это простой способ измерения относительности событий.

Прерывание таймера очень важно для управления работой всей операционной системы. Большое количество функций ядра действуют и завершаются в соответствии с ходом времени. Следующие действия периодически выполняются системным таймером.

- Обновление значения времени работы системы (uptime).
- Обновление значения абсолютного времени (time of day).
- Для SMP-систем выполняется проверка балансировки очередей выполнения планировщика, и если они не сбалансированы, то их необходимо сбалансировать (как было рассказано в главе 4, "Планирование выполнения процессов").

- Проверка, не израсходовал ли текущий процесс свой квант времени, и если израсходовал, то выполнятся планирование выполнения нового процесса (как это было рассказано в главе 4).
- Выполнение обработчиков всех динамических таймеров, для которых истек период времени.
- Обновление статистики по использованию процессорного времени и других ресурсов.

Некоторые из этих действий выполняются при каждом прерывании таймера, т.е. эта работа выполняется с частотой системного таймера. Другие действия также выполняются периодически, но только через каждые n прерываний системного таймера. Иными словами, эти функции выполняются с частотой, которая равна некоторой доле частоты системного таймера. В разделе "Обработчик прерываний таймера" будет рассмотрена сама функция обработки прерываний системного таймера.

Частота импульсов таймера: HZ

Частота системного таймера (частота импульсов, tick rate) программируется при загрузке системы на основании параметра ядра HZ, который определен с помощью директивы препроцессора. Значение параметра HZ отличается для различных поддерживаемых аппаратных платформ. На самом деле, для некоторых аппаратных платформ значение параметра HZ отличается даже для разных типов машин.

Данный параметр ядра определен в файле <asm/param.h>. Частота системного таймера равна значению параметра HZ, период таймера равен 1/HZ. Например, в файле include/asm-i386/param.h для аппаратной платформы i386 этот параметр определен следующим образом.

```
#define HZ 1000 /* internal kernel time frequency */
```

Поэтому для аппаратной платформы i368 прерывание таймера генерируется с частотой 1000 Гц, т.е. 1000 раз в секунду (каждую тысячную долю секунды или одну миллисекунду). Для большинства других аппаратных платформ значение частоты системного таймера равно 100 Гц. В табл. 10.1 приведен полный список всех поддерживаемых аппаратных платформ и определенных для них значений частоты системного таймера.

При написании кода ядра нельзя считать, что параметр HZ имеет определенное заданное значение. В наши дни это уже не такая часто встречающаяся ошибка, так как поддерживается много различных аппаратных платформ с разными частотами системного таймера. Раньше аппаратная платформа Alpha была единственной, для которой частота системного таймера отличалась от 100 Гц, и часто можно было встретить код, в котором жестко было прописано значение 100 там, где нужно использовать параметр HZ. Примеры использования параметра HZ в коде ядра будут приведены ниже.

Частота системного таймера достаточно важна. Как будет видно, обработчик прерыпания таймера выполняет много работы. Вся информация о времени в ядре получается из периодичности системного таймера. Весь компромисс состоит только в том, чтобы выбрать правильное значение данного параметра исходя из взаимоотношения между разными факторами.

Таблица 10.1. Значение частоты системного таймера

Аппаратная платформа	Частота (в герцах)
alpha	1024
arm	100
cris	100
h8300	100
i386	1000
ia64	32 или 1024[2]
m68k	100
m68knommu	50, 100 или 1000
mips	100
mips64	100
parisc	100 или 1000
ppc	100
ppc64	1000
s390	100
sti	100
spare	100
sparc64	100
um	100
v850	24, 100 или 122
x86-64	1000

Идеальное значение параметра HZ

Для аппаратной платформы i386, начиная с самых первых версий операционной системы Linux, значение частоты системного таймера было равно 100 Гц. Однако во время разработки ядер серии 2.5 это значение было увеличено до 1000 Гц, что (как всегда бывает в подобных ситуациях) вызвало споры. Так как в системе очень многое зависит от прерывания таймера, то изменение значения частоты системного таймера должно оказывать сильное влияние на систему. Конечно, как в случае больших, так и в случае маленьких значений параметра HZ есть свои положительные и отрицательные стороны.

Увеличение значения частоты системного таймера означает, что обработчик прерываний таймера выполняется более часто. Следовательно, вся работа, которую он делает, также выполняется более часто. Это позволяет получить следующие преимущества.

- Прерывание таймера имеет большую разрешающую способность по времени, и следовательно, все событии, которые выполняются во времени, также имеют большую разрешающую способность.
- Увеличивается точность выполнения событий во времени.

[2] Эмулятор платформы IA-64 имеет частоту 32 Гц. Настоящая машина платформы IA-64 имеет частоту 1024 Гц.

Разрешающая способность увеличивается во столько же раз, во сколько раз возрастает частота импульсов. Например, гранулярность таймеров при частоте импульсов 100 Гц равна 10 миллисекунд. Другими словами, все периодические события выполняются прерыванием таймера, которое генерируется с предельной точностью по времени, равной 10 миллисекунд, и большая точность[3] не гарантируется. При частоте, равной 1000 Гц, разрешающая способность равна 1 миллисекунде, т.е. в 10 раз выше. Хотя ядро позволяет создавать таймеры с временным разрешением, равным 1 миллисекунде, однако при частоте системного таймера в 100 Гц нет возможности гарантированно получить временной интервал, короче 10 миллисекунд.

Точность измерения времени также возрастает аналогичным образом. Допустим, что таймеры ядра запускаются в случайные моменты времени, тогда в среднем таймеры будут срабатывать с точностью по времени до половины периода прерывания таймера, потому что период времени таймера может закончиться в любой момент, а обработчик таймера может выполниться, только когда генерируется прерывание таймера. Например, при частоте 100 Гц описанные события в среднем будут возникать с точностью +/- 5 миллисекунд от желаемого момента времени. Поэтому ошибка измерения в среднем составит 5 миллисекунд. При частоте 1000 Гц ошибка измерения в среднем уменьшается до 0.5 миллисекунд — получает десятикратное улучшение.

Более высокое разрешение и большая точность обеспечивают следующие преимущества.

- Таймеры ядра выполняются с большим разрешением и с лучшей точностью (это позволяет получить много разных улучшений, некоторые из которых описаны дальше).
- Системные вызовы, такие как `poll()` и `select()`, которые позволяют при желании использовать время ожидания (timeout) в качестве параметра, выполняются с большей точностью.
- Измерения, такие как учет использования ресурсов или измерения времени работы системы, выполняются с большей точностью.
- Вытеснение процессов выполняется более правильно.

Некоторые из наиболее заметных улучшений производительности — это улучшения точности измерения периодов времени ожидания при выполнении системных вызовов `poll()` и `select()`. Это улучшение может быть достаточно большим. Прикладная программа, которая интенсивно использует эти системные вызовы, может тратить достаточно много времени, ожидая на прерывания таймера, хотя в действительности интервал времени ожидания уже истек. Следует вспомнить, что средняя ошибка измерения времени (т.е. потенциально зря потраченное время) равна половине периода прерывания таймера.

Еще одно преимущество более высокой частоты следования импульсов таймера— это более правильное вытеснение процессов, что проявляется в уменьшении задержки за счет планирования выполнения процессов. Вспомним из материала главы 4, что прерывание таймера ответственно за уменьшение кванта времени вы-

[3] Здесь имеется в виду не точность измерения, а точность в вычислительном плане. Точность измерения (в общенаучном смысле) — это статистическая мера повторяемости результата. В вычислительном (компьютерном) смысле точность— это количество значащих цифр, которые используются для представления того или другого значения.

полняющегося процесса. Когда это значение уменьшается до нуля, устанавливается флаг need_resched, и ядро активизирует планировщик как только появляется такая возможность. Теперь рассмотрим ситуацию, когда процесс в данный момент выполняется и у него остался квант времени, равный 2 миллисекундам. Это означает, что через 2 миллисекунды планировщик должен вытеснить этот процесс и запустить на выполнение другой процесс. К сожалению, это событие не может произойти до того момента, пока не будет сгенерировано следующее прерывание таймера. В самом худшем случае следующее прерывание таймера может возникнуть через 1/HZ секунд! В случае, когда параметр HZ=100, процесс может получить порядка 10 лишних миллисекунд. Конечно, в конце концов все будет сбалансировано и равнодоступность ресурсов не нарушится, потому что все задания планируются с одинаковыми ошибками, и проблема состоит не в этом. Проблемы возникают из-за латентности, которую вносят задержки вытеснения процессов. Если задание, которое планируется на выполнение, должно выполнить какие-нибудь чувствительные ко времени действия, как, например, заполнить буфер аудиоустройства, то задержка не допустима. Увеличение частоты до 1000 Гц уменьшает задержку планировщика в худшем случае до 1 миллисекунды, а в среднем — до 0.5 миллисекунды.

Должна, однако, существовать и обратная сторона увеличения частоты системного таймера, иначе она была бы с самого начала равна 1000 Гц (или даже больше). На самом деле существует одна большая проблема. Более высокая частота вызывает более частые прерывания таймера, что означает большие накладные затраты. Чем выше частота, тем больше времени процессор должен тратить на выполнение прерываний таймера. Это приводит не только к тому, что другим задачам отводится меньше процессорного времени, но и к периодическому трешингу (trashing) кэша процессора (т.е. кэш заполняется данными, которые не используются процессором). Проблема, связанная с накладными расходами, вызывает споры. Ясно, что переход от значения HZ=100 до значения HZ=1000 в 10 раз увеличивает накладные затраты, связанные с прерываниями таймера. Однако от какого реального значения накладных затрат следует отталкиваться? Если "ничего" умножить на 10, то получится тоже "ничего". Решающее соглашение состоит в том, что по крайней мере для современных систем, значение параметра HZ=1000 не приводит к недопустимым накладным затратам. Тем не менее для ядер серии 2.6 существует возможность скомпилировать ядро с другим значением параметра HZ[4].

Возможна ли операционная система без периодических отметок времени

Может возникнуть вопрос, всегда ли для функционирования операционной системы необходимо использовать фиксированное прерывание таймера? Можно ли создать операционную систему, в которой не используются периодические отметки времени? Да, можно, *но результат будет не очень привлекательным.*

Нет строгой необходимости в использовании прерывания таймера, которое возникает с фиксированной частотой. Вместо этого ядро может использовать динамически программируемый таймер для каждого ожидающего события. Такое решение сразу же приведет к дополнительным накладным затратам процессорного времени в связи с обработкой событий таймера, поэтому лучшим решением будет использовать один таймер и программировать его так, чтобы он срабатывал тогда, когда должно наступить ближайшее событие.

[4] В связи с ограничениями аппаратной платформы и протокола NTP, значение переменной HZ не может быть произвольным. Для платформы x86 значения 100, 500 и 1000 работают хорошо.

Когда обработчик таймера сработает, создается новый таймер для следующего события и так повторяется постоянно. При таком подходе не требуется периодическое прерывание таймера и нет необходимости в параметре HZ.

Однако при указанном подходе необходимо решить две проблемы. Первая проблема — это как в таком случае реализовать концепцию периодических отметок времени, хотя бы для того, чтобы ядро могло отслеживать относительные интервалы времени. Эту проблему решить не сложно. Вторая проблема — это как избежать накладных затрат, связанных с управлением динамическими таймерами, даже при наличии оптимизации. Данную проблему решить сложнее. Накладные расходы и сложность реализации получаются настолько высокими, что в операционной системе Linux такой подход решили не использовать. Тем не менее так пробовали делать, и результаты получаются интересными. Если интересно, то можно поискать в Интернет-архивах.

Переменная jiffies

Глобальная переменная jiffies содержит количество импульсов системного таймера, которые были получены со времени загрузки системы. При загрузке ядро устанавливает значение этого параметра в нуль и он увеличивается на единицу при каждом прерывании системного таймера. Так как в секунду возникает HZ прерываний системного таймера, то за секунду значение переменной jiffies увеличивается на HZ. Время работы системы (uptime) поэтому равно jiffies/HZ секунд.

Этимология слова jiffy

Происхождение слова *jiffy* (миг, мгновение) точно неизвестно. Считается, что фразы типа *"in a jiffy" (в одно мгновение)* появились в Англии в восемнадцатом веке. В быту термин *jiffy [миг)* означает неопределенный, но очень короткий промежуток времени.

В научных приложениях слово *jiffy* используется для обозначения различных интервалов времени (обычно порядка 10 ms). В физике это слово иногда используется для указания интервала времени, который требуется свету, чтобы пройти определенное расстояние (обычно, фут, сантиметр, или расстояние, равное размеру нуклона).

В вычислительной технике термин *jiffy* — это обычно интервал времени между двумя соседними импульсами системного таймера, которые были успешно обработаны. В электричестве *jiffy* — период переменного тока. В США *jiffy*— это 1/60 секунды.

В приложении к операционным системам, в частности к Unix, *jiffy*— это интервал времени между двумя соседними успешно обработанными импульсами системного таймера. Исторически это значение равно 100 ms. Как уже было показано, интервал времени jiffy в операционной системе Linux может иметь разные значения.

Переменная jiffies определена в файле <linux/jiffies.h> следующим образом.

```
extern unsigned long volatile jiffies;
```

Определение этой переменной достаточно специфичное, и оно будет рассмотрено более подробно в следующем разделе. Сейчас давайте рассмотрим пример кода ядра. Пересчет из секунд в значение переменной jiffies можно выполнить следующим образом.

```
(секунды * HZ)
```

Отсюда следует, что преобразование из значения переменной jiffies в секунды можно выполнить, как показано ниже.

```
(jiffies / HZ)
```

Первый вариант встречается более часто. Например, часто необходимо установить значение некоторого момента времени в будущем.

```
unsigned long time_starnp = jiffies;     /* сейчас */
unsigned long next_tick = jiffies + 1;  /* через один импульс таймера
                                           от текущего момента */
unsigned long later = jiffies + 5*HZ;   /* через пять секунд от текущего
                                           момента */
```

Последний пример обычно используется при взаимодействии с пространством пользователя, так как в самом ядре редко используется абсолютное время.

Заметим, что переменная jiffies имеет тип unsigned long и использовать какой-либо другой тип будет неправильным.

Внутреннее представление переменной jiffies

Переменная jiffies исторически всегда представлялась с помощью типа unsigned long и, следовательно, имеет длину 32 бит для 32-разрядных аппаратных платформ и 64 бит для 64-разрядных. В случае 32-разрядного значения переменной jiffies и частоты появления временных отметок 100 раз в секунду, переполнение этой переменной будет происходить примерно каждые 497 дней, что является вполне возможным событием. Увеличение значения параметра HZ до 1000 уменьшает период переполнения до 47.9 дней! В случае 64-разрядного типа переменной jiffies, переполнение этой переменной невозможно за время существования чего-либо при любых возможных значениях параметра HZ для любой аппаратной платформы.

Из соображений производительности и по историческим причинам — в основном, для совместимости с уже существующим кодом ядра — разработчики ядра предпочли оставить тип переменной jiffies — unsigned long. Для решения проблемы пришлось немного подумать и применить возможности компоновщика.

Как уже говорилось, переменная jiffies определяется в следующем виде и имеет тип unsigned long.

```
extern unsigned long volatile jiffies;
```

Вторая переменная определяется в файле <linux/jiffies.h> в следующем виде.

```
extern u64 jiffies_64;
```

Директивы компоновщика ld (1), которые используются для сборки главного образа ядра (для аппаратной платформы x86 описаны в файле arch/i386/kernel/vmlinux.lds.S), указывают компоновщику, что переменную jiffies необходимо совместить с началом переменной jiffies_64.

```
Jiffies = jiffies_64;
```

Следовательно, переменная jiffies — это просто 32 младших разряда полной 64-разрядной переменной jiffies_64. Так как в большинстве случаев переменная

jiffies используется для измерения промежутков времени, то для большей части кода существенными являются только младшие 32 бит.

В случае применения 64-разрядного значения, переполнение не может возникнуть за время существования чего-либо. В следующем разделе будут рассмотрены проблемы, связанные с переполнением (хотя переполнение счетчика импульсов системного таймера и не желательно, но это вполне нормальное и ожидаемое событие). Код, который используется для управления ходом времени, использует все 64 бит, и это предотвращает возможность переполнения 64-разрядного значения. На рис. 10.1 показана структура переменных jiffies и jiffies_64.

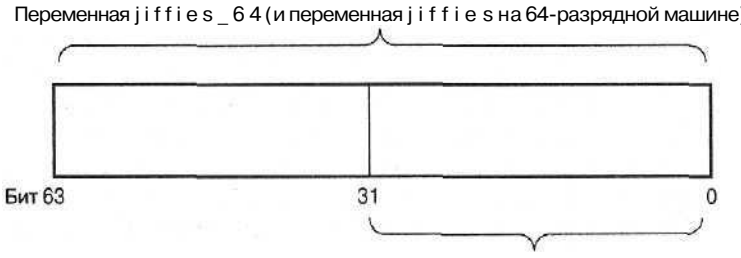


Рис. 10.1. Структура переменных jiffies и jiffies_64

Код, который использует переменную jiffies, просто получает доступ к тридцати двум младшим битам переменной jiffies_64. Функция get_jiffies_64 () может быть использована для получения полного 64-разрядного значения[5]. Такая необходимость возникает редко, следовательно большая часть кода просто продолжает считывать младшие 32 разряда непосредственно из переменной jiffies.

На 64-разрядных аппаратных платформах переменные jiffies_64 и jiffies просто совпадают. Код может либо непосредственно считывать значение переменной jiffies, либо использовать функцию get_jiffies_64 (), так как оба этих способа позволяют получить аналогичный эффект.

Переполнение переменной jiffies

Переменная jiffies, так же как и любое целое число языка программирования C, после достижения максимально возможного значения переполняется. Для 32-разрядного беззнакового целого числа максимальное значение равно 2^{32}-1. Поэтому перед тем как счетчик импульсов системного таймера переполнится, должно прийти 4294967295 импульсов таймера. Если значение счетчика равно этому значению и счетчик увеличивается на 1, то значение счетчика становится равным нулю.

Рассмотрим пример переполнения.

```
unsigned long timeout = jiffies + HZ/2; /* значение лимита времени
                                           равно 0.5 с */
```

[5] Необходима специальная функция, так как на 32-разрядных аппаратных платформах нельзя атомарно обращаться к двум машинным словам 64-разрядного значения, Специальная функция, перед тем как считать значение, блокирует счетчик импульсов системного таймера с помощью блокировки *xtime_lock*.

```
/* выполним некоторые действия и проверим, не слишком ли это много
   заняло времени . . . */
if (timeout < jiffies) {
        /* мы превысили лимит времени — это ошибка ... */
} else {
        /* мы не превысили лимит времени — это хорошо ... */
}
```

Назначение этого участка кода — установить лимит времени до наступления некоторого события в будущем, а точнее полсекунды от текущего момента. Код может продолжить выполнение некоторой работы — возможно, записать некоторые данные в аппаратное устройство и ожидать ответа. После выполнения, если весь процесс превысил лимит установленного времени, код соответственным образом обрабатывает ошибку.

В данном примере может возникнуть несколько потенциальных проблем, связанных с переполнением. Рассмотрим одну из них. Что произойдет, если переменная jiffies переполнится и снова начнет увеличиваться с нуля после того, как ей было присвоено значение переменной timeout? При этом условие гарантированно не выполнится, так как значение переменной jiffies будет меньше, чем значение переменной timeout, хотя логически оно должно быть больше. По идее значение переменной jiffies должно быть огромным числом, всегда большим значения переменной timeout. Так как эта переменная переполнилась, то теперь ее значение стало очень маленьким числом, которое, возможно, отличается от нуля на несколько импульсов таймера. Из-за переполнения результат выполнения оператора if меняется на противоположный!

К счастью, ядро предоставляет четыре макроса для сравнения двух значений счетчика импульсов таймера, которые корректно обрабатывают переполнение счетчиков. Они определены в файле <linux/jiffies.h> следующим образом.

```
#define time_after(unknown, known)  ((long)(known) - (long)(unknown) < 0)
#define time_before(unknown, known)  ((long) (unknown) - (long)(known) < 0)
#define time_after_eq(unknown, known)  ((long) (unknown) - (long)(known) >= 0)
#define
  time_before_eq(unknown, known) ((long) (known) - (long)(unknown) >= 0)
```

Параметр unknown — это обычно значение переменной jiffies, а параметр known — значение, с которым его необходимо сравнить.

Макрос time_after (unknown, known) возвращает значение true, если момент времени unknown происходит после момента времени known, в противном случае возвращается значение false. Макрос time_before (unknown, known) возвращает значение true, если момент времени unknown происходит раньше, чем момент времени known, в противном случае возвращается значение false. Последние два макроса работают аналогично первым двум, за исключением того, что возвращается значение "истинно", если оба параметра равны друг другу.

Версия кода из предыдущего примера, которая предотвращает ошибки, связанные с переполнением, будет выглядеть следующим образом.

```
unsigned long timeout = jiffies + HZ/2; /* значение лимита времени
                                           равно 0.5 с */
```

```
/* выполним некоторые действия и проверим, не слишком ли это много
   заняло времени ... */
if (time_after(jiffies, timeout)) {
        /* мы превысили лимит времени — это ошибка ... */
} else {
        /* мы не превысили лимит времени — это хорошо ... */
}
```

Если любопытно, каким образом эти макросы предотвращают ошибки, связанные с переполнением, то попробуйте подставить различные значения параметров. А затем представьте, что один из параметров переполнился, и посмотрите, что при этом произойдет.

Пространство пользователя и параметр HZ

Раньше изменение параметра HZ приводило к аномалиям в пользовательских программах. Это происходило потому, что значения параметров, связанных со временем, экспортировались в пространство пользователя в единицах, равных количеству импульсов системного таймера в секунду. Так как такой интерфейс использовался давно, то в пользовательских приложениях считалось, что параметр HZ имеет определенное конкретное значение. Следовательно, при изменении значения параметра HZ изменялись значения, которые экспортируются в пространство пользователя, в одинаковое число раз. Информация о том, во сколько раз изменились значения, в пространство пользователя не передавалась! Полученное от ядра значение времени работы системы могло интерпретироваться как 20 часов, хотя на самом деле оно равнялось только двум часам.

Чтобы исправить это, код ядра должен нормировать все значения переменной `jiffies`, которые экспортируются в пространство пользователя. Нормировка реализуется путем определения константы USER_HZ, равной значению параметра HZ, которое *ожидается* в пространстве пользователя. Та как для аппаратной платформы x86 значение параметра HZ исторически равно 100, то значение константы USER_HZ=100. Макрос `jiffies_to_clock_t()` используется для нормировки значения счетчика импульсов системного таймера, выраженного в единицах HZ, в значение счетчика импульсов, выраженное в единицах USER_HZ. Используемый макрос зависит от того, кратны ли значения параметров HZ и USER_HZ один другому. Если кратны, то этот макрос имеет следующий очень простой вид.

```
#define jiffies_to_clock_t(x) ((x) / (HZ / USER_HZ))
```

Если не кратны, то используется более сложный алгоритм.

Функция `jiffies_64_to_clock_t ()` используется для конвертирования 64-битового значения переменной `jiffies` из единиц HZ в единицы USER_HZ.

Эти функции используются везде, где значения данных, выраженных в единицах числа импульсов системного таймера в секунду, должны экспортироваться в пространство пользователя, как в следующем примере.

```
unsigned long start = jiffies;
unsigned long total_time;
/* выполнить некоторую работу ... */
total_time = jiffies - start;
printk("Это заняло %lu импульсов таймера\n", jiffies_to_clock_t(total_time));
```

В пространстве пользователя передаваемое значение должно быть таким, каким оно было бы, если бы выполнялось равенство HZ=USER_HZ. Если это равенство не справедливо, то макрос выполнит нужную нормировку и все будут счастливы. Конечно, этот пример несколько нелогичный: больше смысла имело бы печатать значение времени не в импульсах системного таймера, а в секундах следующим образом.

```
printk("Это заняло %lu секунд\n", total time / HZ);
```

Аппаратные часы и таймеры

Различные аппаратные платформы предоставляют два аппаратных устройства, которые помогают вести учет времени, — это системный таймер, о котором уже было рассказано, и часы реального времени. Реализация и поведение этих устройств могут быть различными для машин разного типа, но общее их назначение и принципы работы с ними почти всегда одинаковы.

Часы реального времени

Часы реального времени (real-time clock, RTC) представляют собой энергонезависимое устройство для сохранения системного времени. Устройство RTC продолжает отслеживать время, даже когда система отключена, благодаря небольшой батарее, которая обычно находится на системной плате. Для аппаратной платформы PC устройство RTC интегрironано в КМОП-микросхему BIOS. При этом используется общая батарея и для работы устройства RTC и для сохранения установок BIOS.

При загрузке ядро считывает информацию из устройства RTC и использует ее для инициализации значения абсолютного времени, которое хранится в переменной xtime. Обычно ядро не считывает это значение снова, однако для некоторых поддерживаемых аппаратных платформ, таких как x86, значение абсолютного времени периодически записывается в устройство RTC. Тем не менее, часы реального времени важны в первую очередь на этапе загрузки системы, когда инициализируется переменная xtime.

Системный таймер

Системный таймер играет более значительную роль для отслеживания хода времени ядром. Независимо от аппаратной платформы, идея, которая лежит в основе системного таймера, одна и та же — это обеспечение механизма управления прерываниями, которые возникают периодически с постоянной частотой. Для некоторых аппаратных платформ это реализуется с помощью электронных часов, которые генерируют колебания с программируемой частотой. В других аппаратных платформах используется декрементный счетчик (decrementer), куда можно записать некоторое начальное значение, которое будет периодически, с фиксированной частотой, уменьшаться на единицу, пока значение счетчика не станет равным нулю. Когда значение счетчика становится равным нулю, генерируется прерывание. В любом случае эффект получается один и тот же.

Для аппаратной платформы x86 главный системный таймер — это программируемый интервальный таймер (programmable interval timer, PIT). Таймер PIT существует

на всех машинах платформы PC. Со времен операционной системы DOS он используется для управления прерываниями. Ядро программирует таймер PIT при загрузке, для того чтобы периодически генерировать прерывание номер нуль с частотой HZ. Этот таймер— простое устройство с ограниченными возможностями, но, тем не менее, хорошо выполняющее свою работу. Другие эталоны времени для аппаратной платформы x86 включают таймер APIC (Advanced Programmable Interrupt Controller, расширенный программируемый контроллер прерываний) и счетчик отметок времени (TSC, Time Stamp Counter).

Обработчик прерываний таймера

Теперь, когда мы разобрались, что такое jiffies и HZ, а также какова роль системного таймера, рассмотрим реализацию обработчика прерываний системного таймера. Обработчик прерываний таймера разбит на две части: часть, зависимую от аппаратной платформы, и независимую часть.

Подпрограмма, которая зависит от аппаратной платформы, регистрируется в качестве обработчика прерываний системного таймера и выполняется, когда срабатывает системный таймер. Конкретная работа, конечно, зависит от аппаратной платформы, но большинство обработчиков выполняют следующие действия.

- Захватывается блокировка xtime_lock, которая защищает доступ к переменной jiffies_64 и значению текущего времени— переменной xtirne.
- Считывается или сбрасывается состояние системного таймера, если это необходимо.
- Периодически записывается новое значение абсолютного времени в часы реального времени.
- Вызывается аппаратно-независимая подпрограмма таймера do_timer ().

Аппаратно-независимая функция do_timer () выполняет значительно больше действий.

- Увеличивается значение переменной jiffies_64 на единицу (это безопасная операция даже для 32-разрядных аппаратных платформ, так как блокировка xtime_lock была захвачена раньше).
- Обновляется статистка использования системных ресурсов, таких как затраченное процессорное время в режиме пользователя и в режиме ядра, для процесса, который в данный момент выполняется.
- Выполняются обработчики динамических таймеров, для которых истек период времени ожидания (это будет рассмотрено в следующем разделе).
- Вызывается функция scheduler_tick () , как было рассмотрено в главе 4.
- Обновляется значение абсолютного времени, которое хранится в переменной xtime.
- Вычисляются значения печально известной средней загруженности системы (load average).

Сама по себе подпрограмма очень проста, так как большинство рассмотренных действий выполняются другими функциями.

```
void do_timer(struct pt_regs *regs)
{
        jiffies_64++;
        update_process_times(user_mode(regs));
        update_times();
}
```

Макрос user_mode () просматривает состояние регистров процессора, regs, и возвращает значение 1, если прерывание таймера возникло в пространстве пользователя, и значение 0— если в пространстве ядра. Это позволяет функции update_process_times О учесть, что за время между предыдущим и данным импульсами системного таймера процесс выполнялся в режиме задачи или в режиме ядра.

```
void update_process_times(int user_tick)
{
        struct task_struct *p = current;
        int cpu = smp_processor_id();
        int system = user_tick ^ 1;

        update_one_process(p,user_tick,system,cpu);
        run_local_timers();
        scheduler_tick(user_tick,  system);
}
```

Функция update_process () собственно обновляет значения параметров времени выполнения процесса. Эта функция тщательно продумана. Следует обратить внимание, каким образом с помощью операции исключающее ИЛИ (XOR) достигается, что одна из переменных user_tick и system имеет значение, равное нулю, а другая— единице. Поэтому в функции update_one_process () можно просто прибавить необходимое значение к соответствующим счетчикам без использования оператора ветвления.

```
/*
* увеличиваем значения соответствующего счетчика импульсов таймера на единицу
*/
p->utime += user;
p->stime += system;
```

Необходимое значение увеличивается на 1, а другое остается без изменений. Легко заметить, что в таком случае предполагается, что за время импульса системного таймера процесс выполнялся *в том же* режиме, в котором он выполняется во время прихода прерывания. На самом деле процесс мог несколько раз переходить в режим задачи и в режим ядра за последний период системного таймера. Кроме того, текущий процесс может оказаться не единственным процессом, который выполнялся за последний период системного таймера. К сожалению, без применения более сложной системы учета, такой способ является лучшим из всех тех, которые предоставляет ядро. Это также одна из причин увеличения частоты системного таймера.

Далее функция run_local_timers () помечает отложенные прерывания, как готовые к выполнению (см. главу 7, "Обработка нижних половин и отложенные дей-

ствия"), для выполнения всех таймеров, для которых закончился период времени ожидания. Таймеры будут рассмотрены ниже, в разделе "Таймеры".

Наконец, функция schedule_tick () уменьшает значение кванта времени для текущего выполняющегося процесса и устанавливает флаг need_resched при необходимости. Для SMP-машин в этой функции также при необходимости выполняется балансировка очередей выполнения. Все это обсуждалось в главе 4.

После возврата из функции update_process_times () вызывается функция update_times (), которая обновляет значение абсолютного времени.

```
void update_times(void)
{
        unsigned long ticks;

        ticks = jiffies - wall_jiffies;
        if (ticks) {
             wall_jiffies += ticks;
             update_wall_time(ticks);
        }
        last_time_offset = 0;
        calc_load(ticks);
}
```

Значение переменной ticks вычисляется как изменение количества импульсов системного таймера с момента последнего обновления абсолютного времени. В нормальной ситуации это значение, конечно, равно 1. В редких случаях прерывание таймера может быть пропущено, и в таком случае говорят, что импульсы таймера *потеряны*. Это может произойти, если прерывания запрещены в течение длительного времени. Такая ситуация не является нормальной и часто указывает на ошибку программного кода. Значение переменной wall_jiffies увеличивается на значение ticks, поэтому она равна значению переменной jiffies в момент самого последнего обновления абсолютного времени. Далее вызывается функция update_wall_time () для того, чтобы обновить значение переменной xtime, которая содержит значение абсолютного времени. Наконец вызывается функция calc_load () для того, чтобы обновить значение средней загруженности системы, после чего функция update_times () возвращает управление.

Функция do_timer () возвращается в аппаратао-зависимый обработчик прерывания, который выполняет все необходимые завершающие операции, освобождает блокировку xtirae_lock и в конце концов возвращает управление.

Всё это происходит каждые 1/HZ секунд, т.е. 1000 раз в секунду на машине типа PC.

Абсолютное время

Текущее значение абсолютного времени (time of day, wall time, время дня) определено в файле kernel/timer.c следующим образом.

```
struct timespec xtime;
```

Структура данных timespec определена в файле <linux/time.h> в следующем виде.

```
struct timespec {
        time_t tv_sec; /* seconds */
        long tv_nsec; /* nanoseconds */
        };
```

Поле xtime.tv_sec содержит количество секунд, которые прошли с 1 января 1970 года (UTC, Universal Coordinated Time, всеобщее скоординированное время). Указанная дата называется *epoch* (начало эпохи). В большинстве Unix-подобных операционных систем счет времени ведется с начала эпохи. В поле xtime.tv_nsec хранится количество наносекунд, которые прошли в последней секунде.

Чтение или запись переменной xtime требует захвата блокировки xtime_lock. Это блокировка — не обычная спин-блокировка, а *секвентная* блокировка, которая рассматривается в главе 9, "Средства синхронизации в ядре".

Для обновления значения переменной xtime необходимо захватить секвентную блокировку на запись следующим образом.

```
write_seqlock(&xtime_lock);

/* обновить значение переменной xtime ... */

write_sequnlock(&xtime_lock);
```

Считывание значения переменной xtime требует применения функций read_seqbegin() и read_seqretry () следующим образом.

```
do {
        unsigned long lost;
        seq = read_seqbegin(&xtime_lock);

        usec = timer->get_offset();
        lost = jiffies - wall_jiffies;
        if (lost)
              usec += lost * (1000000 / HZ);
        sec = xtime.tv_sec;
        usec += (xtime.tv_nsec / 1000);
} while (read_seqretry(&xtime_lock, seq));
```

Этот цикл повторяется до тех пор, пока не будет гарантии того, что во время считывания данных не было записи. Если во время выполнения цикла приходит прерывание таймера и переменная xtime обновляется во время выполнения цикла, возвращаемый номер последовательности будет неправильным и цикл повторится снова.

Главный пользовательский интерфейс для получения значения абсолютного времени — это системный вызов gettimeofday () , который реализован как функция sys_gettimeofday() следующим образом.

```
asmlinkage long sys_gettimeofday(struct timeval *tv, struct timezone *tz)
{
```

```
        if (likely(tv !=NULL)) {
              struct timeval_ktv;
              do_gettimeofday(&ktv);
                    if (copy_to_userftv, &ktv, sizeof(ktv))
                          return -EFAULT;
        }
        if (unlikely(tz !=NULL)) {
              if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
                    return -EFAULT;
        }
        return 0;
}
```

Если из пространства пользователя передано ненулевое значение параметра tv, то вызывается аппаратно-зависимая функция do_gettimeofday() . Эта функция главным образом выполняет цикл считывания переменной xtime, который был только что рассмотрен. Аналогично, если параметр tz не равен нулю, пользователю возвращается значение часового пояса (time zone), в котором находится операционная система. Этот параметр хранится в переменной sys_tz. Если при копировании в пространство пользователя значения абсолютного времени или часового пояса возникли ошибки, то функция возвращает значение -EFAULT. В случае успеха возвращается нулевое значение.

Ядро предоставляет системный вызов time()[6], однако системный вызов gettimeofday() полностью перекрывает его возможности. Библиотека функций языка С также предоставляет другие функции, связанные с абсолютным временем, такие как ftime() и ctirae().

Системный вызов settimeofday() позволяет установить абсолютное время в указанное значение. Для того чтобы его выполнить, процесс должен иметь возможность использования CAP_SYS_TIME.

Если не считать обновления переменной xtime, то ядро не так часто использует абсолютное время, как пространство пользователя. Одно важное исключение— это код файловых систем, который хранят в индексах файлов значения моментов времени доступа к файлам.

Таймеры

Таймеры (timers), или, как их еще иногда называют, *динамические таймеры,* или *таймеры ядра,* необходимы для управления ходом времени в ядре. Коду ядра часто необходимо откладывать выполнение некоторых функций на более позднее время. Здесь намеренно выбрано не очень четкое понятие *"позже".* Назначение механизма нижних половин — это не *задерживать* выполнение, а *не выполнять работу прямо сейчас.* В связи с этим необходим инструмент, который позволяет задержать выполнение работы на некоторый интервал времени. Если этот интервал времени не очень маленький, но и не очень большой, то решение проблемы — таймеры ядра.

[6] Для некоторых аппаратных платформ функция *sys_time ()* не реализована, а вместо этого она эмулируется библиотекой функций языка С на основании вызова *gettimeofday ().*

Таймеры очень легко использовать. Необходимо выполнить некоторые начальные действия, указать момент времени окончания ожидания, указать функцию, которая будет выполнена, когда закончится интервал времени ожидания, и активизировать таймер. Указанная функция будет выполнена, когда закончится интервал времени таймера. Таймеры *не являются* циклическими. Когда заканчивается интервал времени ожидания, таймер ликвидируется. Это одна из причин, почему таймеры называют *динамическими*[7]. Таймеры постоянно создаются и ликвидируются, на количество таймеров не существует ограничений. Использование таймеров очень популярно во всех частях ядра.

Использование таймеров

Таймеры представлены с помощью структур timer list, которая определена в файле <linux/timer.h> следующим образом.

```
struct tirner_list {
        struct list_head entry; /* таймеры хранятся в связанном списке */
        unsigned long expires;  /* время окончание срока ожидания в
                                   импульсах системного таймера (jiffies) */
        spinlock_t lock; /* блокировка для защиты данного таймера */
        void (*function) (unsigned long); /*функция-обработчик таймера*/
        unsigned long data; /* единственный аргумент обработчика */
        struct tvec_t_base_s *base; /*внутренние данные таймера, не трогать! */
};
```

К счастью, использование таймеров не требует глубокого понимания назначения полей этой структуры. На самом деле, крайне не рекомендуется использовать поля этой структуры не по назначению, чтобы сохранить совместимость с возможными будущими изменениями кода. Ядро предоставляет семейство интерфейсов для работы с таймерами, чтобы упростить эту работу. Все необходимые определения находятся в файле <linux/timer.h>. Большинство реализаций находится в файле kernel/timer.c.

Первый шаг в создании таймера — это его объявление в следующем виде.

```
struct timer_list my_timer;
```

Далее должны быть инициализированы поля структуры, которые предназначены для внутреннего использования. Это делается с помощью вспомогательной функции перед вызовом любых функций, которые работают с таймером.

```
init_timer(&my_timer);
```

Далее необходимо заполнить все остальные поля структуры, например, следующим образом.

```
my_timer.expires = jiffies + delay; /* интервал времени таймера
                                       закончится через delay импульсов */
```

[7] Другая причина состоит в том, что в ядрах старых версий (до 2.3) существовали статические таймеры. Такие таймеры создавались во время компиляции, а не во время выполнения. Они имели ограниченные возможности и из-за их отсутствия сейчас никто не огорчается.

```
my_timer.data = 0;  /* в функцию-обработчик Судет передан параметр,
                       равный нулю */
my_timer.function = my_function; /* функция, которая будет выполнена,
                                    когда интервал времени таймера истечет */
```

Значение поля my_timer.expires указывает время ожидания в импульсах системного таймера (необходимо указывать абсолютное количество импульсов). Когда текущее значение переменной jiffies становится большим или равным значению поля my_timer . expires, вызывается функция-обработчик my_timer.function с параметром my_timer.data. Как видно из описания структуры timer_list, функция-обработчик должна соответствовать следующему прототипу.

```
void my_timer_function(unsigned long data);
```

Параметр data позволяет регистрировать несколько таймеров с одним обработчиком и отличать таймеры с различными значениями этого параметра. Если в аргументе нет необходимости, то можно просто указать нулевое (или любое другое) значение.

Последняя операция — это активизация таймера.

```
add_timer(&my_timer);
```

И таймер запускается! Следует обратить внимание на важность значения поля expired. Ядро выполняет обработчик, когда текущее значение счетчика импульсов системного таймера *больше,* чем указанное значение времени срабатывания таймера, *или равно* ему. Хотя ядро и гарантирует, что никакой обработчик таймера не будет выполняться до истечения срока ожидания таймера, тем не менее возможны задержки с выполнением обработчика таймера. Обычно обработчики таймеров выполняются в момент времени, близкий к моменту времени срабатывания, однако они могут быть отложены и до следующего импульса системного таймера. Следовательно, таймеры нельзя использовать для работы в жестком режиме реального времени.

Иногда может потребоваться изменить момент времени срабатывания таймера, который уже активизирован. В ядре реализована функция mod_timer(), которая позволяет изменить момент времени срабатывания активного таймера.

```
mod_timer(&my_timer, jiffies + new_delay); /* установка нового времени
                                              срабатывания */
```

Функция mod_timer () позволяет также работать с таймером, который проинициализирован, но не активен. Если таймер не активен, то функция mod_timer() активизирует его. Эта функция возвращает значение 0, если таймер был неактивным, и значение 1, если таймер был активным. В любом случае перед возвратом из функции mod_timer () таймер будут активизирован, и его время срабатывания будет установлено в указанное значение.

Для того чтобы деактивизировать таймер до момента его срабатывания, необходимо использовать функцию del_timer () следующим образом.

```
del_timer(&my_timer);
```

Эта функция работает как с активными, так и неактивными таймерами. Если таймер уже неактивен, то функция возвращает значение 0, в другом случае возвращается значение 1. Следует обратить внимание, что нет необходимости вызывать эту

функцию для таймеров, интервал ожидания которых истек, так как они автоматически деактивипируются.

При удалении таймеров потенциально может возникнуть состояние конкуренции. Когда функция del_timer () возвращает управление, она гарантирует только то, что таймер будет неактивным (т.е. его обработчик не будет выполнен в будущем). Однако на многопроцессорной машине обработчик таймера может выполняться в этот момент на другом процессоре. Для того чтобы деактивизировать таймер и подождать, пока завершится его обработчик, который потенциально может выполняться, необходимо использовать функцию del_timer_sync () :

```
del_timer_sync(&my_timer);
```

В отличие от функции del_timer(), функция del_timer_sync() не может вызываться из контекста прерывания.

Состояния конкуренции, связанные с таймерами

Так как таймеры выполняются асинхронно по отношению к выполняемому в данный момент коду, то потенциально могут возникнуть несколько типов состояний конкуренции за ресурсы. Во-первых, никогда нельзя использовать следующий код, как замену функции inod_timer ().

```
del_timer (my_timer) ;
my_timer->expires = jiffies + new_delay;
add_timer(my_timer);
```

Во-вторых, практически во всех случаях следует использовать функцию del_timer _sync (), а не функцию del_timer (). В противном случае нельзя гарантировать, что обработчик таймера в данный момент не выполняется. Представьте себе, что после удаления таймера код освободит память или каким-либо другим образом вмешается в ресурсы, которые использует обработчик таймера. Поэтому синхронная версия более предпочтительна.

Наконец, необходимо гарантировать защиту всех совместно используемых данпых, к которым обращается функция-обработчик таймера. Ядро выполняет эту функцию асинхронно по отношению к другому коду. Совместно используемые данные должны защищаться так, как рассматривалось в главах 8 и 9.

Реализация таймеров

Ядро выполняет обработчики таймеров в контексте обработчика отложенного прерывания после завершения обработки прерывания таймера. Обработчик прерывания таймера вызывает функцию update_process_times (), которая в свою очередь вызывает функцию run_local_timers () , имеющую следующий вид.

```
void run_local_timers(void)
{
        raise_softirq(TIMER_SOFTIRQ);
}
```

Отложенное прерывание с номером TIMER_SOFTIRQ обрабатывается функцией run_tirner_softirq (). Эта функция выполняет на локальном процессоре обработчики всех таймеров, для которых истек период времени ожидания (если такие есть).

Таймеры хранятся в связанном списке. Однако в ядре было бы неразумным просматривать весь список в поисках таймеров, для которых истекло время ожидания, или поддерживать список в отсортированном состоянии на основании времени срабатывания таймеров. В последнем случае вставка и удаление таймеров заняли бы много времени. Вместо этого таймеры разбиваются на 5 групп на основании времени срабатывания. Таймеры перемещаются из одной группы в другую, по мере того как приближается момент времени срабатывания. Такое разбиение на группы гарантирует, что в большинстве случаев при выполнении обработчика отложенного прерывания, ответственного за выполнение обработчиков таймеров, ядро будет выполнять мало работы для поиска таймеров, у которых истек период ожидания. Следовательно, код управления таймерами очень эффективен.

Задержка выполнения

Часто коду ядра (особенно драйверам) необходимо задерживать выполнение действий на некоторый период времени без использования таймеров или механизма нижних половин. Это обычно необходимо для того, чтобы дать аппаратному обеспечению время на завершение выполнения задачи. Такой интервал времени обычно достаточно короткий. Например, в спецификации сетевой интерфейсной платы может быть указано время изменения режима работы Ethernel-контроллера, равное 2 микросекундам, т.е. после установки желаемой скорости передачи драйвер должен ожидать хотя бы в течение двух микросекунд перед тем, как продолжить работу.

Ядро предоставляет несколько решений этой задачи, в зависимости от семантики задержки. Эти решения имеют разные свойства. Некоторые решения во время задержки загружают процессор, не давая возможности выполнять другую, более полезную работу. Другие решения не загружают процессор, но не дают гарантии того, что код возобновит выполнение точно в необходимый момент времени[8].

Задержка с помощью цикла

Наиболее простое для реализации (хотя обычно не оптимальное) решение — это использование *задержки с помощью цикла* или *ожидания в состоянии занятости (busy loop, busy wailing)*. Эта техника работает, только если интервал времени задержки является кратным периоду системного таймера или когда точность не очень важна.

Идея проста — выполнить постоянный цикл, пока не будет получено необходимое количество импульсов системного таймера, как в следующем примере.

```
unsigned long delay = jiffies + 10; /* десять импульсов таймера */

while (time_before (jiffies, delay));
```

[8] На самом деле, ни один подход не гарантирует, что время задержки будет точно равно указанному значению. Некоторые подходы обеспечивают задержки, очень близкие к точному значению, тем не менее все подходы гарантируют, что время ожидания будет, по крайней мере, не меньше, чем нужно. В некоторых случаях период ожидания получается существенно больше указанного.

Цикл будет выполняться, пока значение переменной jiffies не станет больше, чем значение переменной delay, что может произойти только после того, как будут получены 10 импульсов системного таймера. Для аппаратной платформы x86 со значением параметра HZ, равным 1000, этот интервал равен 10 миллисекунд.

Аналогично можно поступить следующим образом.

```
unsigned long delay = jiffies + 2*HZ; /* две секунды */

while (time_before(jiffies, delay));
```

В этом случае цикл будет выполняться, пока не поступит 2*H2 импульсов системного таймера, что всегда равно 2 секундам, независимо от частоты системного таймера.

Такой подход не очень хорош для всей системы. Пока код ожидает, процессор загружен выполнением бесполезного цикла и никакой полезной работы при этом не выполняется! На самом деле к такому "глупому" подходу нужно прибегать по возможности реже, и он показан здесь, потому что является понятным и простым способом осуществить задержку. Его можно встретить в чьем-нибудь не очеиь хорошем коде.

Лучшим решением япляется перепланирование для того, чтобы процессор мог выполнить полезную работу, пока ваш код ожидает:

```
unsigned long delay = jiffies + 5*HZ;

while (time_before(jiffies, delay))
        cond_reschcd();
```

Вызов функции cond_resched() планирует выполнение другого процесса, но только в случае, если установлен флаг need_resched. Другими словами, данное решение позволяет активизировать планировщик, но только в случае, когда есть более важное задание, которое нужно выполнить. Следует обратить внимание, что. поскольку используется планировщик, такое решение нельзя применять в контексте прерывания, а только в контексте процесса. Задержки лучше использовать только в контексте процесса, поскольку обработчики прерываний должны выполняться по возможности быстро (а цикл задержки не дает такой возможности!). Более того, любые задержки выполнения, по возможности, не должны использоваться при захваченных блокировках и при запрещенных прерываниях.

Поклонники языка C могут поинтересоваться, какие есть гарантии, что указанные циклы будут действительно выполняться? Обычно компилятор C может выполнить чтение указанной переменной всего один раз. В обычной ситуации нет никакой гарантии, что переменная jiffies будет считываться на каждой итерации цикла. Нам же необходимо, чтобы значение переменной jiffies считывалось на каждой итерации цикла, так как это значение увеличивается в другом месте, а именно в прерывании таймера. Именно поэтому данная переменная определена в файле <linux/jiffies.h> с атрибутом volatile. Ключевое слово volatile указывает компилятору, что эту переменную необходимо считывать из того места, где она хранится в оперативной памяти, и никогда не использовать копию, хранящуюся в регистре процессора. Это гарантирует, что указанный цикл выполнится, как и ожидается.

Короткие задержки

Иногда коду ядра (и снопа обычно драйверам) необходимы задержки на очень короткие интервалы времени (короче, чем период системного таймера), причем интервал должен отслеживаться с достаточно высокой точностью. Это часто необходимо для синхронизации с аппаратным обеспечением, для которого описано некоторое минимальное время выполнения действий, и которое часто бывает меньше одной миллисекунды. В случае таких малых значений времени невозможно использовать задержки на основании переменной jiffies, как показано в предыдущем примере. При частоте системного таймера, равной 100 Гц, значение периода системного таймера достаточно большое— 10 миллисекунд! Даже при частоте системного таймера 1000 Гц, период системного таймера равен одной миллисекунде. Ясно, что необходимо другое решение, которое обеспечивает более короткие и точные задержки.

Ядро предоставляет две функции для обеспечения микросекундных и миллисекундных задержек, которые определены в файле <linux/delay.h> и не используют переменную jiffies.

```
void udelay(unsigned long usecs);
void mdelay(unsigned long msecs);
```

Первая функция позволяет задержать выполнение на указанное количество *микросекунд с* использованием цикла. Вторая функция задерживает выполнение на указанное количество *миллисекунд.* Следует вспомнить, что одна секунда равна 1000 миллисекундам, что эквивалентно 1000000 *микросекунд.* Использование этих функций тривиально.

```
udelay(150); /* задержка на 150 ms */
```

Функция udelay () выполнена па основе цикла, для которого известно, сколько итераций необходимо выполнить за указанный период времени. Функция mdelay () выполнена на основе функции udelay (). Так как в ядре известно, сколько циклов процессор может выполнить в одну секунду (смотрите ниже замечание по поводу характеристики BogoMIPS), функция udelay () просто масштабирует это значение для того, чтобы скорректировать количество итераций цикла для получения указанной задержки.

Мой BogoMIPS больше, чем у Вас!

Характеристика BogoMIPS всегда была источником недоразумений и шуток. На самом деле вычисленное значение BogoMIPS не имеет ничего общего с производительностью компьютера и используется только для функций udelay() и mdelay(). Название этого параметра состоит из двух частей bogus (фиктивный) и *MIPS* (million of instructions per second, миллион инструкций в секунду). Все знакомы с сообщением, которое выдается при загрузке системы и похоже на следующее (данное сообщение соответствует процессору Pentium III с частотой 1 ГГц).

```
Detected 1004.932 MHz processor.
Calibrating delay loop... 1990.65 BogoMIPS
```

Значение параметра BogoMIPS - это количество циклов, которые процессор может выполнить за указанный период времени, В действительности эта характеристика показывает, насколько быстро процессор может ничего не делать! Это значение хранится в переменной loops_per_jiffy, и его можно считать из файла /proc/cpuinfo.

функции, использующие циклы задержки, используют данное значение, чтобы вычислить (и это получается достаточно точно), сколько итераций цикла необходимо выполнить, чтобы обеспечить необходимую задержку.

Ядро вычисляет значение переменной `loops_per_jiffy` при загрузке системы в функции `calibrate_delay()`, реализация которой описана в файле `init/main.c`.

Функция `udelay()` должна вызываться только для небольших задержек, поскольку при большом времени задержки на быстрой машине может возникнуть переполнение в переменных цикла. Общее правило: по возможности не использовать функцию `udelay()` для задержек, больше одной миллисекунды. Для более продолжительных задержек хорошо работает функция `mdelay()`. Так же как и другие методы задержки выполнения, основанные на циклах, эти функции (особенно функция `mdelay()`, так как она дает длительные задержки) должны использоваться, только если это абсолютно необходимо. Следует помнить, что очень плохо использовать циклы задержек, когда удерживается блокировка или запрещены прерывания, потому что это очень сильно влияет на производительность и время реакции системы. Если необходимо обеспечить точное время задержки, то эти функции — наилучшее решение. Обычное использование этих функций — это задержки на не очень короткий период времени, который лежит в микросекундном диапазоне.

Функция schedule_timeout()

Более оптимальный метод задержки выполнения — это использование функции `schedule_timeouit ()`. Этот вызов переводит вызывающее задание в состояние ожидания (sleep) по крайней до тех пор, пока не пройдет указанный период времени. Нет никакой гарантии, что время ожидания будет *точно* равно указанному значению, гарантируется только, что задержка будет не меньше указанной. Когда проходит указанный период времени, ядро возвращает задание в состояние готовности к выполнению (wake up) и помещает его в очередь выполнения. Использовать эту функцию просто.

```
/* установить состояние задания в значение прерываемого ожидания */
set_current_state(TASK  INTERRUPTIBLE);

/* перейти в приостановленное состояние на s секунд */
schedule_timeout(s * HZ);
```

Единственный параметр функции — это желаемое относительное время, выраженное в количестве импульсов системного таймера. В этом примере задание переводится в прерываемое состояние ожидания, которое будет длиться s секунд. Поскольку задание отмечено как TASK_INTERRUPTIBLE, то оно может быть возвращено к выполнению раньше времени, как только оно получит сигнал. Если не нужно, чтобы код обрабатывал сигналы, то можно использовать состояние TASK_ UNINTERRUPTIBLE. Перед вызовом функции `schedule_tirneout()` задание должно быть в одном из этих двух состояний, иначе задание в состояние ожидания переведено не будет.

Следует обратить внимание, что поскольку функция `schedule_tiraeout()` использует планировщик, то код, который ее вызывает, должен быть совместим с состоянием ожидания. Обсуждение, посвященное атомарности и переходу в состояние

ожидания, приведено в главах 8 и 9. Если коротко, то эту функцию необходимо вызывать в контексте процесса и не удерживать при этом блокировку.

Функция schedule_tiraeout() достаточно проста. Она просто использует таймеры ядра. Рассмотрим эту функцию подробнее.

```
signed long schedule_timeout(signed long timeout)
{
        timer_t timer;
        unsigned long expire;

        switch (timeout)
        {
        case MAX_SCHEDULE_TIMEOUT:
              schedule();
              goto out;
        default:
              if (timeout < 0)
              {
                    printk(KERN_ERR "schedule_timeout: wrong timeout "
                          "value %lx from %p\n", timeout,
                            builtin_return_address(0));
                    current->state = TASK_RUNNING;
                    goto out;
              }
        }

        expire = timeout + jiffies;

        init timer(&timer);
        timer.expires = expire;
        timer.data = (unsigned long) current;
        timer.function = process_timeout;

        add_timer(&timer);
        schedule();
        del_timer_sync(&timer);

        timeout = expire - jiffies;
out:
        return timeout < 0 ? 0 : timeout;
}
```

Эта функция создает таймер timer и устанавливает время срабатывания в значение timeout импульсов системного таймера в будущем. В качестве обработчика таймера устанавливается функция process timeout () , которая вызывается, когда истекает период времени таймера. Далее таймер активизируется, и вызывается функция schedule (). Так как предполагается, что текущее задание находится в состоянии TASK_INTERRUPTIBLE или TASK_UNINTERRUPTIBLE, то планировщик не будет выполнять текущее задание, а выберет для выполнения другой процесс.

Когда интервал времени таймера истекает, то вызывается функция process_ timeout (), которая имеет следующий вид.

```
void process_timeout(unsigned long data)
{
        wake_up_process((task_t *) data);
}
```

Эта функция устанавливает задание в состояние TASK_RUNNING и помещает его в очередь выполнения.

Когда задание снова планируется на выполнение, то оно возвращается в функцию schedule_tiraeout () (сразу после вызова функции schedule ()). Если задание возвращается к выполнению преждевременно, то таймер ликвидируется. После этого задание возвращается из функции ожидания по тайм-ауту.

Код оператора switch () служит для обработки специальных случаев и не является основной частью функции. Проверка на значение MAX_SCHEDULE_TIMEOUT позволяет заданию находиться в состоянии ожидания неопределенное время. В этом случае таймер не устанавливается (поскольку нет ограничений на интервал времени ожидания), и сразу же активизируется планировщик. Если вы это применяете, то, наверное, у вас есть лучший способ вернуть задание в состояние выполнения!

Ожидание в очереди wait queue в течение интервала времени

В главе 4 рассматривалось, как контекст процесса в ядре может поместить себя в очередь ожидания для того, чтобы ждать наступления некоторого события, а затем вызвать планировщик, который выберет новое задание для выполнения. Если где-то в другом месте произойдет указанное событие, то вызывается функция wake_up () для всех заданий, которые ожидают в очереди. Эти задания возвращаются к выполнению и могут продолжать работу.

Иногда желательно ожидать наступления некоторого события *или* пока не пройдет определенный интервал времени, в зависимости от того, что наступит раньше, В этом случае код должен просто вызвать функцию schedule_timeout () вместо функции schedule () после того, как он поместил себя в очередь ожидания. Задание будет возвращено к выполнению, когда произойдет желаемое событие или пройдет указанный интервал времени. Код обязательно должен проверить, *почему* он возвратился к выполнению — это может произойти потому, что произошло событие, прошел интервал времени или был получен сигнал — после этого необходимо соответственным образом продолжить выполнение.

Время вышло

В этой главе были рассмотрены понятия, связанные с представлением о времени в ядре и с тем, как при этом происходит управление абсолютным и относительным ходом времени. Были показаны отличия абсолютного и относительного времени, а также периодических и относительных событий. Далее были рассмотрены прерывания таймера, импульсы таймера, константа HZ и переменная jiffies.

После этого было рассказано о том, как реализованы таймеры ядра и как их можно использовать в собственном коде ядра. В конце главы были представлены другие методы, которые разработчики могут использовать для учета времени.

Большая часть кода ядра, который вам придется писать, требует понимания того, как время течет в ядре и как его отслеживать. С очень большой вероятностью, особенно при разработке драйверов, вам необходимо будет иметь дело с таймерами ядра. Материал этой главы принесет практическую пользу.

11

Управление памятью

Выделить память *внутри* ядра не так просто, как *вне* ядра. Это связано со многими факторами. Главным образом, причина в том, что в ядре не доступны те элементы роскоши, которыми можно пользоваться в пространстве пользователя, В отличие от пространства пользователя, в ядре не всегда можно позволить себе легко выделять память. Например, в режиме ядра часто нельзя переходить в состояние ожидания. Более того, в ядре не так просто справиться с ошибками, которые возникают при работе с памятью. Из-за этих ограничений и из-за необходимости, чтобы схема выделения памяти была быстрой, работа с памятью в режиме ядра становится более сложной, чем в режиме пользователя. Конечно, нельзя сказать, что выделение памяти в ядре — очень сложная процедура, однако скоро все будет ясно — просто это делается несколько по-другому.

В этой главе рассматриваются средства, которые предназначены для выделения памяти внутри ядра. Перед изучением интерфейсов, предназначенных для выделения памяти, необходимо рассмотреть, как ядро управляет памятью.

Страницы памяти

Ядро рассматривает страницы физической памяти как основные единицы управления памятью. Хотя наименьшая единица памяти, которую может адресовать процессор, — это машинное слово, модуль управления памятью (MMU, Memory Management Unit) — аппаратное устройство, которое управляет памятью и отвечает за трансляцию виртуальных адресов в физические — обычно работает со страницами. Поэтому модуль MMU управляет таблицами страниц на уровне страничной детализации (отсюда и название). С точки зрения виртуальной памяти, страница — это наименьшая значащая единица.

Как будет показано в главе 19, "Переносимость", каждая аппаратная платформа поддерживает свой характерный размер страницы. Многие аппаратные платформы поддерживают даже несколько разных размеров страниц. Большинство 32-разрядных платформ имеют размер страницы, равный 4 Кбайт, а большинство 64-разрядных платформ— 8 Кбайт. Это значит, что на машине, размер страницы которой равен 4 Кбайт, при объеме физической памяти, равном 1 Гбайт, эта физическая память разбивается на 262 144 страницы.

Ядро сопоставляет *каждой* странице физической памяти в системе структуру struct page. Эта структура определена в файле <linux/mrn.h> следующим образом.

```
struct page {
        page_flags_t              flags;
        atomic_t                  _count;
        atornic_t                 _mapcount;
        unsigned long             private;
        struct address_space      *mapping;
        pgoff_t                   index;
        struct list_head          lru;
        void                      *virtual;
};
```

Рассмотрим самые важные поля этой структуры. Поле flags содержит состояние страницы. Это поле включает следующую информацию: является ли страница измененной (dirty) или заблокированной (locked) в памяти. Значение каждого флага представлено одним битом, поэтому всего может быть до 32 разных флагов. Значения флагов определены в файле <linux/page-flags.h>.

Поле _count содержит счетчик использования страницы — т.е. сколько на эту страницу имеется ссылок. Когда это значение равно нулю, это значит, что никто не использует страницу, и она становится доступной для использования при новом выделении памяти. Код ядра не должен явно проверять значение этого поля, вместо этого необходимо использовать функцию page_count () , которая принимает указатель на структуру page в качестве единственного параметра. Хотя в случае незанятой страницы памяти значение счетчика _count может быть отрицательным (во внутреннем представлении), функция page_count () возвращает значение нуль для незанятой страницы памяти и положительное значение — для страницы, которая в данный момент используется. Страница может использоваться страничным кэшем (в таком случае поле mapping указывает на объект типа address_space, который связан с данной страницей памяти), может использоваться в качестве частных данных (на которые в таком случае указывает поле private) или отображаться в таблицу страниц процесса.

Поле virtual — это виртуальный адрес страницы. Обычно это просто адрес данной страницы в виртуальной памяти ядра. Некоторая часть памяти (называемая областью верхней памяти, high memory) не отображается в адресное пространство ядра (т.е. не входит в него постоянно). В этом случае значение данного поля равно NULL и страница при необходимости должна отображаться динамически. Верхняя память будет рассмотрена в одном из следующих разделов.

Наиболее важный момент, который необходимо понять, это то, что структура page связана со страницами физической, а не виртуальной памяти. Поэтому то, чему соответствует экземпляр этой структуры, в лучшем случае, очень быстро изменяется. Даже если данные, которые содержались в физической странице, продолжают существовать, то это не значит, что эти данные будут всегда соответствовать одной и той же физической странице памяти и соответственно одной и той же структуре раде, например из-за вытеснения страниц (swapping) или по другим причинам. Ядро использует эту структуру данных для описания всего того, что содержится в данный момент в странице физической памяти, соответствующей данной структуре.

Назначение этой структуры— описывать область физической памяти, а не данных, которые в ней содержатся.

Ядро использует рассматриваемую структуру данных для отслеживания всех страниц физической памяти в системе, так как ядру необходима информация о том, свободна ли страница (т.е. соответствующая область физической памяти никому не выделена). Если страница не свободна, то ядро должно иметь информацию о том, чему принадлежит эта страница. Возможные обладатели: процесс пространства пользователя, данные в динамически выделенной памяти в пространстве ядра, статический код ядра, страничный кэш (page cache) и т.д.

Разработчики часто удивляются, что для каждой физической страницы в системе создается экземпляр данной структуры. Они думают: "Как много для этого используется памяти!" Давайте посмотрим, насколько плохо (или хорошо) расходуется адресное пространство для хранения информации о страницах памяти. Размер структуры `struct page` равен 40 байт. Допустим, что система имеет страницы размером 1 Кбайт, а объем физической памяти равен 128 Мбайт. Тогда все структуры раде в системе займут немного больше 1 Мбайт памяти — не очень большая плата за возможность управления всеми страницами физической памяти.

Зоны

В связи с ограничениями аппаратного обеспечения, ядро не может рассматривать все страницы памяти как идентичные. Некоторые страницы, в связи со значениями их физических адресов памяти, не могут использоваться для некоторых типов задач. Из-за этого ограничения ядро делит физическую память на *зоны*. Ядро использует зоны, чтобы группировать страницы памяти с аналогичными свойствами. В частности, операционная система Linux должна учитывать следующие недостатки аппаратного обеспечения, связанные с адресацией памяти.

- Некоторые аппаратные устройства могут выполнять прямой доступ к памяти (ПДП, DMA, Direct Memory Access) только в определенную область адресов.
- На некоторых аппаратных платформах для физической адресации доступны большие объемы памяти, чем для виртуальной адресации. Следовательно, часть памяти не может постоянно отображаться в адресное пространство ядра.

В связи с этими ограничениями, в операционной системе Linux выделяют три зоны памяти.

- ZONE_DMA. Содержит страницы, которые совместимы с режимом DMA.
- ZONE_NORMAL. Содержит страницы памяти, которые отображаются в адресные пространства обычным образом.
- ZONE_HIGHMEM. Содержит "верхнюю память", состоящую из страниц, которые не могут постоянно отображаться в адресное пространство ядра.

Эти зоны определяются в заголовочном файле <linux/mmzone.h>.

То, как используется разделение памяти на зоны, зависит от аппаратной платформы. Например, для некоторых аппаратных платформ нет проблем с прямым доступом к памяти ни по какому адресу. Для таких платформ зона ZONE_DMA является пустой, и для всех типов выделения памяти используется зона ZONE_NORMAL.

Как противоположный пример можно привести платформу x86, для которой устройства ISA[1] не могут выполнять операции DMA в полном 32-разрядном пространстве адресов, так как устройства ISA могут обращаться только к первым 16 Мбайт физической памяти. Следовательно, зона ZONE_DMA для платформы x8б содержит только страницы памяти с физическими адресами в диапазоне 0-16 Мбайт.

Аналогично используется и зона ZONE_HIGHMEM. То, что аппаратная платформа может отображать и чего она не может отображать в адресное пространство ядра, отличается для разных аппаратных платформ. Для платформы x86 зона ZONE_ HIGHMEM— это вся память, адреса которой лежат выше отметки 896 Мбайт. Для других аппаратных платформ зона ZONE_HIGHMEM пуста, так как вся память может непосредственно отображаться. Память, которая содержится в зоне ZONE_HIGHMEM, называется *верхней памятью*[2] *(high memory).* Вся остальная память в системе называется *нижней памятью (low memory).*

Зона ZONE_NORMAL обычно содержит все, что не попало в две предыдущие зоны памяти. Для аппаратной платформы x86, например, зона ZONE_NORMAL содержит всю физическую память от 16 до 896 Мбайт. Для других, более удачных аппаратных платформ, SONE_NORMAL— это вся доступная память. В табл. 11.1 приведен список зон для аппаратной платформы x86.

Таблица 11.1. Зоны памяти для аппаратной платформы x86

Зона	Описание	физическая память
ZONE_DMA	Страницы памяти, совместимые с ПДП	< 16 Мбайт
ZONE_NORMAL	Нормально адресуемые страницы	16 - 896 Мбайт
ZONE_HIGHMEM	Динамически отображаемые страницы	> 896 Мбайт

Операционная система разделяет страницы системной памяти на зоны, чтобы иметь пулы страниц для удовлетворения требований выделения памяти. Например, пул зоны ZONE_DMA дает возможность ядру удовлетворить запрос на выделение памяти, которая необходима для операций DMA. Если нужна такая память, ядро может просто выделить необходимое количество страниц из зоны ZONE_DMA. Следует обратить внимание, что зоны не связаны с аппаратным обеспечением— это логическое группирование, которое позволяет ядру вести учет страниц; памяти.

Хотя некоторые запросы на выделение памяти могут требовать страницы из определенной зоны, это требование не обязательно может быть жестким. Например, выделение памяти для ПДП требует страницы из зоны ZONE DMA, а для обычного выделения памяти могут подойти страницы как из зоны ZONE_NORMAL, так и из зоны ZONE_DMA. Конечно, для удовлетворения запросов по обычному выделению памяти ядро будет стараться выделять страницы из зоны ZONE_NORMAL, чтобы сохранить страницы в зоне ZONE_DMA для случая, когда эти страницы действительно нужны, Если же наступает решающий момент (становится недостаточно памяти), то ядро может обратиться к любой доступной и подходящей зоне.

[1] Некоторые некачественные устройства PCI также могут выполнять прямой доступ к памяти только к 24-битовом адресном пространстве. Но эти устройства работают не правильно.

[2] Это не имеет ничего общего с верхней памятью в операционной системе DOS.

Каждая зона представлена с помощью структуры struct zone, которая определена в файле <linux/mmzone.h> в следующем виде.

```
struct zone {
        spinlock t                lock;
        unsigned ]ong             free_pages;
        unsigned long             pages_min;
        unsigned long             pages_low;
        unsigned long             pages_high;
        unsigned long             protection[MAX_NR_ZONES];
        spinlock_t                lru_lock;
        struct list_head          active_list;
        struct list_head          inactive_list;
        unsigned long             nr_scan_active;
        unsigned long             nr_scan_inactive;
        unsigned long             nr_active;
        unsigned long             nr_inactive;
        int                       all_unreclaimable;
        unsigned long             pages_scanned;
        int                       temp_priority;
        int                       prev_priority;
        struct free_area          free_area[MAX_ORDER];
        wait_queue_head_t         *wait_table;
        unsigned long             wait_table_size;
        unsigned long             wait_table_bits;
        struct per_cpu_pageset    pageset[NR_CPUS];
        struct pglist_data        *zone_pgdat;
        struct page               *zone_mem_map;
        unsigned long             zone_start_pfn;
        char                      *name;
        unsigned long             spanned_pages;
        unsigned long             prcsent_pages;
};
```

Эта структура большая, но в системе всего три зоны и соответственно три такие структуры. Рассмотрим наиболее важные поля данной структуры.

Поле lock— это спин-блокировка, которая защищает структуру от параллельного доступа. Обратите внимание, что она защищает только структуру, а не страницы, которые принадлежат зоне. Для защиты отдельных страниц нет блокировок, хотя отдельные части кода могут блокировать данные, которые могут оказаться в указанных страницах.

Поле free_pages — это количество свободных страниц в соответствующей зоне. Ядро старается поддерживать свободными хотя бы pages_min страниц зоны, если это возможно (например, с помощью вытеснения на диск).

Поле name— это строка, оканчивающаяся нулем, которая содержит имя соответствующей зоны (что не удивительно). Ядро инициализирует указанное поле при загрузке системы с помощью кода, который описан п файле mm/page_alloc.c. Три зоны имеют имена "DMA", "Normal" и "HighMem".

Получение страниц памяти

Теперь, имея некоторое понятие о том, как ядро управляет памятью с помощью страниц, зон и так далее, давайте рассмотрим интерфейсы, которые реализованы в ядре для того, чтобы выделять и освобождать память внутри ядра. Ядро предоставляет один низкоуровневый интерфейс для выделения памяти и несколько интерфейсов для доступа к ней. Все эти интерфейсы выделяют память в объеме, кратном размеру страницы, и определены в файле <linux/gfp.h>. Основная функция выделения памяти следующая.

```
struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)
```

Данная функция позволяет выделить 2^{order} (т.е. 1 << order) смежных страниц (один непрерывный участок) физической памяти и возвращает указатель на структуру page, которая соответствует первой выделенной странице памяти. В случае ошибки возвращается значение NULL. Параметр gfp_mask будет рассмотрен несколько позже. Полученную страницу памяти можно конвертировать в ее логический адрес с помощью следующей функции.

```
void * page_address(struct page *page)
```

Эта функция возвращает указатель на логический адрес, которому в данный момент соответствует начало указанной страницы физической памяти. Если нет необходимости в соответствующей структуре struct page, то можно использовать следующую функцию.

```
unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int
order)
```

Эта функция работает так же, как и функция alloc_pages(), за исключением того, что она сразу возвращает логический адрес первой выделенной страницы памяти. Так как выделяются смежные страницы памяти, то другие страницы просто следуют за первой.

Если необходима всего одна страница памяти, то для этой цели определены следующие функции-оболочки, которые позволяют уменьшить количество работы по набору кода программы.

```
struct page * alloc_page(unsigned int gfp_mask)
unsigned long __get_free_page(unsigned int gfp_mask)
```

Эти функции работают так же, как и ранее описанные, по для них в качестве параметра order передается нуль (2^0 = одна страница памяти).

Получение страниц заполненных нулями

Для того чтобы получаемые страницы памяти были заполнены нулями, необходимо использовать следующую функцию.

```
unsigned long get_zeroed_page(unsigned int gfp_mask)
```

Эта функция аналогична функции __get_free_page(), за исключением того, что после выделения страницы памяти она заполняется нулями. Это полезно для страниц памяти, которые возвращаются в пространство пользователя, так как слу-

чайный "мусор", который находится в страницах памяти, может оказаться не совсем случайным и случайно может содержать некоторые (например, секретные) данные. Все данные необходимо обнулить или очистить каким-либо другим образом перед тем, как возвращать информацию в пространство пользователя, чтобы при этом не пострадала безопасность системы. В табл. 11.2 приведен список всех низкоуровневых средств выделения памяти.

Таблица 11.2. Низкоуровневые средства выделения памяти

Функция	Описание
alloc_page (gfp_mask)	Выделяет одну страницу памяти и возвращает указатель на соответствующую ей структуру page
alloc_pages (gfp_mask, order)	Выделяет 2^{order} страниц памяти и возвращает указатель на структуру page первой страницы
__get_free_page (gfp_mask)	Выделяет одну страницу памяти и возвращает указатель на ее логический адрес
__get_free_pages (gfp_mask, order)	Выделяет 2^{order} страниц памяти и возвращает указатель на логический адрес первой страницы
get_zeroed_page (gfp_mask)	Выделяет одну страницу памяти, обнуляет ее содержимое и возвращает указатель на ее логический адрес

Освобождение страниц

Для освобождения страниц, которые больше не нужны, можно использовать следующие функции.

```
void __free_pages(struct page *page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
void free_page(unsigned long addr)
```

Необходимо быть внимательными и освобождать только те страницы памяти, которые вам выделены. Передача неправильного значения параметра page, addr или order может привести к порче данных. Следует помнить, что ядро доверяет себе. В отличие от пространства пользователя, ядро с удовольствием зависнет, если вы попросите. Рассмотрим пример. Мы хотим выделить 8 страниц памяти.

```
page=__get_free_pages(GFP_KERNEL,3);

if(!page){
        /* недостаточно памяти: эту ошибку необходимо обработать самим! */
        return-ENOMEM;
}

/* переменная 'page' теперь содержит адрес первой из восьми страниц памяти*/

free_pages(page, 3);

/*
* наши страницы памяти теперь освобождены и нам больше нельзя
* обращаться по адресу, который хранится в переменной 'page'
*/
```

Значение GFP_KERNEL, которое передается в качестве параметра, — это пример флага gfp_mask, который скоро будет рассмотрен детально.

Обратите внимание на проверку ошибок после вызова функции __get_free_pages(). Выделение памяти в ядре *может* быть неудачным, поэтому код *должен* проверить и при необходимости обработать соответствующую ошибку. Это может означать, что придется пересмотреть все операции, которые были до этого сделаны. В связи с этим, часто имеет смысл выделять память в самом начале подпрограммы, чтобы упростить обработку ошибок. В противном случае после попытки выделения памяти отмена ранее выполненных действий может оказаться сложной.

Низкоуровневые функции выделения памяти полезны, когда необходимы участки памяти, которые находятся в смежных физических страницах, особенно если необходима одна страница или очень большое количество страниц. Для более общего случая, когда необходимо выделить заданное количество байтов памяти, ядро предоставляет функцию kmalloc().

Функция kmalloc()

Функция kmalloc() работает аналогично функции malloc() пространства пользователя, за исключением того, что добавляется еще один параметр flags. Функция kmalloc() — это простой интерфейс для выделения в ядре участков памяти размером в заданное количество байтов. Если необходимы смежные страницы физической памяти, особенно когда их количество близко целой степени двойки, то ранее рассмотренные интерфейсы могут оказаться лучшим выбором. Однако для большинства операций выделения памяти в ядре функция kmalloc() — наиболее предпочтительный интерфейс.

Рассматриваемая функция определена в файле <linux/slab.h> следующим образом.

```
void * kmalloc(size_t size, int flags)
```

Данная функция возвращает указатель на участок памяти, который имеет размер *хотя бы* size байт[3]. Выделенный участок памяти содержит физически смежные страницы. В случае ошибки функция возвращает значение NULL. Выделение памяти в ядре заканчивается успешно, только если доступно достаточное количество памяти. Поэтому после вызова функции kmalloc() всегда необходимо проверять возвращаемое значение на равенство значению NULL и соответственным образом обрабатывать ошибку.

Рассмотрим пример. Допустим, нам необходимо выделить достаточно памяти для того, чтобы в ней можно было разместить некоторую воображаемую структуру dog.

```
struct dog *ptr;
ptr = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!ptr)
        /* здесь обработать ошибку ... */
```

[3] Данная функция может выделить памяти больше, чем указано, и нет никакой возможности узнать, на сколько больше! Поскольку в своей основе система выделения памяти в ядре базируется на страницах, некоторые запросы на выделение памяти могут округляться, чтобы хорошо вписываться е области доступной памяти. Ядро никогда не выделит меньше памяти, чем необходимо. Если ядро не *v,* состоянии найти хотя бы указанное количество байтов, то операция завершится неудачно и функции возвратит значение *NULL.*

Если вызов функции kmalloc () завершится успешно, то переменная ptr будет указывать на область памяти, размер которой больше указанного значения или равен ему. Флаг GFP_KERNEL определяет тип поведения системы выделения памяти, когда она пытается выделить необходимую память при вызове функции kmalloc ().

Флаги gfp_mask

Выше были показаны различные примеры использования флагов, которые модифицируют работу системы выделения памяти, как при вызове низкоуровневых функций, работающих на уровне страниц, так и при использовании функции kmalloc {) . Теперь давайте рассмотрим их более детально.

Флаги разбиты на три категории: модификаторы операций, модификаторы зон и флаги типов. Модификаторы операций указывают, *каким образом* ядро должно выделять указанную память. В некоторых ситуациях только некоторые методы могут использоваться для выделения памяти. Например, обработчики прерываний могут потребовать от ядра, что нельзя переходить в состояние ожидания при выделении памяти (поскольку обработчики прерывания не могут быть перепланированы), Модификаторы зоны указывают, *откуда* нужно выделять память. Как было рассказано, ядро делит физическую память на несколько зон, каждая из которых служит для различных целей. Модификаторы зоны указывают, из какой зоны выделять память. Флаги типов представляют собой различные комбинации модификаторов операций и зон, которые необходимы для определенного *типа* выделения памяти. Флаги типов содержат в себе различные модификаторы, вместо которых можно просто использовать один флаг типа. Флаг GFP_KERNEL — это флаг типа, который используется кодом, выполняющимся в ядре в контексте процесса. Рассмотрим все флаги отдельно.

Модификаторы операций

Все флаги, включая модификаторы операций, определены в заголовочном файле <linux/gfp.h>. Подключение файла <linux/slab.h> также подключает и этот заголовочный файл, поэтому его не часто приходится подключать явно. На практике обычно лучше использовать флаги типов, которые будут рассмотрены дальше. Тем не менее полезно иметь представление об индивидуальных флагах. В табл. 11.3 показан список модификаторов операций.

Описанные модификаторы можно указывать вместе, как показано в следующем примере.

```
ptr = kmalioc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
```

Этот код дает инструкцию ядру (а именно функции alloc_pages ()), что операция выделения памяти может быть блокирующей, выполнять операции ввода-вывода и операции файловой системы, если это необходимо. В данном случае ядру предоставляется большая свобода в отношении того, где оно будет искать необходимую память, чтобы удовлетворить запрос.

Большинство запросов на выделение памяти указывают эти модификаторы, но это делается косвенным образом с помощью флагов типа, которые скоро будут рассмотрены. Не нужно волноваться, у вас не будет необходимости каждый раз разбираться, какие из этих ужасных флагов использовать при выделении памяти!

Таблица 11.3. Модификаторы операций

Флаг	Описание
__GFP_WAIT	Операция выделения памяти может переводить текущий процесс в состояние ожидания
__GFP_HIGH	Операция выделения памяти может обращаться к аварийным запасам
__GFP_IO	Операция выделения памяти может использовать дисковые операции ввода-вывода
__GFP_FS	Операция выделения памяти может использовать операции ввода-вывода файловой системы
__GFP_COLD	Операция выделения памяти должна использовать страницы памяти, содержимое которых не находится в кэше процессора (cache cold)
__GFP_NOWARN	Операция выделения памяти не будет печатать сообщения об ошибках
__GFP_REPEAT	Операция выделения памяти повторит попытку выделения в случае ошибки
__GFP_NOFAIL	Операция выделения памяти будет повторять попытки выделения неопределенное количество раз
__GFP_NORETRY	Операция выделения памяти никогда не будет повторять попытку выделения памяти
	Используется внутри слябового распределителя памяти (slab layer)
__GFP_NO_GROW	Добавить метаданные составной (compound) страницы памяти.
__GFP_COMP	Используется кодом поддержки больших страниц памяти (hugetlb)

Модификаторы зоны

Модификаторы зоны указывают, из какой зоны должна выделяться память. Обычно выделение может происходить из любой зоны. Однако ядро предпочитает зону ZONE_NORMAL, чтобы гарантировать, что в других зонах, когда это необходимо, есть свободные страницы.

Существует всего два модификатора зоны, поскольку, кроме зоны ZONE_N0RMAL (из которой по умолчанию идет выделение памяти), существует всего две зоны. В табл. 11.4 приведен список модификаторов зоны.

Таблица 11.4. Модификаторы зоны

Флаг	Описание
__GFP_DMA	Выделять память только из зоны ZONE_DMA
__GFP_HIGHMEM	Выделять память только из зон ZONE_HIGHMEM и ZONE_NORMAL

Указание одного из этих флагов изменяет зону, из которой ядро пытается выделить память. Флаг __GFP_DMA требует, чтобы ядро выделило память только из зоны ZONE_DMA. Этот флаг эквивалентен следующему высказыванию в форме жесткого требования: *"Мне абсолютно необходима память, в которой можно выполнять операции прямого доступа к памяти"*. Флаг __GFP_HIGHMEM, наоборот, требует, чтобы выделение памяти было из зон ZONE_NORMAL и ZOHE_HIGHMEM (вторая более предпочтительна). Этот флаг эквивалентен запросу: *"Я могу использовать верхнюю память, но мне на самом деле, все равно, и делайте, что хотите, обычная память тоже подойдет"*.

Если не указан ни один из флагов, то ядро пытается выделять память из зон ZONE_NORMAL и ZONE_DMA, отдавая значительное предпочтение зоне ZONE_NORMAL.

Флаг __GFP_HIGHMEM нельзя укалывать при вызове функций __get_free_pages () или kmalloc (> . Это связано с тем, что они возвращают логический адрес, а не структуру page, и появляется возможность, что эти функции выделят память, которая в данный момент не отображается п виртуальное адресное пространство ядра и поэтому не имеет логического адреса. Только функция alloc_pagea () может выделять страницы в верхней памяти. Однако в большинстве случаев в запросах на выделение памяти не нужно указывать модификаторы зоны, так как достаточно того, что используется зона ZONE_NORMAL.

Флаги типов

Флаги типов указывают модификаторы операций и зон, которые необходимы для выполнения запросов определенных типов. В связи с этим, в коде ядра стараются использовать правильный флаг типа и не использовать больших наборов модификаторов. Это одновременно и проще и при этом меньше шансов ошибиться. В табл. 11.5 приведен список возможных флагов типов, а в табл. 11.6 показано, какие модификаторы соответствуют какому флагу.

Таблица 11.5. Флаги типов

Флаг	Описание
GFP_ATOMIC	Запрос на выделение памяти высокоприоритетный и в состояние ожидания переходить нельзя. Этот флаг предназначен для использования в обработчиках прерываний, нижних половин и в других ситуациях, когда нельзя переходить в состояние ожидания
GFP_NOIO	Запрос на выделение памяти может блокироваться, но при его выполнении нельзя выполнять операции дискового ввода-вывода. Этот флаг предназначен для использования в коде блочного ввода-вывода, когда нельзя инициировать новые операции ввода-вывода
GFP_NOFS	Запрос на выделение памяти может блокироваться и выполнять дисковые операции ввода-вывода, но запрещено выполнять операции, связанные с файловыми системами. Этот флаг предназначен для использования в коде файловых систем, когда нельзя начинать выполнение новых файловых операций
GFP_KERNEL	Обычный запрос на выделение памяти, который может блокироваться. Этот флаг предназначен для использования в коде, который выполняется в контексте процесса, когда безопасно переходить в состояние ожидания
GFP_USER	Обычный запрос на выделение памяти, который может блокироваться. Этот флаг используется для выделения памяти процессам пространства пользователя
GFP_HIGHUSER	Запрос на выделение памяти из зоны ZONE_HIGHMEM, который может блокироваться. Этот флаг используется для выделения памяти процессам пространства пользователя
GFP_DMA	Запрос на выделение памяти из зоны ZONE_DMA. Драйверам устройств, которым нужна память для выполнения операций по ПДП, необходимо использовать этот флаг обычно в комбинации с одним из описанных выше флагов

Таблица 11.6. Список модификаторов, соответствующих каждому флагу типа

Флаг	Модификаторы
GFP_ATOMIC	__GFP_HIGH
GFP_NOIO	__GFP_WAIT
GFP_NOFS	(__GFP_WAIT | __GFP_IO)
GFP_KERNEL	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_USER	(__GFP_WAIT | __GFP_IO | __GFP_FS)
GFP_HIGHUSER	(__GFP_WAIT | __GFP_IO | __GFP_FS | __GFP_HIGHMEM)
GFP_DMA	__GFP_DMA

Рассмотрим наиболее часто используемые флаги и для чего они могут быть нужны. Подавляющее большинство операций выделения памяти в ядре используют флаг GFP_KERNEL. В результате операция выделения памяти имеет обычный приоритет и может переводить процесс в состояние ожидания. Поскольку этот вызов может блокироваться, его можно использовать только в контексте процесса, выполнение которого может быть безопасно перепланировано (т.е. нет удерживаемых блокировок и т.д.). При использовании этого флага нет никаких оговорок по поводу того, каким образом ядро может получить необходимую память, поэтому операция выделения памяти имеет большой шанс выполниться успешно.

Можно сказать, что свойства флага GFP_ATOMIC лежат на противоположном конце спектра. Так как этот флаг указывает, что операция выделения памяти не может переходить в состояние ожидания, то такая операция очень ограничена в том, какую память можно использовать для выделения. Если нет доступного участка памяти заданного размера, то ядро, скорее всего, не будет пытаться освободить память, поскольку вызывающий код не может переходить в состояние ожидания. При использовании флага GFP_KERNEL, наоборот, ядро может перевести вызывающий код в состояние ожидания, чтобы во время ожидания вытеснить страницы на диск (swap out), очистить измененные страницы памяти путем записи их в дисковый файл (flush dirty pages) и т.д. Поскольку при использовании флага GFP_ATOMIC нет возможности выполнить ни одну из этих операций, то и шансов успешно выполнить выделение памяти тоже меньше (по крайней мере, когда в системе недостаточно памяти). Тем не менее использование флага GFP_ATOMIC— это единственная возможность, когда вызывающий код не может переходить в состояние ожидания, как в случае обработчиков прерываний и нижних половин.

По своим свойствам между рассмотренными флагами находятся флаги GFP_NOIC и GFP_NOFS. Операции пыделения памяти, которые запущены с этими флагами, могут блокироваться, но они воздерживаются от выполнения некоторых действий. Выделение памяти с флагом GFP_NOIO не будет запускать никаких операций дискового ввода-вывода. С другой стороны, при использовании флага GFP_NOFS могут запускаться операции дискового ввода-вывода, но не могут запускаться операции файловых систем. Когда эти флаги могут быть полезны? Они соответственно необходимы для определенного низкоуровневого кода блочного ввода-вывода или кода файловых систем. Представьте себе, что в некотором часто используемом участке кода файловых систем используется выделение памяти *без указания* флага GFP_NOFS. Если выделение памяти требует выполнения операций файловой системы, то вы-

деление памяти приведет к еще *большему* количеству операций файловой системы, которые потребуют дополнительного выделения памяти и еще большего количества файловых операций! При разработке кода, который использует выделение памяти, необходимо гарантировать, что операции выделения памяти не будут использовать этот код, как в рассмотренном случае, иначе может возникнуть самоблокировка. Не удивительно, что и ядре рассматриваемые флаги используются только в небольшом количестве мест.

Флаг GFP_DMA применяется для указания, что система выделения памяти должна при выполнении запроса предоставить память из зоны ZONE_DMA. Этот флаг используется драйверами устройств, для которых необходимо выполнение операций прямого доступа к памяти. Обычно этот флаг должен комбинироваться с флагами CFP_ATOMIC или GFP_KERNEL.

В подавляющем большинстве случаев при разработке кода вам будет необходимо использовать флаги GFP_ATOMIC или GFP_KERNEL. В табл. 11.7 показано какие флаги и в каких ситуациях необходимо использовать. Независимо от типа операции выделения памяти, необходимо проверять результат и обрабатывать ошибки.

Таблица 11.7. Какой флаг и когда необходимо использовать

Ситуация	Решение
Контекст процесса, можно переходить в состояние ожидания	Используется флаг GFP_KERNEL
Контекст процесса, нельзя переходить в состояние ожидания	Используется флаг GFP_ATOMIC или память выделяется с использованием флага GFP_KERNEL но в более ранний или поздний момент, когда можно переходить в состояние ожидания
Обработчик прерывания	Используется флаг GFP_ATOMIC
Обработка нижней половины	Используется флаг GFP_ATOMIC
Необходима память для выполнения операций ПДП, можно переходить в состояние ожидания	Используются флаги (GFP_DMA | GFP_KERNEL)
Необходима память для выполнения операций ПДП, нельзя переходить в состояние ожидания	Используются флаги (GFP_DMA | GFP_ATOMIC) или выделение выполняется в более поздний или более ранний момент времени

Функция kfree()

Обратной к функции kmalloc () является функция kfree (), которая определена в файле <linux/slab.h> следующим образом.

```
void kfree(const void *ptr)
```

Функция kfree () позволяет освободить память, ранее выделенную с помощью функции kmalloc (). Вызов этой функции для памяти, которая ранее не была выделена с помощью функции kmalloc () или уже была освобождена, приводит к очень плохим последствиям, таким как освобождение памяти, которая принадлежит другим частям ядра. Так же как и в пространстве пользователя, количество операций выделения памяти должно быть равно количеству операций освобождения, чтобы предотвратить утечку памяти и другие проблемы. Следует обратить внимание, что случай вызова kfree (NULL) специально проверяется и поэтому является безопасным.

Рассмотрим пример выделения памяти в обработчике прерывания. В этом примере обработчику прерывания необходимо выделить буфер памяти для хранения входных данных. С помощью препроцессора определяется константа. BUF_SIZE, как размер буфера памяти в байтах, который, скорее всего, должен быть больше, чем несколько байт.

```
char *buf;

buf = kmalloc(BUF_SIZE, GFP_ATOMIC);
if (!buf)
        /* ошибка выделения памяти! */
```

Позже, когда память больше не нужна, нужно не забыть освободить ее с помощью вызова функции

```
kfree(buf);
```

Функция vmalloc ()

Функция vmalloc () работает аналогично функции kmalloc (), за исключением того, что она выделяет страницы памяти, которые только виртуально смежные и необязательно смежные физически. Точно так же работают и функции выделения памяти в пространстве пользователя: страницы памяти, которые выделяются с помощью функции malloc (), япляются смежными в виртуальном адресном пространстве процесса, но нет никакой гарантии, что они смежные в физической оперативной памяти. Функция kmalloc () отличается тем, что гарантирует, что страницы памяти будут физически (и виртуально) смежными. Функция vmalloc () гарантирует только, что страницы будут смежными в виртуальном адресном пространстве ядра. Это реализуется путем выделения потенциально несмежных участков физической памяти и "исправления" таблиц страниц, чтобы отобразить эту физическую память в непрерывный участок логического адресного пространства.

Большей частью, только аппаратным устройствам необходимо выделение физически непрерывных участков памяти. Аппаратные устройства существуют по другую сторону модуля управления памятью и не "понимают" виртуальной адресации. Следовательно, все области памяти, с которыми работают аппаратные устройства, должны состоять из физически смежных блоков, а не из виртуально непрерывных участков. Для участков памяти, которые используются только программным обеспечением, например буферы памяти, связанные с процессами, прекрасно подходят области памяти, которые только виртуально непрерывны. При программировании заметить разницу невозможно. Это связано с тем, что память ядром воспринимается как логически непрерывная.

Несмотря на то что физически смежные страницы памяти необходимы только в определенных случаях, большая часть кода ядра использует для выделения памяти функцию kmalloc () , а не vmalloc () . Это, в основном, делается из соображений производительности. Для того чтобы физически несмежные страницы памяти сделать смежными в виртуальном адресном пространстве, функция vmalloc () должна соответствующим образом заполнить таблицы страниц. Хуже того, страницы памяти, которые получаются с помощью функции vmalloc () , должны отображаться посредством страниц памяти, которые принадлежат к таблицам страниц (потому что

выделяемые страницы памяти физически несмежные). Это приводит к значительно менее эффективному использованию буфера TLB[4], чем в случае, когда страницы памяти отображаются напрямую. Исходя из этих соображений функция vmalloc () используется только тогда, когда она абсолютно необходима, обычно это делается для выделения очень больших областей памяти. Например, при динамической загрузке модулей ядра, модули загружаются в память, которая выделяется с помощью функции vmalloc ().

Функция vmalloc () объявлена в файле <linux/vmalloc.h> и определена в файле mm/vmalloc.c. Использование этой функции аналогично функции malloc () пространства пользователя.

```
void * vmalloc(unsigned long size)
```

Функция возвращает указатель на виртуально непрерывную область памяти размером по крайней мере size байт. В случае ошибки эта функция возвращает значение NULL. Данная функция может переводить процесс в состояние ожидания и соответственно не может вызываться в контексте прерывания или в других ситуациях, когда блокирование недопустимо.

Для освобождения памяти, выделенной с помощью функции vmalloc () , необходимо использовать функцию

```
void vfree(void *addr)
```

Эта функция освобождает участок памяти, который начинается с адреса addr и был ранее выделен с помощью функции vmalloc (). Данная функция также может переводить процесс в состояние ожидания и поэтому не может вызываться из контекста прерывания. Функция не возвращает никаких значений.

Использовать рассмотренные функции очень просто. Например, следующим образом.

```
char *buf;

buf = vmalloc (16 * PAGE_SIZE); /* получить 16 страниц памяти */
if (!buf)
        /* ошибка! Не удалось выделить память */
/*
* переменная buf теперь указывает на область памяти
* размером, по крайней мере, 16*PAGE_SIZE байт, которая состоит
* из виртуально смежных блоков памяти
*/
```

После того как память больше не нужна, необходимо убедиться, что она освобождается с помощью следующего вызова.

```
vfree (buf);
```

[4] Буфер TLB (translation lookside buffer или буфер быстрого преобразования адреса) — это аппаратный буфер памяти, который используется в большинстве аппаратных платформ для кэширования отображений виртуальных адресов памяти в физические адреса. Этот буфер позволяет существенно повысить производительность системы, так как большинство операций доступа к памяти выполняются с использованием виртуальной адресации.

Уровень слябового распределителя памяти

Выделение и освобождение структур данных — это одна из наиболее частых операций, которые выполняются в любом ядре. Для того чтобы облегчить процедуру частого выделения и освобождения данных при программировании, вводятся *списки свободных ресурсов (free list)*. Список свободных ресурсов содержит некоторый набор уже выделенных структур данных. Когда коду необходим новый экземпляр структуры данных, он может взять одну структуру из списка свободных ресурсов, вместо того чтобы выделять необходимый объем памяти и заполнять его данными соответствующей структуры. Позже, когда структура данных больше не нужна, она снова возвращается в список свободных ресурсов, вместо того чтобы освобождать память. В этом смысле список свободных ресурсов представляет собой кэш объектов, в котором хранятся объекты одного определенного, часто используемого типа.

Одна из наибольших проблем, связанных со списком свободных ресурсов в ядре, это то, что над ними нет никакого централизованного контроля. Когда ощущается недостаток свободной памяти, нет никакой возможности взаимодействовать между ядром и всеми списками свободных ресурсов, которые в такой ситуации должны уменьшить размер своего кэша, чтобы освободить память. В ядре нет никакой информации о случайно созданных списках свободных ресурсов. Для исправления положения и для универсальности кода ядро предоставляет уровень слябового распределения памяти (slab layer), который также называется просто слябовым распределителем памяти (slab allocator). Уровень слябового распределения памяти выполняет функции общего уровня кэширования структур данных.

Концепции слябового распределения памяти впервые были реализованы в операционной системе SunOS 5.4 фирмы Sun Microsystems'. Для уровня кэширования структур данных в операционной системе Linux используется такое же название и похожие особенности реализации.

Уровень слябового распределения памяти служит для достижения следующих целей.

- Часто используемые структуры данных, скорее всего, будут часто выделяться и освобождаться, поэтому их следует кэшировать.
- Частые операции выделения и освобождения памяти могут привести к фрагментации памяти (к невозможности найти большие участки физически однородной памяти). Для предотвращения этого, кэшированные списки свободных ресурсов организованы непрерывным образом в физической памяти. Так как структуры данных возвращаются снова в список свободных ресурсов, то в результате никакой фрагментации не возникает.
- Список свободных ресурсов обеспечивает улучшенную производительность при частых выделениях и освобождениях объектов, так как освобожденные объекты сразу же готовы для нового выделения.
- Если распределитель памяти может использовать дополнительную информацию, такую как размер объекта, размер страницы памяти и общий размер кэша, то появляется возможность принимать в критических ситуациях более интеллектуальные решения.

[5] И позже документированы в работе Bonwirk J."The Slab Allocator: An Object-Caching Kernel Memory Allocator," USENIX, 1994.

- Если кэш организован, как связанный с определенным процессором (т.е. для каждого процессора в системе используется свой уникальный отдельный кэш), то выделение и освобождение структур данных может выполняться без использования SMP-блокиропок.
- Если распределитель памяти рассчитан на доступ к неоднородной памяти (Non-Uniform Memory Access NUMA), то появляется возможность выделения памяти с того же узла (node), на котором эта память запрашивается.
- Хранимые объекты могут быть *"окрашены',* чтобы предотвратить отображение разных объектов на одни и те же строки системного кэша.

Уровень слябового распределения памяти в ОС Linux был реализован с учетом указанных принципов.

Устройство слябового распределителя памяти

Уровень слябового распределения памяти делит объекты па группы, которые называются *кэшами (cache).* Разные кэши используются для хранения объектов различных типов. Для каждого типа объектов существует свой уникальный кэш. Например, один кэш используется для дескрипторов процессов (список свободных структур `struct task_struct`), а другой— для индексов файловых систем (`struct inode`). Интересно, что интерфейс krnalloc () построен на базе уровня слябового распределения памяти и использует семейство кэшей общего назначения.

Далее кэши делятся на *слябы* (буквально slab — однородная плитка, отсюда и название всей подсистемы). Слябы занимают одну или несколько физически смежных страниц памяти. Обычно сляб занимает только одну страницу памяти. Каждый кэш может содержать несколько слябов.

Каждый сляб содержит некоторое количество *объектов,* которые представляют собой кэшируемые структуры данных. Каждый сляб может быть в одном из трех состояний: полный (full), частично заполненный (partial) и пустой (empty). Полный сляб не содержит свободных объектов (все объекты сляба выделены для использования). Частично заполненный сляб содержит часть выделенных и часть свободных объектов. Когда некоторая часть ядра запрашивает новый объект, то этот запрос удовлетворяется из частично заполненного сляба, если такой существует. В противном случае запрос выполняется из пустого сляба. Если пустого сляба не существует, то он создается. Очевидно, что полный сляб никогда не может удовлетворить запрос, поскольку в нем нет свободных объектов. Такая политика уменьшает фрагментацию памяти.

В качестве примера рассмотрим структуры `inode`, которые являются представлением в оперативной памяти индексов дисковых файлов (см. главу 12). Эти структуры часто создаются и удаляются, поэтому есть смысл управлять ими с помощью слябового распределителя памяти. Структуры `struct inode` выделяются из кэша `inode_cachep` (такое соглашение по присваиванию названий является стандартом). Этот кэш состоит из одного или более слябов, скорее всего слябов много, поскольку много объектов. Каждый сляб содержит максимально возможное количество объектов типа `struct inode`. Когда ядро выделяет новую структуру типа `struct inode`, возвращается указатель на уже выделенную, но не используемую структуру из частично заполненного сляба или, если такого нет, из пустого сляба. Когда ядру больше не нужен объект типа `inode`, то слябовый распределитель памяти помечает этот объект

как свободный. На рис. 11.1 показана диаграмма взаимоотношений между кэшами, слябами и объектами.

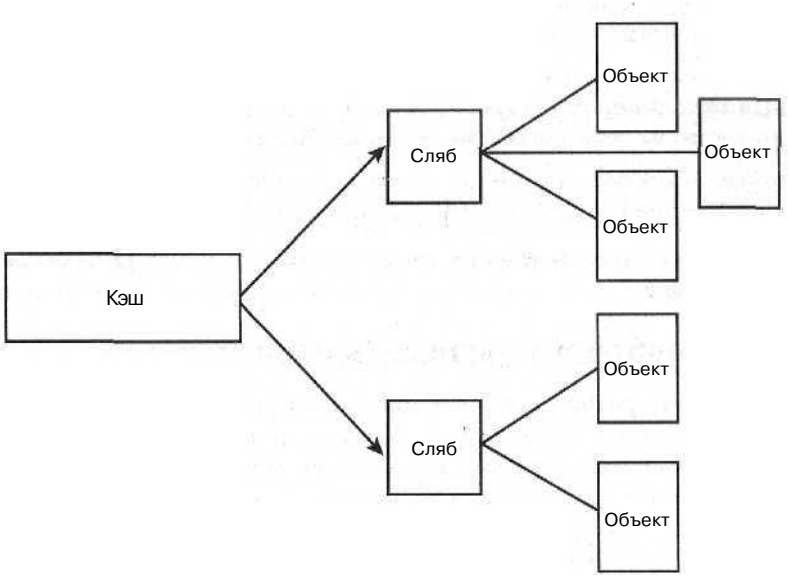


Рис. 11.1. *Взаимоотношения между кэшами, слябами и объектами*

Каждый кэш представляется структурой kmem_cache_s. Эта структура содержит три списка slab_full, slab_partial и slab_empty, которые хранятся в структуре kmem_list3. Эти списки содержат все слябы, связанные с данным кэшем. Каждый сляб представлен следующей структурой struct slab, которая является дескриптором сляба.

```
struct slab {
        struct list head list;  /* список полных, частично заполненных
                                   или пустых слябов */
        unsigned long colouroff; /* смещение для окрашивания слябов */
        void *s_mem;            /* первый объект сляба */
        unsigned int inuse;     /* количество выделенных объектов */
        kmem_bufctl_t free;     /* первый свободный объект, если есть*/
};
```

Дескриптор сляба выделяется или за пределами сляба, в кэше общего назначения, или в начале самого сляба. Дескриптор хранится внутри сляба, либо если общий размер сляба достаточно мал, либо если внутри самого сляба остается достаточно места, чтобы разместить дескриптор.

Слябовый распределитель создает новые слябы, вызывая интерфейс ядра нижнего уровня для выделения памяти __get_free_pages () следующим образом.

```
static void *kmem getpagss(kmem cache_t *cachep, int flags, int nodeid)
{
        struct page *page;
        void *addr;
        int i;
```

```
        flags |= cachep->gfpflags;
        if (likely(nodeid == -1)) {
                addr = (void*)__get_free_pages(flags, cachep->gfporder);
                if (!addr)
                        return NULL;
                page = virt_to_page(addr);
        } else {
                page = alloc_pages_node(nodeid, flags, cachep->gfporder);
                if (!page)
                        return NULL;
                addr=page_address(page);
        }

        i = (1 << cachep->gfporder);
        if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
                atomic_add(i, &slab_reclaim_pages);
        add_page_state(nr_slab, i);
        while (i--) {
                SetPageSlab(page);
                page++;
        }
        return addr;
}
```

Первый параметр этой функции указывает на определенный кэш, для которого нужны новые страницы памяти. Второй параметр содержит флаги, которые предаются в функцию `__get_free_pages ()`. Следует обратить внимание на то, как значения этих флагов объединяются с другими значениями с помощью логической операции ИЛИ. Данная операция дополняет флаги значением флагов кэша, которые используются по умолчанию и которые обязательно должны присутствовать *и* значении параметра flags. Количество страниц памяти — целая степень двойки — хранится в поле `cachep->gfporder`. Рассмотренная функция выглядит более сложной, чем это может показаться сначала, поскольку она также рассчитана на NUMA-системы (Non-Uniform Memory Access, системы с неоднородным доступом к памяти). Если параметр `nodeid` на равен `-1`, то предпринимается попытка выделить память с того же узла памяти, на котором выполняется запрос. Такое решение позволяет получить более высокую производительность для NUMA-систем. Для таких систем обращение к памяти других узлов приводит к снижению производительности.

Для образовательных целей можно убрать код, рассчитанный на NUMA-системы, и получить более простой вариант функции kraem_getpages () в следующем виде.

```
static inline void * kmem_getpages(kmem_cache_t *cachep, unsigned long flags)
{
        void *addr;

        flags |= cachep->gfpflags;
        addr = (void*) __get_free_pages(flags, cachep->gfporder);

        return addr;
}
```

Память освобождается с помощью функции kmem_freepages (), которая вызывает функцию free_pages () для освобождения необходимых страниц кэша. Конечно, назначение уровня слябового распределения — это воздержаться от выделения и освобождения страниц памяти. На самом деле слябовый распределитель использует функции выделения памяти только тогда, когда в данном кэше не доступен ни один частично заполненный или пустой сляб. Функция освобождения памяти вызывается только тогда, когда становится мало доступной памяти и система пытается освободить память или когда кэш полностью ликвидируется.

Уровень слябового распределения управляется с помощью простого интерфейса, и это можно делать отдельно для каждого кэша. Интерфейс позволяет создавать или ликвидировать новые кэши, а также выделять или уничтожать объекты в этих кэшах. Все механизмы оптимизации управления кэшами и слябами полностью управляются внутренними элементами уровня слябового распределения памяти. После того как кэш создан, слябовый распределитель памяти работает, как специализированная система создания объектов определенного типа.

Интерфейс слябового распределителя памяти

Новый кэш можно создать с помощью вызова следующей функции.

```
kmern_cache_t * kmem_cache_create (const char *name, size_t size,
        size_t offset, unsigned long flags,
        void (*ctor) (void*, kmem_cache_t *,unsigned long),
        void (*dtor) (void*, kmem_cache_t *,unsigned long))
```

Первый параметр — это строка, которая содержит имя кэша. Второй параметр — это размер каждого элемента кэша. Третий параметр — это смещение первого объекта в слябе. Он нужен для того, чтобы обеспечить необходимое выравнивание по границам страниц памяти. Обычно достаточно указать значение, равное нулю, которое соответствует выравниванию по умолчанию. Параметр flags указывает опциональные параметры, которые управляют поведением кэша. Он может быть равен нулю, что выключает все специальные особенности поведения, или состоять из одного или более значений, показанных ниже и объединенных с помощью логической операции ИЛИ.

- SLAB_NO_REAP — этот флаг указывает, что кэш не должен автоматически "убирать мусор" (т.е. освобождать память, в которой хранятся неиспользуемые объекты) при нехватке памяти в системе. Обычно этот флаг *не нужно* устанавливать, поскольку если этот флаг установлен, то созданный кэш может помешать нормальной работе системы при нехватке памяти.
- SLAB_HWCACHE_ALIGN — этот флаг указывает уровню слябового распределения памяти, что расположение каждого объекта в слябе должно выравниваться по строкам процессорного кэша. Это предотвращает так называемое "ошибочное распределение", когда два или более объектов отображаются в одну и ту же строку системного кэша, несмотря на то что они находятся по разным адресам памяти. При использовании этого флага возрастает производительность, но это делается ценой увеличения занимаемой памяти, потому что строгое выравнивание приводит к тому, что часть памяти сляба не используется. Степень

увеличения занимаемой памяти зависит от размера объектов кэша и от того, каким образом происходит их выравнинание по отношению к строкам системного кэша. Для кэшей, которые часто используются в коде, критичном к производительности, будет правильным установить этот флаг, в остальных случаях следует подумать, стоит ли это делать.

- SLAB_MUST_HWCACHE_ALIGN. Если включен режим отладки, то может оказаться невозможным одновременная отладка и выравнивание положения объектов по строкам системного кэша. Этот флаг указывает уровню слябового распределения памяти, что необходимо форсировать выравнивание положения объектов по строкам системного кэша. В обычной ситуации этот флаг указывать необязательно, и достаточно использовать предыдущий. Установка этого флага в режиме отладки слябового распределителя памяти (по умолчанию отладка запрещена) может привести к значительному увеличению затрат памяти. Только для объектов, которые критичны к выравниванию по строкам системного кэша, таких как дескрипторы процессов, необходимо указывать данный флаг.
- SLABPOSON— этот флаг указывает на необходимость заполнения слябов известным числовым значением (a5a5a5a5). Эта операция называется "отравлением" *(poisoning)* и служит для отслеживания доступа к неинициализированной памяти.
- SLAB_RED_ZONE — этот флаг указывает на необходимость выделения так называемых "красных зон" *(red zone)* для облегчения детектирования переполнений буфера.
- SLAB_PANIC — этот флаг указывает на необходимость перевода ядра в состояние паники, если выделение памяти было неудачным. Данный флаг полезен, если выделение памяти всегда должно завершаться успешно, как, например, в случае создания кэша структур VMA (областей виртуальной памяти, см. главу 14, "Адресное пространство процесса") при загрузке системы.
- SLAB_CACHE_DMA — этот флаг указывает уровню слябового распределения, что все слябы должны выделяться в памяти, с которой возможны операции прямого доступа к памяти. Это необходимо, когда объекты используются в операциях ПДП и должны находиться в зоне ZONE_DMA. В противном случае эта возможность не нужна и этот флаг не нужно устанавливать.

Два последних параметра `ctor` и `dtor` — это конструктор и деструктор кэша соответственно. Конструктор вызывается, когда в кэш добавляются новые страницы памяти. Деструктор вызывается, когда из кэша удаляются страницы памяти. Если указан деструктор, то должен быть указан и конструктор. На практике кэши ядра ОС Linux обычно не используют функции конструктора и деструктора. В качестве этих параметров можно указывать значение NULL.

В случае успешного выполнения функция `kmem_cache_create ()` возвращает указатель на созданный кэш. В противном случае возвращается NULL. Данная функция не может вызываться в контексте прерывания, так как она может переводить процесс в состояние ожидания. Для ликвидации кэша необходимо вызвать следующую функцию.

```
int kmem_cache_destroy(kmem_cache_t *cachep)
```

Эта функция ликвидирует указанный кэш. Она обычно вызывается при выгрузке модуля, который создает свой кэш. Из контекста прерывания эту функцию вызывать нельзя, так как она может переводить вызывающий процесс в состояние ожидания. Перед вызовом этой функции необходимо, чтобы были выполнены следующие два условия.

- Все слябы кэша являются пустыми. Действительно, если в каком-либо слябе существует объект, который все еще используется, то как можно ликвидировать кэш?
- Никто не будет обращаться к кэшу во время и особенно после вызова функции kmem_cache_destroy (). Эту синхронизацию должен обеспечить вызывающий код.

В случае успешного выполнения функция возвращает нуль, в других случаях возвращается ненулевое значение.

После того как кэш создан, из него можно получить объект путем вызова следующей функции.

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

Эта функция возвращает указатель на объект из кэша, на который указывает параметр cachep. Если ни в одном из слябов нет свободных объектов, то уровень слябового распределения должен получить новые страницы памяти с помощью функции kmem_getpages (), значение параметра flags передается *в* функцию__get_free_pages (). Это те самые флаги, которые были рассмотрены ранее. Скорее всего, необходимо указывать GFP_KERNEL или GFP_ATOMIC.

Далее для удаления объекта и возвращения его в сляб, из которого он был выделен, необходимо использовать следующую функцию.

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

Данная функция помечает объект, на который указывает параметр objp, как свободный.

Пример использования слябового распределителя памяти

Давайте рассмотрим пример из реальной жизни, связанный с работой со структурами task_struct (дескрипторы процессов). Показанный ниже код в несколько более сложной форме приведен в файле kernel/fork.c.

В ядре определена глобальная переменная, в которой хранится указатель на кэш объектов task_struct:

```
kmem_cache_t *task_struct_cachep;
```

Во время инициализации ядра, в функции forkinit (), этот кэш создается следующим образом.

```
task_struct_cachep = kmem_cache_create("task_struct",
                                sizeof(struct task_struct),
                                ARCH_M1N_TASKALIGN,
                                SLAB_PANIC,
                                NULL,
                                NULL);
```

Данный вызов создает кэш с именем "task_struct", который предназначен для хранения объектов тина struct task_struct. Объекты создаются с начальным смещением в слябе, равным ARCH_MIN_TASKALIGN байт, и положение всех объектов выравнивается по границам строк системного кэша, значение этого выравнивания зависит от аппаратной платформы. Обычно значение выравнивания задается для каждой аппаратной платформы с помощью определения препроцессора LI_CACHE_BYTES, которое равно размеру процессорного кэша первого уровня в байтах. Конструктор и деструктор отсутствуют. Следует обратить внимание, что возвращаемое значение не проверяется на рапенство NULL, поскольку указан флаг SLAB_PANIC. В случае, когда при выделении памяти произошла ошибка, слябовый распределитель памяти вызовет функцию panic (). Если этот флаг не указан, то нужно проверять возвращаемое значение на равенство NULL, что сигнализирует об ошибке. Флаг SLAB_PANIC здесь используется потому, что этот каш является необходимым для работы системы (без дескрипторов процессов работать как-то не хорошо).

Каждый раз, когда процесс вызывает функцию fork (), должен создаваться новый дескриптор процесса (вспомните главу 3, "Управление процессами"). Это выполняется следующим образом в функции dup_task_struct () , которая вызывается из функции do_fork () .

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task struct_cachep, GFP_KERNEL);
if (!tsk)
       return NULL;
```

Когда процесс завершается, если нет порожденных процессов, которые ожидают на завершение родительского процесса, то дескриптор освобождается и возвращается обратно в кэш task_struct_cachep. Эти действия выполняются в функции free_task_struct (), как показано ниже (где параметр tsk указывает на удаляемый дескриптор).

```
kmem_cache_free(task_struct_cachep, tsk);
```

Так как дескрипторы процессов принадлежат к основным компонентам ядра и всегда необходимы, то кэш task_struct_cachep никогда не ликвидируется. Если бы он ликвидировался, то делать это необходимо было бы следующим образом.

```
int err;

err = kmem_cache_destroy (task_struct_cachep);
if (err)
       /* ошибка ликвидации кэша */
```

Достаточно просто, не так ли? Уровень слябопого распределения памяти скрывает все низкоуровневые операции, связанные с выравниванием, "раскрашипанием", выделением и освобождением памяти, "сборкой мусора" в случае нехватки памяти. Коли часто необходимо создавать много объектов одного типа, то следует подумать об использовании слябового кэша. И уж точно не нужно писать свою реализацию списка свободных ресурсов!

Статическое выделение памяти в стеке

В пространстве пользователя многие операции выделения памяти, в частности некоторые рассмотренные ранее примеры, могут быть выполнены с использованием стека, потому что априори известен размер выделяемой области памяти. В пространстве пользователя доступна такая роскошь, как очень большой и динамически увеличивающийся стек задачи, однако в режиме ядра такой роскоши нет — стек ядра маленький и фиксирован по размеру. Когда процессу выделяется небольшой и фиксированный по размеру стек, то затраты памяти уменьшаются и ядру нет необходимости выполнять дополнительные функции по управлению памятью.

Размер стека зависит как от аппаратной платформы, так и от конфигурационных параметров, которые были указаны на этапе компиляции. Исторически размер стека ядра был равен двум страницам памяти для каждого процесса. Это соответствует 8 Кбайт для 32-разрядных аппаратных платформ и 16 Кбайт для 64-разрядных аппаратных платформ.

В первых версиях ядер серии 2.6 была введена возможность конфигурации, для которой размер стека ядра равен одной странице памяти. Когда устанавливается такая конфигурация, то процесс получает стек, по размеру равный всего одной странице памяти: 4 Кбайт на 32-разрядных аппаратных платформах и 8 Кбайт — на 64-разрядных. Это сделано по двум причинам. Во-первых это уменьшает затраты памяти на одну страницу для каждого процесса. Во-вторых, что наиболее важно, при увеличении времени работы системы (uptime) становится все тяжелее искать две физически смежные страницы памяти. Физическая память становится все более фрагментированной, и нагрузка на систему управления виртуальной памятью при создании новых процессов становится все более существенной.

Существует еще одна сложность (оставайтесь с нами, и Вы узнаете все о стеках ядра). Вся последовательность вложенных вызовов функций в режиме ядра должна поместиться в стеке. Исторически обработчики прерываний используют стек того процесса, выполнение которого они прервали. Это означает, что в худшем случае 8 Кбайт стека должно использоваться совместно всеми вложенными вызовами функций и еще парой обработчиков прерываний. Все это эффективно и просто, но это накладывает еще больше ограничений на использование стека ядра. Когда размер стека сократился до одной страницы памяти, обработчики прерываний туда перестали помещаться.

Для решения указанной проблемы была реализована еще одна возможность — стеки обработчиков прерываний. Стеки прерываний представляют собой один стек на каждый процессор, которые используются для обработки прерываний. При такой конфигурации обработчики прерываний больше не используют стеки ядра тех процессов, которые этими обработчиками прерываются. Вместо этого они используют свои собственные стеки. Это требует только одну страницу памяти на процессор.

Подведем итоги. Стек ядра занимает одну или две страницы памяти, в зависимости от конфигурации, которая выполняется перед компиляцией ядра. Следовательно, размер стека ядра может иметь диапазон от 4 до 16 Кбайт. Исторически обработчики прерываний совместно использовали стек прерванного ими процесса. При появлении стеков ядра размером в одну страницу памяти обработчикам прерываний были назначены свои стеки. В любом случае неограниченная рекурсия и использование функций вроде `alloca()` явно не допустимы.

Честная игра со стеком

В любой функции необходимо сокращать использование стека до минимума. Хотя не существует твердых правил, тем не менее следует поддерживать максимальный суммарный объем всех локальных переменных (также известных как автоматические переменные или переменные, выделенные в стеке) не больше нескольких сотен байтов. Опасно статически выделять большие объекты в стеке, такие как большие массивы структур. В противном случае выделение памяти в стеке будет выполняться так же, как и в пространстве пользователя. Переполнение стека происходит незаметно и обычно приводит к проблемам. Так как ядро не выполняет никакого управления стеком, то данные стека просто перепишут все, что находится за стеком. В первую очередь пострадает структура `thread_info`, которая расположена в самом конце стека процесса (вспомните главу 3). За пределами стека все данные ядра могут пропасть. В лучшем случае при переполнении стека произойдет сбой в работе машины. В худших случаях может произойти повреждение данных.

В связи с этим, для выделения больших объемов памяти необходимо использовать одну из динамических схем выделения памяти, которые были рассмотрены раньше в этой главе.

Отображение верхней памяти

По определению, страницы верхней памяти не могут постоянно отображаться в адресное пространство ядра. Поэтому страницы памяти, которые были выделены с помощью функции `alloc_pages ()`, при использовании флага `__GFP_HIGHMEM` могут не иметь логического адреса.

Для аппаратной платформы x86 вся физическая память свыше 896 Мбайт помечается как верхняя память, и она не может автоматически или постоянно отображаться в адресное пространство ядра, несмотря на то что процессоры платформы x86 могут адресовать до 4 Гбайт физической памяти (до 64 Гбайт при наличии расширения PAE[6]). После выделения эти страницы должны быть отображены в логическое адресное пространство ядра. Для платформы x86 страницы верхней памяти отображаются где-то между отметками 3 и 4 Гбайт.

Постоянное отображение

Для того чтобы отобразить заданную структуру `page` в адресное пространство ядра, необходимо использовать следующую функцию.

```
void *kmap(struct page *page)
```

Эта функция работает как со страницами нижней, так и верхней памяти. Если структура `page` соответствует странице нижней памяти, то просто возвращается виртуальный адрес. Если страница расположена в верхней памяти, то создается постоянное отображение этой страницы памяти и возвращается полученный логический

[6] PAE — Physical Address Extension (расширение физической адресации). Эта функция процессоров x86 позволяет физически адресовать до 36 разрядов (64 Гбайт) памяти, несмотря на то что размер виртуального адресного пространства соответствует только 32 бит.

адрес. Функция kmap () может переводить процесс в состояние ожидания, поэтому ее можно вызывать только в контексте процесса.

Поскольку количество постоянных отображений ограничено (если бы это было не так, то мы бы не мучились, а просто отобразили всю необходимую память), то отображение страниц верхней памяти должно быть отменено, если оно больше не нужно. Это можно сделать с помощью вызова следующей функции.

```
void kunmap(struct page *page)
```

Данная функция отменяет отображение страницы памяти, связанной с параметром page.

Временное отображение

В случаях, когда необходимо создать отображение страниц памяти в адресное пространство, а текущий контекст не может переходить в состояние ожидания, ядро предоставляет функцию *временного отображении* (которое также называется *атомарным отображением).* Существует некоторое количество зарезервированных постоянных отображений, которые могут временно выполнять отображение "на лету". Ядро может автоматически отображать страницу верхней памяти в одно из зарезервированных отображений. Временное отображение может использоваться в коде, который не может переходить в состояние ожидания, как, например, контекст прерывания, потому что полученное отображение никогда не блокируется.

Установка временного отображения выполняется с помощью следующей функции.

```
void *kmap_atomic(struct page *page, enum km_type type)
```

Параметр type — это одно из значений показанного ниже перечисления, определенного в файле <asm/kmap_types.h>, которое описывает цель временного отображения.

```
enum km_type {
        KM_BOUNCE_READ,
        KM_SKB_SUNRPC_DATA,
        KM_SKB_DATA_SOFTIRQ,
        KM_USER0,
        KM_USER1,
        KM_BIO_SRC_IRQ,
        KM_BIO_DST_IRQ,
        KM_PTE0,
        KM_PTE1,
        KM_PTE2,
        KM_IRQ0,
        KM_IRQ1,
        KM_SOFTIRQ0,
        KM_SOFTIRQ1,
        KM TYPE_NR
);
```

Данная функция не блокируется и поэтому может использоваться в контексте прерывания и в других случаях, когда нельзя перепланировать выполнение. Эта

функция также запрещает преемптивность ядра, что необходимо потому, что отображения являются уникальными для каждого процессора (а перепланирование может переместить задание для выполнения на другом процессоре).

Отменить отображение можно с помощью следующей функции.

```
void kunmap_atomic(void *kvaddr, enum km_type type)
```

Эта функция также не блокирующая. На самом деле для большинства аппаратных платформ она ничего не делает, за исключением разрешения преемптивности ядра, потому что временное отображение действует только до тех пор, пока не создано новое временное отображение. Поэтому ядро просто "забывает" о вызове функции kmap_atomic () , и функции kunmap atomic () практически ничего не нужно делать. Следующее атомарное отображение просто заменяет предыдущее.

Выделение памяти, связанной с определенным процессором

В современных операционных системах широко используются данные, связанные с определенными процессорами (per-CPU data). Это данные, которые являются уникальными для каждого процессора. Данные, связанные с процессорами, хранятся в массиве. Каждый элемент массива соответствует своему процессору системы. Номер процессора является индексом в этом массиве. Таким образом была реализована работа с данными, связанными с определенным процессором, в ядрах серии 2.4. В таком подходе нет ничего плохого, поэтому значительная часть кода ядра в серии 2.6 все еще использует этот интерфейс. Данные объявляются следующим образом,

```
unsigned long my_percpu[NR_CPUS];
```

Доступ к этим данным выполняется, как показано ниже.

```
int cpu;

cpu = get_cpu();    /* получить номер текущего процессора и запретить
                       вытеснение в режиме ядра */
my_percpu[cpu]++;
printk("значение данных процессора cpu=%d равно %ld\n",
        cpu, my_percpu[cpu]);
put_cpu(); /* разрешить вытеснение в режиме ядра */
```

Обратите внимание, что не нужно использовать никаких блокировок, потому что данные уникальны для каждого процессора. Поскольку никакой процессор, кроме текущего, не может обратиться к соответствующему элементу данных, то не может возникнуть и никаких проблем с конкурентным доступом, а следовательно, текущий процессор может безопасно обращаться к данным без блокировок.

Возможность вытеснения процессов в режиме ядра— единственное, из-за чего могут возникнуть проблемы. В преемптивном ядре могут возникнуть следующие две проблемы.

- Если выполняющийся код вытесняется и позже планируется для выполнения на другом процессоре, то значение переменной cpu больше не будет действительным, потому что эта переменная будет содержать номер другого процессо-

ра. (По той же причине, после получения номера текущего процессора, нельзя переходить в состояние ожидания.)

- Если некоторый другой код вытеснит текущий, то он может параллельно обратиться к переменной my_percpu на том же процессоре, что соответствует состоянию гонок за ресурс.

Однако все опасения напрасны, потому что вызов функции `getcpu ()`, которая возвращает номер текущего процессора, также запрещает вытеснение в режиме ядра. Соответствующий вызов функции `put_cpu ()` разрешает вытеснение кода в режиме ядра. Обратите внимание, что функция `smp_processor_icl ()`, которая также позволяет получить номер текущего процессора, не запрещает вытеснения кода в режиме ядра, поэтому для безопасной работы следует использовать указанный выше метод.

Новый интерфейс percpu

В ядрах серии 2.6 предложен новый интерфейс, именуемый *percpu,* который служит для создания данных и работы с данными, связанными с определенным процессором. Этот интерфейс обобщает предыдущий пример. При использовании нового подхода работа с per-CPU-данными упрощается.

Рассмотренный ранее метод работы с данными, которые связаны с определенным процессором, является вполне законным. Новый интерфейс возник из необходимости иметь более простой и мощный метод работы с данными, связанными с процессорами, на больших компьютерах с симметричной мультипроцессорностью.

Все подпрограммы объявлены в файле `<linux/percpu.h>`. Описания же находятся в файлах `mm/slab.c` и `<asm/percpu.h>`.

Работа с данными, связанными с процессорами, на этапе компиляции

Описать переменную, которая связана с определенным процессором, на этапе компиляции можно достаточно просто следующим образом.

```
DEFINE_PER_CPU(type, name);
```

Это описание создает переменную типа `type` с именем `name`, которая имеет интерфейс связи с каждым процессором в системе. Если необходимо объявить соответствующую переменную с целью избежания предупреждений компилятора, то необходимо использовать следующий макрос.

```
DECLARE_PER_CPU(type, name);
```

Работать с этими переменными можно с помощью функций `get_cpu_var ()` и `put_cpu_var ()`. Вызов функции `get_cpu_var ()` возвращает l-значение (левый операнд, l-value) указанной переменной на текущем процессоре. Этот вызов также запрещает вытеснение кода в режиме ядра, а соответственный вызов функции `put_cpu_var ()` разрешает вытеснение.

```
get_cpu_var(name)++;    /* увеличить на единицу значение переменной
                           name, связанное с текущим процессором */
put_cpu_var();         /* разрешить вытеснение кода в режиме ядра */
```

Можно также получить доступ к переменной, связанной с *другим* процессором.

```
per_cpu(name, cpu)++;   /* увеличить значение переменной name
                           на указанном процессоре */
```

Использовать функцию per_cpu () необходимо осторожно, так как этот вызов не запрещает вытеснение кода и не обеспечивает никаких блокировок. Необходимость использования блокировок при работе с данными, связанными с определенным процессором, отпадает, только если к этим данным может обращаться один процессор. Если процессоры обращаются к данным других процессоров, то необходимо использовать блокировки. Будьте осторожны! Применение блокировок рассматривается в главе 8, "Введение в синхронизацию выполнения кода ядра", и главе 9, "Средства синхронизации в ядре".

Необходимо сделать еще одно важное замечание относительно создания данных. связанных с процессорами, на этапе компиляции. Загружаемые модули не могут использовать те из них, которые объявлены не в самом модуле, потому что компоновщик создает эти данные в специальных сегментах кода (а именно, .data.percpu). Если необходимо использовать данные, связанные с процессорами, в загружаемых модулях ядра, то нужно создать эти данные для каждого модуля отдельно или использовать динамически создаваемые данные.

Работа с данными процессоров на этапе выполнения

Для динамического создания данных, связанных с процессорами, в ядре реализован специальный распределитель памяти, который имеет интерфейс, аналогичный kmalloc () . Эти функции позволяют создать экземпляр участка памяти для каждого процессора в системе. Прототипы этих функций объявлены в файле <linux/percpu.h> следующим образом.

```
void *alloc percpu(type);  /* макрос */
void *__alloc_percpu(size_t size, size_t align);
voidfree_percpu(constvoid* ) ;
```

Функция alloc percpu () создает экземпляр объекта заданного типа (выделяет память) для каждого процессора в системе. Эта функция является оболочкой вокруг функции __alloc_percpu (). Последняя функция принимает в качестве аргументов количество байтов памяти, которые необходимо выделить, и количество байтов, по которому необходимо выполнить выравнивание этой области памяти. Функция alloc_percpu () выполняет выравнивание по той границе, которая используется для указанного типа данных. Такое выравнивание соответствует обычному поведению, как показано в следующем примере.

```
struct rabid_cheetah = alloc_percpu(struct rabid_cheetah);,
```

что аналогично следующему вызову.

```
struct rabid_cheetah = __alloc_percpu(sizeof (struct rabid_cheetah),
                                      alignof  (struct rabid_cheetah));
```

Оператор __alignof__ — это расширение, предоставляемое компилятором gcc, который возвращает количество байтов, по границе которого необходимо выполнять выравнивание (или рекомендуется выполнять для тех аппаратных платформ, у которых нет жестких требований к выравниванию данных в памяти). Синтаксис этого вызова такой же как и у оператора sizeof (). В примере, показанном ниже, для аппаратной платформы x86 будет возвращено значение 4.

```
__alignof__ (unsigned long)
```

При передаче l-значения (левое значение, lvalue) возвращается максимально возможное выравнивание, которое может потребоваться для этого l-значения. Например, l-значение внутри структуры может иметь большее значение выравнивания, чем это необходимо для хранения того же типа данных за пределами структуры, что связано с особенностями выравнивания структур данных в памяти. Проблемы выравнивания более подробно рассмотрены в главе 19, "Переносимость".

Соответствующий вызов функции freepercpu () освобождает память, которую занимают соответствующие данные на всех процессорах.

Функции alloc_percpu () и __alloc_percpu () возвращают указатель, который используется для косвенной ссылки на динамически созданные данные, связанные с каждым процессором в системе. Для простого доступа к данным ядро предоставляет два следующих макроса.

```
get_cpu_ptr(ptr) ;   /* возвращает указатель типа void на данные,
        соответствующие параметру ptr, связанные с текущим процессом*/
put_cpu_ptr(ptr);   /* готово, разрешаем вытеснение кода в режиме ядра
*/
```

Макрос get_cpu_ptr () возвращает указатель на экземпляр данных, связанных с текущим процессором. Этот вызов также запрещает вытеснение кода в режиме ядра, которое снова разрешается вызовом функции put_cpu_ptr ().

Рассмотрим пример использования этих функций. Конечно, этот пример не совсем логичный, потому что память обычно необходимо выделять один раз (например, в некоторой функции инициализации), использовать ее в разных необходимых местах, а затем освободить также один раз (например, в некоторой функции, которая вызывается при завершении работы). Тем не менее этот пример позволяет пояснить особенности использования.

```
void*percpu_ptr;
unsigned long *foo;

percpu_ptr = alloc_percpu(unsigned long);
if (!ptr)
       /* ошибка выделения памяти .. */

foo = get_cpu_ptr(percpu_ptr);
/* работаем с данными foo .. */
put_cpu_ptr(percpu_ptr);
```

Еще одна функция — per_cpu_ptr() — возвращает экземпляр данных, связанных с указанным процессором.

```
per_cpu_ptr(ptr, cpu);
```

Эта функция не запрещает вытеснение в режиме ядра. Если вы "трогаете" данные, связанные с другим процессором, то, вероятно, необходимо применить блокировки.

Когда лучше использовать данные, связанные с процессорами

Использование данных, связанных с процессорами, позволяет получить ряд преимуществ. Во-первых, это ослабление требований по использованию блокировок. В зависимости от семантики доступа к данным, которые связаны с процессорами, может оказаться, что блокировки вообще не нужны. Следует помнить, что правило *"только один процессор может обращаться к этим данным"* является всего лишь рекомендацией для программиста. Необходимо специально гарантировать, что каждый процессор работает только со своими данными. Ничто не может помешать нарушению этого правила.

Во-вторых, данные, связанные с процессорами, позволяют существенно уменьшить недостоверность данных, хранящихся в кэше. Это происходит потому, что процессоры поддерживают свои кэши в синхронизированном состоянии. Если один процессор начинает работать с данными, которые находятся в кэше другого процессора, то первый процессор должен обновить содержимое своего кэша. Постоянное аннулирование находящихся в кэше данных, именуемое перегрузкой кэша (cash thrashing), существенно снижает производительность системы. Использование данных, связанных с процессорами, позволяет приблизить эффективность работы с кэшем к максимально возможной, потому что в идеале каждый процессор работает только со своими данными.

Следовательно, использование данных, которые связаны с процессорами, часто избавляет от необходимости использования блокировок (или снижает требования, связанные с блокировками). Единственное требование, предъявляемое к этим данным для безопасной работы, — это запрещение вытеснения кода, который работает в режиме ядра. Запрещение вытеснения — значительно более эффективная операция по сравнению с использованием блокировок, а существующие интерфейсы выполняют запрещение и разрешение вытеснения автоматически. Данные, связанные с процессорами, можно легко использовать как в контексте прерывания, так и в контексте процесса. Тем не менее следует обратить внимание, что при использовании данных, которые связаны с текущим процессором, нельзя переходить в состояние ожидания {в противном случае выполнение может быть продолжено на другом процессоре).

Сейчас нет строгой необходимости где-либо использовать новый интерфейс работы с данными, которые связаны с процессорами. Вполне можно организовать такую работу вручную (на основании массива, как было рассказано ранее), если при этом запрещается вытеснение кода в режиме ядра. Тем не менее новый интерфейс более простой в использовании и, возможно, позволит в будущем выполнять дополнительные оптимизации. Если вы собираетесь использовать в своем коде данные, связанные с процессорами, то лучше использовать новый интерфейс. Единственный *недостаток* нового интерфейса — он не совместим с более ранними версиями ядер.

Какой способ выделения памяти необходимо использовать

Если необходимы смежные страницы физической памяти, то нужно использовать один из низкоуровневых интерфейсов выделения памяти, или функцию kmalloc(). Это стандартный способ выделения памяти в ядре, и, скорее всего, в большинстве случаев следует использовать именно его. Необходимо вспомнить, что два наиболее часто встречающихся флага, которые передаются этой функции, это флаги GFP_ATOMIC и GFP_KERNEL. Для высокоприоритетных операций выделения памяти, которые не переводят процесс в состояние ожидания, необходимо указывать флаг GFP_ATOMIC. Это обязательно для обработчиков прерываний и других случаев, когда нельзя переходить в состояние ожидания. В коде, который может переходить в состояние ожидания, как, например код, выполняющийся в контексте процесса и не удерживающий спин-блокировку, необходимо использовать флаг GFP_KERNEL. Такой флаг указывает, что должна выполняться операция выделения памяти, которая при необходимости может перейти в состояние ожидания для получения необходимой памяти.

Если есть необходимость выделить страницы верхней памяти, то следует использовать функцию alloc_pages(). Функция alloc_pages () возвращает структуру struct page, а не логический адрес. Поскольку страницы верхней памяти могут не отображаться в адресное пространство ядра, единственный способ доступа к этой памяти — через структуру struct page. Для получения "настоящего" указателя на область памяти необходимо использовать функцию kmap(), которая позволяет отобразить верхнюю память в логическое адресное пространство ядра.

Если нет необходимости в физически смежных страницах памяти, а необходима только виртуально непрерывная область памяти, то следует использовать функцию vmalloc() (также следует помнить о небольшой потере производительности при использовании функции vmalloc() по сравнению с функцией kmalloc()) Функция vmalloc() выделяет область памяти, которая содержит только виртуально смежные страницы, но не обязательно физически смежные. Это выполняется почти так же, как и в программах пользователя путем отображения физически несмежных участков памяти в логически непрерывную область памяти.

Если необходимо создавать и освобождать много больших структур данных, то следует рассмотреть возможность построения слябового кэша. Уровень слябового распределения памяти позволяет поддерживать кэш объектов (список свободных объектов), уникальный для каждого процессора, который может значительно улучшить производительность операций выделения и освобождения объектов. Вместо того чтобы часто выделять и освобождать память, слябовый распределитель сохраняет кэш уже выделенных объектов. При необходимости получения нового участка памяти для хранения структуры данных, уровню слябового распределения часто нет необходимости выделять новые страницы памяти, вместо этого можно просто возвращать объект из кэша.

12

Виртуальная файловая система

Виртуальная файловая система (Virtual File System), иногда называемая виртуальным файловым коммутатором *(Virtual File Switch)* или просто *VFS,* — это подсистема ядра, которая реализует интерфейс пользовательских программ к файловой системе. Все файловые системы зависят от подсистемы VFS, что позволяет не только сосуществовать разным файловым системам, но и совместно функционировать. Это также дает возможность использовать стандартные системные вызовы для чтения и записи данных на различные файловые системы, которые находятся на различных физических носителях, как показано на рис. 12.1,

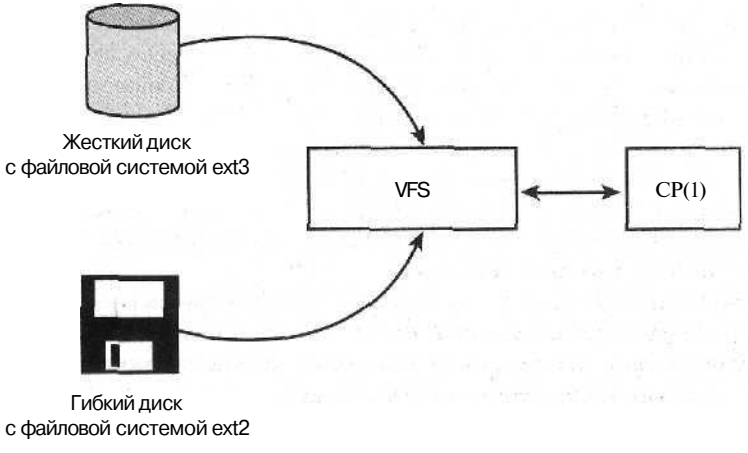


Рис. 12.1. Подсистема VFS в действии: использование команды ср (1) для копирования данных с жесткого диска, на котором монтируется файловая система, ext3, на гибкий диск, на котором монтируется файловая система ext2

Общий интерфейс к файловым системам

Подсистема VFS— это связующее звено, которое позволяет таким системным вызовам, как open (), read () и write (), работать независимо от файловой системы и физической среды носителя информации. Сегодня это может не впечатлять, поскольку такая возможность принимается как должное. Тем не менее сделать так, чтобы общие системные вызовы работали для всех поддерживаемых файловых систем и физических сред хранения данных, — задача не тривиальная. Более того, эти системные вызовы позволяют выполнять операции *между* различными файловыми системами и различными физическими носителями — мы можем копировать и перемещать данные с одной файловой системы на другую с помощью стандартных системных вызовов. В старых операционных системах (например, DOS) таких возможностей не было. Любые операции доступа к "неродным" файловым системам требовали использования специальных утилит. Сейчас такие возможности существуют, потому что все современные операционные системы, включая Linux, абстрагируют доступ к файловым системам с помощью виртуального интерфейса, который дает возможность совместной работы с данными и обобщенного доступа к данным. В операционной системе Linux может появиться поддержка новых типов файловых систем или новых физических средств хранения данных, при этом нет необходимости переписывать или перекомпилировать существующие программы.

Уровень обобщенной файловой системы

Общий интерфейс для всех типов файловых систем возможен только благодаря тому, что в ядре реализован обобщающий уровень, который скрывает низкоуровневый интерфейс файловых систем. Данный обобщающий уровень позволяет операционной системе Linux поддерживать различные файловые системы, даже если эти файловые системы существенно отличаются друг от друга своими функциями и особенностями работы. Это в свою очередь становится возможным благодаря тому, что подсистема VFS реализует общую файловую модель, которая в состоянии представить общие функции и особенности работы потенциально возможных файловых систем. Конечно, эта модель имеет уклон в сторону файловых систем в стиле Unix (что представляют собой файловые системы в стиле Unix, будет рассказано в следующем разделе). Несмотря на это в ОС Linux поддерживается довольно большой диапазон различных файловых систем.

Обобщенный уровень работает путем определения базовых интерфейсов и структур данных, которые нужны для поддержки всех файловых систем. Код поддержки каждой файловой системы должен формировать все концепции своей работы в соответствии с шаблонными требованиями подсистемы VFS, например *"так открываем файл"*, а *"так представляем каталог"*. Код файловой системы скрывает все детали реализации. По отношению к уровню VFS и остальным частям ядра все файловые системы выглядят одинаково, т.е. все файловые системы начинают поддерживать такие объекты, как файлы и каталоги, и такие операции, как создание и удаление файла.

В результате получается общий уровень абстракции, который позволяет ядру легко и просто поддерживать множество типов файловых систем. Код файловых систем программируется таким образом, чтобы поддерживать общие интерфейсы и структуры данных, которые нужны для работы с виртуальной файловой системой.

В свою очередь, ядро легко может работать со всеми файловыми системами, и соответственно, экспортируемый ядром интерфейс пользователя также позволяет аналогично работать со всеми файловыми системами.

В ядре нет необходимости поддерживать низкоуровневые детали реализации файловых систем нигде, кроме кода самих файловых систем. Например, рассмотрим следующую простую программу, работающую в пространстве пользователя.

```
write(f, &buf, len);
```

Этот системный вызоп записывает len байт из области памяти по адресу &buf в файл, представленный с помощью дескриптора f, начиная с текущей позиции файла. Этот системный вызов вначале обрабатывается общей функцией ядра sys_write (), которая определяет функцию записи в файл для той файловой системы, на которой находится файл, представленный дескриптором f. Далее общий системный вызов вызывает найденную функцию, которая является частью реализации файловой системы и служит для записи данных на физический носитель (или для других действий, которые файловая система выполняет при записи файла). На рис. 12.2 показана диаграмма выполнения операции записи, начиная от пользовательской функции write () и заканчивая поступлением данных на физический носитель. Далее в этой главе будет показано, как подсистема VFS позволяет достичь необходимой абстракции и какие для этого обеспечиваются интерфейсы.

Файловые системы Unix

Исторически так сложилось, что ОС Unix обеспечивает четыре абстракции, связанные с файловыми системами: файлы, элементы каталогов (directory entry), индексы (inode) и точки монтирования (mount point).

Файловая система — это иерархическое хранилище данных определенной структуры. Файловые системы содержат файлы, каталоги и соответствующую управляющую информацию. Обычные операции, которые выполняются с файловыми системами, — это создание (create), удаление (delete) и монтирование (mount). В ОС Unix файловые системы монтируются на определенную точку монтирования в общей иерархии[1], которая называется *пространством имен (namespace).* Это позволяет все файловые системы сделать элементами одной древовидной структуры[2].

Файл (file) — это упорядоченный поток байтов. Первый байт соответствует началу файла, а последний байт - концу файла. Каждому файлу присваивается удобочитаемое имя, по которому файл идентифицируется как пользователями, так и системой. Обычные файловые операции— это чтение (read), запись (write), создание (create) и удаление (delete).

---

[1] Сейчас в операционной системе Linux эта иерархическая структура является уникальной для каждого процесса, т.е. каждый процесс имеет свое пространство имен. По умолчанию каждый процесс наследует пространство имен своего родительского процесса,. поэтому кажется, что существует одно глобальное пространство имен.

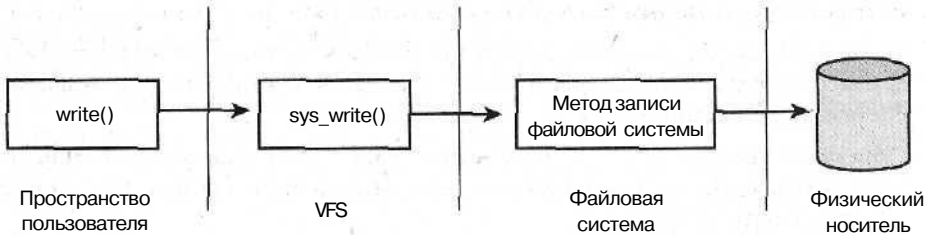
[2] В отличие от указания буквы, которая соответствует определенному диску, например C:. В последнем случае пространство имен разбивается на части, которые соответствуют различным устройствам или разделам устройств. Поскольку такое разделение выполняется случайным образом, в качестве представления для пользователя его можно считать не самым идеальным вариантом.

Файлы помещаются в каталогах (directory). Каталог — это аналог папки, которая обычно содержит связанные между собой файлы. Каталоги могут содержать подкаталоги. В этой связи каталоги могут быть вложены друг в друга и образуют пути (path). Каждый компонент пути называется элементом каталога (directory entry). Пример пути— "/home/wolfman/foo". Корневой каталог "/", каталоги home и wolfman, а также файл foo — это элементы каталогов, которые называются *dentry*. В операционной системе Unix каталоги представляют собой обычные файлы, которые просто содержат список файлов каталога. Так как каталог по отношению к виртуальной файловой системе — это файл, то с каталогами можно выполнять те же операции, что и с файлами.

Unix-подобные операционные системы отличают концепцию файла от любой информации об этом файле (права доступа, размер, владелец, время создания и т.д.). Последняя информация иногда называется *метаданными файла (file, metadata),* т.е. данные о данных, и хранится отдельно от файлов в специальных структурах, которые называются *индексами (inode)*. Это сокращенное название от *index node* (индексный узел), хотя в наши дни термин "inode" используется значительно чаще.

Вся указанная информация, а также связанная с ней информация о самой файловой системе хранится в суперблоке (superblock). Суперблок— это структура данных, которая содержит информацию о файловой системе в целом. Иногда эти общие данные называются *метаданными файловой системы*. Метаданные файловой системы содержат информацию об индивидуальных файлах и о файловой системе в целом.

Традиционно файловые системы ОС Unix реализуют эти понятия как структуры данных, которые определенным образом расположены на физических дисках. Например, информация о файлах хранится в индексе, в отдельном блоке диска, каталоги являются файлами, информация по управлению файловой системой хранится централизованно в суперблоке и т.д. Подсистема VFS операционной системы Linux рассчитана на работу с файловыми системами, в которых поддерживаются аналогичные концепции. Не Unix-подобные файловые системы, такие как FAT или NTFS, также работают в ОС Linux, однако их программный код должен обеспечить наличие аналогичных концепций. Например, если файловая система не поддерживает отдельные индексы файлов, то код должен построить в оперативной памяти структуры данных таким образом, чтобы казалось, что такая поддержка работает. Если файловая система рассматривает каталоги как объекты специальных типов, для VFS каталоги должны представляться как обычные файлы. Часто код не файловых систем не в стиле Unix требует выполнять некоторую дополнительную обработку чтобы уложиться в парадигму Unix и требования VFS. Такие файловые системы также поддерживаются, и обычно качество не особенно страдает.



Рис, 12.2. Схема прохождения данных из пространства пользователя, где вызывается функция wri te (), через общий системный вызов VFS, к специфическому методу записи файловой системы и, наконец, поступление па физический носитель

Объекты VFS и их структуры данных

Виртуальная файловая система (VFS) объектно-ориентированна[3]. Общая файловая модель представлена набором структур данных. Эти структуры данных очень похожи на объекты. Так как ядро программируется строго на языке С, то, при отсутствии возможностей прямой поддержки парадигм ООП в языке программирования, структуры данных представляются структурами языка С. Структуры содержат как указатели на элементы данных, так и указатели на функции, которые работают с этими данными.

Существуют следующие четыре основных типа объектов VFS.

- Объект *суперблок (superblock),* который представляет определенную смонтированную файловую систему.
- Объект *файловый индекс (inode),* который представляет определенный файл.
- Объект *элемент каталога (denlry),* который представляет определенный элемент каталога.
- Объект *файл (file),* который представляет открытый файл, связанный с процессом.

Следует обратить внимание, что поскольку подсистема VFS рассматривает каталоги как обычные файлы, то не существует специальных объектов для каталогов. Как рассказывалось ранее, объект dentry представляет компонент пути, который может содержать обычный файл. Другими словами, deniry — это не то же самое, что каталог, а каталог — это то же, что и файл. Все понятно?

Каждый из рассмотренных основных объектов содержит объект *operations (операции).* Эти объекты описывают методы, которые ядро может применять для основных объектов.

[3] Часто многие этого не замечают и даже отрицают, но тем не менее в ядре много примеров объектно-ориентированного программирования. Хотя разработчики ядра и сторонятся языка C++ и других явно объектно-ориентированных языков программировании (ООП), иногда очень полезно мыслить в терминах объектов. Подсистема VFS— это хороший пример того, как просто и эффективно объектно-ориентированное программирование реализуется на языке С, в котором нет объектно-ориентированныхконструкций.

В частности, существуют следующие объекты операций.

- Объект super_operations (операции с суперблоком файловой системы) содержит методы, которые ядро может вызывать для определенной файловой системы, как, например, read_inode () или sync_fs ().
- Объект inodeoperations (операции с файловыми индексами) содержит методы, которые ядро может вызывать для определенного файла, как, например, created или link ().
- Объект dentry_operations (операции с элементами каталогов) содержит методы, которые ядро может вызывать для определенного элемента каталога, как, например, d_compare () или d_delete ().
- Объект file_operations (операции с файлами) содержит методы, которые процесс может вызывать для открытого файла, как например, read () и write ().

Объекты операций реализованы в виде структур, содержащих указатели на функции. Эти функции оперируют объектом, которому принадлежит объект операций. Для многих методов объект может унаследовать общую функцию, если для работы достаточно базовой функциональности. В противном случае каждая файловая система присваивает указателям адреса своих специальных методов.

И еще раз повторимся, что под *объектами* мы будем понимать структуры, которые явно не являются объектными типами (в отличие от языков программирования C++ и Java). Однако эти структуры представляют определенные экземпляры объектов, данные связанные с объектами, и методы, которые ими оперируют. Это практически то же, что и объектные типы.

Другие объекты подсистемы VFS

Структуры для VFS — это самая "любимая" вещь, и в этой подсистеме существуют не только рассмотренные структуры, но и еще некоторые. Каждая зарегистрированная файловая система представлена структурой file_system_type, Объекты этого типа описывают файловую систему и ее свойства. Более того, каждая точка монтирования предстаплена в виде структуры vfsmount. Эта структура содержит информацию о точке монтирования, такую как ее положение и флаги, с которыми выполнена операция монтирования.

И наконец, каждый процесс имеет три структуры, которые описывают файловую систему и файлы, связанные с процессом. Это структуры file_struct, fs_struct и namespace.

Далее в этой главе будут рассматриваться эти объекты и их роль в функционировании уровня VFS.

Объект superblock

Объект *суперблок* должен быть реализован для каждой файловой системы. Он используется для хранения информации, которая описывает определенную файловую систему. Этот объект обычно соответствует *суперблоку (superblock) или управляющему блоку (control block)* файловой системы, который хранится в специальном секторе диска (отсюда и имя объекта). Файловые системы, которые не располагаются на дисках

(например, файловые системы в виртуальной памяти, как *sysfs),* генерируют информацию суперблока "на лету" и хранят в памяти.

Объект *суперблак* представляется с помощью структуры struct super_block, которая определена в файле <linux/fs.h>. Она выглядит следующим образом (комментарии описывают назначение каждого поля).

```
struct super_block {
 struct list_head        s_list;        /* список всех суперблоков */
 dev_t                   s_dev;         /* идентификатор */
 unsigned long       s_blocksize;       /* размер блока в байтах */
 unsigned long           s_old_blocksize; /*старый размер блока в байтах*/
 unsigned char           s_blocksize_bits; /* размер блока в битах */
 unsigned char           s_dirt;        /*флаг того, что суперблок изменен*/
 unsigned long long     s_maxbytes;     /* максимальный размер файла */
 struct file_system_type  *s_type;       /* тип файловой системы */
 struct super_operations  *s_op;         /* операции суперблока */
 struct dquot_operations    *dq_op;     /* операции с квотами */
 struct quotactl_ops      *s_qcop;      /* операции управления квотами */
 struct export_operations *s_export_op; /* операции экспортирования */
 unsigned long           s_flags;       /* флаги монтирования */
 unsigned long      s_magic;      /*      магический номер файловой системы*/
 struct dentry           *s_root;      /* каталог, точка монтирования */
 struct rw_semaphore     s_umount;      /* семафор размонтирования */
 struct semaphore        s_lock;        /* семафор суперблока */
 int                     s_count;       /* счетчик ссылок на суперблок */
 int                     s_syncing;     /*флаг синхронизации файловой системы*/
 int                    s_nesd_sync_fs; /*флаг того, что файловая
                                          система еще не синхронизирована*/
 atomic_t                s_active;      /* счетчик активных ссыпок */
 void                    *s_security;   /* модуль безопасности */
 struct list_head        s_dirty;       /* список измененных индексов */
 struct list_head         s_io;         /*список обратной записи */
 struct hlist_head       s_anon;        /* анонимные элементы каталога
                                           для экспортирования */
 struct list_head        s_files;       /* список связанных файлов */
 struct block_device     *s_bdev;  "     /* соответствующий драйвер
                                           блочного устройства */
 struct list_head        s_instances;   /* список файловых систем
                                           данного типа */
 struct quota_info       s_dquot;       /* параметры квот */
 char             s_id[32];             /* текстовое имя */
 void            *s_fs_info;            /* специфическая информация
                                           файловой системы */
 struct semaphore        s_vfs_rename_sem; /* семафор переименования */
};
```

Код для создания, управления и ликвидации объектов *суперблок* находится в файле fs/super.c. Объект *суперблок* создается и инициализируется в функции alloc_super (). Эта функция вызывается при монтировании файловой системы, которая считывает суперблок файловой системы с диска и заполняет поля объекта *суперблок*.

Операции суперблока

Наиболее важный элемент суперблока — это поле s_op, которое является указателем на таблицу операций суперблока. Таблица операций суперблока представлена с помощью структуры struct super_operations, которая определена в файле <linux/fs.h>. Она выглядит следующим образом.

```
struct super_operations {
        struct inode *(*alloc_inode) (struct super_block *sb);
        void (*destroy_inode) (struct inode * ) ;
        void (*read_inode) (struct inode * ) ;
        void (*dirty_inode) (struct inode * ) ;
        void (*write_inode) (struct inode *, int);
        void (*put inode) (struct inode * ) ;
        void (*drop_inode) (struct inode *) ;
        void (*delete_inode) (struct inode * ) ;
        void (*put_super) (struct super_block * ) ;
        void (*write_super) (struct super block * ) ;
        int (*sync_fs) (struct super_block *, int};
        void (*write_super_lockfs) (struct super_block * ) ;
        void (*unlockfs) (struct super_block * ) ;
        int (*statfs) (struct super_block *, struct statfs *};
        int (*remount_fs) (struct super_block *, int *, char * ) ;
        void (*clear_inode) (struct inode * ) ;
        void (*umount_begin) (struct super block * ) ;
        int (*show_options) (struct seq_file *, struct vfsmount * ) ;
};
```

Каждое поле этой структуры представляет собой указатель на функцию, которая работает с объектом *суперблок*. Операции суперблока выполняют низкоуровневые действия с файловой системой и ее файловыми индексами.

Когда для файловой системы необходимо выполнить операции с суперблоком, то выполняется разыменование указателя на суперблок, и далее получается указатель на необходимый метод. Например, если файловой системе необходимо записать суперблок, то вызывается следующая функция.

```
sb->s_op->write_super(sb);
```

где параметр sb — это указатель на суперблок файловой системы. Следуя по указателю s_op, получаем таблицу операций суперблока и, наконец, необходимую функцию write_super(), которая вызывается непосредственно. Следует обратить внимание на то, что вызову функции write_super() необходимо передать указатель на суперблок в качестве параметра, несмотря на то что метод связан с суперблоком. Это происходит от того, что язык программирования C не объектно-ориентирован. В C++ аналогичный вызов может быть выполнен следующим образом.

```
sb.write_super();
```

В языке C нет простого способа получить указатель на объект, для которого вызван метод, поэтому его необходимо передавать явно.

Рассмотрим операции суперблока, которые описаны в структуре super_operations.

- struct inode * alloc_inode (struct super_block *sb) — эта функция создает и инициализирует новый объект файлового индекса, связанного с данным суперблоком.
- void destroy_inode(struct inode *inode) — эта функция уничтожает данный объект индекса файла.
- void read_inode (struct inode *inode) — эта функция считывает с диска файловый индекс с номером inode->i_ino и заполняет все остальные поля структуры данных индекса.
- void dirty_inode (struct inode *inode) — эта функция вызывается подсистемой VFS, когда в индекс вносятся изменения (dirty). Журналируемые файловые системы (как, например, ext3) используют эту функцию для обновления журнала.
- void write_inode(struct inode inode*, int wait) — эта функция записывает указанный индекс на диск. Параметр wait указывает, должна ли данная операция выполняться синхронно.
- void put_inode (struct inode *inode) — эта функция освобождает указанный индекс.
- void drop_inode (struct inode *inode) — эта функция вызывается подсистемой VFS, когда исчезает последняя ссылка на индекс. Обычные файловые системы Unix никогда не определяют эту функцию, в таком случае подсистема VFS просто удаляет индекс. Вызывающий код должен удержипать блокировку inode_lock.
- void delete_inode (struct inode *inode) — эта функция удаляет индекс файла с диска.
- void put_super (struct super_block *sb) — эта функция вызывается подсистемой VFS при раэмонтировании файловой системы, чтобы освободить указанный суперблок.
- void write_super (struct super_block *sb) — эта функция обновляет суперблок на диске данными из указанного суперблока. Подсистема VFS вызывает эту функцию для синхронизации измененного суперблока в памяти с данными суперблока на диске.
- int sync_fs (struct super_block *sb, int wait) — эта функция синхронизирует метаданные файловой системы с данными на диске. Параметр wait указывает, должна ли операция быть синхронной или асинхронной.
- void write_super_lockfs (struct super_block *sb) — эта функция предотвращает изменения файловой системы и затем обновляет данные суперблока на диске данными из указанного суперблока. Сейчас она используется диспетчером логических томов (LVM, Logical Volume Manager).
- void unlockfs (struct super_block *sb) — эта функция разблокирует файловую систему после выполнения функции write_super_lockfs ().
- int statfs(struct super_block *sb, struct statfs *statfs) — эта функция вызывается подсистемой VFS для получения статистики файловой системы, Статистика указанной файловой системы записывается в структуру statfs.

- int remount_fs (struct super_block *sb, int *flags, char *data) —эта функция вызывается подсистемой VFS, когда файловая система монтируется с другими параметрами монтирования.
- void clear_inode (struct inode *) — эта функция вызывается подсистемой VFS для освобождения индекса и очистки всех страниц памяти, связанных с индексом.
- void umount_begin (struct super_block *sb) — эта функция вызывается подсистемой VFS для прерывания операции монтирования. Она используется сетевыми файловыми системами, такими как NFS.

Все рассмотренные функции вызываются подсистемой VFS в контексте процесса. Все они при необходимости могут блокироваться.

Некоторые из этих функций являются необязательными. Файловая система может установить их значения в структуре операций суперблока равными NULL. Если соответствующий указатель равен NULL, то подсистема VFS или вызывает общий вариант функции, или не происходит ничего, в зависимости от операции.

Объект inode

Объект inode содержит всю информацию, которая необходима ядру для манипуляций с файлами и каталогами. В файловых системах в стиле Unix вся информация просто считывается из дисковых индексов и помещается в объект inode подсистемы VFS. Если файловые системы не имеют индексов, то эту информацию необходимо получить из других дисковых структур[4].

Объект индекса файла представляется с помощью структуры struct inode, которая определена в файле <linux/fs.h>. Эта структура с комментариями, описывающими назначение каждого поля, имеет следующий вид.

```
struct inode {
 struct hlist_node     i_hash;     /* хешированный список */
 struct list_head      i_list;     /* связанный список индексов */
 struct list_head      i_dentry;   /* связанный список объектов dentry */
 unsigned long         i_ino;      /* номер индекса */
 atomic_t              i_count;    /* счетчик ссылок */
 umode_t               i_mode;     /* права доступа */
 unsigned int          i_nlink;    /* количество жестких ссылок */
 uid_t                 i_uid;      /* идентификатор пользователя-владельца */
 gid_t                 i_gid;      /* идентификатор группы-владельца */
 kdev_t                i_rdev;     /* связанное устройство */
 loff_t                i_size;     /* размер файла в байтах */
 struct timespec       i_atime;    /* время последнего доступа к файлу */
 struct timespec       i_mtime;    /* время последнего изменения файла */
 struct timespec       i_ctime;    /* время изменения индекса */
 unsigned int          i_blkbits;  /* размер блока в битах */
```

[4] Файловые системы, которые не имеют индексов, обычно хранят необходимую информацию как часть файла. Некоторые современные файловые системы также применяют базы данных для хранения метаданных файла. В любом случае объект индекса создается тем способом, который подходит для файловой системы.

```
    unsigned long          i_blksize;    /* размер блока в байтах */
    unsigned long          i_version;    /* номер версии */
    unsigned long          i_blocks;     /* размер файла в блоках */
    unsigned short         i_bytes;      /* количество использованных байтов*/
    spinlock_t             i_lock;       /* блокировка для защиты полей */
    struct  rw_semaphore   i_alloc_sem   /* вложенные блокировки при
                                            захваченной i_sem */
    struct semaphore       i_sem;        /* семафор индекса */
    struct inode_operations *i_op;       /* таблица операций с индексом */
    struct file_operations *i_fop;       /* файловые операции */
    struct super_block     *i_sb;        /* связанный суперблок */
    struct file_lock       *i_flock;     /* список блокировок файлов */
    struct address_space   *i_mapping;   /* соответствующее адресное
                                            пространство */
    struct address_space   i_data;       /* адресное пространство устройства*/
    struct dquot          *i_dquot[MAXQUOTAS]; /* дисковые квоты для индекса*/
    struct list_head       i_devices;    /* список блочных устройств */
    struct pipe_inode_info *i_pipe;      /* информация конвейера */
    struct block_device    *i_bdev;      /* драйвер блочного устройства */
    unsigned long         i_dnotify_mask; /* события каталога */
    struct dnotify_struct *i_dnotify;    /* информация о событиях каталога */
    unsigned long          i_state;      /* флаги состояния */
    unsigned long          dirtied_when  /* время первого изменения */
    unsigned int           i_flags;      /* флаги файловой системы */
    unsigned char          i_sock;       /* сокет или нет? */
    atomic_t               i_writecount; /* счетчик использования для записи*/
    void                   *i_security;  /* модуль безопасности */
    __u32                  i_generation; /* номер версии индекса */
    union {
        void               *generic_ip;  /* специфическая информация
                                            файловой системы */
    } u;
};
```

Для каждого файла в системе существует представляющий его индекс (хотя объект файлового индекса создается в памяти только тогда, когда к файлу осуществляется доступ). Это справедливо и для специальных файлов, таких как файлы устройств или конвейеры. Следовательно, некоторые из полей структуры `struct inode` относятся к этим специальным файлам. Например, поле `i_pipe` указывает на структуру данных именованного конвейера. Если индекс не относится к именованному конвейеру, то это поле просто содержит значение NULL. Другие поля, связанные со специальными файлами, — это `i_devices`, `i_bdev`, `i_cdev`.

Может оказаться, что та или иная файловая система не поддерживает тех свойств, которые присутствуют в объекте `inode`. Например, некоторые файловые системы не поддерживают такого атрибута, как время создания файла. В этом случае файловая система может реализовать это свойство как угодно. Например, поле `i_ctime` можно сделать нулевым или равным значению поля `i_mtime`.

Операции с файловыми индексами

Так же как и в случае операций суперблока, важным является поле inode_operations, в котором описаны функции файловой системы, которые могут быть вызваны подсистемой VFS для объекта файлового индекса. Как и для суперблока, операции с файловыми индексами могут быть вызваны следующим образом.

```
i->i_op->truncate(i)
```

где переменная i содержит указатель на определенный объект файлового индекса. В данном случае для индекса I выполняется операция trancate (), которая определена для файловой системы, в которой находится указанный файловый индекс i. Структура inode_operations определена в файле <linux/fs.h>, как показано ниже.

```
struct inode_operations {
       int (*create) (struct inode *, struct dentry *,int);
       struct dentry * (*lookup) (struct inode *, struct dentry * ) ;
       int (*link) (struct dentry *, struct inode *, struct dentry * ) ;
       int (*unlink) (struct inode *, struct dentry * ) ;
       int (*symlink) (struct inode *, struct dentry *, const char * ) ;
       int (*mkdir) (struct inode *, struct dentry *, int);
       int (*rmdir) (struct inode *, struct dentry *);
       int (*mknod) (struct inode *, struct dentry *, int, dev_t);
       int (*rename) (struct inode *, struct dentry *,
                    struct inode *, struct dentry * ) ;
       int (*readlink) (struct dentry *, char *, int);
       int (*follow_link) (struct dentry *, struct nameidata * ) ;
       int (*put_link) (struct dentry *, struct nameidata * ) ;
       void (*truncate) (struct inode * ) ;
       int (*permission) (struct inode *, int);
       int (*setattr) (struct dentry *, struct iattr * ) ;
       int (*getattr) (struct vfsmount *, struct dentry *, struct kstat * ) ;
       int (*setxattr) (struct dentry *, const char *,
                      const void *, size_t, int);
       ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
       ssize_t (*listxattr) (struct dentry *, char *, size_t);
       int (*removexattr) (struct dentry *, const char * ) ;
};
```

Рассмотрим указанные операции более подробно.

- int create(struct inode *dir, struct dentry *dentry, int mode)
  Эта функция вызывается подсистемой VFS из системных вызовов creat () и open () для создания нового файлового индекса, который имеет указанный режим доступа (mode) и связан с указанным элементом каталога (dentry).
- struct dentry * lookup(struct inode *dir, struct dentry *dentry)
  Эта функция производит поиск файлового индекса в указанном каталоге. Файловый индекс должен соответствовать имени файла, хранящемуся в указанном объекте элемента каталога.

- int link(struct ctentry *old_dentry, struct inode *dir,
  struct dentry *dentry)

  Эта функция вызывается из системного вызова link () для создания жесткой ссылки (hard link) на файл, соответствующий элементу каталога old_dentry в каталоге dir. Новая ссылка должна иметь имя, которое хранится в указанном элементе каталога dentry.

- int unlink(struct inode *dir, struct dentry *dentry)

  Эта функция вызывается из системного вызова unlink () для удаления файлового индекса, соответствующего элементу каталога dentry в каталоге dir.

- int symlink(struct inode *dir, struct dentry *dentry, const char *symname)

  Эта функция вызывается из системного вызова symlink() для создания символьной ссылки с именем symname на файл, которому соответствует элемент каталога dentry в каталоге dir.

- int mkdir(struct inode *dir, struct dentry *dentry, int mode)

  Эта функция вызывается из системного вызова mkdir () для создания нового каталога с указанным режимом доступа (mode).

- int rmdir(struct inode *dir, struct dentry *dentry)

  Эта функция вызывается из системного вызова rmdir () для удаления каталога на который указывает элемент каталога dentry из каталога dir.

- int mknod (struct inode *dir, struct dentry *dentry, int mode, dev_t rdev)

  Эта функция вызывается из системного вызова mknod () для создания специального файла (файла устройства, именованного конвейера или сокета), информация о котором хранится в параметре rdev. Файл должен быть создан в каталоге dir с именем, указанным в параметре dentry, и режимом доступа mode.

- int rename(struct inode *old_dir, struct dentry *old_dentry,
  struct inode *new_dir, struct dentry *new_dentry)

  Эта функция вызывается подсистемой VFS для перемещения указанного элемента каталога old_dentry из каталога old_dir в каталог new_dir с новым именем, указанным в параметре new_dentry.

- int readlink(struct dentry *dentry, char *buffer, int buflen)

  Эта функция вызывается из системного вызова readlink () для копирования не более buflen байт полного пути, связанного с символьной ссылкой, соответствующей указанному элементу каталога, в указанный буфер.

- int follow_link(struct dentry *dentry, struct nameidata *nd)

  Эта функция вызывается подсистемой VFS для трансляции символьной ссылки в индекс файла, на который эта ссылка указывает. На ссылку указывает указатель dentry, а результат сохраняется в структуру nameidata, на которую указывает параметр nd.

- int put_link(struct dentry *dentry, struct nameidata* nd)

  Эта функция вызывается подсистемой VFS после вызова функции followlink ().

- void truncate (struct inode *inode). Эта функция вызывается подсистемой VFS для изменения размера заданного файла. Перед вызовом поле i_size указанного индекса файла должно быть установлено в желаемое значение размера.

- `int permission(struct inode *inode, int mask)`

  Эта функция проверяет, разрешен ли указанный режим доступа к файлу, на который ссылается объект inode. Функция должна возвращать нулевое значение, если доступ разрешен, и отрицательное значение кода ошибки в противном случае. Для большинства файловых систем данное поле устанавливается *в* значение NULL, и при этом используется общий метод VFS, который просто сравнивает биты поля режима доступа файлового индекса с указанной маской. Более сложные файловые системы, которые поддерживают списки контроля доступа (ACL), реализуют свой метод `permission ()`.

- `int setattr(struct dentry *dentry, struct iattr *attr)`

  Эта функция вызывается функцией `notify_change ()` для уведомления о том, что произошло "событие изменения" ("change event") после модификации индекса.

- `int getattr(struct vfsmount *mnt, struct dentry *dentry, struct kstat *stat)`

  Эта функция вызывается подсистемой VFS при уведомлении, что индекс должен быть обновлен с диска.

- `int setxattr(struct dentry *dentry, const char *name, const void *value, size_t size, int flags)`

  Эта функция вызывается подсистемой VFS для установки одного из расширенных атрибутов (extended attributes)[5] с именем name в значение value для файла, соответствующего элементу каталога dentry.

- `int getxattr (struct dentry *dentry, const char *name, void *value, size_t size)`

  Эта функция вызывается подсистемой VFS для копирования значения одного из расширенных атрибутов (extended attributes) с именем name в область памяти с указателем value.

- `ssize_t listxattr(struct dentry *dentry, char *list, size_t size)`

  Эта функция должна копировать список всех атрибутов для указанного файла в буфер, соответствующий параметру list.

- `int removexattr(struct dentry *dentry, const char *name)`

  Эта функция удаляет указанный атрибут для указанного файла.

Объект dentry

Как уже рассказывалось, подсистема VFS представляет каталоги так же, как и файлы. В имени пути /bin/vi, и элемент bin, и элемент vi — это файлы, только bin — это специальный файл, который является каталогом, а vi — это обычный файл. Объекты файловых индексов служат для представления обоих этих компонентов. Несмотря на такую полезную унификацию, подсистеме VFS также необходимо

[5] Расширенные атрибуты — это новая функциональность, которая появилась в ядре 2.6 для того, чтобы создавать параметры файлов в виде пар имя/значение по аналогии с базой данных. Эти параметры поддерживаются не многими файловыми системами, и к тому же они еще используются не достаточно широко.

выполнять операции, специфичные для каталогов, такие как поиск компонента пути по его имени, проверка того, что указанный элемент пути существует, и переход на следующий компонент пути.

Для решения этой задачи в подсистеме VFS реализована концепция элемента каталога (directory entry или dentry). Объект dentry — это определенный компонент пути. В предыдущем примере компоненты /, bin и vi — это объекты элементов каталога. Первые два— это каталоги, а последний— обычный файл. Важным моментом является то, что все объекты dentry— это компоненты пути, включая и обычные файлы.

Элементы пути также могут включать в себя точки монтирования. В имени пути /mnt/cdrom/foo, компоненты /, mnt, cdrom и foo — это все объекты типа dentry. Подсистема VFS при выполнении операций с каталогами по необходимости конструирует объекты элементов каталога на лету.

Объекты типа dentry представлены с помощью структуры `struct dentry` и определены в файле <linux/dcache.h>. Эта структура с комментариями, которые определяют назначение каждого поля, имеет следующий вид.

```
struct dentry {
 atomic_t                d_count;    /* счетчик использования */
 unsigned long           d_vfs_flags; /* флаги кэша объектов dentry */
 spinlock_t              d_lock;     /* блокировка данного объекта dentry*/
 struct inode            *d_inode;   /* соответствующий файловый индекс */
 struct list_head        d_lru;      /* список неиспользованных объектов*/
 struct list_head        d_child;    /* список объектов у родительского
                                        экземпляра */
 struct list_head        d_subdirs;  /* подкаталоги */
 struct list_head        d_alias;    /* список альтернативных (alias)
                                        индексов */
 unsigned long           d_time;     /* время проверки правильности */
 struct dentry_operations *d_op;     /* таблица операций с элементом
                                        каталога */
 struct super_block      *d_sb;      /* связанный суперблок */
 unsigned int            d_flags;    /* флаги элемента каталога */
 int           d_mounted;            /* является ли объект точкой
                                        монтирования */
 void          *d_fsdata;           /*специфические данные файловой системы*/
 struct rcu_head          d_rcu;     /*блокировки RCU (read-copy update) */
 struct  dcookie_struct  *d_cookie;  /* cookie-идентификатор */
 struct dentry           *d_parent;  /*объект dentry родительского каталога*/
 struct qstr              d_name;    /* имя dentry */
 struct hlist_node        d_hash;    /* список хеширования */
 struct hlist_head       *d_bucket;  /* сегмент хеш-таблицы */
 unsigned char           d_iname[DNAME_INLINE_LEN_MIN]; /*короткое имя файла*/
};
```

В отличие от предыдущих двух объектов, объект dentry не соответствует какой бы то ни было структуре данных на жестком диске. Подсистема VSF создает эти объекты на лету на основании строкового представления имени пути. Поскольку объекты элементов каталога не хранятся физически на дисках, то в структуре `struct dentry` нет никаких флагов, которые указывают на то, изменен ли объект (т.е. должен ли он быть записан назад на диск).

Состояние элементов каталога

Действительный объект элемента каталога, может быть в одном из трех состояний: используемый fused), неиспользуемый (unused) и негативный (negative).

Используемый объект соответствует существующему файловому индексу (т.е. поле d_inode указывает на связанный объект типа mode) и используется один или более раз (т.е. значение поля d_count — положительное число). Используемый элемент каталога используется подсистемой VFS, а также указывает на существующие данные, поэтому не может быть удален.

Неиспользуемый объект типа dentry соответствует существующему объекту inode (поле d_inode указывает на объект файлового индекса), но подсистема VFS в данный момент не использует этот элемент каталога (поле d_count содержит нулевое значение). Так как элемент каталога указывает на существующий объект, то он сохраняется на случай, если вдруг окажется нужным. Если объект не ликвидировать преждевременно, то его и не нужно будет создавать заново, если вдруг он понадобится в будущем, и поиск по имени пути пройдет быстрее. Когда же появляется необходимость освободить память, то такой объект элемента каталога может быть удален, потому что он никем не используется.

Негативный объект dentry[6] не связан с существующим файловым индексом (поле d_inode равно значению NULL), потому что или файловый индекс был удален, или соответствующий элемент пути никогда не существовал. Такие объекты элементов каталогов сохраняются, чтобы в будущем поиск по имени пути проходил быстрее. Хотя такие объекты dentry и полезны, но они при необходимости могут уничтожаться, поскольку никто их на самом деле не использует.

Объект dentry может быть освобожден, оставаясь в слябовом кэше объектов, как обсуждалось в предыдущей главе. В таком случае на этот объект нет ссылок ни в коде VFS, ни в коде файловых систем.

Кэш объектов dentry

После того как подсистема VFS преодолела все трудности, связанные с переводом всех элементов пути в объекты элементов каталогов, и был достигнут конец пути, то было бы достаточно расточительным выбрасывать на ветер всю проделанную работу. Ядро кэширует объекты в кэше элементов каталога, который называют *dcache*.

Кэш объектов dentry состоит из трех частей.

- Список "используемых" объектов dentry, которые связаны с определенным файловым индексом (поле i_dentry объекта inode). Поскольку указанный файловый индекс может иметь несколько ссылок, то ему может соответсвовать несколько объектов dentry, а следовательно используется связанный список.
- Двухсвязный список неиспользуемых и негативных объектов dentry "с наиболее поздним использованием" (last recently used, LRU). Вставки элементов в этот список отсортированы по времени, поэтому элементы, которые находятся в начале списка, — самые новые. Когда ядро должно удалить элементы каталогов для освобождения памяти, то эти элементы берутся из конца списка.

[6] Это название несколько сбивает с толку. В таких объектах нет ничего негативного или отрицательного. Более удачным было бы, наверное, название invalid denlry или несуществующий элемент каталога.

потому что там находятся элементы, которые использовались наиболее давно и для которых меньше шансов, что они понадобятся в ближайшем будущем.

- Хеш-таблица и хеш-функция, которые позволяют быстро преобразовать заданный путь в объект dentry.

Указанная хеш-таблица представлена с помощью массива dentry_hashtable. Каждый элемент массива — это указатель на список тех объектов dentry, которые соответствуют одному ключу. Размер этого массива зависит от объема физической памяти в системе.

Значение ключа определяется функцией d_hash (), что позволяет для каждой файловой системы реализовать свою хеш-функцию.

Поиск в хеш-таблице выполняется с помощью функции d_lookup (). Если в кэше dcache найден соответствующий объект, то это значение возвращается. В случае ошибки возвращается значение NULL.

В качестве примера рассмотрим редактирование файла исходного кода в вашем домашнем каталоге, /home/dracula/src/foo.c. Каждый раз, когда производится доступ к этому файлу (например, при первом открытии, при последующей записи, при компиляции и так далее), подсистема VFS должна пройти через псе элементы каталогов в соответствии с путем к файлу: /, home, dracula, sre и, наконец, foo.c. Для того чтобы каждый раз при доступе к этому (и любому другому) имени пути избежать выполнения данной операции, которая требует довольно больших затрат времени, подсистема VFS вначале может попытаться найти это имя пути в dentry-кэше. Если поиск проходит успешно, то необходимый конечный элемент каталога нужного пути получается без особых усилий. Если же данного элемента каталога нет в dentry-кэше, то подсистема VFS должна самостоятельно отследить путь. После завершения поиска найденные объекты dentry помещаются в кэш dcache, чтобы ускорить поиск в будущем.

Кэш dcache также является интерфейсом к кэшу файлопых индексов *icache*. Объекты inode связаны с объектами dentry, поскольку объект dentry поддерживает положительное значение счетчика использования для связанного с ним индекса. Это в свою очередь позволяет объектам dentry удерживать связанные с ними объекты mode в памяти. Иными словами, если закэширован элемент каталога, то соответственно оказывается закэшированным и соответствующий ему файловый индекс. Следовательно, если поиск в кэше для некоторого имени пути прошел успешно, то соответствующие файловые индексы уже закэшированы в памяти.

Операции с элементами каталогов

Структура dentry_operations содержит методы, которые подсистема VFS может вызывать для элементов каталогов определенной файловой системы. Эта структура определена в файле <linux/dcache.h> следующим образом.

```
struct dentry_operations {
       int (*d_revalidate) (struct dentry *, int);
       int (*d_hash) (struct dentry *, struct qstr * ) ;
       int (*d_corapare) (struct dentry *, struct qstr *, struct qstr * ) ;
       int (*d_delete) (struct dentry * ) ;
       void (*d_release) (struct dentry * ) ;
       void (*d_iput) (struct dentry *, struct inode * ) ;
};
```

Методы служат для следующих целей

- `int d_revalidate(struct dentry *dentry, int flags)`

  Эта функция определяет, является ли указанный объект элемента каталога действительным. Подсистема VFS вызывает эту функцию, когда она пытается использовать объект dentry из кэша dcache. Для большинства файловых систем этот метод установлен в значение NULL, потому что объекты denry, которые находятся в кэше, всегда действительны.

- `int d_hash(struct dentry *dentry, struct qstr *name)`

  Эта функция создает значение хеш-ключа на основании указанного объекта dentry. Подсистема VFS вызывает эту функцию всякий раз, когда добавляет объект элемента каталога в хеш-таблицу.

- ```
  int d_compare(struct dentry *dentry,
                struct qstr *narael,
                struct qstr *name2)
  ```

  Эта функция вызывается подсистемой VFS для сравнения двух имен файлов namel и name2. Большинство файловых систем используют умолчание VFS, которое соответствует простому сравнению двух строк. Для некоторых файловых систем, таких как FAT, не достаточно простого сравнения строк. Файловая система FAT не чувствительна к регистру символов в именах файлов, поэтому появляется необходимость в реализации функции, которая при сравнении не учитывает регистр символов. Эта функция вызывается при захваченной блокировке dcache_lock[7].

- `int d_delete (struct dentry *dentry)`

  Эта функция вызывается подсистемой VFS, когда количество ссылок d_count указанного объекта dentry становится равным пулю. Функция вызывается при захваченной блокировке dcache_lock.

- `void d_release(struct dentry *dentry)`

  Эта функция вызывается подсистемой VFS, когда она собирается освободить указанный объект dentry. По умолчанию данная функция не выполняет никаких действий.

- `void d_iput(struct dentry *dentry, struct inode *inode)`

  Эта функция вызывается подсистемой VFS, когда элемент каталога теряет связь со своим файловым индексом (например, когда этот элемент каталога удаляется с диска). По умолчанию подсистема VFS просто вызывает функцию `iput ()`, чтобы освободить соответствующий объект inode. Если файловая система переопределяет эту функцию, то она также должна вызывать функцию `iput ()` в дополнение к специфичной для файловой системы работе.

[7] А также при захваченной блокировке *dentry->d_lock*. — *Примеч. перев.*
  ```

Объект file

Последним из основных объектов подсистемы VFS рассмотрим объект файла. Объект File используется для представления файлов, которые открыты процессом. Когда мы думаем о подсистеме VFS с точки зрения пространства пользователя, то объект файла — это то, что первое приходит в голову. Процессы непосредственно работают с файлами, а не с суперблоками, индексами или элементами каталогов. Не удивительно, что информация, которая содержится в объекте file, наиболее привычна (такие данные, как режим доступа или текущее смещение), а файловые операции очень похожи на знакомые системные вызовы, такие как `read()` и `write()`.

Объект файла — это представление открытого файла, которое хранится в оперативной памяти. Объект (а не сам файл) создается в ответ на системный вызов `open()` и уничтожается в результате системного вызова `close()`. Все вызовы, связанные с файлом, на самом деле являются методами, которые определены в таблице операций с файлом. Так как несколько процессов могут одновременно открыть и использовать один и тот же файл, то для одного файла может существовать несколько объектов file. Файловый объект просто представляет открытый файл с точки зрения процесса. Этот объект содержит указатель на соответствующий элемент каталога (который, в свою очередь, указывает на файловый индекс), представляющий открытый файл. Соответствующие объекты inode и dentry, конечно, являются уникальными.

Файловый объект представляется с помощью структуры `struct file`, которая определена в файле <linux/fs.h>. Рассмотрим поля этой структуры с комментариями, которые описывают назначение каждого поля.

```
struct file {
 struct list_head     f_list;     /* список объектов file*/
 struct dentry        *f_dentry;  /* связанный объект dentry */
 struct vfsmount      *f_vfsmnt;  /* связанная смонтированная
                                     файловая система */
 struct file_operations  *f_op;     /* таблица файловых операций */
 atomic_t             f_count;    /* счетчик ссылок на этот объект */
 unsigned int         f_flags;    /*флаги, указанные при вызове функции open*/
 mode_t               f_mode;     /* режим доступа к файлу */
 loff_t               f_pos;      /*смещение в файле (file pointer, offset) */
 struct fown_struct    f_owner;    /* информация о владельце для обработки
                                     сигналов */
 unsigned int          f_uid;     /*идентификатор пользователя владельца, UID*/
 unsigned int          f_gid;   /*  идентификатор группы владельца, GID */
 int                   f_error;    /* код ошибки */
 struct file_ra_state  f_ra;       /*состояние предварительного считывания*/
 unsigned long         f_version;  /* номер версии */
 void                  *f_security; /* модуль безопасности */
 void                  *private_data;  /* привязка для драйвера терминала*/
 struct list_head     f_ep_links; /*список ссылок eventpoll (опрос событий) */
 spinlock_t            f_ep_lock;  /* блокировка eventpoll */
 struct address_space  *f_mapping; /* отображение в страничном кэше */
};
```

По аналогии с объектом элемента каталога объект файла на самом деле не соответствует никакой структуре, которая хранится на жестком диске. Поэтому в этой структуре нет никакого флага, который бы указывал, что объект изменен (dirty) и требует обратной записи на диск. Объект file указывает на связанный с ним объект dentry с помощью указателя f_dentry. Объект dentry в свою очередь содержит указатель на связанный с ним индекс файла, который содержит информацию о том, изменен ли файл.

Файловые операции

Как и для других объектоп подсистемы VFS, таблица файловых операций является важной структурой. Операции, связанные со структурой struct file, — это знакомые системные вызовы, составляющие основу системных вызовов ОС Unix.

Методы работы с файловым объектом хранятся в структуре file_operations и определены в файле <linux/fs.h> следующим образом.

```
struct file_operations {
        struct module *owner;
        loff_t (*llseek) (struct file *, loff_t, int);
        ssize_t (*read) (struct file *, char *, size_t, loff_t *);
        ssize_t (*aio_read) (struct kiocb *, char *, size_t, loff_t);
        ssize_t (*write) (struct file *, const char *, size_t, loff_t * ) ;
        ssize_t (*aio_write) (struct kiocb *, const char *, size_t, loff_t);
        int (*readdir) (struct file *, void *, filldir_t);
        unsigned int (*poll) (struct file *, struct poll_table_struct * ) ;
        int (*ioctl) (struct inode *, struct file *,
                      unsigned int, unsigned long);
        int (*mmap) (struct file *, struct vm_area_struct * ) ;
        int (*open) (struct inode *, struct file * ) ;
        int (*flush) (struct file * ) ;
        int (*release) (struct inode *, struct file * ) ;
        int (*fsync) (struct file *, struct dentry *, int);
        int (*aio_fsync) (struct kiocb *, int);
        int (*fasync) (int, struct file *, int);
        int (*lock) (struct file *, int, struct file_lock * ) ;
        ssize_t (*readv) (struct file *, const struct iovec *,
                      unsigned long, loff_t * ) ;
        ssize_t (*writev) (struct file *, const struct iovec *,
                       unsigned long, loff_t * ) ;
        ssize_t (*sendfile) (struct file *, loff_t *, size_t,
                         read_actor_t,void* ) ;
        ssize_t (*sendpage) (struct file *, struct page *, int,
                         size_t, loff_t *, int);
        unsigned long (*get_unmapped_area) (struct file *, unsigned long,
                                    unsigned long, unsigned long,
                                    unsigned long);
        int (*check_flags) (int flags);
        int (*dir_notify) (struct file *filp, unsigned long arg);
        int (*flock) (struct file *filp, int cmd, struct file_lock *fl);
};
```

Файловые системы могут реализовать уникальную функцию для каждой из этих операций или использовать общий существующий метод. Общие методы нормально работают для обычных Unix-подобных файловых систем. Разработчики файловых систем не обязаны реализовать все эти функции, хотя основные методы должны быть реализованы. Если какой-либо метод не представляет интереса, то его можно установить в значение NULL.

Рассмотрим каждую операцию подробнее.

- `loff_t llseek(struct file *file, loff_t offset, int origin)`

  Эта функция устанавливает значения указателя текущей позиции в файле (file pointer) в заданное значение параметра offset. Функция вызывается из системного вызова lseek().

- ```
  ssize_t read(struct file *file,
               char *buf, size_t count,
               loff_t  *offset)
  ```

  Эта функция считывает count байт данных из указанного файла, начиная с позиции, заданной параметром offset, в буфер памяти, на который указывает параметр buf. После этого значение указателя текущей позиции в файле должно быть обновлено. Данная функция вызывается из системного вызова read().

- ```
  ssize_t aio_read(struct kiocb *iocb,
          char *buf, size_t count,
          loff_t  offset)
  ```

  Эта функция запускает асинхронную операцию считывания count байт данных из файла, который описывается параметром iocb, в буфер памяти, описанный параметром buf. Эта функция вызывается из системного вызова aio_read ().

- ```
  ssize_t write(struct file *file,
          const char *buf, size_t count,
          loff_t  *offset)
  ```

  Эта функция записывает count байт данных в указанный файл, начиная с позиции offset. Данная функция вызывается из системного вызова write ().

- ```
  ssize_t aio_write(struct kiocb *iocb,
                const char *buf,
                size_t count, loff_t offset)
  ```

  Эта функция запускает асинхронную операцию записи count байт данных в файл, описываемый параметром iocb, из буфера памяти, на который указывает параметр buf. Данная функция вызывается из системного вызова aio_write.

- `int readdir(struct file *file, void *dirent, filldir_t filldir)`

  Эта функция возвращает следующий элемент из списка содержимого каталога. Данная функция вызывается из системного вызова readdir ().

- ```
  unsigned int poll(struct file *file,
          struct poll_table_struct  *poll_table)
  ```

  Эта функция переводит вызывающий процесс в состояние ожидания для ожидания действий, которые производятся с указанным файлом. Она вызывается из системного вызова poll().
  ```

- `int ioctl(struct inode *inode,`
  `struct file *file,`
  `unsigned int cmd,`
  `signed long arg)`

  Эта функция используется для того, чтобы отправлять устройствам пары значений команда/аргумент. Функция используется, когда открытый файл— это специальный файл устройства. Данная функция вызывается из системного вызова `ioctl()`.

- `int mmap(struct file *file, struct vra_area_struct *vma)`

  Эта функция отображает указанный файл на область памяти в указанном адресном пространстве и вызывается из системного вызова mmap().

- `int open(struct inode *inode, struct file *file)`

  Эта функция создает новый файловый объект и связывает его с указанным файловым индексом. Она вызывается из системного вызова open ().

- `int flush(struct file *file)`

  Эта функция вызывается подсистемой VFS, когда уменьшается счетчик ссылок на открытый файл. Назначение данной функции зависит от файловой системы.

- `int release(struct inode *inode, struct file *file)`

  Эта функция вызывается подсистемой VFS, когда исчезает последняя ссылка на файл, например, когда последний процесс, который использовал соответствующий файловый дескриптор, вызывает функцию `close ()` или завершается. Назначение этой функции также зависит от файловой системы.

- `int fsync(struct file *file,`
  `struct dentry *dentry,`
  `int datasync)`

  Эта функция вызывается из системного вызова fsync() для записи на диск всех закэшированных данных файла.

- `int aio_fsync(struct kiocb *iocb, int datasync)`

  Эта функция вызывается из системного вызова `aiofsync ()` для записи на диск всех закэшированных данных файла, связанного с параметром iocb.

- `int fasyn (fint fd, struct file *file, int on)`

  Эта функция разрешает или запрещает отправку сигнала для уведомлении о событиях при асинхронном вводе-выводе.

- `int lock(struct file *file, int cmd, struct file_lock *lock)`

  Эта функция управляет файловыми блокировками для данного файла.

- `ssize_t readv(struct file *file,`
  `const struct iovec *vector,`
  `unsigned long count,`
  `loff_t *offset)`

  Эта функция вызывается из системного вызова `readv ()` для считывания данных из указанного файла в count буферов, которые описываются параметром `vector`. После этого указатель текущей позиции файла должен быть соответственным образом увеличен.

- ```
  ssize_t writev(struct file *file,
          const struct iovec *vector,
          unsigned long count,
          loff_t *offset)
  ```

  Эта функция вызывается из системного вызова writev () для записи в указанный файл буферов, описанных параметром vector; количество буферов равно count. После этого должно быть соответственным образом увеличено значение текущей позиции в файле.

- ```
  ssize_t sendfile(struct file *file,
                   loff_t *offset,
                   size_t size,
                   read_actor_t actor,
                   void *target)
  ```

  Эта функция вызывается из системного вызова sendfile () для копирования данных из одного файла в другой. Она выполняет операцию копирования исключительно в режиме ядра и позволяет избежать дополнительного копирования данных в пространство пользователя.

- ```
  ssize_t sendpage(struct file *file,
                   struct page *page,
                   int offset, size_t size,
                   loff_t *pos, int more)
  ```

  Эта функция используется для отправки данных из одного файла в другой.

- ```
  unsigned long get_unmapped_area(struct file*file,
                   unsigned long addr,
                   unsigned long len,
                   unsigned long offset,
                   unsigned long flags)
  ```

  Эта функция получает неиспользуемое пространство адресов для отображения данного файла.

- ```
  int check_flags(int flags)
  ```

  Эта функция используется для проверки корректности флагов, которые передаются в системный вызов fcntl (), при использовании команды SETFL. Как и в случае многих операций подсистемы VFS, для файловой системы нет необходимости реализовать функцию check_flags (). Сейчас это сделано только для файловой системы NFS. Эта функция позволяет файловой системе ограничить некорректные значения флагов команды SETFL в обобщенном системном вызове fcntl(). Для файловой системы NFS не разрешается использовать комбинацию флагов O_APPEND и O_DIRECT.

- ```
  int flock(struct file *filp, int cmd, struct file_lock *fl)
  ```

  Эта функция используется для реализации системного вызова flock(), который служит для выполнения рекомендованных блокировок.

Структуры данных, связанные с файловыми системами

В дополнение к фундаментальным объектам подсистемы VFS, ядро использует и другие стандартные структуры данных для управления данными, связанными с файловыми системами. Первый объект используется для описания конкретного типа файловой системы, как, например, ext.3 или XFS. Вторая структура данных используется для описания каждого экземпляра смонтированной файловой системы.

Поскольку операционная система Linux поддерживает множество файловых систем, то ядро должно иметь специальную структуру для описания возможностей и поведения каждой файловой системы.

```
struct file_system_type {
 const char               *name;     /* название файловой системы */
 struct subsystem         subsys;    /* объект подсистемы sysfs */
 int                      fs_flags;  /* флаги типа файловой системы */

 /* следующая функция используется для считывания суперблока с диска */
 struct super_block * (*get_sb) (struct file_system_type *, int, char*, void * ) ;

 /* эта функция используется для прекращения доступа к суперблоку */
 void (*kill_sb) (struct super_block * ) ;

 struct module            *owner;    /* соответствующий модуль (если есть) */
 struct file_system_type  *next;     /* следующая файловая система в списке */
 struct list_head         fs_supera; /* список объектов типа суперблок */
};
```

Функция get_sb () служит для считывания суперблока с диска и заполнения объекта суперблока соответствующими данными при монтировании файловой системы. Остальные параметры описывают свойства файловой системы.

Для каждого типа файловой системы существует только одна структура file_system_type, независимо от того, сколько таких файловых систем смонтировано и смонтирован ли хотя бы один экземпляр соответствующей файловой системы.

Значительно интереснее становится, когда файловая система монтируется, при этом создается структура vfsmount. Эта структура используется для представления конкретного экземпляра файловой системы, или, другими словами, точки монтирования.

Структура vfsmount определена в файле <linux/mount.h> следующим образом.

```
struct vfsmount {
 struct list_head         mnt_hash;      /* список хеш-таблицы */
 struct vfsmount          *mnt_parent;   /* родительская файловая система */
 struct dentry            *mnt_mountpoint; /* объект элемента каталога
                                              точки монтирования */
 struct dentry            *mnt_root;  /* объект элемента каталога корня
                                         данной файловой системы */
 struct super_block       *mnt_sb;    /* суперблок данной файловой системы */
 struct list_head         mnt_mounts; /* список файловых систем,
                                         смонтированных к данной */
 struct list_head         mnt_child;  /* потомки, связанные с родителем */
```

```
 atomic_t                mnt_count;  /* счетчик использования */
 int                     mnt_flags;  /* флаги монтирования */
 char                    *mnt_devname; /* имя смонтированного устройства */
 struct list_hcad        mnt_list;   /* список дескрипторов */
 struct listhead         mnt_fslinkk;   /* истекший список, специфичный
                                           для файловой системы */
 struct namespace        *mnt_namespace;  /* связанное пространство имен */
};
```

Самая сложная задача — это поддержание списка всех точек монтирования и взаимоотношений между данной файловой системой и другими точками монтирования. Эта информация хранится в различных связанных списках структуры vfsmount.

Структура vfsmount также содержит иоле mnt_flags. В табл. 12.1 приведен список стандартных флагов монтирования.

Таблица 12.1. Список стандартных флагов монтирования

Флаг	Описание
MNT_NOSUID	Запрещает использование флагов setuid и setgid для бинарных файлов на файловой системе
MNT_NODEV	Запрещает доступ к файлам устройств на файловой системе
MNT_NOEXEC	Запрещает выполнение программ на файловой системе

Эти флаги полезны, в основном, для сменных носителей, которым администратор не доверяет.

Структуры данных, связанные с процессом

Каждый процесс в системе имеет свои открытые файлы, корневую файловую систем); текущий рабочий каталог, точки монтирования и т.д. Следующие три структуры данных связывают вместе подсистему VFS и процессы, которые выполняются в системе. Это структуры files_struct, fs_struct и namespace.

Структура files_struct определена в файле <linux/file.h>. Адрес этой структуры хранится в поле files дескриптора процесса. В данной структуре хранится вся информация процесса об открытых файлах и файловых дескрипторах. Эта структура, с комментариями, имеет следующий вид.

```
struct files_struct {
 atomic_t    count;          /* счетчик ссылок на данную структуру */
 spinlock_t  file_lock;      /* блокировка для защиты данной структуры */
 int         max_fds;        /* максимальное количество файловых объектов */
 int         max_fdset;      /*максимальное количество файловых дескрипторов */
 int         next_fd;        /* номер следующего файлового дескриптора */
 struct file **fd;           /* массив всех файловых объектов */
 fd_set      *close on exec;/* файловые дескрипторы, которые должны
                               закрываться при вызове exec() */
 fd_set      *open_fds;      /* указатель на дескрипторы открытых файлов */
 fd_set    close_on_exec    init;  /* первоначальные файлы для закрытия
                                      при вызове exec () */
 fd_set      open_fds_init;  /* первоначальный набор файловых дескрипторов */
 struct file  *fd_array[NR_OPEN_DEFAULT];  /* массив файловых объектов */
};
```

Массив fd указывает на список открытых файловых объектов. По умолчанию это массив fd_array. Так как по умолчанию значение константы NR_OPEN_DEFAULT равно 32, то это соответствует 32 файловым объектам. Если процесс открывает больше 32 файловых объектов, то ядро выделяет новый массив и присваивает полю fd указатель на него. При таком подходе доступ к небольшому количеству файловых объектов осуществляется быстро, потому что они хранятся в статическом массиве. В случае, когда процесс открывает аномально большое количество файлов, ядро может создать новый массив. Если большинство процессов в системе открывает больше 32 файлов, то для получения оптимальной производительности администратор может увеличить значение константы NR_OPEN_DE_FAULT с помощью директивы препроцессора. Следующая структура данных, связанная с процессом, — это структура fs_struct, которая содержит информацию, связанную с процессом, и на которую указывает поле fs дескриптора процесса. Эта структура определена в файле <linux/fs_struct.h> и имеет следующий вид с поясняющими комментариями.

```
struct fs struct {
 atomic_t        count;    /* счетчик ссылок на структуру */
 rwlock_l         lock;    /* блокировка для защиты структуры */
 int             umask;   /* права доступа к файлу, используемые
                             по умолчанию */
 struct dentry   *root;    /* объект dentry корневого каталога */
 struct dentry   *pwd;     /* объект dentry текущего рабочего каталога */
 struct dentry   *allroot; /* объект dentry альтернативного корня */
 struct vfsmounL *rootmnt; /* объект монтирования корневого каталога */
 struct vfsmount *pwdmnt;  /* объект монтирования текущего рабочего каталога*/
 struct vfsmount *altrootrnnt; /* объект монтирования альтернативного корня*/
};
```

Эта структура содержит текущий рабочий каталог и корневой каталог данного процесса.

Третья, и последняя, структура— это структура namespace, которая определена в файле <linux/namespace.h> и на экземпляр которой указывает поле namespace дескриптора процесса. Пространства имен, индивидуальные для каждого процесса, были введены в ядрах Linux серии 2.4. Это позволило создать для каждого процесса уникальное представление о смонтированных файловых системах. Иными словами, процесс может иметь не только уникальный корневой каталог, но и полностью уникальную иерархию смонтированных файловых систем, если это необходимо. Как обычно, ниже приведена соответствующая структура данных с комментариями.

```
struct namespace {
 atomic_t            count;   /* счетчик ссылок на структуру */
 struct vfsmount     *root;   /* объект монтирования корневого каталога */
 struct list_head    list;    /* список точек монтирования */
 struct rw_semaphore sem;     /* семафор для защиты пространства имен */
};
```

Поле list представляет собой двухсвязный список смонтированных файловых систем, которые составляют пространство имен.

Каждый дескриптор процесса имеет связанные с ним рассмотренные структуры данных. Для большинства процессов их дескриптор процесса указывает на уникальную структуру files_struct и структуру fs_struct. Однако для процессов,

созданных с флагами CLONE_FILES и CLONE_FS, эти структуры являются совместно используемыми[8]. Отсюда следует, что несколько дескрипторов процессов могут указывать на одну и ту же структуру files_struct, или структуру fs_struct. Поле count каждой структуры содержит счетчик использования, что предотвращает уничтожение структуры данных, когда ее использует хотя бы один процесс.

Структура namespace используется несколько по-другому. По умолчанию вес процессы совместно используют одно пространство имен (и соответственно одну иерархию файловых систем). Только когда для системного вызова clone () указан флаг CLONE_NEWNS, для процесса создается уникальная копия пространства имен. Поскольку для большинства процессов этот флаг не указывается, процессы обычно наследуют пространство имен родительского процесса. Следовательно, для большинства систем существует только одно пространство имен.

Файловые системы в операционной системе Linux

Операционная система Linux поддерживает большой набор файловых систем, от "родных" ext2 и ext3 до сетевых файловых систем, таких как NFS или Coda. Сейчас в официальном ядре ОС Linux поддерживается более 50 файловых систем. Уровень VFS обеспечивает все эти разнообразные файловые системы общей базой для их реализации и общим интерфейсом для работы со стандартными системными вызовами. Следовательно, уровень виртуальной файловой системы позволяет четким образом реализовать поддержку новых файловых систем в операционной системе Linux, а также дает возможность работать с этими файловыми системами с помощью стандартных системных вызовов Unix.

В этой главе было описано назначение подсистемы VFS и рассмотрены соответствующие структуры данных, включая такие важные объекты, как inode, dentry и superblock. В главе 12, "Виртуальная файловая система", будет рассказано о том, как данные физически поступают на файловые системы.

[8] Для создания потоков обычно укапываются флаги *CLONE_FILES* и *CLONB_FS,* поэтому они совместно используют структуры *£iles_struct* и *fs_struct.* С другой стороны, для обычных процессов эти флаги не указываются, поэтому для каждого процесса существует своя информация о файловой системе и своя таблица открытых файлов.

13

Уровень блочного ввода-вывода

Устройства блочного ввода-вывода (блочные устройства, устройства ввода-вывода блоками, block devices) — это аппаратные устройства, которые позволяют случайным образом (т.е. не обязательно последовательно) осуществлять доступ к фрагментам данных фиксированного размера, называемых блоками. Наиболее часто встречающееся устройство блочного впода-вывода — это жесткий диск, но существуют и другие блочные устройства, например устройства работы с гибкими дисками, оптическими компакт-дисками (CD-ROM) и флеш-памятью. Следует обратить внимание, что файловые системы монтируются с таких устройств. Именно таким образом обычно и осуществляется доступ к устройствам блочного ввода-вывода.

Другой фундаментальный тип устройства — это устройство посимвольного ввода-вывода (символьное устройство, character device, char device). Это— устройство, к которому можно обращаться, только как к последовательному потоку данных, т.е. байт за байтом. Пример символьных устройств — это последовательный порт и клавиатура. Если же устройстпо позволяет обращаться к данным случайным образом (не последовательно), то это блочное устройство.

Существенное различие между этими типами устройств проявляется, в основном, в возможности случайного доступа к данным, т.е. в возможности производить *поиск (seek)* по устройству, перемещаясь из одной позиции в другую. Как пример, рассмотрим клавиатуру. Драйвер устройства клавиатуры на выходе выдает поток данных. Когда печатают слово "fox", то драйвер клавиатуры возвращает поток данных, в котором три символа идут строго в указанном порядке. Считывание символов в другом порядке или считывание какого-нибудь другого символа, кроме следующего символа в потоке, имеет немного смысла. Поэтому драйвер клавиатуры — это устройство посимвольного ввода-вывода, он позволяет на выходе получить поток символов, которые пользователь вводит на клавиатуре. Операция чтения данных с устройства возвращает сначала символ "f", затем символ "o" и в конце символ "x". Когда нажатий клавиш нет, то поток— пустой. Жесткий диск же работает по-другому Драйвер жесткого диска может потребовать чтения содержимого определенного блока, а затем прочитать содержимое другого блока, и эти блоки не обязательно должны следовать друг за другом. Поэтому доступ к данным жесткого диска может выполняться случайным образом, а не как к потоку данных, и поэтому жесткий диск— блочное устройство.

Управление блочными устройствами в ядре требует большего внимания, подготовки и работы, чем управление устройствами посимвольного ввода-вывода. Все это потому, что символьные устройства имеют всего одну позицию — текущую, в то время как блочные устройства должны иметь возможность перемещаться туда и обратно между любыми позициями на физическом носителе информации. Оказывается, что нет необходимости создавать в ядре целую подсистему для обслуживания символьных устройств, а для блочных устройств это необходимо. Такая подсистема необходима отчасти из-за сложности блочных устройств. Однако основная причина такой мощной поддержки в том, что блочные устройства достаточно чувствительны к производительности. Выжать максимум производительности из жесткого диска значительно важнее, чем получить некоторый прирост скорости при работе с клавиатурой. Более того, как будет видно дальше, сложность блочных устройств обеспечивает большой простор для таких оптимизаций. Предмет данной главы — как ядро управляет работой блочных устройств и запросами к этим устройствам. Рассматриваемая часть ядра называется *уровнем, блочного ввода-вывода (block I/O layer)*. Интересно, что усовершенствование подсистемы блочного ввода-вывода было одной из целей разрабатываемой серии ядра 2.5. В этой главе рассматриваются все новые особенности уровня блочного ввода-вывода, которые появились в ядрах серии 2.6.

Анатомия блочного устройства

Наименьший адресуемый элемент блочного устройства называется сектором. Размеры секторов — это числа, которые являются целыми степенями двойки, однако наиболее часто встречающийся размер — 512 байт. Размер сектора— это физическая характеристика устройства, а сектор — фундаментальный элемент блочного устройства. Устройства не могут адресовать или другим образом работать с элементами данных, размер которых меньше, чем один сектор, тем не менее многие блочные устройства могут передавать несколько секторов за один раз. Хотя большинство блочных устройств и имеет размер сектора, равный 512 байт, все же существуют и другие стандартные размеры сектора (например, большинство компакт-дисков CD-ROM имеют размер сектора, равный 2 Кбайт).

У программного обеспечения несколько другие цели, и поэтому там существует другая минимально адресуемая единица, которая называется блок. Блок— это абстракция файловой системы, т.е. все обращения к файловым системам могут выполняться только с данными, кратными размеру блока. Хотя физические устройства сами по себе адресуются на уровне секторов, ядро выполняет все дисковые операции в терминах блоков. Так как наименьший возможный адресуемый элемент —- это сектор, то размер блока не может быть меньше размера одного сектора и должен быть кратен размеру сектора. Более того, для ядра (так же как и для аппаратного обеспечения в случае секторов) необходимо, чтобы размер блока был целой степенью двойки. Ядро также требует, чтобы блок имел размер, не больший, чем размер страницы памяти (см. главу 11, "Управление памятью" и главу 12, "Виртуальная файловая система")[1].

[1] Это ограничение является искусственным и в будущем оно может быть отменено. Тем не менее требование, чтобы размер блока был меньше или равен размеру страницы памяти, позволяет значительно упростить ядро.

Поэтому размер блока равен размеру сектора, умноженному на число, которое является целой степенью двойки. Наиболее часто встречающиеся размеры блока— это 512 байт, один килобайт и четыре килобайта.

Часто сбивает с толку то, что некоторые люди называют секторы и блоки поразному. Секторы, наименьшие адресуемые элементы устройства, иногда называют "аппаратными секторами" (hardware sector) или "блоками аппаратного устройства" (device block). В то время как блоки, наименьшие адресуемые единицы файловых систем, иногда называются "блоками файловой системы" (filesyst.em block) или "блоками ввода-вывода" (I/O block). В этой главе будут использованы термины "сектор" (sector) и "блок" (block), однако следует помнить и о других возможных названиях. На рис. 13.1 показана диаграмма соответствия между секторами и блоками.

Существует еще одна часто встречающаяся терминология. По крайней мере в отношении.к жестким дискам, используются термины *кластеры,* (clusters), *цилиндры* (cylinder, дорожка) и *головки* (head). Такие обозначения являются специфическими только для некоторых типов блочных устройств, они, в основном, невидимы для пользовательских программ и в этой главе рассматриваться не будут. Причина, по которой секторы являются важными для ядра, состоит в том, что все операции ввода-вывода должны выполняться в единицах секторов. Поэтому более высокоуровневые концепции ядра, которые основаны на блоках, являются надстройками над секторами.

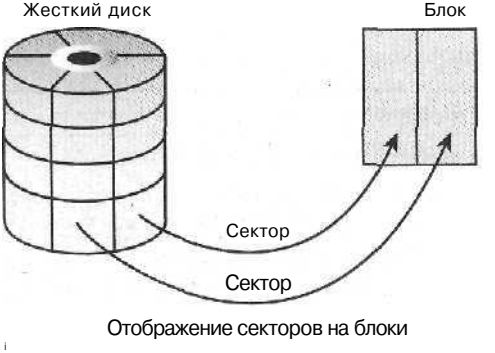


Рис. 13.1. Связь между секторами и бликами

Буферы и заголовки буферов

Когда блок хранится в памяти (скажем, после считывания или в ожидании записи), то он хранится в структуре данных, называемой *буфером (buffer).* Каждый буфер связан строго с одним блоком. Буфер играет роль объекта, который представляет блок в оперативной памяти. Вспомним, что блок состоит из одного или больше секторов, но по размеру не может быть больше одной страницы памяти. Поэтому одна страница памяти может содержать один или больше блоков. Поскольку для ядра требуется некоторая управляющая информация, связанная с данными (например, какому блочному устройству и какому блоку соответствует буфер), то каждый буфер связан со своим дескриптором. Этот дескриптор называется *заголовком, буфера (buffer head)* и представляется с помощью структуры `struct buffer_head`.

Структура buffer_head содержит информацию, которая необходима ядру для управления буферами и определена в файле <linux/buffer_head.h>.

Рассмотрим эту структуру с комментариями, которые описывают назначение каждого поля.

```
. struct buffer_head {
   unsigned long      b_state;          /* флаги состояния буфера */
   atomic_t           b_count;          /* счетчик использования буфера */
   struct buffer_head *b_this_page;     /* список буферов в текущей
                                           странице памяти */
   struct page        *b_page;          /* соответствующая страница памяти */
   sector_t           b_blocknr;        /* логический номер блока */
   u32                b_size;          /* размер блока (в байтах) */
   char               *b_data;         /*указатель на буфер в странице памяти*/
   struct block_device *b_bdev;        /* соответствующее блочное устройство*/
   bh_end_io_t        *b_end_io;       /* метод завершения ввода-вывода */
   void               *b_private;      /* данные метода завершения */
   struct list_head   b_assoc_buffers; /* список связанных отображений */
  };
```

Поле b_state содержит состояние определенного буфера. Это значение может содержать один или несколько флагов, которые перечислены в табл. 13.1. Возможные значения флагов описаны в виде перечисления bh_state_bits, которое описано в файле <Linux/buffer_head.h>.

Таблица 13.1. Значения флагов поля bh_state

Флаг состояния	Назначение
BH_Uptodate	Буфер содержит действительные данные
BH_Dirty	Буфер изменен (содержимое буфера новее соответствующих данных на диске, и поэтому буфер должен быть записан на диск)
BH_Lock	Для буфера выполняется операция чтения-записи дисковых данных, и буфер заблокирован, чтобы предотвратить конкурентный доступ
BH_Req	Буфер включен в запрос
BH_Mapped	Буфер является действительным и отображается на дисковый блок
BH_NEW	Буфер только что выделен и к нему еще не было доступа
BH_Async_ Read	Для буфера выполняется асинхронная операция чтения
вн_Азупс Write	Для буфера выполняется асинхронная операция записи
вн_Delay	С буфером еще не связан дисковый блок
BH_Boundary	Буфер является последним в последовательности смежных блоков — следующий за ним блок не является смежным с этой серией

Перечисление bh_state_bits также содержит в качестве последнего элемента флаг BH_PrivateStart. Этот флаг не является разрешенным значением флага, а соответствует первому биту, который можно использовать по усмотрению разработчиков кода. Все биты, номер которых больше или равен значению BH_PrivateStart, не используются подсистемой блочного ввода-вывода и безопасно могут использоваться драйверами, которым необходимо хранить информацию в поле b_state.

Флаги, которые используются драйверами, могут быть определены на основании значения этого флага, что позволяет гарантированно избежать перекрытия с битами, которые официально используются уровнем блочного ввода-вывода.

Поле b_count — это счетчик использования буфера. Значение этого поля увеличивается и уменьшается двумя функциями, которые определены в файле <linux/buffer_head.h> следующим образом.

```
static inline void get_bh (struct buffer_head *bh)
{
        atomic_inc{&bh->b_count);
}
static inline void put_bh (struct buffer_head *bh)
{
        atomic_dec (&bh->b_count);
}
```

Перед тем как манипулировать заголовком буфера, необходимо увеличить значение счетчика использования с помощью функции get_bh() , что гарантирует, что во время работы буфер не будет освобожден. Когда работа с заголовком буфера закончена, необходимо уменьшить значение счетчика, ссылок с помощью функции put_bh ().

Физический блок на жестком диске, которому соответствует буфер, — это блок с логическим номером b_blocknr, который находится на блочном устройстве b_bdev.

Физическая страница памяти, в которой хранятся данные буфера, соответствует значению поля b_page. Поле b_data — это указатель прямо на данные блока (которые хранятся где-то в странице памяти b_page), размер блока хранится в поле b_size. Следовательно, блок хранится в памяти, начиная с адреса b_data и заканчивая адресом (b_data + b_size).

Назначение заголовка буфера— это описание отображения между блоком на диске и буфером в физической памяти (т.е. последовательностью байтов, которые хранятся в указанной странице памяти). Выполнение роли дескриптора отображения буфер-блок —единственное назначение этой структуры данных ядра.

В ядрах до серии 2.6 заголовок буфера был значительно более важной структурой данных. По существу, это была единица ввода-вывода данных в ядре. Он не только выполнял роль дескриптора для отображения буфер-блок-страница физической памяти, но и выступал контейнером для всех операций блочного ввода-вывода. Это приводило к двум проблемам. Первая проблема заключалась в том, что заголовок буфера был большой и громоздкой структурой данных (сегодня он несколько уменьшился в размерах), а кроме того, выполнение операций блочного ввода-вывода в терминах заголовков буферов было непростой и довольно непонятной задачей. Вместо этого, ядру лучше работать со страницами памяти, что одновременно и проще и позволяет получить большую производительность. Использовать большой заголовок буфера, описывающий отдельный буфер (который может быть размером со страницу памяти), — неэффективно. В связи с этим в ядрах серии 2.6 было сделано много работы, чтобы дать возможность ядру выполнять операции непосредственно со страницами памяти и пространствами адресов, вместо операций с буферами. Некоторые из этих операций обсуждаются в главе 15, "Страничный кэш и обратная запись страниц", где также рассматривается структура address_space и демоны pdflush.

Вторая проблема, связанная с заголовками буферов, — это то, что они описывают только один буфер. Когда заголовок буфера используется в качестве контейнера для операций ввода-вывода, то это требует, чтобы ядро разбивало потенциально большую операцию блочного ввода-вывода на множество мелких структур buffer_head, что в свою очередь приводит к ненужным затратам памяти для храпения структур данных. В результате, основной целью при создании серии ядра 2.5 была разработка нового гибкого и быстрого контейнера для операций блочного ввода-вывода. В результат появилась структура bio, которая будет рассмотрена в следующем разделе.

Структура bio

Основным контейнером для операций ввода-вывода в ядре является структура bio, которая определена в файле <linux/bio.h>. Эта структура представляет активные операции блочного ввода-вывода в виде списка *сегментов (segment)*. Сегмент — это участок буфера, который является непрерывным в физической памяти, т.е. отдельные буферы не обязательно должны быть непрерывными в физической памяти. Благодаря тому, что буфер может представляться в виде нескольких участков, структура bio даст возможность выполнять операции блочного ввода-вывода, даже если данные одного буфера хранятся в разных местах памяти. Ниже показана структура bio с комментариями, описывающими назначение каждого поля.

```
struct bio {
 sector_t       bi_sector;       /* соответствующий сектор на диске */
 struct bio      *bi_next;       /* список запросов */
 struct block_device  *bi_bdev;  /* соответствующее блочное устройство */
 unsigned long   bi_flags;       /* состояние и флаги команды */
 unsigned long   bi_rw;          /* чтение или запись? */
 unsigned short  bi_vcnt;        /* количество структур bio vec
                                    в массиве bi_io_vec */
 unsigned short  bi_idx;         /* текущий индекс в массиве bi_io_vec */
 unsigned short  bi_phys_segments; /*количество сегментов после объединения*/
 unsigned short  bi_hw_segments;  /* количество сегментов после
                                    перестройки отображения */
 unsigned int    bi_size;        /* объем данных для ввода-вывода */
 unsigned int    bi_hw_front_size;/* размер первого объединяемого сегмента*/
 unsigned int    bi_hw_front_size;/* размер последнего объединяемого
                                    сегмента */
 unsigned int    bi_max_vecs;    /* максимально возможное количество
                                    структур bio_vecs */
 struct bio_vec  *bi_io_vec;     /* массив структур bio_vec */
 bio_end_io_t    *bi_end_io;     /* метод завершения ввода-вывода */
 atomic_t        bi_cnb;         /* счетчик использования */
 void            *bi_private;    /* поле для информации создателя */
 bio_destructor_t *bi_destructor;  /* деструктор */
};
```

Главное назначение структуры bio — это представление активной (выполняющейся) операции блочного ввода-вывода. В связи с этим большинство полей этой структуры являются служебными. Наиболее важные поля — это bi_io_vecs, bi_vcnt и bi_idx.

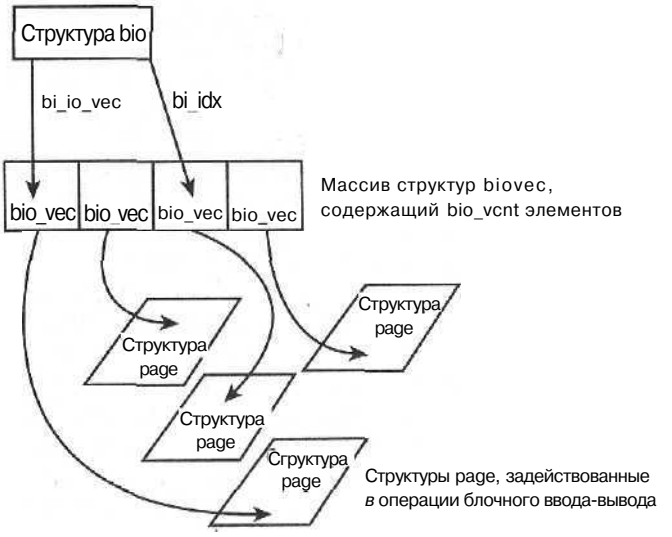


Рис. 13.2. Связь между структурами struct bio, struct b,io_vec и struct page

Поле bi_io_vecs указывает па начало массива структур bio_vec, Эти структуры используются в качестве списка отдельных сегментов в соответствующей операции блочного ввода-вывода. Каждый экземпляр структуры bio_vec представляет собой вектор следующего вида: <страница памяти, смещение, размер>, который описывает определенный сегмент, соответственно страницу памяти, где этот сегмент хранится, положение блока — смещение внутри страницы — и размер блока. Массив рассмотренных векторов описывает весь буфер полностью. Структура bio_vec определена в файле <linux/bio.h> следующим образом.

```
struct bio_vec {
   /* указатель на страницу физической памяти, где находится этот буфер */
   struct   page   *bv_page;

   /* размер буфера в байтах */
   Unsigned int    bv_len;

   /* смещение в байтах внутри страницы памяти, где находится буфер */
   unsigned  int  bv_offset;
   };
```

Для каждой операции блочного ввода-выпода создается массив из bi_vcnt элементов типа bio_vec, начало которого содержится в поле bi _io_vecs. В процессе выполнения операции блочного ввода-вывода поле bi_idx используется для указания па текущий элемент массива.

В общем, каждый запрос на выполнение блочного ввода-вывода представляется с помощью структуры bio. Каждый такой запрос состоит из одного или более блоков, которые хранятся в массиве структур bio_vec. Каждая из этих структур представляет собой вектор, который описывает положение в физической памяти каждого сегмента запроса. На первый сегмент для операции ввода-вывода указывает поле bi_io_vec. Каждый следующий сегмент следует сразу за предыдущим. Всего

в массиве bi_vcnt сегментов. В процессе того, как уровень блочного ввода-вывода обрабатывает сегменты запроса, обновляется значение поля bi_idx, чтобы его значение соответствовало номеру текущего сегмента. На рис. 13.2 показана связь между структурами bio, bio_vec и page.

Поле bi_idx указывает на текущую структуру bio_vec в массиве, что позволяет уровню блочного ввода-вывода поддерживать частично выполненные операции блочного ввода-вывода. Однако более важное использование состоит в том, что драйверы таких устройств, как RAID (Redundant Array of Inexpensive/Independent Disks, массив недорогих/независимых дисковых устройств с избыточностью — специальный способ использования жестких дисков, при котором один логический том может быть распределен но нескольким физическим дискам для увеличения надежности или производительности), могут одну структуру bio, которая изначально была адресована одному устройству, разбивать на несколько частей, которые предназначаются различным дискам RAID массива. Все, что необходимо сделать драйверу RAID, это создать необходимое количество копий структуры bio, которая предназначалась одному устройству, и изменить для каждой копии значение поля bi_idx, чтобы оно указывало на ту часть массива, откуда каждый диск должен начать свою операцию ввода-вывода.

Структура bio содержит счетчик использования, который хранится в поле bi_cnt. Когда значение этого поля становится равным нулю, структура удаляется, и занятая память освобождается. Следующие две функции позволяют управлять счетчиком использования.

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

Первая увеличивает на единицу значение счетчика использования, а вторая-уменьшает значение этого счетчика на единицу и, если это значение становится равным нулю, уничтожает соответствующую структуру bio. Перед тем как работать с активной структурой bio, необходимо увеличить счетчик использования, чтобы гарантировать, что экземпляр структуры не будет удален во время работы. После окончания работы необходимо уменьшить счетчик использования.

И наконец, поле bio_private — это поле данных создателя (владельца) структуры. Как правило, это поле необходимо считывать или записывать только тому, кто создал данный экземпляр структуры bio.

Сравнение старой и новой реализаций

Между заголовками буферов и новой структурой bio существуют важные отличия. Структура bio представляет операцию ввода-вывода, которая может включать одну или больше страниц в физической памяти. С другой стороны, заголовок буфера связан с одним дисковым блоком, который занимает не более одной страницы памяти. Поэтому использование заголовков буферов приводит к ненужному делению запроса ввода-вывода на части, размером в один блок, только для того, чтобы их потом снова объединить. Работа со структурами bio выполняется быстрее, эта структура может описывать несмежные блоки и не требует без необходимости разбивать операции ввода-вывода на части.

Переход от структуры struct buffer_head к структурам struct bio позволяет получить также и другие преимущества.

- Структура bio может легко представлять верхнюю память (см. главу 11), так как структура struct bio работает только со страницами физической памяти, а не с указателями.
- Структура bio может представлять как обычные страничные операции ввода-вывода, так и операции непосредственного (direct) ввода-вывода (т.е. те, которые не проходят через страничный кэш; страничный кэш обсуждается в главе 15).
- Структура bio позволяет легко выполнять операции блочного ввода-вывода типа распределения-аккумуляции (scatter-gather), в которых данные находятся в нескольких страницах физической памяти.
- Структура bio значительно проще заголовка буфера, потому что она содержит только минимум информации, необходимой для представления операции блочного ввода-вывода, а не информацию, которая связана с самим буфером.

Тем не менее заголовки буферов все еще необходимы для функций, которые выполняют отображение дисковых блоков на страницы физической памяти. Структура bio не содержит никакой информации о состоянии буфера, это просто массив векторов, которые описывают один или более сегментов данных одной операции блочного ввода-вывода, плюс соответствующая дополнительная информация. Структура buffer_head необходима для хранения информации о буферах. Применение двух отдельных структур позволяет сделать размер обеих этих структур минимальным.

Очереди запросов

Для блочных устройств поддерживаются *очереди запросов (request queue),* в которых хранятся ожидающие запросы на выполнение операций блочного ввода-вывода. Очередь запросов представляется с помощью структуры request_queue, которая определена в файле <linux/blkdev.h>. Очередь запросов содержит двухсвязный список запросов и соответствующую управляющую информацию. Запросы добавляются в очередь кодом ядра более высокого уровня, таким как файловые системы. Пока очередь запросов не пуста, драйвер блочного устройства, связанный с очередью, извлекает запросы из головы очереди и отправляет их на соответствующее блочное устройство. Каждый элемент списка запросов очереди— это один запрос, представленный с помощью структуры struct request.

Запросы

Отдельные запросы представляются с помощью структуры struct request, которая тоже определена в файле <linux/blkdev.h>. Каждый запрос может состоять из более чем одной структуры bio, потому что один запрос может содержать обращение к нескольким смежным дисковым блокам. Обратите внимание, что хотя блоки на диске и должны быть смежными, данные этих блоков не обязательно должны быть смежными в физической памяти— каждая структура bio может содержать несколько сегментов (вспомните, сегменты — это непрерывные участки памяти, в которых хранятся данные блока), а запрос может состоять из нескольких структур bio.

Планировщики ввода-вывода

Простая отправка запросов на устройство ввода-вывода в том же порядке, в котором эти запросы направляет ядро, приводит к очень плохой производительности. Одна из наиболее медленных операций, которые *вообще могут* быть в компьютере,— это поиск по жесткому диску. Операция поиска — это позиционирование головки жесткого диска на определенный блок, которое может запять много миллисекунд. Минимизация количества операций поиска чрезвычайно критична для производительности всей системы.

Поэтому ядро не отправляет все запросы на выполнение операций блочного ввода-вывода жесткому диску в том же порядке, в котором они были получены, или сразу же, как только они были получены. Вместо этого, оно выполняет так называемые операции *слияния (объединения, merging)* и *сортировка (sorting),* которые позволяют значительно увеличить производительность всей системы[2]. Подсистема ядра, которая выполняет эти операции называется *планировщиком ввода-вывода (J/0 scheduler).*

Планировщик ввода-вывода разделяет дисковые ресурсы между ожидающими в очереди запросами впода-вывода. Это выполняется путем объединения и сортировки запросов ввода-вывода, которые находятся в очереди. Планировщик ввода-вывода не нужно путать с планировщиком выполнения процессов (см. главу 4, "Планирование выполнения процессов"), который делит ресурсы процессора между процессами системы. Эти две подсистемы похожи, но это— не одно и то же. Как планировщик выполнения процессов, так и планировщик ввода-вывода выполняют виртуализацию ресурсов для нескольких объектов. В случае планировщика процессов выполняется виртуализация процессора, который совместно используется процессами системы. Это обеспечивает иллюзию того, что процессы выполняются одновременно, и является основой многозадачных операционных систем с разделением времени, таких как Unix. С другой стороны, планировщики ввода-вывода выполняют виртуализацию блочных устройств для ожидающих выполнения запросов блочного ввода-вывода. Это делается для минимизации количества операций поиска по жесткому диску и для получения оптимальной производительности дисковых операций.

Задачи планировщика ввода-вывода

Планировщик ввода-вывода работает управляя очередью запросов блочного устройства. Он определяет, в каком порядке должны следовать запросы в очереди и в какое время каждый запрос должен передаваться на блочное устройство. Управление производится с целью уменьшения количества операций поиска по жесткому диску, что значительно повышает *общее быстродействие.* Определение "общее" здесь существенно. Планировщик ввода-вывода ведет себя "нечестно" по отношению к некоторым запросам и за счет этого повышает производительность системы в целом.

[2] Это необходимо подчеркнуть особо. Системы, не имеющие таких функций или в которых эти функции плохо реализованы, будут иметь очень плохую производительность даже при небольшом количестве операций блочного ввода-вывода.

Планировщики ввода-вывода выполняют две основные задачи: объединение и сортировку. Объединение — это слияние нескольких запросов в один. В качестве примера рассмотрим запрос, который поставлен в очередь кодом файловой системы, например чтобы прочитать порцию данных из файла (конечно, на данном этапе все уже выполняется на уровне секторов и блоков, а не на уровне файлов). Если в очереди уже есть запрос на чтение из соседнего сектора диска (например, другой участок того же файла), то два запроса могут быть объединены в один запрос для работы с одним или несколькими, расположенными рядом, секторами диска. Путем слияния запросов планировщик ввода-вывода уменьшает затраты ресурсов, связанные с обработкой нескольких запросов, до уровня необходимого на обработку одного запроса. Наиболее важно здесь то, что диск выполняет всего одну команду и обслуживание нескольких запросов может быть выполнено вообще без применения поиска. Следовательно, слияние запросов уменьшает накладные расходы и минимизирует количество операций поиска.

Теперь предположим, что наш запрос на чтение помещается в очередь запросов, но там нет других запросов на чтение соседних секторов. Поэтому нет возможности выполнить объединение этого запроса с другими запросами, находящимися в очереди. Можно просто поместить запрос в конец очереди. Но что если в очереди есть запросы к близкорасположенным секторам диска? Не лучше ли будет поместить новый запрос в очередь где-то рядом с запросами к физически близко расположенным секторам диска. На самом деле планировщики ввода-вывода именно так и делают, Вся очередь запросов поддерживается в отсортированном состоянии по секторам, чтобы последовательность запросов в очереди (насколько это возможно) соответствовала линейному движению по секторам жесткого диска. Цель состоит в том, чтобы не только уменьшить количество перемещений в каждом индивидуальном случае, но и минимизировать общее количество операций поиска таким образом, чтобы головка двигалась по прямой линии. Это аналогично алгоритму, который используется в лифте (elevator) — лифт не прыгает между этажами. Вместо этого он плавно пытается двигаться в одном направлении. Когда лифт доходит до последнего этажа в одном направлении, он начинает двигаться в другую сторону. Из-за такой аналогии планировщик ввода-вывода (а иногда только алгоритм сортировки) называют *лифтовым планировщиком (алгоритмом лифта, elevator).*

Лифтовой алгоритм Линуса

Рассмотрим некоторые планировщики ввода-вывода, применяемые в реальной жизни. Первый планировщик ввода-вывода, который мы рассмотрим, *называется Linus Elevator (лифтовой алгоритм Линуса).* Это не опечатка, действительно существует лифтовой планировщик, разработанный Лисусом Торвальдсом и названный в его честь! Это основной планировщик ввода-вывода в ядре 2.4. В ядре 2.6 его заменили другими планировщиками, которые мы ниже рассмотрим. Однако поскольку этот алгоритм значительно проще новых и в то же время позволяет выполнять почти те же функции, то он заслуживает внимания.

Лифтовой алгоритм Линуса позволяет выполнять как объединение, так и сортировку запросов. Когда запрос добавляется в очередь, вначале он сравнивается со всеми ожидающими запросами, чтобы обнаружить все возможные кандидаты па объединение. Алгоритм Линуса выполняет два типа объединения: *добавление в начало*

запроса (front merging) и *добавление в конец запроса (back merging)*. Тип объединения соответствует тому, с какой стороны найдено соседство. Если новый запрос следует перед существующим, то выполняется вставка в начало запроса. Если новый запрос следует сразу за существующим — добавление выполняется в конец очереди. В связи с тем, что секторы, в которых хранится файл, расположены по мере увеличения номера сектора и операции ввода-вывода чаще всего выполняются от начала файла до конца, а не наоборот, то при обычной работе вставка в начало запроса встречается значительно реже, чем вставка в конец. Тем не менее алгоритм Линуса проверяет и выполняет оба типа объединения,

Если попытка объединения была неудачной, то определяется возможное место вставки запроса в очередь (положение в очереди, в котором новый запрос наилучшим образом вписывается по номеру сектора между окружающими запросами). Если такое положение находится, то новый запрос помещается туда. Если подходящего места не найдено, то запрос помещается в конец очереди. В дополнение к этому, если в очереди найден запрос, который является достаточно старым, то новый запрос также добавляется в конец очереди. Это предотвращает ситуацию, в которой наличие большого количества запросов к близко расположенным секторам приводит к недостатку обслуживания других запросов. К сожалению, такая проверка "на старость" не очень эффективна. В рассмотренном алгоритме не предпринимается никаких попыток обслуживания запросов в заданных временных рамках, а просто прекращается процесс сортировки-вставки при наличии определенной задержки. Это в свою очередь приводит к задержке в обслуживании, что было веской причиной для доработки планировщика ввода-пывода ядра 2.4.

Итак, когда запрос добавляется в очередь возможны четыре типа действий. Вот эти действия в необходимой последовательности.

- Если запрос к соседнему сектору находится в очереди, то существующий запрос и новый объединяются в один.
- Если в очереди существует достаточно старый запрос, то новый запрос помещается в конец очереди, чтобы предотвратить отказ обслуживания для других запросов, которые долгое время находятся в очереди.
- Если для секторов данного запроса в очереди существует позиция, которая соответствует рациональному перемещению между секторами, то данный запрос помещается в эту позицию, что позволяет поддерживать очередь в отсортированном состоянии.
- И наконец, если такая позиция не найдена, то запрос помещается в конец очереди.

Планировщик ввода-вывода с лимитом по времени

Планировщик ввода-выпода с лимитом по времени (Deadline I/O scheduler, deadline-планировщик ввода-вывода) разработан с целью предотвращения задержек обслуживания, которые могут возникать для алгоритма Линуса. Если задаться целью только минимизировать количество операций поиска, то при большом количестве операций ввода-вывода из одной области диска могут возникать задержки обслуживания для операций с другими областями диска, причем на неопределенное время. Более того, поток запросов к одной и той же области диска может привести к тому,

что запросы к области диска, которая находится далеко от первой, никогда не будут обработаны. Такой алгоритм не может обеспечить равнодоступность ресурсов.

Хуже того, общая проблема задержки обслуживания запросов приводит к частной проблеме *задержки обслуживания чтения при обслуживании записи (writes-starving-reads)*. Обычно операции записи могут быть отправлены на обработку диском в любой момент, когда ядру это необходимо, причем это выполняется полностью асинхронно по отношению к пользовательской программе, которая сгенерировала запрос записи. Обработка же операций чтения достаточно сильно отличается. Обычно, когда пользовательское приложение отправляет запрос на чтение, это приложение блокируется до тех пор, пока запрос не будет выполнен, т.е. запросы чтения возникают синхронно по отношению к приложению, которое эти запросы генерирует. В связи с этим время реакции системы, в основном, не зависит от латентности записи (времени задержки, которое необходимо на выполнение запроса записи), а задержки чтения (время, которое необходимо на выполнение операции чтения) очень важно минимизировать. Латентность записи мало влияет на производительность пользовательских программ[3], но эти программы должны "с дрожащими руками" ждать завершение каждого запроса чтения. Следовательно, задержки чтения очень важны для производительности системы.

Проблему усугубляет то, что запросы чтения обычно зависят друг от друга. Например, рассмотрим чтение большого количества файлов. Каждая операция чтения выполняется небольшими порциями, которые соответствуют размеру буфера. Приложение не станет считывать следующую порцию данных (или следующий файл), пока предыдущая порция данных не будет считана с диска и возвращена приложению. Следовательно, если существует задержка в обслуживании одного запроса чтения, то для программы эти задержки складываются, и общая задержка может стать недопустимой. Принимая во внимание, что синхронность и взаимозависимость запросов чтения приводят к большим задержкам обработки этих запросов (что в свою очередь сильно влияет на производительность системы), в планировщике ввода-вывода с лимитом по времени были добавлены некоторые функции, которые позволяют гарантированно минимизировать задержки в обработке запросов вообще и в обработке запросов чтения в частности.

Следует обратить внимание, что уменьшение времени задержки в обслуживании может быть выполнено только за счет уменьшения общего быстродействия системы. Даже для алгоритма Линуса такой компромисс существует, хотя и в более мягкой форме. Алгоритм Линуса мог бы обеспечить и большую общую пропускную способность (путем уменьшения количества операций поиска), если бы запросы *всегда* помещались в очередь в соответствии с номерами секторов и не выполнялась проверка на наличие старых запросов и вставка в конец очереди. Хотя минимизация количества операций поиска и важна, тем не менее неопределенное время задержки тоже не очень хорошая вещь. Поэтому deadline-планировщик и выполняет много работы для уменьшения задержек в обслуживании. Достаточно сложно одновременно обеспечить равнодоступность и максимизировать общую пропускную способность.

---

[3] Однако все же не желательно задерживать операции записи на неопределенное время. .Запросы записи также должны немедленно отправляться на диск, но это не так критично, как в случае запросов чтения.

В планировщике ввода-вывода, с лимитом по времени с запросом связано предельное время ожидания (expiration time). По умолчанию этот момент времени равен 500 миллисекунд в будущем для запросов чтения и 5 секунд в будущем для запросов записи. Планировщик ввода-вывода с лимитом по времени работает аналогично планировщику Линуса — он также поддерживает очередь запросов в отсортированном состоянии в соответствии с физическим расположением сектора на диске. Эта очередь называется отсортированной (sorted queue). Когда запрос помещается в отсортированную очередь, то deadlme-планировщик ввода-вывода выполняет объединение и вставку запросов так же, как это делается в лифтовом алгоритме Линуса[4]. Кроме того, планировщик с лимитом по времени помещает каждый запрос и во вторую очередь, в зависимости от типа запроса. Запросы чтения помещаются в специальную очередь FIFO запросов чтения, а запросы записи— в очередь FIFO запросов записи. В то время как обычная очередь отсортирована по номеру сектора на диске, две очереди FIFO (first-in first-out— первым поступил, первым обслужен) сортируются по времени поступления запроса, так как новые запросы всегда добавляются в конец очереди. При нормальной работе deadline-планировщик ввода-вывода получает запросы из головы отсортированной очереди и помещает их в очередь диспетчеризации. Очередь диспетчеризации отправляет запросы жесткому диску. Это приводит к минимизации количества операций поиска.

Если же для запроса, который находится в голове FIFO-очереди записи или FIFO-очереди чтения, истекает период ожидания (т.е. текущий момент времени становится большим, чем момент времени, когда истекает период ожидания, связанный с запросом), то deadline-планировщик начинает обрабатывать запросы из соответствующей очереди FIFO. Таким образом планировщик с лимитом по времени пытается гарантировать, что запросы не будут ожидать дольше максимального периода ожидания (рис. 13.3).

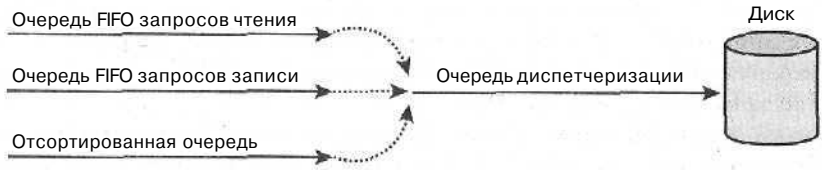


Рис. 13.3. *Три очереди планировщика ввода-вывода с лимитом по времени*

Следует заметить, что deadline-плаиировщик ввода-вывода не дает строгой гарантии времени задержки запроса. Однако он, в общем, позволяет отправлять на обработку запросы до или вскоре после того, как истек их период ожидания. Это позволяет предотвратить ситуацию недостатка обслуживания запросов. Так как для запросов чтения максимальное время ожидания значительно больше, чем для запросов записи, то планировщик с лимитом по времени также позполяет гарантировать, что обслуживание запросов записи не приведет к недостатку обслуживания запросов чтения. Больший приоритет запросов чтения позволяет минимизировать время задержки при операциях чтения.

[4] Для deadline-планировщика операция вставки в начало запроса выполняется опционально. Обычно невыполнение вставки в начало запроса не приводит к проблемам, так как в большинстве случаев количество запросов, которые могут быть добавлены в начало, очень незначительно.

Код планировщика ввода-вывода с лимитом по времени находится в файле drivers/block/deadline-iosched.c.

Прогнозирующий планировщик ввода-вывода

Хотя планировщик с лимитом по времени ввода-вывода и выполняет работу по минимизации задержек чтения, это делается ценой уменьшения глобального быстродействия. Рассмотрим систему с большой активностью записи. Каждый раз, когда приходит запрос на чтение, планировщик сразу же начинает его выполнять. Это приводит к тому, что сразу же запускается операция поиска того места на диске, где будет выполнено чтение и сразу после выполнения чтения снова осуществляется поиск того места, где будет выполнена запись, и так повторяется при каждом запросе чтения. Большой приоритет запросов чтения вещь хорошая, но две операции поиска на каждый запрос чтения (перемещение в точку чтения и обратно в точку записи) очень плохо сказываются на общей дисковой производительности. Цель прогнозирующего планировщика ввода-вывода (anticipatory I/O scheduler) — обеспечение хороших характеристик по задержкам чтения и в то же время обеспечение отличной общей производительности.

Прогнозирующий планировщик построен на базе планировщика ввода-вывода с лимитом но времени. Поэтому он не особо отличается. В прогнозирующем планировщике реализованы три очереди (плюс очередь диспетчеризации) и обработка времени ожидания для каждого запроса, так же как и В случае deadline-планировщика. Главное отличие состоит в наличии дополнительного *эвристического прогнозирования (anticipation heuristic)*.

Прогнозирующий планировщик ввода-вывода пытается минимизировать "шторм операций поиска", который следует за каждым запросом чтения во время выполнения других дисковых операций. Когда поступает запрос на чтение, он обрабатывается в рамках своего времени ожидания как обычно. После того как запрос отправлен жесткому диску, прогнозирующий планировщик сразу не возвращается к выполнению предыдущих запросов и не запускает операцию поиска сразу же. Он абсолютно ничего не делает в течение нескольких миллисекунд (значение этого периода времени можно настраивать, а по умолчанию оно равно 6 миллисекунд). Существует большой шанс, что за эти несколько миллисекунд приложение отправит еще один запрос на чтение. Все запросы к соседним секторам диска выполняются немедленно. После .того как время ожидания истекает, прогнозирующий планировщик возвращается к выполнению ранее оставленных запросов и выполняет поиск соответствующего места на диске.

Важно обратить внимание, что те несколько миллисекунд, в течение которых планировщик ожидает на новые запросы (т.е. время, которое планировщик тратит в предвещании нового запроса), полностью окупаются, даже если это позволяет минимизировать всего лишь небольшой процент операций поиска при выполнении запросов чтения в случае большого количества других запросов. Если во время ожидания приходит запрос к соседней области диска, то это позволяет избежать двух операций поиска. Чем больше за это время приходит запросов к соседним областям диска, тем большего количества операций поиска можно избежать.

Конечно, если в течение периода ожидания не было никакой активности, то прогнозирующий планировщик зря потратит эти несколько миллисекунд.

Ключевой момент для получения максимальной производительности от прогнозирующего планировщика — правильно предсказать действия приложений и файловых систем. Это выполняется на основе эвристических алгоритмов и сбора статики. Прогнозирующий планировщик ведет статистику операций блочного ввода-вывода по каждому процессу в надежде предсказать действия приложений. При достаточно высоком проценте точных предсказаний прогнозирующий планировщик способен значительно снизить затраты на поиск при выполнении операций чтения и в то же время уделить внимание тем запросам, которые критичны для производительности системы. Это позволяет прогнозирующему планировщику минимизировать задержки чтения и в то же время уменьшить количество и продолжительность операций поиска, что в свою очередь проявляется в уменьшении времени реакции системы и в увеличении ее производительности.

Код прогнозирующего планировщика находится в файле drivers/block/as-iosched.c дерева исходных кодов ядра.

Этот планировщик используется в ядре Linux по умолчанию и хорошо работает для большинства типов нагрузки на систему. Он идеальный для серверов, однако работает очень плохо в случае определенных типов загрузки, которые встречаются не очень часто, как, например, в случае баз данных, рассчитанных на большое количество операций поиска но диску.

Планировщик ввода-вывода с полностью равноправными очередями

Планировщик ввода-вывода с полностью равноправными очередями (Complete Fair Queuing, CFQ) был разработан для определенного типа нагрузок на систему, но на практике он позволяет получить хорошую производительность для широкого диапазона типов нагрузки. Он фундаментальным образом отличается от всех ранее рассмотренных планировщиков ввода-вывода.

Планировщик CFQ распределяет все приходящие запросы ввода-вывода по определенным очередям на основании того, какой процесс прислал этот запрос. Например, запросы от процесса foo идут в очередь foo, запросы от процесса bar — в очередь bar. В пределах каждой очереди запросы объединяются со смежными и сортируются. Таким образом очереди поддерживаются в отсортированном состоянии, так же как и в случае других планировщиков ввода-вывода. Отличие планировщика CFQ состоит в том, что он поддерживает отдельную очередь для каждого процесса, который выполняет операции ввода-вывода.

После этого планировщик CFQ выполняет запросы из разных очередей по круговому алгоритму, выполняя конфигурируемое количество запросов (по умолчанию 4) из каждой очереди, перед тем как перейти к следующей. Это позволяет получить равномерное распределение пропускной способности диска для каждого процесса в системе. Предполагаемое использование такого планировщика— мультимедийные приложения, для которых он позволяет гарантировать, что, например, аудиопроигрыватель всегда будет успевать вовремя заполнять аудиобуферы с диска. Тем не менее планировщик CFQ на практике хорошо работает для многих сценариев загруженности системы.

Код CFQ планировщика находится в файле drivers/block/cfq-iosched.c. Этот планировщик рекомендуется для офисных компьютеров, хотя хорошо работает практически для всех типов нагрузок, за исключением, может быть, уж очень экстремальных типов загруженности.

Планировщик ввода-вывода noop

Четвертый, и последний, тип планировщика ввода-вывода— это планировщик noop (no operation, с отсутствием операций). Он назван так потому, что практически ничего не делает. Этот планировщик не выполняет никакой сортировки или других операций для предотвращения поиска по устройству. Ему нет необходимости выполнять ничего, включая алгоритмы, которые минимизируют задержки и были рассмотрены для предыдущих планировщиков.

Планировщик ввода-вывода noop выполняет только объединение приходящих запросов со смежными, которые находятся в очереди. Кроме этого, больше никаких функций у данного планировщика нет. Он просто обслуживает очередь запросов, которые передаются драйверу блочного устройства, в режиме FIFO.

Планировщик noop не является полностью бесполезным. В том, что он ничего не делает, есть большой смысл. Он рассчитан на блочные устройства, которые позволяют выполнять истинно произвольный доступ, такие как платы флеш-памяти. Если для блочного устройства нет накладных затрат, связанных с поиском по устройству, то нет и необходимости выполнять сортировку и вставку вновь приходящих запросов, и планировщик noop — идеальный вариант.

Код планировщика noop находится в файле drivers/block/noop-iosched.c. Он предназначен только для устройств с произвольным доступом.

Выбор планировщика ввода-вывода

В ядрах серии 2.6 есть четыре планировщика ввода-вывода. Каждый из этих планировщиков может быть активизирован. По умолчанию все блочные устройства используют прогнозирующий планировщик ввода-вывода. Планировщик можно изменить, указав параметр ядра elevator=<планировщик> в командной строке при загрузке системы, где <планировщик> — это один из поддерживаемых типов планировщика, которые показаны в табл. 13.2.

Таблица 13.2. Возможные значения параметра elevator

Значение	Тип планировщика
as	Прогнозирующий
cfq	С полностью равноправными очередями
deadline	С лимитом по времени
noop	С отсутствием операций (noop)

Например, указание параметра elevator=cfq в командной строке ядра при загрузке системы означает, что для всех блочных устройств будет использоваться планировщик с полностью равноправными очередями.

Резюме

В этой главе были рассмотрены основы работы устройств блочного ввода-вывода, а также структуры данных, используемые для работы уровня ввода-вывода блоками: структура bio, которая представляет выполняемую операцию ввода-вывода; структура buffer_head, которая представляет отображение блоков на страницы памяти; структура request, которая представляет собой отдельный запрос ввода-вывода. После рассмотрения запросов ввода-вывода был описан их короткий, но важный путь, кульминацией которого является прохождение через планировщик ввода-вывода. Были рассмотрены дилеммы, возникающие при планировании операций ввода-вывода, и четыре типа планировщика, которые на данный момент существуют в ядре Linux, а также планировщик ввода вывода из ядра 2.4 — лифтовой алгоритм Линуса.

Далее мы рассмотрим адресное пространство процесса.

14

Адресное пространство процесса

В главе 11, "Управление памятью", было рассказано о том, как ядро управляет физической намятью. В дополнение к тому, что ядро должно управлять своей памятью, оно также должно,управлять и адресным пространством процессов— тем, как память видится для каждого процесса в системе. Операционная система Linux — это операционная система с виртуальной памятью (virtual memory operating system), т.е. для всех процессов выполняется виртуализация ресурсов памяти. Для каждого процесса создается иллюзия того, что он один использует всю физическую память в системе. Еще более важно, что адресное пространство процессов может быть даже значительно больше объема физической памяти. В этой главе рассказывается о том, как ядро управляет адресным пространством процесса.

Адресное пространство процесса состоит из диапазона адресов, которые выделены процессу, и, что более важно, в этом диапазоне выделяются адреса, которые процесс может так или иначе использовать. Каждому процессу выделяется "плоское" 32- или 64-битовое адресное пространство. Термин "плоское" обозначает, что адресное пространство состоит из одного диапазона адресов (например, 32-разрядное адресное пространство занимает диапазон адресов от 0 до 429496729). Некоторые операционные системы предоставляют сегментированное адресное пространство — адресное пространство состоит больше чем из одного диапазона адресов, т.е. состоит из сегментов. Современные операционные системы обычно предоставляют плоское адресное пространство. Размер адресного пространства зависит от аппаратной платформы. Обычно для каждого процесса существует свое адресное пространство. Адрес памяти в адресном пространстве одного процесса не имеет никакого отношения к такому же адресу памяти в адресном пространстве другого процесса. Тем не менее несколько процессов могут совместно использовать одно общее адресное пространство. Такие процессы называются потоками.

Значение адреса памяти — это заданное значение из диапазона адресов адресного пространства, как, например, 41021f000. Это значение идентифицирует определенный байт в 32-битовом адресном пространстве. Важной частью адресного пространства являются интервалы адресов памяти, к которым процесс имеет право доступа, как, например, 08048000-0804c000. Такие интервалы разрешенных адресов называются *областями памяти (memory area)*. С помощью ядра процесс может динамически добавлять и удалять области памяти своего адресного пространства.

Процесс имеет право доступа только к действительным областям памяти. Более того, на область памяти могут быть установлены права только для чтения или запрет на выполнение. Если процесс обращается к адресу памяти, который не находится в действительной области памяти, или доступ к действительной области выполняется запрещенным образом, то ядро уничтожает процесс с ужасным сообщением "Segmentation Fault" (ошибка сегментации).

Области памяти могут содержать следующую нужную информацию.

- Отображение выполняемого кода из выполняемого файла в область памяти процесса, которая называется *сегментом кода (text section)*.
- Отображение инициализированных переменных из выполняемого файла в область памяти процесса, которая называется *сегментом данных (data section)*.
- Отображение страницы памяти, заполненной нулями, в область памяти процесса, которая содержит неинициализированные глобальные переменные и называется *сегментом bss*[1] *(bss section)*. Нулевая страница памяти (zero page, страница памяти, заполненная нулями) — это страница памяти, которая полностью заполнена нулевыми значениями и используется, например, для указанной выше цели.
- Отображение страницы памяти, заполненной нулями, в память процесса, которая используется в качестве стека процесса пространства пользователя (не нужно путать со стеком процесса в пространстве ядра, который является отдельной структурой данных и управляется и используется ядром).
- Дополнительные сегменты кода, данных и BSS каждой совместно используемой библиотеки, таких как библиотека libc и динамический компоновщик, которые загружаются в адресное пространство процесса.
- Все файлы, содержимое которых отображено в память.
- Все области совместно используемой памяти.
- Все анонимные отображения в память, как, например, связанные с функцией `malloc()`[2].

Каждое действительное значение адреса памяти в адресном пространстве процесса принадлежит только и только одной области памяти (области памяти не перекрываются). Как будет показано, для каждого отдельного участка памяти в выполняющемся процессе существует своя область: стек, объектный код, глобальные переменные, отображенный в память файл и т.д.

---

[1] Термин "BSS" сложился исторически и является достаточно старым. Он означает *block started by symbol (блок, начинающийся с символа)*. Неинициализированные переменные в выполняемом файле не хранятся, поскольку с ними не связано никакого значения. Тем не менее стандарт языка С требует, чтобы неинициализированным переменным присваивалось определенное значение по умолчанию (обычно все заполняется нулями). Поэтому ядро загружает переменные (без их значений) из выполняемого файла в память и отображает в эту память нулевую страницу, тем самым переменным присваивается нулевое значение без необходимости зря тратить место в объектном файле на ненужную инициализацию.

[2] В более новых версиях библиотеки giibc фушщия *malloc()* реализована через системный вызов *mmap(),* а не через вызов *brk()*.

Дескриптор памяти

Ядро представляет адресное пространство процесса в виде структуры данных, которая называется *дескриптором памяти*. Эта структура содержит всю информацию, которая относится к адресному пространству процесса. Дескриптор памяти представляется с помощью структуры `struct mm_struct`, которая определена в файле `<linux/sched.h>`[3].

Рассмотрим эту структуру с комментариями, поясняющими назначение каждого поля.

```
struct mm_struct {
 struct vm_area_struct  *mmap;      /* список областей памяти */
 struct rb_root       mm_rb;       /* красно-черное дерево областей памяти*/
 struct vm_area_struct *mmap_cache; /*последняя использованная область памяти*/
 unsigned long         free_area_cache;  /* первый незанятый участок
                                            адресного пространства */
 pgd_t                  *pgd;       /* глобальный каталог страниц */
 atomic_t         mm_users;         /* счетчик пользователей адресного
                                       пространства */
 atomic_t              mm_count;    /* основной счетчик использования */
 int                  map_count;    /* количество областей памяти */
 struct rw_semaphore   mmap_sem;    /* семафор для областей памяти */
 spinlock_t           page_table_lock; /* спин-блокировка таблиц страниц*/
 struct list_head      mmlist;      /* список всех структур mm_struct */
 unsigned long        start_code; /* начальный адрес сегмента кода */
 unsigned long        end code;   /* конечный адрес сегмента кода */
 unsigned long        start_data; /* начальный адрес сегмента данных */
 unsigned long     end_data;      /* конечный адрес сегмента данных */
 unsigned long        start_brk;  /* начальный адрес сегмента "кучи" */
 unsigned long         brk;       /* конечный адрес сегмента "кучи" */
 unsigned long        start_stack;   /* начало стека процесса */
 unsigned long        arg_start;  /* начальный адрес области аргументов */
 unsigned long        arg_end;    /* конечный адрес области аргументов */
 unsigned long  env_start;  /*начальный адрес области переменных среды */
 unsigned long        env_end;    /*конечный адрес области переменных среды*/
 unsigned long         rss;       /* количество физических страниц памяти*/
 unsigned long     total_vm;      /* общее количество страниц памяти */
 unsigned long        locked_vm;  /* количество заблокированных страниц
                                       памяти */
 unsigned long        def_flags;  /* флаги доступа, используемые
                                       по умолчанию */
 unsigned long       cpu_vm_mask; /*МасКа отложенного переключения буфера TLB*/
 unsigned long        swap_address;  /* последний сканированный адрес */
 unsigned            dumpable:1;    /* можно ли создавать файл core? */
 int                  used_hugetlb;  /* используются ли гигантские
                                        страницы памяти (hugetlb)? */
```

[3] Между дескриптором процесса, дескриптором памяти и соответствующими функциями существует тесная связь. Поэтому структура `struct mm_struct` и определена в заголовочном файле *sched.h*.

```
    mm_context_t        context;    /* данные, специфичные для аппаратной
                                       платформы */
    int                 core_waiters; /* количество потоков, ожидающих на
                                         создание файла core */
    struct completion   *core_startup_donc; /* условная переменная начала
                                               создания файла core */
    struct completion   core_done; /* условная переменная завершения
                                      создания файла core */
    rwlock_t            ioctx_l.ist_lock; /* блокировка списка асинхронного
                                             ввода-вывода (AIO) */
    struct kioctx       *ioctx_list; /* список асинхронного ввода-вывода (AIO) V
    struct kioctx       default kioctx; /* контекст асинхронного ввода-
                                           вывода, используемый по умолчанию */
};
```

Поле mm_users — это количество процессов, которые используют данное адресное пространство. Например, если одно и то же адресное пространство совместно используется двумя потоками, то значение поля mm_users равно двум. Поле ram_count — это основной счетчик использования структуры mm_struct. Наличие пользователей структуры, которым соответствует поле mm_users, приводит к увеличению счетчика mm_count на единицу. В предыдущем примере значение поля mm_count равно единице. Когда значение поля mm_users становится равным нулю (т.е. когда два потока завершатся), только тогда значение поля mm_count уменьшается на единицу. Когда значение поля mm_count становится равным нулю, то на соответствующую структуру mm_struct больше нет ссылок, и она освобождается, Поддержка двух счетчиков позволяет ядру отличать главный счетчик использования (mm_count) от количества процессов, которые используют данную структуру (mm_users).

Поля mmap и mm_rb— это два различных контейнера данных, которые содержат одну и ту же информацию: информацию обо всех областях памяти в соответствующем адресном пространстве. В первом контейнере эта информация хранится в виде связанного списка, а во втором— в виде красно-черного бинарного дерева. Поскольку красно-черное дерево — это разновидность бинарного дерева, то, как и для всех типов бинарного дерева, количество операций поиска заданного элемента в нем равно O(log (n)). Более детальное рассмотрение красно-черных деревьев найдете в разделе "Списки и деревья областей памяти".

Хотя обычно в ядре избегают избыточности, связанной с введением нескольких структур для хранения одних и тех же данных, тем не менее в данном случае эта избыточность очень кстати. Контейнер mmap — это связанный список, который позволяет очень быстро проходить по всем элементам. С другой стороны, контейнер mm_rb — это красно-черное дерево, которое очень хорошо подходит для поиска заданного элемента. Области памяти будут рассмотрены в этой главе несколько ниже,

Все структуры mm_struct объединены в двухсвязный список с помощью нолей mmlist. Первым элементом этого списка является дескриптор памяти init_mm, который является дескриптором памяти процесса ink. Этот список защищен от конкурентного доступа с помощью блокировки mmlist_lock, которая определена в файле kernel/fork.c. Общее количество дескрипторов памяти хранится в глобальной целочисленной переменной mmlist_nr, которая определена в том же файле.

Выделение дескриптора памяти

Указатель на дескриптор памяти, выделенный для какой-либо задачи, хранится в поле mm дескриптора процесса этой задачи. Следовательно, выражение current->rnm позволяет получить дескриптор памяти текущего процесса. Функция copy_mm() используется для копирования дескриптора родительского процесса в дескриптор порожденного процесса во время выполнения вызова fork (). Структура mm_struct выделяется из слябового кэша mm_cachep с помощью макроса allocate_mm (). Это реализовано в файле kernel/fork.c. Обычно каждый процесс получает уникальный экземпляр структуры mm_struct и соответственно уникальное адресное пространство.

Процесс может использовать одно и то же адресное пространство совместно со своими порожденными процессами, путем указания флага CLONE_VM при выполнении вызова clone (). Такие процессы называются потоками. Вспомните из материала главы 3, "Управление процессами", что в операционной системе Linux в этом и состоит *единственное* существенное отличие между обычными процессами и потоками. Ядро Linux больше никаким другим образом их не различает. Потоки с точки зрения ядра— это обычные процессы, которые просто совместно используют некоторые общие ресурсы.

В случае, когда указан флаг CLONE_VM, макрос allocate_mm() не вызывается, а в поле mm дескриптора порожденного процесса записывается значение указателя на дескриптор памяти родительского процесса. Это реализовано с. помощью следующего оператора ветвления в функции copy_mm().

```
if (clone_flags & CLONE_VM) {
        /*
        * current — это родительский процесс
        * tsk — это процесс, порожденный в вызове fork()
        */
        atomic_inc(&current->mm->mm_users);
        tsk->mm = current->mm;
}
```

Удаление дескриптора памяти

Когда процесс, связанный с определенным адресным пространством, завершается, то вызывается функция exit_mm() . Эта функция выполняет некоторые служебные действия и обновляет некоторую статистическую информацию. Далее вызывается функция input() , которая уменьшает на единицу значение счетчика количества пользователей mm_users для дескриптора памяти. Когда значение счетчика количества пользователей становится равным нулю, то вызывается функция mmdrop(), которая уменьшает значение основного счетчика использования mm_count. Когда и этот счетчик использования наконец достигает нулевого значения, то вызывается функция free_mm(), которая возвращает экземпляр структуры mm_struct в слябовый кэш mm_cachep с помощью вызова функции kmem_cache_free(), поскольку дескриптор памяти больше не используется.

Структура mm_struct и потоки пространства ядра

Потоки пространства ядра не имеют своего адресного пространства процесса и, следовательно, связанного с ним дескриптора памяти. Значение поля mm для потока пространства ядра равно NULL. Еще одно *определение* потока ядра — это процесс, который не имеет пользовательского контекста.

Отсутствие адресного пространства— хорошее свойство, поскольку потоки ядра вообще не обращаются к памяти в пространстве пользователя (действительно, к какому адресному пространству им обращаться?). Поскольку потоки ядра не обращаются к страницам памяти в пространстве пользователя, им вообще не нужен дескриптор памяти и таблицы страниц (таблицы страниц обсуждаются дальше в этой главе). Несмотря на это, потокам пространства ядра все же нужны некоторые структуры данных, такие как таблицы страниц, чтобы обращаться к памяти ядра. Чтобы обеспечить потоки ядра всеми данными без необходимости тратить память на дескриптор памяти и таблицы страниц, а также процессорное время на переключение на новое адресное пространство и так далее, каждый поток ядра использует дескриптор памяти задания, которое выполнялось перед ним.

Когда процесс запланирован на выполнение, то загружается адресное пространство, на которое указывает поле mm этого процесса. Поле active_mm дескриптора процесса обновляется таким образом, чтобы указывать на новое адресное пространство. Потоки ядра не имеют своего адресного пространства, поэтому значение поля mm для них равно NULL. Поэтому, когда поток ядра планируется на выполнение, ядро определяет, что значение ноля mm равно NULL, и оставляет загруженным предыдущее адресное пространство. После этого ядро обновляет поле active_mm дескриптора процесса для потока ядра, чтобы он указывал на дескриптор памяти предыдущего процесса. При необходимости поток ядра может использовать таблицы страниц предыдущего процесса. Так как потоки ядра не обращаются к памяти в пространстве пользователя, то они используют только ту информацию об адресном пространстве ядра, которая связана с памятью ядра и является общей для всех процессов.

Области памяти

Области памяти (memory areas) представляются с помощью объектов областей памяти, которые хранятся в структурах типа vm_area_struct. Эта структура определена в файле <linux/mm.h>. Области памяти часто называются *областями виртуальной памяти (virtual memory area,* или *VMA).*

Структура vm_area_struct описывает одну непрерывную область памяти в данном адресном пространстве. Ядро рассматривает каждую область памяти, как уникальный объект. Для каждой области памяти определены некоторые общие свойства, такие как права доступа и набор соответствующих операций. Таким образом, одна структура VMA может представлять различные типы областей памяти, например файлы, отображаемые в память, или стек пространства пользователя. Это аналогично объектно-ориентированному подходу, который используется в подсистеме VFS (см .главу 12, "Виртуальная файловая система").

Ниже показана эта структура данных с комментариями, описывающими назначение каждого поля.

```
struct vm_area_struct {
 struct mm_struct      *vm_mm;  /* соответствующая структура mm_struct */
 unsigned long         vm_start;     /* начало диапазона адресов */
 unsigned     long     vm_end;       /* конец диапазона адресов */
 struct vm_area_struct *vm_next;     /* список областей VMA */
 pgprot_t              vm_page_prot; /* права доступа */
 unsigned long         vm_flags;     /* флаги */
 struct rb_node        vm_rb;        /* узел текущей области VMA */
 union { /* связь с address_space->i_mmap, или i_mmap_nonlinear */
        struct {
           struct list_head       list;
           void                   *parent;
           struct vm_area_struct  *head;
        } vm_set;
        struct prio_tree_node prio_tree_node;
 } shared;
 struct list_head     anon_vma_node; /* анонимные области */
 struct anon_vma       *anon_vma;    /* объект анонимной VMA */
 struct vm_operations_struct   *vm_ops; /* операции */
 unsigned long         vm_pgoff;     /* смещение в файле */
 struct file           *vm_file;     /* отображенный файл (если есть) */
 void                 *vm_private_data; /* приватные данные */
};
```

Как уже было рассказано, каждый дескриптор памяти связан с уникальным диапазоном (интервалом) адресов в адресном пространстве процесса. Поле vm_start — это начальный (минимальный) адрес, а поле vm_end— конечный (максимальный) адрес данного интервала. Следовательно, значение (vm_end - vm_start) — это размер (длина) интервала адресов в байтах. Интервалы адресов разных областей памяти одного адресного пространства не могут перекрываться.

Поле vm_mm указывает па структуру mm_struct, связанную с данной областью VMA. Заметим, что каждая область VMA уникальна для той структуры mm_struct, с которой эта область связана. Поэтому, даже если два разных процесса отображают один и тот же файл на свои адресные пространства, то для каждого процесса создается своя структура vm_area_struct, чтобы идентифицировать уникальные области памяти каждого процесса. Следовательно, два потока, которые совместно используют адресное пространство, также совместно используют и все структуры vm_area_struct в этом адресном пространстве.

Флаги областей VMA

Поле флагов vm_flags содержит битовые флаги, которые определены в файле <linux/mm.h>. Они указывают особенности поведения и содержат описательную информацию о страницах памяти, которые входят в данную область памяти. В отличие от прав доступа, которые связаны с определенной физической страницей памяти, флаги областей VMA указывают особенности поведения, за которые отвечает ядро, а не аппаратное обеспечение. Более того, поле vm_flags содержит информацию, которая относится к каждой странице в области памяти или, .что то же самое, ко всей области памяти в целом. В табл. 14.1 приведен список возможных значений флагов vm_flags.

Таблица 14.1. Флаги областей VMA

Флаг	Влияние на область VMA и на ее страницы памяти
VM_READ	Из страниц памяти можно считывать информацию
VM_WRITE	В страницы памяти можно записывать информацию
VM_EXEC	Можно выполнять код, хранящийся в страницах памяти
VM_SHARED	Страницы памяти являются совместно используемыми
VM_MAYREAD	Можно устанавливать флаг VM_READ
VM_MAYWRITE	Можно устанавливать флаг VM_WRITE
VM_MAYEXEC	Можно устанавливать флаг VM_EXEC
VM_MAYSHARE	Можно устанавливать флаг VM_SHARED
VM_GROWSDOWN	Область памяти может расширяться "вниз"
VM_GROWSUP	Область памяти может расширяться "вверх"
VM_SHM	Область используется для разделяемой (совместно используемой) памяти
VM_DENYWRITE	В область отображается файл, в который нельзя выполнять запись
VM_EXECUTABLE	В область отображается выполняемый файл
VM_LOCKED	Страницы памяти в области являются заблокированными
VM_IO	В область памяти отображается пространство ввода-вывода аппаратного устройства
VM_SEQ_READ	К страницам памяти, вероятнее всего, осуществляется последовательный доступ
VM_RAND_READ	К страницам памяти, вероятнее всего, осуществляется случайный доступ
VM_DONTCOPY	Область памяти не должна копироваться при вызове fork ()
VM_DONTEXPAND	Область памяти не может быть увеличена с помощью вызова remap ()
VM_RESERVED	Область памяти не должна откачиваться на диск
VM_ACCOUNT	Область памяти является объектом, по которому выполняется учет ресурсов
VM_HUGETLB	В области памяти используются гигантские (hugetlb) страницы памяти
VM_NONLINEAR	Область памяти содержит нелинейное отображение

Рассмотрим подробнее назначение наиболее интересных и важных флагов. Флаги VM_READ, VM_WRITE и VM_EXEC указыпают обычные права на чтение-запись и выполнение для страниц памяти, которые принадлежат данной области памяти. При необходимости их можно комбинировать для формирования соответствующих прав доступа. Например, отображение выполняемого кода процесса может быть выполнено с указанием флагов VM_READ и VM_EXEC, но никак не с указанием флага VM_WRITE. С другой стороны, сегмент данных из выполняемого файла может отображаться с указанием флагов VM_READ и VM_WRITE, указывать при этом флаг VM_EXEC не имеет смысла. Файл данных, который отображается только для чтения, должен отображаться с указанием только флага VM_READ.

Флаг VM_SHARED указывает на то, что область памяти содержит отображение, которое может совместно использоваться несколькими процессами. Если этот флаг установлен, то такое отображение называют совместно используемым *(shared mapping),* что интуитивно понятно. Если этот флаг не установлен, то такое отображение доступно только одному процессу и оно называется *частным отображением, (private mapping).*

Флаг VM_IO указывает, что область памяти содержит отображение области ввода-вывода аппаратного устройства. Этот флаг обычно устанавливается драйверами устройств при выполнении вызова mmap () для отображения в память области ввода-вывода аппаратного устройства. Кроме всего прочего, этот флаг указывает, что область памяти не должна включаться в файл core процесса. Флаг VM_RESERVED указывает, что область памяти не должна откачиваться на диск. Этот флаг также укалывается при отображении на память областей ввода-вывода аппаратных устройств.

Флаг VM_SEQ_READ является подсказкой ядру, что приложение выполняет последовательное (т.е. линейное и непрерывное) чтение из соответствующего отображения. При этом ядро может повысить производительность чтения за счет выполнения упреждающего чтения (read-ahead) из отображаемого файла. Флаг VM_RAND_READ указывает обратное, т.е. приложение выполняет операции чтения из случайно выбранных мест отображения (т.е. не последовательно). При этом ядро может уменьшить или совсем отключить выполнение упреждающего чтения из отображаемого файла. Эти флаги устанавливаются с помощью системного вызова madvice () путем указания соответственно флагов MADV_SEQUENTIAL и MADV_RANDOM для этого вызова. Упреждающее чтение — это последовательное чтение несколько большего количества данных, чем было запрошено, в надежде на то, что дополнительно считанные данные могут скоро понадобиться. Такой режим полезен для приложений, которые считывают данные последовательно. Однако если считывание данных выполняется случайным образом, то режим упреждающего чтения не эффективен.

Операции с областями VMA

Поле vm_ops структуры vm_area_struct содержит указатель на таблицу операций, которые связаны с данной областью памяти и которые ядро может вызывать для манипуляций с областью VMA. Структура vm_area_struct служит общим объектом для представления всех типов областей виртуальной памяти, а в таблице операций описаны конкретные методы, которые могут быть применены к каждому конкретному экземпляру объекта.

Таблица операций представлена с помощью структуры vm_operations_struct, которая определена в файле <linux/mm.h> следующим образом.

```
struct vm_operations_struct {
   void (*open) (struct vm_area_struct *) ;
   void (*close) (struct vm_area_struct * ) ;
   struct page * (*nopage) (struct vm_area_struct *, unsigned long, int);
   int (*populate) (struct vm_area struct *, unsigned long,
        unsigned long, pgprot_t, unsigned long, int);
        };
```

Рассмотрим каждый метод в отдельности.

- `void open (struct vm_area_struct *area)`
  Эта функция вызывается, когда соответствующая область памяти добавляется в адресное пространство.
- `void close(struct vm_area_struct *area)`
  Эта функция вызывается, когда соответствующая область памяти удаляется из адресного пространства.

- `struct page * nopage(struct vm_area_sruct *area, unsigned long address, int unused)`

  Эта функция вызывается обработчиком прерывания из-за отсутствия страницы (page fault), когда производится доступ к странице, которая отсутствует в физической памяти.

- `int populate{struct vm_area_struct *area, unsigned long address, unsigned long len, pgprot_t prot, unsigned long pgoff, int nonblock)`

  Эта функция вызывается из системного вызова. remap_pages() для предварительного заполнения таблиц страниц области памяти (prefault) при создании нового отображения.

Списки и деревья областей памяти

Как уже рассказывалось, к областям памяти осуществляется доступ с помощью двух структур данных дескриптора памяти: полей mmap и mm_rb. Эти две структуры данных независимо друг от друга указывают на все области памяти, связанные с данным дескриптором памяти. Они содержат указатели на одни и те же структуры vm_area_struct, просто эти указатели связаны друг с другом по-разному.

Первый контейнер, поле mmap, объединяет все объекты областей памяти в односвязный список. Структуры vm_area_struct объединяются в список с помощью своих полей vm_next. Области памяти отсортированы в порядке увеличения адресов (от наименьшего и до наибольшего). Первой области памяти соответствует структура vm area_struct, на которую указывает само поле mmap. Указатель на самую последнюю структуру равен значению NULL.

Второе поле, mm_rb, объединяет все объекты областей памяти в красно-черное (red-black) дерево. На корень дерева указывает поле mm_rb, а каждая структура vm_area struct присоединяется к дереву с помощью поля vm_rb.

Красно-черное дерево — это один из типов бинарного дерева. Каждый элемент красно-черного дерева называется узлом. Начальный узел является корнем дерева. Большинство узлов имеет два дочерних узла: левый дочерний узел и правый дочерний узел. Некоторые узлы имеют всего один дочерний узел, и, наконец, узлы, которые не имеют дочерних, называются листьями. Для любого узла все элементы дерева, которые находятся слева от данного узла, всегда меньше по своему значению, чем значение данного узла, а все элементы дерева, которые находятся справа от некоторого узла, всегда больше по значению, чем значение этого узла. Более того, каждому узлу присвоен цвет (красный или черный, отсюда и название этого типа деревьев) в соответствии со следующими двумя правилами: дочерние элементы красного узла являются черными и любой путь по дереву от узла к листьям должен содержать одинаковое количество черных узлов. Корень дерева всегда красный. Поиск, вставка и удаление элементов из такого дерева требуют количество операций порядка O (log (n)).

Связанный список используется, когда необходимо пройти по всем узлам. Красно-черное дерево используется, когда необходимо найти определенную область памяти адресного пространства. Таким образом, ядро использует избыточные структуры данных для обеспечения оптимальной производительности независимо от того, какие операции выполняются с областями памяти.

Области памяти в реальной жизни

Рассмотрим пример адресного пространства процесса и области памяти в этом адресном пространстве. Для этой цели можно воспользоваться полезной файловой системой /proc и утилитой pmap (1). В качестве примера рассмотрим следующую простую прикладную программу, которая работает в пространстве пользователя. Эта программа не делает абсолютно ничего, кроме того, что служит примером.

```
int main(int argc, char *argv[])

        return 0;
}
```

Рассмотрим список областей памяти из адресного пространства этого процесса. Этих областей немного. Мы уже знаем, что среди них есть сегмент кода, сегмент данных сегмент bss. Если учесть, что эта программа динамически скомпонована с библиотекой функций языка C, то соответствующие области существуют также для модуля libc.so и для модуля ld.so. И наконец, среди областей памяти также есть стек процесса.

Результат вывода списка областей адресного пространства этого процесса из файла /proc/<pid>/maps имеет следующий вид.

```
rml@phantasy:~$ cat /proc/1426/maps
00e80000-00faf000  r-xp 00000000 03:01 208530  /lib/tls/libc-2.3.2.so
00faf000-00fb2000  rw-p 0012f000 03:01 208530  /lib/tls/libc-2.3.2.so
00fb2000-00fb4000  rw-p 00000000 00:00 0
08048000-08049000  r-xp 00000000 03:03 439029  /home/rml/src/example
08049000-0804a000  rw-p 00000000 03:03 439029  /home/rml/src/example
40000000-40015000  r-xp 00000000 03:01 80276   /lib/ld-2.3.2.so
40015000-40016000  rw-p 00015000 03:01 80276   /lib/ld-2.3.2.so
4001e000-4001f000  rw-p 00000000 00:00 0
bfffe000-c0000000  rwxp fffff000 00:00 0
```

Информация об областях памяти выдается в следующем формате.

начало-конец права доступа смещение старший:младший номера устройства файловый индекс файл

Утилита pmap (1)[4] форматирует эту информацию в следующем, более удобочитаемом виде.

```
rml@phantasy:~$ pmap 1426
example[1426]
00e80000 (1212 KB)  r-xp (03:01 208530)         /lib/tls/libc-2.3.2.so
00faf000 (12 KB)    rw-p (03:01 208530)         /lib/tls/libc-2.3.2.so
00fb2000 (8 KB)     rw-p (00:00 0)
08048000 (4 KB)     r-xp (03:03 439029)         /home/rml/src/example
08049000 (4 KB)     rw-p (03:03 439029)         /home/rml/src/example
40000000 (84 KB)    r-xp (03:01 80276)          /lib/ld-2.3.2.so
```

[4]Утилита pmap(1) печатает форматированный список областей памяти процесса. Результат ее вывода несколько более удобочитаем, чем информация, получаемая из файловой системы /proc, но это одна и та же информация. Данная утилита включена в новые версии пакета procps.

```
40015000 (4KB)     rw-p (03:01 80276)          /lib/ld-2.3.2.so
4001e000 (4 KB)    rw-p (00:00 0)
bfffeOOO (8 KB)    rwxp (00:00 0)
mapped: 1340 KB   writable/private: 40 KB   shared: 0 KB
```

Первые три строчки соответствуют сегменту кода, сегменту данных и сегменту bss модуля libc.so (библиотека функций языка С). Следующие две строчки описывают соответственно сегмент кода и сегмент данных выполняемого образа. Далее три строчки— описание сегментов кода, данных и bss модуля ld. so (динамический компоновщик). Последняя строчка описывает стек процесса.

Обратите внимание, что все сегменты кода имеют права на чтение и выполнение, что и должно быть для выполняемых образов. С другой стороны, сегменты данных и bss, которые содержат глобальные переменные, помечаются как имеющие права на запись и чтение, а не на выполнение.

Все адресное пространство составляет порядка 1340 Кбайт, но только 40 Кбайт из них имеют право на запись и соответствуют частному отображению. Если область памяти является совместно используемой и не имеет прав на запись, то ядро хранит в памяти всего одну копию отображаемого файла. Это может показаться обычным для совместно используемых отображений; однако, случай, когда при этом еще и отсутствуют права на запись, проявляется несколько неожиданно. Если учесть факт, что когда на отображение нет прав записи, то соответствующая информация никогда не может быть изменена (из отображения возможно только чтение), становится ясно, что можно совершенно безопасно загрузить выполняемый образ в память всего один раз. Поэтому динамически загружаемая библиотека функций языка С и занимает в памяти всего 1212 Кбайт, а не 1212 Кбайт, умноженное на количество процессов, которые эту библиотеку используют. В связи с этим, процесс, код и данные которого имеют объем порядка 1340 Кбайт, на самом деле занимает всего 40 Кбайт физической памяти. Экономия памяти из-за такого совместного использования получается существенной.

Обратите внимание на области памяти, которые не имеют отображаемого файла, находятся на устройстве с номерами 00:00 и номер файлового индекса для которых равен нулю. Это отображение страницы, заполненной нулями (zero page, пулевая страница). Если отобразить страницу, заполненную нулями, на область памяти, которая имеет права на запись, то побочным эффектом является инициализация всех переменных в нулевые значения. Это важно, поскольку в таком случае получается область памяти, заполненная нулями, которая нужна для сегмента bss.

Каждой области памяти, связанной с процессом, соответствует структура vm_area_struct. Так как процесс не является потоком (thread), то для него существует отдельная структура min_struct, на которую есть ссылка из структуры task_struct.

Работа с областями памяти

Ядру часто необходимо определять, соответствует ли та или иная область памяти в адресном пространстве процесса заданному критерию, например, существует ли заданный адрес в области памяти. Эти операции являются основой работы функции mmap () , которая будет рассмотрена в следующем разделе, и выполнять их приходится часто. Несколько полезных для этого функций объявлены в файле <linux/mm.h>.

Функция find_vma()

Функция find_vma () определена в файле mm/mmap.c.

Эта функция позволяет найти в заданном адресном пространстве ту первую область памяти, для которой значение поля vm_end больше заданного адреса addr. Другими словами, эта функция позволяет найти первую область памяти, которая содержит адрес addr или начинается с адреса, большего адреса addr. Если такой области памяти не существует, то функция возвращает значение NULL.

В противном случае возвращается указатель на соответствующую структуру vm_area_struct. Обратите внимание, что найденная область VMA может начинаться с адреса, большего адреса addr, и этот адрес не обязательно *принадлежит,* найденной области памяти. Результат выполнения функции find_vma () кэшируется в поле map_cache дескриптора памяти. Поскольку очень велика вероятность того, что после одной операции с областью памяти последуют еще операции с ней же, то процент попаданий в кэш получается достаточно большим (на практике получаются значения порядка 30-40%). Проверка кэшированных результатов выполняется очень быстро. Если нужный адрес в кэше не найден, то выполняется поиск по всем областям памяти, связанным с заданным дескриптором. Этот поиск выполняется с помощью красно-черного дерева следующим образом.

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long addr)
{
        struct vm_area_struct *vma = NULL;

        if (mm) {
              vma = mm->mmap_cache;
              if (! (vma && vma->vm_end > addr && vma->vm start <= addr)) {
                    struct rb node * rb_node;

                    rb node = mm->mm_rb.rb_node;
                    vma = NULL;
                    while (rb_node) {
                          struct vm_area_struct * vma_tmp;

                          vma_tmp = rb_entry (rb_node,
                                        struct vm_area_struct, vm_rb);
                          if (vma_tmp->vm_end > addr) {
                                vma = vma_tmp;
                                if (vma_tmp->vm_start <= addr)
                                      break;
                                rb_node = rb_node->rb_left;
                          } else
                                rb_node = rb_node->rb_right;
                    }
                    if (vma)
                          mm->mmap_cache = vma;
              }
        }
        return vma;
        }
```

Вначале выполняется проверка поля vma_cache на предмет того, содержит ли кэшированная область VMA необходимый адрес. Обратите внимание, что простая проверка того, является ли значение поля vm_end большим addr, не гарантирует что проверяемая область памяти является первой, в которой есть адреса, большие addr. Поэтому, для того чтобы кэш в этой ситуации оказался полезным, проверяемый адрес должен принадлежать кэшированной области памяти. К счастью, это как раз и соответствует случаю выполнения последовательных операций с одной и той же областью VMA.

Если кэш не содержит нужную область VMA, то функция должна выполнять поиск по красно-черному дереву Это выполняется путем проверки узлов дерева. Если значение поля vma_end для области памяти текущего узла больше addr, то текущим становится левый дочерний узел, в противном случае — правый. Функция завершает свою работу, как только находится область памяти, которая содержит адрес addr. Если такая область VMA не найдена, то функция продолжает поиск по дереву и возвращает ту область памяти, которая начинается после адреса addr. Если вообще не найдена ни одна область памяти, то возвращается значение NULL.

Функция find_vma_prev()

Функция find_vma_prev () работает аналогично функции find vma () , но дополнительно она еще возвращает последнюю область VMA, которая заканчивается перед адресом addr. Эта функция также определена в файле mma/mmap.c и объявлена в файле <linux/ram.h> следующим образом.

```
struct vm_area_struct * find vma_prev (struct mm_struct *mm,
        unsigned long addr, struct vm_area_struct **pprev)
```

Параметр pprev после возвращения из функции содержит указатель на предыдущую область VMA.

Функция find_VMA_intersection()

Функция find_vma_intersection () возвращает первую область памяти, которая перекрывается с указанным интервалом адресов. Эта функция определена в файле <linux/mm.h> следующим образом. Это функция с подстановкой тела.

```
static inline struct vm_area_struct * find_vma_intersection(
struct mm_struct *mm, unsigned long start_addr, unsigned long end addr)
{
        struct vm_area_struct *vma;
        vma = find_vma (mm, start_addr) ;
        if (vma && end_addr <= vma->vm_start)
             vma = NULL;
        return vma;
}
```

Первый параметр — адресное пространство, в котором выполняется поиск, параметр start_addr — это первый адрес интервала адресов, а параметр end_addr — последний адрес интервала.

Очевидно, что если функция find_vma() возвращает значение NULL, то это же значение будет возвращать и функция find_vma_intersection(). Если функция

find_vma () возвращает существующую область VMA, то функция find_vma_intersection () возвратит ту же область только тогда, когда эта область *не* начинается после конца данного диапазона адресов. Если область памяти, которая возвращается функцией find_vma (), начинается после последнего адреса из указанного диапазона, то функция find_vma_intersection () возвращает значение NULL.

Функции mmap() и do_mmap(): создание интервала адресов

Функция do_mmap() используется ядром для создания нового линейного интервала адресов. Говорить, что эта функция создает новую область VMA, — технически не корректно, поскольку если создаваемый интервал адресов является смежным с существующим интервалом адресов и у этих интервалов одинаковые права доступа, то два интервала объединяются в один. Если это невозможно, то создается новая область VMA. В любом случае функция do_mmap() — это функция, которая добавляет интервал адресов к адресному пространству процесса, независимо от того, создается ли при этом новая область VMA или расширяется существующая.

Функция do_ramap() объявлена в файле <linux/mm.h> следующим образом.

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long len, unsigned long prot,
                      unsigned long flag, unsigned long offset)
```

Эта функция выполняет отображение на память содержимого файла file начиная с позиции в файле offset; размер отображаемого участка равен len байт. Значения параметров file и offset могут быть нулевыми, в этом случае отображение не будет резервироваться (сохраняться) в файле. Такое отображение называется *анонимным (anonymous mapping)*. Если указан файл и смещение, то отображение называется *отображением файла в память (file-backed mapping)*.

Параметр addr указывает (точнее, всего лишь подсказывает), откуда начинать поиск свободного интервала адресов.

Параметр prot указывает права доступа для страниц памяти в данной области. Возможные значение флагов зависят от аппаратной платформы и описаны в файле <asm/mman.h>. Хотя на практике для всех аппаратных платформ определены флаги, приведенные в табл. 14.2.

Параметр flags позволяет указать все остальные флаги области VMA. Эти флаги также определены в <asm/mman.h> и приведены в табл. 14.3.

Таблица 14.2. Флаги защиты страниц памяти

Флаг	Влияние на страницы памяти в созданном интервале адресов
PROT_READ	Соответствует флагу VM_READ
PROT_WRITE	Соответствует флагу VM_WRITE
PROT_EXEC	Соответствует флагу VM_EXEC
PROT_NONE	К страницам памяти нет доступа

Таблица 14.3. Флаги защиты страниц памяти

Флаг	Влияние на созданный интервал адресов
MAP_SHARED	Отображение может быть совместно используемым
MAP_PRIVATE	Отображение не может быть совместно используемым
MAP_FIXED	Создаваемый интервал адресов *должен* начинаться с указанного адреса addr
MAP_ANONYMOUS	Отображение является анонимным, а не отображением файла
MAP_GROWSDOWN	Соответствует флагу VM_GROWSDOWN
MAP_DENYWRIIE	Соответствует флагу VM DENYWRITE
MAP_EXECUTABLE	Соответствует флагу VM_EXECUTABLE
MAP_LOCKED	Соответствует флагу VM_LOCKED
MAP_NORESERVE	Нет необходимости резервировать память для отображения
MAP_POPULATE	Предварительно заполнить (prefault) таблицы страниц
MAP_NONBLOCK	Не блокировать при операциях ввода-вывода

Если какой-либо из параметров имеет недопустимое значение, то функция do_mmap() возвращает отрицательное число. В протипном случае создастся необходимый интервал адресов. Если это возможно, то этот интервал объединяется с соседней областью памяти. Если это невозможно, то создается новая структура vm_area_struct, которая выделяется в слябовом кэше vm_area_cachep. После этого новая область памяти добавляется в связанный список и красно-черное дерево областей памяти адресного пространства с помощью функции vma_link() . Затем обновляется значение поля total_vm в дескрипторе памяти. В конце концов, функция возвращает начальный адрес вновь созданного интервала адресов.

Системный вызов mmap()

Возможности функции do_mmap() экспортируются в пространство пользователя с помощью системного вызова mmap() , который определен следующим образом.

```
void * mmap2 (void *start,
              size_t length,
              int prot,
              int flags,
              int fd,
              off_t pgoff)
```

Этот системный вызов имеет имя mmap2() ,т.е. второй вариант функции mmap(). Первоначальный вариант mmap() требовал в качестве последнего параметра смещение в байтах, а текущий вариант, mmap2(), — смещение в единицах размера страницы памяти. Это позволяет отображать файлы большего размера с большим значением смещения. Первоначальный вариант функции mmap() , который соответствует стандарту POSTX, доступен через библиотеку функций языка С, как функция mmap() , но в ядре уже не реализован. Новый вариант библиотечной функции называется mmap2() . Обе эти библиотечные функции используют системный вызов mmap2 (), При этом библиотечная функция mmap() переводит значение смещения из байтов в количество страниц памяти.

Функции munmap() и do_munmap(): удаление интервала адресов

Функция do_manmap() удаляет интервал адресов из указанного адресного пространства процесса. Эта функция объявлена в файле <asm/mman.h> следующим образом.

```
int do_munmap(struct mm_struct *mm, unsigned long start, size t_len)
```

Первый параметр указывает адресное пространство, из которого удаляется интервал адресов, начинающийся с адреса start и имеющий длину len байт. В случае успеха возвращается нуль, а в случае ошибки — отрицательное значение.

Системный вызов munmap()

Системный вызов munmap () экспортируется в адресное пространство пользователя, чтобы иметь возможность удалять интервалы адресов из адресного пространства. Эта функция является комплиментарной к системному вызову mmap () и имеет следующий прототип.

```
int munmap(void *start, size_t length)
```

Данный системный вызов реализован в виде очень простой интерфейсной оболочки (wrapper) функции do_munmap ().

```
asmlinkage long sys_munmap(unsigned long addr, size_t len)
{
        int ret;
        struct mm_struct *mm;
        mm = current->mm;
        down_write(&mm->mmap_sem);
        ret = do_munmap(mm, addr, len);
        p_write(&mm->mmap_sem);
        return ret;
}
```

Таблицы страниц

Хотя пользовательские программы и работают с виртуальной памятью, которая отображается на физические адреса, процессоры работают непосредственно с этими физическими адресами. Следовательно, когда приложение обращается к адресу виртуальной памяти, этот адрес должен быть конвертирован в физический адрес, чтобы процессор смог выполнить запрос. Соответствующий поиск выполняется с помощью таблиц страниц. Таблицы страниц работают путем разбиения виртуального адреса на части. Каждая часть используется в качестве индекса (номера) записи в таблице. Таблица содержит или указатель на другую таблицу, или указатель на соответствующую страницу физической памяти.

В операционной системе Linux таблицы страниц состоят из трех уровней[3]. Несколько уровней позволяют эффективно поддерживать неравномерно заполненные адресные пространства даже для 64-разрядных машин. Если бы таблицы страниц были выполнены в виде одного статического массива, то их размер, даже для 32-разрядных аппаратных платформ, был бы чрезвычайно большим. В операционной системе Linux трехуровневые таблицы страниц используются даже для тех аппаратных платформ, которые аппаратно не поддерживают трехуровневых таблиц (например, для некоторых аппаратных платформ поддерживается только два уровня или аппаратно реализовано хеширование). Три уровня соответствуют своего рода "наибольшему общему знаменателю". Для аппаратных платформ с менее сложной реализацией работа с таблицами страниц в ядре при необходимости может быть упрощена с помощью оптимизаций компилятора.

Таблица страниц самого верхнего уровня называется глобальным каталогом страниц (page global directory, PGD). Таблица PGD представляет собой массив элементов типа pgd_t. Для большинства аппаратных платформ тип pgd_t соответствует типу unsigned long. Записи в таблице PGD содержат указатели на каталоги страниц более низкого уровня, PMD.

Каталоги страниц второго уровня еще называются каталогами страниц; среднего уровня (page middle directory, PMD). Каждый каталог PMD— это массив элементов типа prad_t. Записи таблиц PMD укалывают на таблицы PTE (page table entry, запись таблицы страниц).

Таблицы страниц последнего уровня называются просто таблицами страниц и содержат элементы типа pte_t. Записи таблиц страниц указывают на страницы памяти.

Для большинства аппаратных платформ поиск в таблицах страниц выполняется аппаратным обеспечением (по крайней мере частично). При нормальной работе аппаратное обеспечение берет на себя большую часть ответственности по использованию таблиц страниц. Однако для этого ядро должно все настроить так, чтобы аппаратное обеспечение могло нормально работать. На рис. 14.1 показана диаграмма того, как происходит перевод виртуального адреса в физический с помощью таблицы страниц.

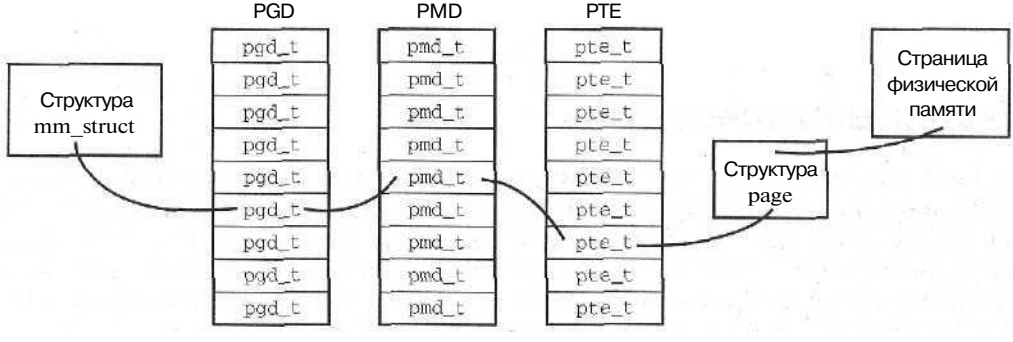


Рис. 14.1. *Таблицы страниц*

[3] Начиная с ядра версии 2.6.11 таблицы страниц в ОС Linux для 64-разрядных аппаратных платформ стали 4-уровневыми, что позволяет в полном объеме использовать все виртуальное адресное пространство. Для 32-разрядных аппаратных платформ осталось 3 уровня, как и раньше. — *Примеч. ред.*

Каждый процесс имеет свои таблицы страниц (разумеется, потоки эти таблицы используют совместно). Поле pgd дескриптора памяти указывает на глобальный каталог страниц. Манипуляции с таблицами и прохождение по ним требуют захвата блокировки page_table_lock, которая также находится в соответствующем дескрипторе памяти.

Структуры данных, связанные с таблицами страниц, сильно зависят от аппаратной платформы и определены в файле <asm/page.h>.

Поскольку практически каждое обращение к страницам виртуальной памяти требует определения соответствующего адреса физической памяти, производительность операций с таблицами страниц является очень критичной. Поиск всех этих адресов в памяти должен всегда выполняться очень быстро. Чтобы посодействовать этому, большинство процессоров имеют *буфер быстрого преобразования адреса (translation lookaside buffer,* или *TLB),* который работает, как аппаратный кэш отображения виртуальных адресов на физические. При обращении к виртуальному адресу процессор вначале проверяет, не кэшировано ли это отображение в TLB. Если обращение в кэш было удачным, то сразу же возвращается физический адрес. В противном случае поиск физического адреса выполняется с помощью таблиц страниц.

Несмотря на это, управление таблицами страниц все же остается критичной и развивающейся частью ядра. Изменения в ядре 2.6 включают выделение частей таблиц страниц не в области верхней памяти. В будущем, вероятно, появится возможность совместного использования таблиц страниц с копированием при записи. В такой схеме таблицы страниц будут совместно использоваться родительским и порожденным процессами даже после выполнения вызова fork(). Если же родительский или порожденный процесс изменит некоторую запись таблицы страниц, то будет создана копия этой записи, и эти процессы больше не будут совместно использовать данную запись. Совместное использование таблиц страниц позволит устранить затраты, связанные с копированием таблиц страниц при вызове fork().

Заключение

В этой главе была рассмотрена абстракция виртуальной памяти, которая предоставляется каждому процессу. Было рассказано, как ядро представляет адресное пространство процесса (с помощью структуры struct mm_struct) и каким образом ядро представляет области памяти внутри этого адресного пространства (struct vrn_area_struct). Также рассказывалось о том, как ядро создает (с помощью функции mmap()) и удаляет (с помощью функции munmap()) области памяти. Б конце были рассмотрены таблицы страниц. Так как операционная система Linux — это система с виртуальной памятью, то все эти понятия очень важны для понимания работы системы и используемой модели процессов.

В следующей главе рассматривается страничный кэш - общий кэш данных, который используется для выполнения страничных операций ввода-вывода и обратной записи страниц. Оставайтесь с нами!

15

Страничный кэш и обратная запись страниц

В ядре операционной системы Linux реализован один главный дисковый кэш, который называется страничным (page cache). Назначение этого кэша— минимизировать количество дисковых операций ввода-вывода путем хранения в памяти тех данных, для обращения к которым необходимо выполнять дисковые операции, Эта глава посвящена рассмотрению страничного кэша.

Кэширование дисковых данных важно по двум причинам. Во-первых, доступ к диску выполняется значительно медленнее, чем доступ к памяти. Доступ к данным в памяти выполняется значительно быстрее, чем к данным на диске. Во-вторых, если к некоторым данным осуществлялся доступ, то с достаточно большой вероятностью к этим же данным в ближайшем будущем потребуется обратиться снова. Принцип, согласно которому операции обращения к некоторым данным имеют тенденцию группироваться друг с другом во времени, называется сосредоточенностью во времени (temporal locality). Сосредоточенность во времени гарантирует, что если данные кэшируются при первом доступе к ним, то существует большая вероятность удачного обращения в кэш к этим данным в ближайшем будущем.

Страничный кэш состоит из физических страниц, которые находятся в оперативной памяти. Каждая страница памяти в кэше соответствует нескольким дисковым блокам. Когда ядро начинает некоторую операцию страничного ввода-вывода (дисковые, обычно файловые, операции ввода-вывода, которые выполняются порциями, равными размеру страницы памяти), то оно вначале проверяет, нет ли соответствующих данных в страничном кэше. Если эти данные есть в кэше, то ядро может не обращаться к диску и использовать данные прямо из страничного кэша.

Отдельные дисковые блоки также могут быть привязаны к страничному кэшу с помощью буферов блочного ввода-вывода. Вспомните из материала главы 13, "Уровень блочного ввода-вывода", что буфер — это представление в памяти одного физического дискового блока. Буферы играют роль дескрипторов, которые отображают страницы памяти на дисковые блоки. Поэтому страничный кэш также позволяет сократить количество обращений к диску при выполнении операций блочного ввода-вывода как за счет кэширования, так и за счет буферизации операций блочно-

го ввода-вывода для выполнения в будущем. Такой тип кэширования часто называют "буферным кэшем", хотя на самом деле это не отдельный кэш, а часть страничного кэша.

Рассмотрим те типы операций и данных, которые связаны со страничным кэшем. Страничный кэш в основном пополняется при выполнении страничных операций ввода-вывода, таких как `read()` и `write()`. Страничные операции ввода-вывода выполняются с целыми страницами памяти, в которых хранятся данные, что соответствует операциям с более, чем одним дисковым блоком. В страничном кэше данные файлов хранятся порциями. Размер одной порции равен одной странице памяти.

Операции блочного ввода-вывода работают в каждый отдельный момент времени с одним дисковым блоком. Часто встречающаяся операция блочного ввода-вывода — это чтение и запись файловых индексов. Ядро предоставляет функцию `bread()` , которая выполняет низкоуровневое чтение одного блока с диска. С помощью буферов дисковые блоки отображаются на связанные с ними страницы памяти и благодаря этому сохраняются в страничном кэше.

Например, при первом открытии в текстовом редакторе дискового файла с исходным кодом, данные считываются с диска и записываются в память. При редактировании файла считывается вес больше данных в страницы памяти. Когда этот файл позже начинают компилировать, то ядро может считывать соответствующие страницы памяти из дискового кэша. Нет необходимости снова считывать данные с диска. Поскольку пользователи склонны к тому, чтобы периодически работать с одними и теми же файлами, страничный кэш уменьшает необходимость выполнения большого количества дисковых операций.

Страничный кэш

Как следует из названия, страничный кэш (page cache) — это кэш страниц; памяти. Соответствующие страницы памяти получаются в результате чтения и записи обычных файлов на файловых системах, специальных файлов блочных устройств и файлов, отображаемых в память. Таким образом, в страничном кэше содержатся страницы памяти, полностью заполненные данными из файлов, к которым только что производился доступ. Перед выполнением операции страничного ввода-вывода, как, например, `read()`[1], ядро проверяет, есть ли те данные, которые нужно считать, в страничном кэше. Если данные находятся в кэше, то ядро может быстро возвратить требуемую страницу памяти.

Объект address_space

Физическая страница памяти может содержать данные из нескольких несмежных физических дисковых блоков[2].

[1] Какбыло показано в главе 12 ," Виртуальная файловая система", операции страничного ввода-вывода непосредственно выполняются не системными вызовами read() и write(), а специфичными для файловых систем методами *file->f_op->read()* и *file~>f_op->wriie()*.

[2] Например,размер страницы физической памяти для аппаратной платформы x86 равен 4 Кбайт, в то время как размер дискового блока для большинства устройств и файловых систем равен 512 байт. Следовательно, в одной странице памяти может храниться 8 блоков. Блоки не обязательно должны быть смежными, так как один файл может быть физически "разбросанным" по диску.

Проверка наличия определенных данных *в* страничном кэше может быть затруднена, если смежные блоки принадлежат совершенно разным страницам памяти. Невозможно проиндексировать данные в страничном кэше, используя только имя устройства и номер блока, что было бы наиболее простым решением.

Более того, страничный кэш ядра Linux является хранилищем данных достаточно общего характера в отношении того, какие страницы памяти в нем могут кэшироваться. Первоначально страничный кэш был предложен в операционной системе System V (SVR 4) для кэширования только данных из файловых систем. Следовательно, для управления страничным кэшем операционной системы SVR 4 использовался эквивалент файлового объекта, который назывался `struct vnode`. Кэш операционной системы Linux разрабатывался с целью кэширования *любых* объектов, основанных на страницах памяти, что включает множество типов файлов и отображений в память.

Для получения необходимой общности в страничном кэше операционной системы Linux используется структура `address_space` (адресное пространство), которая позволяет идентифицировать страницы памяти, находящиеся в кэше. Эта структура определена в файле <linux/fs.h> следующим образом.

```
struct address_space {
 struct inode            *host;            /* файловый индекс, которому
                                              принадлежит объект */
 struct radix_tree_root  page_tree;        /* базисное дерево всех страниц */
                                           /* блокировка для защиты
 spinlock_ t             tree_lock;           поля page_tree */
                                           /* количество областей памяти
 unsigned int            i_mmap_wrltable;     с флагом VM_SHARED */
                                           /* список всех отображений */
 struct prio_tree_root   i_mmap;           /* список областей памяти
 struct list_head        i_mmap_nonlinear;    с флагом VM_NONLINEAR */
 spinlock_t              i_mmap_lock;      /* Блокировка поля i_mmap */
 atomic_t                truncate_counl;   /* счетчик запросов truncate */
 unsigned long           nrpages;          /* общее количество страниц */
 pgoff_t                 writeback_index;  /* смещения начала обратной записи*/
 struct address_space_operations *a_ops;   /* таблица операций */
 unsigned long           flags;            /* маска gfp_mask и флаги ошибок */
 struct backing_dev_info  *backing_dev_info; /* информация упреждающего чтения */
 spinlock_t              private_lock;     /* блокировка для частных отображений */
 struct list_head    private_list;         /* список частных отображений */
 struct address_spacs    *assoc_mapping;   /* соответствующие буферы */
};
```

Поле i_mmap — это дерепо поиска по приоритетам для всех совместно используемых и частных отображений. Дерево поиска по приоритетам— это хитрая смесь базисных и частично упорядоченных бинарных деревьев[3].

Всего в адресном пространстве nrpages страниц памяти.

[3] Реализация ядра основана на базисном дереве поиска по приоритетам, предложенном в работе Edward M. McCreight, опубликованной в журнале SIAM Journal of Computing, May 1985, vol. 14. № 2, P. 257-276.

Объект addressspace связан с некоторым другим объектом ядра, обычно с файловым индексом. Если это так, то поле host указывает на соответствующий файловый индекс. Если значение поля host равно NULL, то соответствующий объект не является файловым индексом; например, объект address_space может быть связан с процессом подкачки страниц (swapper).

Поле a_ops указывает на таблицу операгций с адресным пространством так же, как и в случае объектов подсистемы VFS. Таблица операций представлена с помощью структуры struct address_space_operations, которая определена в файле <linux/fs.h> следующим образом.

```
struct address_space_operations {
        int (*writepage) (struct page *, struct writeback_control * ) ;
        int (*readpage) (struct file *, struct page * ) ;
        int (*sync_page) (struct page * ) ;
        int (*writepages) (struct address_space *,
                        structwriteback_control* ) ;
        int (*set_page_dirty) (struct page * ) ;
        int (*readpages) (struct file *, struct address_space *,
                        struct list_head *, unsigned);
        int (*prepare_write) (struct file *, struct page *,
                        unsigned, unsigned);
        int (*commit_write) (struct file *, struct page *,
                        unsigned, unsigned);
        sector_t (*bmap)(struct address_space *, sector_t);
        int (*invalidatepage) (struct page *, unsigned long);
        int (*releasepage) (struct page *, int);
        int (*direct_IO) (int, struct kiocb *, const struct iovec *,
                        loff_t, unsigned long);
};
```

Методы read_page и write_page являются наиболее важными. Рассмотрим шаги, которые выполняются при страничной операции чтения.

Методу чтения в качестве параметров передается пара значений: объект address_space и смещение. Эти значения используются следующим образом для поиска необходимых данных в страничном кэше.

```
page = find_get_page(mapping, index);
```

где параметр mapping — это заданное адресное пространство, а index - заданная позиция в файле.

Если в кэше нет необходимой страницы памяти, то новая страница памяти выделяется и добавляется в кэш следующим образом.

```
struct page *cached_page;
int error;

cached_page = page_cache_alloc_cold(mapping);
if (!cached_page)
        /* ошибка выделения памяти */
error = add_to_page_cache_lru(cached_page, mapping, index, GFP_KERNEL);
if (error)
        /* ошибка добавления страницы памяти в страничный кэш */
```

Наконец, необходимые данные могут быть считаны с диска, добавлены в страничный кэш и возвращены пользователю. Это делается следующим образом.

```
error = mapping->a_ops->readpage(file, page);
```

Операции записи несколько отличаются. Для отображаемых в память файлов при изменении страницы памяти система управления виртуальной памятью просто вызывает следующую функцию.

```
SetPageDirty(page);
```

Ядро выполняет запись этой страницы памяти позже с помощью вызова метода writepage (). Операции записи для файлов, открытых обычным образом (без отображения в память), выполняются более сложным путем. В основном, общая операция записи, которая реализована в файле mm/filemap.c, включает следующие шаги.

```
page = __grab_cache_page(mapping, index, &cached_page, &lru_pvec);
status  a_ops->prepare_write(file,  page, offset, offset+bytes);
page_fault = filemap_copy_from_user(page, offset, buf, bytes);
status = a_ops->commit_write(file, page, offset, offset+bytes};
```

Выполняется поиск необходимой страницы памяти в кэше. Если такая страница в кэше не найдена, то создается соответствующий элемент кэша. Затем вызывается метод prepare_write (), чтобы подготовить запрос на запись. После этого данные копируются из пространства пользователя в буфер памяти в пространстве ядра. И наконец данные записываются на диск с помощью функции comnit_write ().

Поскольку все описанные шаги выполняются при всех операциях страничного ввода-вывода, то все операции страничного ввода-вывода выполняются только через страничный каш. Ядро пытается выполнить все запросы чтения из страничного кэша. Если этого сделать не удается, то страница считывается с диска и добавляется в страничный кэш. Для операций записи страничный кэш выполняет роль "стартовой площадки". Следовательно, все записанные страницы также добавляются в страничный кэш.

Базисное дерево

Так как ядро должно проверять наличие страниц в страничном кэше перед тем, как запускать любую операцию страничного ввода-вывода, то этот поиск должен выполняться быстро. В противном случае затраты на поиск могут свести на нет все выгоды кэширования (по крайней мере, в случае незначительного количества удачных обращений в кэш, эти затраты времени будут сводить на нет все преимущества считывания данных из памяти по сравнению со считыванием напрямую с диска).

Как было показано в предыдущем разделе, поиск в страничном кэше выполняется на основании информации объекта address_space и значения смещения. Каждый объект address_space имеет свое уникальное базисное дерево (radix tree), которое хранится в поле pagetree. Базисное дерево - это один из типов бинарных деревьев. Базисное дерево позволяет выполнять очень быстрый поиск необходимой страницы только на основании значения смещения в файле. Функции поиска в страничном кэше, такие как find_get_page () и radix_tree_lookup (), выполняют поиск с использованием заданного дерева и заданного объекта.

Основной код для работы с базисными деревьями находится в файле lib/radix-tree.c. Для использования базисных деревьев необходимо подключить заголовочный файл <linux/radix-tree.h>.

Старая хеш-таблица страниц

Для ядер до серии 2.6 поиск в страничном кэше не выполнялся с помощью базисных деревьев. Вместо этого поддерживалась глобальная хеш-таблица всех страниц памяти в системе. Специальная хеш-функция возвращала двухсвязный список значений, связанных с одним значением ключа. Если нужная страница находится в кэше, то один из элементов этого списка соответствует этой нужной странице. Если страница в кэше отсутствует, то хеш-функция возвращает значение NULL.

Использование глобальной хеш-таблицы приводило к четырем основным проблемам.

- Хеш-таблица защищалась одной глобальной блокировкой. Количество конфликтов при захвате этой блокировки было достаточно большим даже для не очень больших машин. В результате страдала производительность.
- Размер хеш-таблицы был большим, потому что в ней содержалась информация обо всех страницах памяти в страничном кэше, в то время как важными являются лишь страницы, связанные с одним конкретным файлом.
- Производительность в случае неудачного обращения в кэш (когда искомая страница памяти не находится в кэше) падала из-за необходимости просматривать все элементы списка, связанного с заданным ключом.
- Хеш-таблица требовала больше памяти, чем другие возможные решения.

Применение в ядрах серии 2.6 страничного кэша на основании базисных деревьев позволило решить эти проблемы.

Буферный кэш

В операционной системе Linux больше нет отдельного буферного кэша. В ядрах серии 2.2 существовало два отдельных кэша: страничный и буферный. В первом кэшировались страницы памяти, а в другом — буферы. Эти два кэша не были объединены между собой. Дисковый блок мог находиться в обоих кэшах одновременно. Это требовало больших усилий по синхронизации двух кэшированных копий, не говоря уже о напрасной трате памяти.

Так было в ядрах серии 2.2 и более ранних, но начиная с ядер Linux серии 2.4 оба кэша объединили вместе. Сегодня существует только один дисковый кэш — страничный кэш.

Ядру все еще необходимо использовать буферы для того, чтобы представлять дисковые блоки в памяти. К счастью, буферы описывают отображение блоков на страницы памяти, которые в свою очередь находятся в страничном кэше.

Демон pdflush

Измененные (dirty, "грязные") страницы памяти когда-нибудь должны быть записаны на диск. Обратная запись страниц памяти выполняется в следующих двух случаях.

- Когда объем свободной памяти становится меньше определенного порога, ядро должно записать измененные данные обратно на диск, чтобы освободить память.
- Когда несохраненные данные хранятся в памяти достаточно долго, то ядро должно их записать на диск, чтобы гарантировать, что эти данные не будут находиться в Несохраненном состоянии неопределенное время.

Эти два типа записи имеют разные цели. В более старых ядрах они выполнялись двумя разными потоками пространства ядра (см. следующий раздел). Однако в ядре 2.6 эту работу выполняет группа (gang[4]) потоков ядра pdflush, которые называются демонами фоновой обратной записи (или просто потоками pdflush). Ходят слухи, что название pdflush — это сокращение от "dirty page flush" ("очистка грязных страниц"). Не обращайте внимание на это сомнительное название, давайте лучше более детально рассмотрим, для чего нужны эти процессы.

Во-первых, потоки pdflush служат для записи измененных страниц на диск, когда объем свободной памяти в системе уменьшается до определенного уровня. Цель такой фоновой записи— освобождение памяти, которую занимают незаписанные страницы, в случае недостатка физических страниц памяти. Уровень, когда начинается обратная запись, может быть сконфигурирован с помощью параметра dirty_background_ratio утилиты sysctl. Когда объем свободной памяти становится меньше этого порога, ядро вызывает функцию wakeup_bdflush ()[5] для перевода в состояние выполнения потока pdflush, который пыполняет функцию обратной записи измененных страниц памяти background_writeout (). Эта функция получает один параметр, равный количеству страниц, которые функция должна попытаться записать на диск.

Функция продолжает запись до тех пор, пока не выполнятся два следующих условия.

- Указанное минимальное количество страниц записано на диск.
- Объем свободной памяти превышает соответствующее значение параметра dirty_background_ratio.

Выполнение этих условий гарантирует, что демон pdflush выполнил свою работу по предотвращению нехватки памяти. Если эти условия не выполняются, то обратная запись может остановиться только тогда, когда демон pdflush запишет на диск *все* несохраненные страницы и для него больше не будет работы.

Во-вторых, назначение демона pdflush — периодически переходить в состояние выполнения (независимо от состояния нехватки памяти) и записывать на диск очень

[4] Слово "gang" не является жаргонным. Этот термин часто используется в компьютерных науках, чтобы указать группу чего-либо, что может выполняться параллельно.

[5] Да, название функции не совсем верное. Должно было бы быть *wakeup_pdflush ()*. В следующем разделе рассказано, откуда произошло это название.

давно измененные страницы памяти. Это гарантирует, что измененные страницы не будут находиться в памяти неопределенное время. При сбоях системы будут потеряны те страницы памяти, которые не были сохранены на диске, так как содержимое памяти после перегрузки не сохраняется. Следовательно, периодическая синхронизация страничного кэша с данными на диске является важным делом. При загрузке системы инициализируется таймер, периодически возвращающий к выполнению поток pdflush, который выполняет функцию wb_kupdate (). Эта функция выполняет обратную запись данных, которые были изменены более чем dirty_expire_centisecs сотых секунды тому назад. После этого таймер снова инициализируется, чтобы сработать через dirty_expire_centisecs сотых секунды. Таким образом потоки pdflush периодически возвращаются к выполнению и записывают на диск все измененные страницы, данные в которых старше, чем указанный лимит.

Системный администратор может установить эти значения с помощью каталога /proc/sys/vrn и утилиты sysctl. Втабл. 15.1 приведен список всех соответствующих переменных.

Таблица 15.1. Параметры для настройки демона pdflush

Переменная	Описание
dirty_background_ratio	Объем свободной оперативной памяти, при котором демон pdflush начинает обратную запись незаписанных данных
dirty_expire_centisecs	Время, в сотых долях секунды, в течение которого незаписанные данные могут оставаться в памяти, перед тем как демон pdflush не запишет их на диск при следующем периоде обратной записи
dirty_ratio	Процент от общей оперативной памяти, соответствующий страницам памяти одного процесса, при котором начинается обратная запись незаписанных данных
dirty_writeback_centisecs	Насколько часто, в сотых долях секунды, процесс bdflush возвращается к выполнению для обратной записи данных
laptop_mode	Переменная булевого типа, которая включает или выключает режим ноутбука (см. следующий раздел)

Код потока pdflush находится в файлах mm/page-writeback.c и fs/fs-writeback.c.

Режим ноутбука

Режим ноутбука — это специальная политика обратной записи страниц с целью оптимизации использования батареи и продления срока ее работы. Это делается путем минимизации активности жестких дисков, чтобы они оставались в остановленном состоянии по возможности долго. Конфигурировать этот режим можно с помощью файла /proc/sys/vm/laptop_mode. По умолчанию в этом файле записано значение 0 и режим ноутбука выключен. Запись значения 1 в этот файл позволяет включить режим ноутбука.

В режиме ноутбука существует всего одно изменение в выполнении обратной записи страниц. В дополнение к обратной записи измененных страниц; памяти, когда они становятся достаточно старыми, демон pdflush также выполняет и все остальные операции дискового ввода-вывода, записывая все дисковые буферы на

диск. Таким образом демон pdflush пользуется тем преимуществом, что диск уже запущен, а также он гарантирует, что в ближайшем будущем диск снова запущен не будет.

Такое поведение имеет смысл, когда параметры dirty_expire_centisecs и dirty_writeback_centisecs установлены в большие значения, скажем 10 минут. При таких задержках обратной записи диск запускается не часто, а когда он все-таки запускается, то работа в режиме ноутбука гарантирует, что этот момент будет использован с максимальной эффективностью.

Во многих поставках ОС Linux режим ноутбука автоматически включается и выключается, при этом также могут изменяться и другие параметры демона pbflush, когда заряд батареи уменьшается. Такое поведение позволяет получать преимущества от режима ноутбука при работе от батареи и автоматически возвращаться к нормальному поведению, когда машина включается в электрическую сеть.

Демоны bdflush и kupdated

В ядрах серий до 2.6 работа потоков pdflush выполнялась двумя другими потоками ядра bdflush и kupdated.

Поток пространства ядра bdflush выполнял фоновую обратную запись измененных страниц, когда количество доступной памяти становилось достаточно малым. Также был определен ряд пороговых значений, аналогично тому как это делается для демона pdflush. Демон bdflush возвращался к выполнению с помощью функции wakeup_bdflush () , когда количество свободной памяти становилось меньше этих пороговых значений.

Между демонами bdflush и pdflush существует два главных отличия. Первое состоит в том, что демон bdflush был всего один, а количество потоков pdflush может меняться динамически. Об этом более подробно будет рассказано в следующем разделе. Второе отличие состоит в том, что демон bdflush работал с буферами, он записывал на диск измененные буферы. Демон pdflush работает со страницами, он записывает на диск целые измененные страницы памяти. Конечно, страницы памяти могут соответствовать буферам, но единицей ввода-вывода является целая страница памяти, а не один буфер. Это дает преимущество, поскольку работать со страницами памяти проще, чем с буферами, так как страница памяти — более общий и более часто используемый объект.

Так как демон bdflush выполнял обратную запись, только когда количество свободной памяти очень сильно уменьшалось или количество буферов было очень большим, то был введен поток ядра kupdated, который периодически выполнял обратную запись измененных страниц памяти. Он использовался для целей, аналогичных функции wb_kupdate () демона pdflush.

Потоки bdflush и kupdated и их функциональность сейчас заменены потоками pdflush.

Предотвращение перегруженности: для чего нужны несколько потоков

Один из главных недостатков решения на основе демона bdflush состоит в том, что демон bdflush имел всего один поток выполнения. Это приводило к возможности зависания демона при большом количестве операций обратной записи, когда

один поток демона bdflush блокировался на очереди запросов ввода-вывода перегруженного устройства, в то время как очереди запросов других устройств могли быть в этот момент сравнительно свободными. Если система имеет несколько дисков и соответствующую процессорную мощность, то ядро должно иметь возможность загрузить работой все диски. К сожалению, даже при большом количестве данных, для которых необходима обратная запись, демон bdflush может оказаться загруженным работой с одной очередью и не сможет поддерживать все диски в нагруженном состоянии. Это происходит потому, что пропускная способность диском конечна и, к несчастью, очень низкая. Если только один поток выполняет обратную запись страниц, то он может проводить много времени в ожидании одного диска, так как пропускная способность диска ограничена. Для облегчения этой ситуации ядру необходима многопоточная обратная запись. В таком случае ни одна очередь запросов не может стать узким местом.

В ядрах серии 2.6 эта проблема решается путем введения нескольких потоков pdflush. Каждый поток самостоятельно выполняет обратную запись страниц памяти на диск, что позволяет различным потокам pdflush работать с разными очередями запросов устройств.

Количество потоков изменяется в процессе работы системы в соответствии с простым алгоритмом. Если все существующие потоки pdflush оказываются занятыми в течение одной секунды, то создается новый поток pdflush. Общее количество потоков не может превышать значения константы MAX_PDFLUSH_THREADS, которая по умолчанию равна 8. И наоборот, если поток pdflush находился в состоянии ожидания больше одной секунды, то он уничтожается. Минимальное количество потоков равно, по крайней мере, значению константы MIN_PDFLUSH_THREADS, что по умолчанию соответствует 2. Таким образом, количество потоков pdflush изменяется динамически в зависимости от количества страниц, для которых необходима обратная запись, и загруженности этих потоков. Если все потоки pdflush заняты обратной записью, то создается новый поток. Это гарантирует, что ни одна из очередей запросов устройств не будет перегружена, в то время как другие очереди устройств не так загружены и в них тоже можно выполнять обратную запись. Если перегрузка предотвращается, то количество потоков pdflush уменьшается, чтобы освободить память.

Все это хорошо, но что если все потоки pdflush зависнут в ожидании записи в одну и ту же перегруженную очередь? В этом случае производительность нескольких потоков pdflush не будет выше производительности одного потока, а количество занятой памяти станет значительно большим. Чтобы уменьшить такой эффект, для потоков pdflush реализован алгоритм предотвращения зависания (congestion avoidance). Потоки активно начинают обратную запись страниц для тех очередей, которые не перегружены. В результате потоки pdflush распределяют свою работу по разным очередям и воздерживаются от записи в перегруженную очередь. Когда все потоки pdflush заняты работой и запускается новый поток, то это означает, что они действительно заняты.

В связи с усовершенствованием алгоритмов обратной записи страниц, включая введение демона bdflush, ядро серии 2.6 позволяет поддерживать в загруженном состоянии значительно большее количество дисков, чем в более старых версиях ядер. При активной работе потоки pdflush могут обеспечить большую пропускную способность сразу для большого количества дисковых устройств.

Коротко о главном

В этой главе был рассмотрен страничный кэш и обратная запись страниц. Было показано, как ядро выполняет все операции страничного ввода-вывода, как операции записи откладываются с помощью дискового кэша и как данные записываются на диск с помощью группы потоков пространства ядра pdflush.

На основании материала последних нескольких глав вы получили устойчивое представление о том, как выполняется управление памятью и файловыми системами. Теперь давайте перейдем к теме модулей и посмотрим, ядро Linux обеспечивает модульную и динамическую инфраструктуру для загрузки кода ядра во время работы системы.

16

Модули

Несмотря на то что ядро является монолитным, в том смысле что все ядро выполняется в общем защищенном адресном домене, ядро Linux также является модульным, что позволяет выполнять динамическую вставку и удаление кода ядра в процессе работы системы. Соответствующие подпрограммы, данные, а также точки входа и выхода группируются в общий бинарный образ, загружаемый объект ядра, который называется *модулем*. Поддержка модулей позволяет системам иметь минимальное базовое ядро с опциональными возможностями и драйверами, которые компилируются в качестве модулей. Модули также позволяют просто удалять и перегружать код ядра, что помогает при отладке, а также дает возможность загружать драйверы по необходимости в ответ на появление новых устройств с функциями горячего подключения.

В этой главе рассказывается о хитростях, которые стоят за поддержкой модулей в ядре, и о том, как написать свой собственный модуль.

Модуль "Hello,World!"

В отличие от разработки основных подсистем ядра, большинство из которых были уже рассмотрено, разработка модулей подобна созданию новой прикладной программы, по крайней мере в том, что модули имеют точку входа, точку выхода и находятся каждый в своем бинарном файле.

Может показаться банальным, но иметь возможность написать программу, которая выводит сообщение "Hello World!", и не сделать этого- просто смешно. Итак, леди и джентльмены, модуль "Hello, World!".

```
 /*
 * hello.c - модуль ядра Hello, World!
 /*
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
/*
 * hello_init - функция инициализации, вызывается при загрузке модуля,
 * В случае успешной загрузки модуля возвращает значение нуль,
 * и ненулевое значение в противном случае.
 */
```

```
static int hello_init(void)
{
       printk(KERN_ALERT "I bear a charmed life.\n");
       return 0;
}

/*
 * hello_exit - функция завершения, вызывается при выгрузке модуля.
 */
static void hello_exit (void)
{
       printk(KERN_ALERT "Out, out, brief candle!\n");
}

module_init(hello_init);
module_exit(hello_exit);

MODULE_LICENSE{"GPL");
MODULE_AUTHOR("Shakespeare");
```

Это самый простой модуль ядра, который только может быть. Функция hello_init() регистрируется с помощью макроса module_init() в качестве точки входа в модуль. Она вызывается ядром при загрузке модуля. Вызов module_init() — это не вызов функции, а макрос, который устанавливает значение своего параметра в качестве функции инициализации. Все функции инициализации должны соотпетствовать следующему прототипу.

```
int my_init(void);
```

Так как функция инициализации редко вызывается за пределами модуля, ее обычно не нужно экспортировать и можно объявить с ключевым словом static.

Функции инициализации возвращают значение тина int. Если инициализация (или то, что делает функция инициализации) прошла успешно, то функция должна возвратить значение нуль. В случае ошибки возвращается ненулевое значение.

В данном случае эта функция просто печатает сообщение и возвращает значение нуль. В настоящих модулях функция инициализации регистрирует ресурсы, выделяет структуры данных и т.д. Даже если рассматриваемый файл будет статически скомпилирован с ядром, то функция инициализации останется и будет вызвана при загрузке ядра.

Функция hello_exit() регистрируется в качестве точки выхода из модуля с помощью макроса module_exit(). Ядро вызывает функцию hello_exit(), когда модуль удаляется из памяти. Завершающая функция должна выполнить очистку ресурсов, гарантировать, что аппаратное обеспечение находится в непротиворечивом состоянии, и т.д. После того как эта функция завершается, модуль выгружается.

Завершающая функция должна соответствовать следующему прототипу.

```
void my_exit(void);
```

Так же как и в случае функции инициализации, ее можно объявить как static.

Если этот файл будет статически скомпилирован с образом ядра, то данная функция не будет включена в образ и никогда не будет вызвана (так как если нет модуля, то код никогда не может быть удален из памяти).

Макрос MODULE_LICENSE () позволяет указать лицензию на право копирования модуля. Загрузка в память модуля, для которого лицензия не соответствует GPL, приведет к установке в ядре флага tainted (буквально, испорченное). Этот флаг служит для информационных целей, кроме того, многие разработчики уделяют меньше внимания сообщениям об ошибках, в которых указан этот флаг. Более того, модули, у которых лицензия не соответствует GPL, не могут использовать символы, которые служат "только для GPL" (см. раздел "Экспортируемые символы" ниже в этой главе).

Наконец, макрос MODULE_AUTHOR () позволяет указать автора модуля. Значение этого макроса служит только для информационных целей.

Сборка модулей

Благодаря новой системе сборки "kbuild", в ядрах серии 2.6 сборка модулей выполняется значительно проще, чем в старых сериях. Первое, что нужно сделать при сборке модулей, — это решить, где будет находиться исходный код модуля. Исходный код модуля необходимо правильно объединить с деревом исходных кодов ядра. Это можно сделать в виде заплаты или путем добавления в официальное дерево исходного кода ядра. Кроме этого, можно компилировать исходный код модуля отдельно от исходных кодов ядра.

Использование дерева каталогов исходных кодов ядра

В идеале модуль является частью официального ядра и находится в каталоге исходных кодов ядра. Введение вашей разработки непосредственно в ядро может вначале потребовать больше работы, но обычно такое решение более предпочтительно.

На первом этапе необходимо решить, *где* именно будет находиться модуль в дереве исходных кодов ядра. Драйверы необходимо хранить в подкаталогах каталога drivers/, который находится в корне дерева исходных кодов ядра. Внутри этого каталога драйверы делятся на классы, типы и собственно на отдельные драйверы. Символьные устройства находятся в каталоге drivers/char/, блочные— в каталоге drivers/block/, устройства USB — в каталоге drivers/usb/. Эти правила не есть жесткими, так как многие устройства USB также являются и символьными устройствами. Но такая организация является понятной и четкой.

Допустим, что вы хотите создать свой подкаталог и ваш воображаемый драйвер разработан для удочки с числовым программным управлением, которая имеет интерфейс Fish Master XL 2000 Titanium для подключения к компьютеру. Следовательно, необходимо создать подкаталог fishing внутри каталога drivers/char/.

После этого необходимо добавить новую строку в файл Makefile, который находится в каталоге drivers/char/. Для этого отредактируйте файл drivers/char/Makefile и добавьте в него следующую запись.

```
obj-m += fishing/
```

Эта строка указывает системе компиляции, что необходимо войти в подкаталог fishing/ при компиляции модулей. Скорее всего, компиляция драйвера определяется отдельным конфигурационным параметром, например, CONFIG_FISHING_POLE (как создавать новые конфигурационные параметры, рассмотрено ниже в этой главе в разделе "Управление конфигурационными параметрами"). В этом случае необходимо добавить строку следующего вида.

```
obj-$(CONFIG_FISHING_POLE) += fishing/
```

И наконец, в каталоге drivers/char/fishing необходимо добавить новый файл Makefile, содержащий следующую строку.

```
obj-m += fishing.o
```

При таких настройках система компиляции перейдет в каталог fishing/ и скомпилирует модуль fishing.ko из исходного файла fishing.c. Да, расширение объектного файла указано как .o, но в результате будет создан модуль с расширением .ko.

И снова, скорее всего, факт компиляции модуля будет зависеть от конфигурационного параметра, в таком случае в Makefile необходимо добавить следующую строку.

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
```

Однажды драйвер удочки может стать очень сложным. Введение функции автодетектирования наличия лески может привести к тому, что модуль станет очень большим и теперь будет занимать больше одного файла исходного кода. Никаких проблем! Просто нужно внести в Makefile следующую запись.

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

В последнем случае будут скомпилированы файлы fishing-main.c и fishing-line.c и скомпонованы в файл модуля fishing.ko.

Наконец, может потребоваться передать компилятору gcc дополнительные конфигурационные параметры. Для этого в файле Makefile необходимо добавить следующую строку.

```
EXTRA_CFLAGS += -DTITANIUM_POLE
```

Если вы желаете поместить ваши файлы в каталог drivers/char/, вместо того чтобы создавать новый подкаталог, то необходимо просто прописать указанные строки (те, что должны быть прописаны в файле Makefile подкаталога drivers/char/fishing/) в файле drivers/char/Makefile.

Для компиляции просто запустите процесс сборки ядра, как обычно. Если компиляция модуля зависит от конфигурационного параметра, как в данном случае она зависит от параметра CONFIG_FISHING_POLE, то необходимо включить этот конфигурационный параметр перед компиляцией.

Компиляция вне дерева исходных кодов ядра

Если вы предпочитаете разрабатывать и поддерживать ваш модуль отдельно от дерева исходных кодов ядра и жить жизнью аутсайдера, просто создайте файл Makefile следующего вида в том каталоге, где находится модуль.

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

Такая конфигурация позволяет скомпилировать файл fishing.c в файл fishing.ko. Если ваш исходный код занимает несколько файлов, то необходимо добавить две строки.

```
obj-m := fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

Такая конфигурация позволяет скомпилировать файлы fishing-main.c и fishing-line.c и создать модуль fishing, ko.

Главное отличие от случая, когда модуль находится внутри дерева исходного кода, состоит в процессе сборки. Так как модуль находится за пределами дерева исходных кодов ядра, необходимо указать утилите make местонахождение исходных файлов ядра и файл Makefile ядра. Это также делается просто с помощью следующей команды.

```
make -C /kerncl/source/location SUBDTRS=$PWD modules
```

В этом примере /kernel/source/location — путь к сконфигурированному дереву исходных кодов ядра. Вспомните, что *не нужно* хранить копию дерева исходных кодов ядра, с которой вы работаете, в каталоге /usr/src/linux, эта копия должна быть где-то в другом месте, скажем где-нибудь в вашем домашнем каталоге.

Инсталляция модулей

Скомпилированные модули должны быть инсталлированы в каталог /lib/modules/*version*/kernel. Например, для ядра 2.6.10 скомпилированный модуль управления удочкой будет находиться в файле /lib/modules/2.6.10/kernel/drivers/char/fishing.ko, если исходный код находился непосредственно в каталоге drivers/char/.

Для инсталляции скомпилированных модулей в правильные каталоги используется следующая команда.

```
make modules_install
```

Разумеется, эту команду необходимо выполнять от пользователя root.

Генерация зависимостей между модулями

Утилиты работы с модулями ОС Linux поддерживают зависимости между модулями. Это означает, что если модуль chum зависит от модуля bait, то при загрузке модуля chum модуль bait будет загружен автоматически. Информация о зависимостях между модулями должна быть сгенерирована администратором. В большинстве поставок ОС Linux эта информация генерируется автоматически и обновляется при

загрузке системы. Для генерации информации о зависимостях между модулями необходимо от пользователя root выполнить следующую команду.

```
depmod
```

Для быстрого обновления и генерации информации только о более новых модулях, чем сам файл информации, необходимо от пользователя root выполнить другую команду.

```
depmod -A
```

Информация о зависимостях между модулями хранится в файле /lib/modules/*version/modules*.dep.

Загрузка модулей

Наиболее простой способ загрузки модуля — это воспользоваться утилитой insmod. Эта утилита выполняет самые общие действия. Она просто загружает тот модуль, который ей указан в качестве параметра. Утилита insmod не отслеживает зависимости и не выполняет никакой интеллектуальной обработки ошибок. Использовать ее очень просто. От пользователя root необходимо просто выполнить команду

```
insmod module
```

где module— это имя модуля, который необходимо загрузить. Для загрузки модуля управления удочкой необходимо выполнить команду.

```
insmod fishing
```

Удалить модуль можно аналогичным образом с помощью утилиты rmmod. Для этого от пользователя root нужно просто выполнить команду.

```
rmmod module
```

Например, удалить модуль управления удочкой можно следующим образом.

```
rmmod fishing
```

Тем не менее, эти утилиты тривиальные и не обладают интеллектуальным поведением. Утилита modprobe позволяет обеспечить удовлетворение зависимостей, оповещение об ошибках, интеллектуальную обработку ошибок, а также выполняет множество других расширенных функций. Её настоятельно рекомендуется использовать.

Для загрузки модуля в ядро с помощью утилиты modprobe необходимо от пользователя root выполнить команду

```
modprobe module [ module parameters ]
```

где параметр module — это имя модуля, который необходимо загрузить. Все следующие аргументы интерпретируются как параметры, которые передаются модулю при загрузке. Параметры модулей обсуждаются ниже в одноименном разделе.

Утилита modprobe пытается загрузить не только указанный модуль, но и все модули, от которых он зависит. Следовательно, это наиболее предпочтительный механизм загрузки модулей ядра.

Команда modprobe также может использоваться для удаления модулей из ядра. Для этого с правами пользователя root необходимо выполнить ее следующим образом.

```
modprobe Pr modules
```

где параметр modules — имя одного или нескольких модулей, которые необходимо удалить. В отличие от команды rmmod, утилита modprobe также удаляет и все модули, от которых указанный модуль зависит, если последние не используются.

В восьмом разделе страниц руководства операционной системы Linux приведен список других, менее используемых ключей этой команды.

Управление конфигурационными параметрами

В предыдущих разделах рассматривалась компиляция модуля управления удочкой при условии, что установлен конфигурационный параметр CONFIG_FISHING_POLE. Конфигурационные параметры рассматривались в предыдущих главах, а теперь давайте рассмотрим добавление нового параметра в продолжение примера модуля управления удочкой.

Благодаря новой системе компиляции ядра "kbuild", которая появилась в серии ядер 2.6, добавление нового конфигурационного параметра является очень простым делом. Все, что необходимо сделать, — это добавить новую запись в файл Kconfig, который отвечает за конфигурацию дерева исходных кодов ядра. Для драйверов этот файл обычно находится в том же каталоге, в котором находится и исходный код. Если код драйвера удочки находится в каталоге drivers/char/, то необходимо использовать файл drivers/char/Kconfig.

Если был создан новый каталог и есть желание, чтобы файл конфигурации находился в этом новом каталоге, то необходимо на него сослаться из существующего файла Kconfig. Это можно сделать путем добавления строки

```
source Tdrivers/char/fishing/Kconfig
```

в существующий файл Kconfig, скажем в файл drivers/char/Kconfig.

Конфигурационные записи в файле Kconfig добавляются очень просто. Для модуля управления удочкой эта запись может выглядеть следующим образом.

```
config FISHING_POLE
       tristate "Fish Master XL support"
       default n
       help
         If you say Y here, support for the Fish Master XL 2000 Titanium with
         computer interface will be compiled into the kernel
         and accessible via
         device node. You can also say M here and the driver will be
         built as a
         module named fishing.ko.

         If unsure, say N.
```

Первая строка определяет, какой конфигурационный параметр создается. Обратите внимание, что префикс CONFIG_ указывать не нужно, он добавляется автоматически.

Вторая строка указывает на то, что параметр может иметь три состояния *(tristate),* которые соответствуют следующим значениям: статическая компиляция в ядро (Y), компиляция в качестве модуля (M) или не компилировать драйвер вообще (N). Для того чтобы запретить компиляцию кода, который соответствует конфигурационному параметру, в качестве модуля (допустим, что этот параметр определяет не драйвер. а просто некоторую дополнительную функцию) необходимо указать ТИП параметра `bool` вместо `tristate`. Текст в кавычках, который следует после этой директивы, определяет название конфигурационного параметра и будет отображаться различными утилитами конфигурации.

Третья строка позволяет указать значение этого параметра по умолчанию, который соответствует в данном случае запрещению компиляции.

Директива `help` указывает на то, что остальная часть текста будет интерпретироваться как описание данного модуля. Различные конфигурационные утилиты могут при необходимости отображать этот текст. Так как этот текст предназначен для пользователей и разработчиков, которые будут компилировать ядро, то он должен быть коротким и ясным. Обычные пользователя, скорее всего, не будут компилировать ядро, а сели будут, то тогда они должны понимать, что в этом описании сказано.

Существуют также и другие директивы файла конфигурации. Директива `depends` указывает на конфигурационные параметры, которые должны быть установлены перед тем, как может быть установлен текущий параметр. Если зависимости не будут удовлетворены, то текущий параметр будет запрещен. Например, можно указать следующую директиву.

```
depends on FISH_TANK
```

При этом текущий модуль не будет разрешен, пока не будет разрешен модуль, соответствующий конфигурационному параметру CONFIG_FISH_TANK.

Директива `select` аналогична директиве `depends`, за исключением того, что она принудительно включает указанный конфигурационный параметр, если включается текущая конфигурационная опция. Ее не нужно использовать так же часто, как директиву `depends`, потому что она включает другие конфигурационные опции. Использовать ее так же просто.

```
select BAIT
```

В этом случае конфигурационный параметр CONFIG_BAIT автоматически активизируется при включении конфигурационного параметра CONFIG_FISHING_POLE.

Как для директивы `select`, так и для директивы `depends` можно указывать несколько параметров с помощью оператора &&. В директиве `depends` с помощью восклицательного знака перед параметром можно указать требование, что некоторый конфигурационный параметр *не* должен быть установлен. Например, следующая запись указывает, что для компиляции текущего драйвера необходимо, чтобы был установлен конфигурационный параметр CONFIG_DUMB_DRIVERS и не был установлен параметр CONFIG_NO_FISHING_ALLOWED.

```
depends on DUMB_DRIVERS && !NO_FISHING_ALLOWED
```

После директив tristate и bool можно указать директиву if, что позволяет сделать соответствующий параметр зависимым от другого конфигурационного параметра. Если условие не выполняется, то конфигурационный параметр не только запрещается, но и не будет отображаться утилитами конфигурации. Например, следующая строка указывает, что функция "Deep Sea Mode" будет доступна, только если разрешен конфигурационный параметр CONFIG_OKEAN.

```
bool TDeep Sea ModeY if OCEAN
```

Директива if также может быть указана после директивы default, что означает, что значение по умолчанию будет установлено, только если выполняется условие, указанное в директиве if.

Система конфигурации экспортирует несколько метапараметров, чтобы упростить процесс конфигурации. Параметр CONFIG_EMBEDDED устанавливается только тогда, когда пользователь указывает, что он хочет видеть вес параметры, отвечающие за запрещение некоторых ключевых позможностей ядра (обычно с целью сохранения памяти на встраиваемых системах). Параметр CONFIG_BROKEN_ON_SMP используется, чтобы указать, что драйвер не рассчитан на системы с симметричной многопроцессорностью. Обычно этот параметр не устанавливается, при этом от пользователя требуется, чтобы он сам убедился в возможности компиляции драйвера для SMP. Новые драйверы этот флаг использовать не должны.

Параметр CONFIG_EXPERIMENTAL используется для указания экспериментальных или не очень хорошо оттестированных возможностей. По умолчанию этот параметр отключен, что требует от пользователя лично убедиться в степени риска при разрешении компиляции того или иного драйвера.

Параметры модулей

Ядро Linux предоставляет возможность драйверам определять параметры, которые пользователь будет указывать при загрузке ядра или модуля. Эти параметры будут доступны коду модуля в качестве глобальных переменных. Указанные параметры модулей также будут отображаться в файловой системе sysfs (см. главу 17, "Объекты kobject и файловая система sysf "). Определять параметры модуля и управлять ими просто.

Параметр модуля определяется с помощью макроса module_param () следующим образом.

```
module_param(name, type, perm);
```

где аргумент name — это имя неременной, которая появляется в модуле, и имя параметра, который может указать пользователь. Аргумент type — это тип данных параметра. Значения типа могут быть следующими: byte, short, ushort, int. uint, long, ulong, charp, bool или invbool. Эти значения соответствуют следующим типам данных: байт; короткое целое число; короткое целое число без знака; целое число; целое число без знака; длинное целое; длинное целое число без знака; указатель на строку символов; булев тип; булев тип, значение которого инвертируется по сравнению с тем, которое указывает пользователь. Данные типа byte хранятся в переменной типа char, а данные булевых типов — в переменных типа int. Остальные типы соответствуют аналогичным типам языка С. Наконец, аргумент perm указыва-

ет права доступа к соответствующему файлу в файловой системе sysfs. Права доступа можно указать как в обычном восьмеричном формате, например 0644 (владелец имеет права на чтение и запись, группа имеет права на чтение и запись, остальные пользователи имеют право только на чтение), так и в виде определений препроцессора, объединенных с помощью оператора "|", например S_IRUGO | S_IWUSR (все могут считывать данные, а владелец также и записывать). Нулевое значение этого параметра приводит к тому, что соответствующий файл в файловой системе sysfs не появляется.

Этот макрос не определяет переменную. *Перед* тем как использовать макрос, соответствующую переменную нужно определить. В связи с этим типичный пример использования может быть следующим.

```
/* параметр модуля, который управляет переменной bait */
static int allow_live_bait = 1;            /* по умолчанию включено */
module_param(allow_live_bait, bool, 0644); /* булев тип */
```

Это определение должно быть в глобальной области видимости, т.е. переменная allow_live_bait должна быть глобальной.

Существует возможность дать внешнему параметру модуля имя, отличное от имени переменной. Это можно сделать с помощью макроса module_param_named().

```
module_param_named(name, variable, type, perm);
```

где name — это имя внешнего параметра модуля, а variable — имя внутренней глобальной переменной модуля, как показано ниже.

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

Для того чтобы определить параметр модуля, значением которого является строка символов, необходимо использовать тип charp. Ядро копирует переданную пользователем строку символов в память и присваивает переменной указатель на эту строку, как в следующем примере.

```
static char *name;
module_param(name, charp, 0);
```

При необходимости ядро может скопировать строку в заранее определенный массив символов, который указывает разработчик. Это делается с помощью макроса module_param_string().

```
module_param_string(name, string, len, perm);
```

где name — это имя внешнего параметра, string — имя внутренней переменной, которая содержит указатель на область памяти массива, len — размер буфера string (или некоторое меньшее число, чем размер буфера, что, однако, обычно не имеет смысла), perm — права доступа к файлу на файловой системе sysfs (нулевое значение запрещает доступ к параметру через sysfs). Пример показан ниже.

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

В качестве параметров модуля также можно передавать список значений, которые разделены запятой и в коде модуля будут записаны в массив данных. Эти пара-

метры модуля можно обработать с помощью макроса module_param_array() следующим образом.

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

В данном случае аргумент name — это имя внешнего параметра и внутренней переменной, type — это тип данных одного значения, а perm — это права доступа к файлу на файловой системе sysfs. Новый аргумент nump — это указатель на целочисленное значение, где ядро сохраняет количество элементов, записанных в массив. Обратите внимание, что массив, который передается в качестве параметра name, должен быть выделен статически. Ядро определяет размер массива на этапе компиляции и гарантирует, что он не будет переполнен. Как использовать данный макрос, показано в следующем примере.

```
static int fish[MAX_FISH];
static int nr_fish;
module_param_array(fish, int, &nr_fish, 0444);
```

Внутренний массив может иметь имя, отличное от имени внешнего параметра, в этом случае следует использовать макрос module_param_array_named ().

```
module_param_array_named(name, array, type, nump, perm);
```

Параметры идентичны аналогичным параметрам других макросов.

Наконец, параметры модуля можно документировать, используя макрос MODULE_PARM_DESC().

```
static unsigned short size = 1;
module_param(size, ushort, 0644);
MODULE_PARM__DESC(size, "The size in inches of the fishing pole " \
                  "connected to this computer.");
```

Вес описанные в этом разделе макросы требуют включения заголовочного файла <linux/moduleparam.h>.

Экспортируемые символы

При загрузке модули динамически компонуются с ядром. Так же как и в случае динамически загружаемых бинарных файлов пространства пользователя, в коде модулей могут вызываться только те функции ядра (основного образа или других модулей), которые явно *экспортируются* для использования. В ядре экспортирование осуществляется с помощью специальных директив EXPORT_SYMBOL () и EXPORT_SYMBOL_GPL().

Функции, которые экспортируются, доступны для использования модулями. Функции, которые не экспортируются, не могут быть вызваны из модулей. Правила компоновки и вызова функций для модулей значительно более строгие, чем для основного образа ядра. Код ядра может использовать любые интерфейсы ядра (кроме тех, которые определены с ключевым словом static), потому что код ядра компонуется в один выполняемый образ. Экспортируемые символы, конечно, тоже не должны определяться как static.

Набор символов ядра, которые экспортируются, называется экспортируемым интерфейсом ядра или даже (здесь не нужно удивляться) *API ядра*.

Экспортировать символы просто. После того как функция определена, необходимо вызвать директиву EXPORT_SYMBOL ().

```
/*
 * get_pirate_beard_color — возвратить значение цвета бороды текущего
 * пирата pirate — это глобальная переменная, доступная из данной
 * функции цвета определены в файле <linux/beard_colors.h>
 */
int get_pirate_beard_color(void)
{
        return pirate->beard->color;
}
EXPORT_SYMBOL(get_pirate_beard_color);
```

Допустим, что функция get_pirate_beard_color() объявлена в заголовочном файле и ее может использовать любой модуль.

Некоторые разработчики хотят, чтобы их интерфейсы были доступны только для модулей с лицензией GPL. Такая возможность обеспечивается компоновщиком ядра с помощью макроса MODULE_LICENSE (). Если есть желание, чтобы рассматриваемая функция была доступна только для модулей, которые помеченные как соответствующие лицензии GPL, то экспортировать функцию можно следующим образом.

```
EXPORT_SYMBOL_GPL(get_pirate_beard_color);
```

Если код ядра конфигурируется для компиляции в виде модуля, то необходимо гарантировать, что все используемые интерфейсы экспортируются. В противном случае будут возникать ошибки компоновщика и загружаемый модуль не будет работать.

Вокруг модулей

В этой главе были рассмотрены особенности написания, сборки, загрузки и выгрузки модулей ядра. Мы обсудили, что такое модули и каким образом ядро операционной системы Linux, несмотря на то что оно является монолитным, может загружать код динамически. Были также рассмотрены параметры модулей и экспортируемые символы. На примере воображаемого модуля ядра (драйвера устройства) управления удочкой был показан процесс написания модуля и процесс добавления к нему различных возможностей, таких как внешние параметры.

В следующей главе будут рассмотрены объекты kobject и файловая система sysfs, которые являются основным интерфейсом к драйверам устройств и, следовательно, к модулям ядра.

17

Объекты kobject и файловая система sysfs

Унифицированная *модель представления устройств* — это существенно новая особенность, которая появилась в ядрах серии 2.6. Модель устройств — это единый механизм для представления устройств и описания их топологии в системе. Использование единого представления устройств позволяет получить следующие преимущества.

- Уменьшается дублирование кода.
- Используется механизм для выполнения общих, часто встречающихся функций, таких как счетчики использования.
- Появляется возможность систематизации всех устройств в системе, возможность просмотра состояний устройств и определения, к какой шине то или другое устройство подключено.
- Появляется возможность генерации полной и корректной информации о древовидной структуре всех устройств в системе, включая все шины и соединения.
- Обеспечивается возможность связывания устройств с их драйверами и наоборот.
- Появляется возможность разделения устройств ва категории в соответствии с различными классификациями, таких как устройства ввода, без знания физической топологии устройств.
- Обеспечивается возможность просмотра иерархии устройств от листьев к корню и выключения питания устройств в правильном порядке.

Последний пункт был самой первой мотивацией необходимости создания общей модели представления устройств. Для того чтобы реализовать интеллектуальное управление электропитанием в ядре, необходимо построить дерево, которое представляет топологию устройств в системе. Для выключения питания устройств, которые организованы в виде древовидной топологии, ориентированной сверху вниз, ядро должно выключить питание нижних узлов (листьев) перед выключением питания верхних узлов. Например, ядро должно выключить питание USB-мыши перед

тем, как выключать питание контроллера шины USB, а питание контроллера шины USB должно быть выключено перед выключением питания шины PCI. Чтобы делать это эффективно и правильно для всей системы, ядру необходимо отслеживать топологию дерева всех устройств в системе.

Объекты kobject

Сердцем модели представления устройств являются объекты *kobject,* которые представляются с помощью структуры struct kobject, определенной в файле <linux/kobject.h>. Тип kobject аналогичен классу Object таких объектно-ориентированных языков программирования, как C# и Java. Этот тип определяет общую функциональность, такую как счетчик ссылок, имя, указатель на родительский объект, что позволяет создавать объектную иерархию.

Структура, с помощью которой реализованы объекты kobject, имеет следующий вид.

```
struct kobject {
        char                *k_name;
        char                name[KOBJ_NAME_LEN];
        struct kref         kref;
        struct list_head    entry;
        struct kobject      *parent;
        struct kset         *kset;
        struct kobj_type    *ktype;
        struct dentry       *dentry;
};
```

Поле k_name содержит указатель на имя объекта. Если длина имени меньше KOBJ_NAME_LEN, что сейчас составляет 20 байт, то имя хранится в массиве name, а поле kname указывает на первый элемент этого массива. Если длина имени больше KOBJ_NAME_LEN байт, то динамически выделяется буфер, размер которого достаточен для хранения строки символов имени, имя записывается в этот буфер, а поле k_name указывает на него.

Указатель parent указывает на родительский объект данного объекта kobject. Таким образом, с помощью структур kobject может быть создана иерархия объектов в ядре, которая позволяет устанавливать соотношения родства между различными объектами. Как будет видно дальше, с помощью файловой системы sysfs осуществляется представление в пространстве пользователя той иерархии объектов kobject, которая существует в ядре.

Указатель dentry содержит адрес структуры struct dentry, которая представляет этот объект в файловой системе sysfs.

Поля kref, ktype и kset указывают на экземпляры структур, которые используются для поддержки объектов kobject. Поле entry используется совместно с полем kset. Сами эти структуры и их использование будут обсуждаться ниже.

Обычно структуры kobject встраиваются в другие структуры данных и сами по себе не используются. Например, такая важная структура, как struct cdev, имеет поле kobj.

```
/* структура cdev - объект для представления символьных устройств */
struct cdev {
       struct kobject           kobj;
       struct module            *owner;
       struct file_operations   *ops;
       struct list_head         list;
       dev_t                    dev;
       unsigned int             count;
};
```

Когда структуры kobject встраиваются в другие структуры данных, то последние получают те стандартизированные возможности, которые обеспечиваются структурами kobject. Еще более важно, что структуры, которые содержат в себе объекты kobject, становятся частью объектной иерархии. Например, структура cdev представляется в объектной иерархии с помощью указателя на родительский объект cdev->kobj->parent и списка cdev->kobj->entry.

Типы ktype

Объекты kobject могут быть связаны с определенным типом, который называется ktype. Типы ktype представляются с помощью структуры struct kobj_type, определенной в файле <linux/kobject.h> следующим образом.

```
struct kobj_type {
       void (*release)(struct kobject * ) ;
       struct sysfs_ops       *sysfs_ops;
       struct attribute       **default_attrs;
};
```

Тип ktype имеет простое назначение— представлять общее поведение для некоторого семейства объектов kobject. Вместо того чтобы для каждого отдельного объекта задавать особенности поведения, эти особенности связываются с их полем ktype, и объекты одного "типа" характеризуются одинаковым поведением.

Поле release содержит указатель на деструктор, который вызывается, когда количество ссылок на объект становится равным нулю. Эта функция отвечает за освобождение памяти, связанной с объектом, и за другие операции очистки.

Поле sysfs_ops указывает на структуру sysfs_ops. Эта структура определяет поведение файлов на файловой системе sysfs при выполнении операций записи и чтения. Более детально она рассматривается в разделе "Добавлепие файлов на файловой системе sysfs".

Наконец, поле default_attrs указывает на массив структур attribute. Эти структуры определяют *атрибуты,* которые связаны с объектом kobject и используются но умолчанию. Атрибуты соответствуют свойствам данного объекта. Если некоторый объект kobject экспортируется через файловую систему sysfs, то атрибуты экспортируются как отдельные файлы. Последний элемент этого массива должен содержать значению NULL.

Множества объектов kset

Множества kset представляют собой коллекции объектов kobject. Множество kset работает как базовый контейнерный класс для объектов, например, "все блочные устройства". Множества kset очень похожи на типы ktype, и возникает вопрос: "Для чего нужны два разных обобщения?" Множество kset объединяет несколько объектов kobject, а типы ktype определяют общие свойства, которые связаны с объектами kobject одного типа. Существует возможность объединить объекты одного типа ktype в различные множества kset.

Поле kset объекта kobject указывает на связанное с данным объектом множество kset. Множество объектов kset представляется с помощью структуры kset, которая определена в файле <linux/koject.h> следующим образом.

```
struct kset {
       struct subsystem          *subsys;
       struct kobj_type          *ktype;
       struct list_head          list;
       struct kobject            kobj;
       struct kset_hotplug_ops   *hotplug_ops;
};
```

Указатель ktype указывает на структуру ktype, которая определяет тип всех объектов данного множества, поле list - список всех объектов kobject данного множества, поле kobj — объект kobject, который представляет базовый класс для всех объектов данного множества, а поле hotplug_ops указывает на структуру, которая определяет поведение объектов kobject при горячем подключении устройств, связанных с данным множеством.

Наконец, поле sybsys указывает на структуру struct subsystem, которая связана с данным множеством kset.

Подсистемы

Подсистемы используются для представления высокоуровневых концепций ядра и являются коллекцией одного или нескольких множеств kset. Множества kset содержат объекты kobject, подсистемы — множества kset, но связь между множествами в подсистеме значительно более слабая, чем связь между объектами kobject в множестве. Множества kset одной подсистемы могут иметь только наиболее общие объединяющие факторы.

Несмотря на их важную роль, подсистемы представляются с помощью очень простой структуры данных — struct subsystem.

```
struct subsystem {
       struct kset               ksot;
       struct rw_semaphore       rwsem;
       };
```

Структура subsystem содержит только одно множество kset, тем не менее несколько множеств kset могут указывать на общую структуру subsystem с помощью

поля subsys. Такие однонаправленные взаимоотношения означают, что нет возможности определить все множестпа подсистемы, только имея ее структуру subsystem.

Поле kset, которое содержится в структуре subsystem, — это множество kset подсистемы, которое используется по умолчанию, чтобы зафиксировать положение этой подсистемы в иерархии объектов.

Поле rwsem структуры subsystem — это семафор чтения-записи (см. главу 9, "Средства синхронизации в ядре"), который используется для защиты подсистемы и ее множеств kset от конкурентного доступа. Все множества kset должны принадлежать какой-нибудь подсистеме, поскольку они используют семафор подсистемы для защиты своих данных от конкурентного доступа.

Путаница со структурами

Те несколько структур, которые только что были описаны, приводят к путанице не потому, что их много (только четыре) или они сложные (все они достаточно просты), а потому что они сильно друг с другом переплетаются. При использовании объектов kobject достаточно сложно рассказать об одной структуре, не упоминая другие. Тем не менее, на основании рассмотренных особенностей этих структур можно построить прочное понимание их взаимоотношений.

Самым важным является объект kobject, который представляется с помощью структуры struct kobject. Структура kobject используется для представления наиболее общих объектных свойств структур данных ядра, таких как счетчик ссылок, взаимоотношения родитель-порожденный и имя объекта. С помощью структуры kobject эти свойства можно обеспечить одинаковым для всех стандартным способом. Сами по себе структуры kobject не очень полезны, они обычно встраиваются в другие структуры данных.

С каждым объектом kobject связан один определенный тип данных— ktype, который представляется с помощью структуры struct kobj_type. На экземпляр такой структуры указывает поле ktype каждого объекта kobject. С помощью типов ktype определяются некоторые общие свойства объектов: поведение при удалении объекта, поведение, связанное с файловой системой sysfs, а также атрибуты объекта.

Объекты kobject группируются в множества, которые называются kset. Множества kset представляются с помощью структур данных struct kset. Эти множества предназначены для двух целей. Во-первых, они позволяют использовать встроенный в них объект kobject в качестве базового класса для группы других объектов kobject. Во-вторых, они позволяют объединять вместе несколько связанных между собой объектов kobject. На файловой системе sysfs объекты kobject представляются отдельными каталогами файловой системы. Связанные между собой каталоги, например все подкаталоги одного каталога, могут быть включены в одно множество kset.

Подсистемы соответствуют большим участкам ядра и являются набором множеств kset. Подсистемы представляются с помощью структур struct subsystem. Все каталоги, которые находятся в корне файловой системы sysfs, соответствуют подсистемам ядра.

На рис. 17.1 показаны взаимоотношения между этими структурами данных.

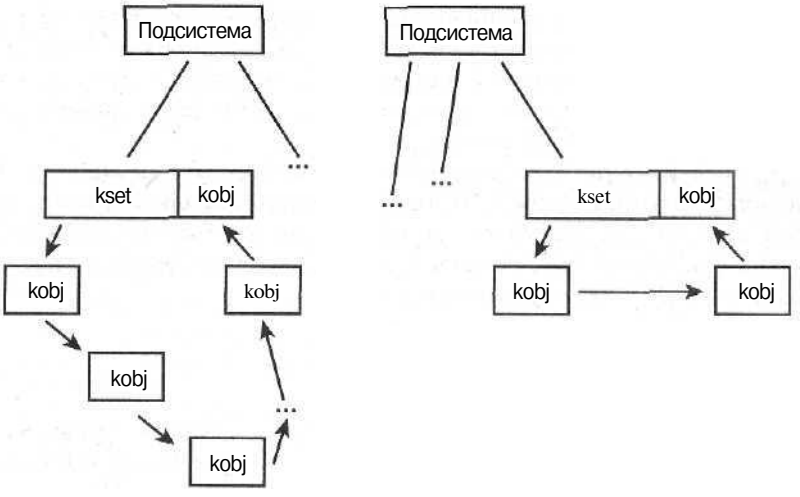


Рис. 17.1. Взаимоотношения между объектами kobject, множествами kset и подсистемами

Управление и манипуляции с объектами kobject

Теперь, когда у нас уже есть представление о внутреннем устройстве объектов kobject и связанных с ними структурах данных, самое время рассмотреть экспортируемые интерфейсы, которые дают возможность управлять объектами kobject и выполнять с ними другие манипуляции. В основном, разработчикам драйверов непосредственно не приходится иметь дело с объектами kobject. Структуры kobject встраиваются в некоторые специальные структуры данных (как это было в примере структуры устройства посимвольного ввода-вывода) и управляются "за кадром" с помощью соответствующей подсистемы драйверов. Тем не менее, объекты kobject не всегда могут оставаться невидимыми, иногда с ними приходится иметь дело, как при разработке кода драйверов, так и при разработке кода управления подсистемами ядра.

Первый шаг при работе с объектами kobject - это их декларация и инициализация. Инициализируются объекты kobject с помощью функции kobject_init(), которая определена в файле <linux/kobject.h> следующим образом.

```
void kobject_init(struct kobject *kobj);
```

Единственным параметром этой функции является объект kobject, который необходимо проинициализировать. Перед вызовом этой функции область памяти, в которой хранится объект, должна быть заполнена нулевыми значениями. Обычно это делается при инициализации большой структуры данных, в которую встраивается объект kobject. В других случаях просто необходимо вызвать функцию memset().

```
memset(kobj, 0, sizeof (*kobj));
```

После заполнения нулями безопасным будет инициализация полей parent и kset, как показано в следующем примере.

```
kobj = kmalloc(sizeof (*kobj), GFP_KERNEL);
if (!kobj)
       return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj->ksct - kset;
kobj->parent = parent_kobj;
kobject_init(kobj);
```

После инициализации необходимо установить имя объекта с помощью функции kobject_set_name(), которая имеет следующий прототип.

```
int kobject_set_name(struct kobject *kobj, const char* fmt, , . . . ) ;
```

Эта функция принимает переменное количество параметров, по аналогии с функциями printf() и printk(). Как уже было сказано, на имя объекта указывает поле k_name структуры kobject. Если это имя достаточно короткое, то оно хранится в статически выделенном массиве name, поэтому есть смысл без необходимости не указывать длинные имена.

После того как для объекта выделена память и объекту присвоено имя, нужно установить значение его поля kset, а также опционально поле ktype. Последнее необходимо делать только в том случае, если множество kset не предоставляет типа ktype для данного объекта, в противном случае значение поля ktype, которое указано в структуре kset, имеет преимущество. Если интересно, почему объекты kobject имеют свое поле ktype, то добро пожаловать в клуб!

Счетчики ссылок

Одно из главных свойств, которое реализуется с помощью объектов kobject, — это унифицированная система поддержки счетчиков ссылок. После инициализации количество ссылок на объект устанавливается равным единице. Пока значение счетчика ссылок на объект не равно нулю, объект существует в памяти, и говорят, что он *захвачен (pinned,* буквально, пришпилен). Любой код, который работает с объектом, вначале должен увеличить значение счетчика ссылок. После того как код закончил работу с объектом, он должен уменьшить значение счетчика ссылок. Увеличение значения счетчика называют *захватом (getting),* уменьшение — *освобождением (putting)* ссылки на объект. Когда значение счетчика становится равным нулю, объект может быть уничтожен, а занимаемая им память освобождена.

Увеличение значения счетчика ссылок выполняется с помощью функции kobject_get().

```
struct kobject * kobject_get(struct kobject *kobj);
```

Эта функция возвращает указатель на объект kobject в случае успеха и значение NULL в случае ошибки.

Уменьшение значения счетчика ссылок выполняется с помощью функции kobject_put().

```
void kobject put(struct kobject *kobj);
```

Если значение счетчика ссылок объекта, который передается в качестве параметра, становится равным нулю, то вызывается функция, на которую указывает указатель release поля ktype этого объекта.

Структуры kref

Внутреннее представление счетчика ссылок выполнено с помощью структуры kref, которая определена в файле <linux/kref. h> следующим образом.

```
struct kref{
        atomic_trefcount;
};
```

Единственное поле этой структуры — атомарная переменная, в которой хранится значение счетчика ссылок. Структура используется просто для того, чтобы выполнять проверку типов. Чтобы воспользоваться структурой kref, необходимо ее инициализировать с помощью функции kref_init () .

```
void kref_init(struct kref *krcf)
{
        atomic_set(&kref->refcount, 1);
        }
```

Как видно из определения, эта функция просто инициализирует атомарную переменную тина atomic_t в значение, равное единице.

Следовательно, структура kref является захваченной сразу же после инициализации, так же ведут себя и объекты kobject.

Для того чтобы захватить ссылку на структуру kref, необходимо использовать функцию kref_get ().

```
void kref_get(struct kref *kref)
{
        WARN_ON(!atomic_read(&kref->refcount));
        atomic_inc(&kref->refcount);
}
```

Эта функция увеличивает значение счетчика ссылок на единицу. Она не возвращает никаких значений. Чтобы освободить ссылку на структуру kref, необходимо использовать функцию kref_put ().

```
void kref_put(struct kref *kref, void (*release) (struct kref *kref))
{
        WARN_ON(release == NULL);
        WARN_ON(release== (void (*)(struct kref *))kfree);

        if (atomicdec_and_test(&kref->refcount))
                release(kref);
}
```

Эта функция уменьшает значение счетчика ссылок на единицу и вызывает функцию release (), которая передастся ей в качестве параметра, когда значение счетчика ссылок становится равным нулю. Как видно из использованного выражения WARM_ON (), функция release () не может просто совпадать с функцией kfree (),

а должна быть специальной функцией, которая принимает указатель на структуру struct kref в качестве своего единственного параметра и не возвращает никаких значений.

Вместо того чтобы разрабатывать свои функции управления счетчиками ссылок на основании типа данных atomic_t, настоятельно рекомендуется использовать тип данных kref и соответствующие функции, которые обеспечивают общий и правильно работающий механизм поддержки счетчиков ссылок в ядре.

Все эти функции определены в файле lib/kref.c и объявлены в файле <linux/kref.h>.

Файловая система sysfs

Файловая система sysfs — это виртуальная файловая система, которая существует только в оперативной памяти и позволяет просматривать иерархию объектов kobject. Она позволяет пользопателям просматривать топологию устройств операционной системы в виде простой файловой системы. Атрибуты объектов kobject могут экспортироваться в виде файлов, которые позволяют считывать значения переменных ядра, а также опционально записывать их.

Хотя изначально целью создания модели представления устройств было описание топологии устройств системы для управления электропитанием, файловая система sysfs стала удачным продолжением этой идеи. Для того чтобы упростить отладку, разработчик унифицированной модели устройств решил экспортировать дерево устройств в виде файловой системы. Такое решение показало свою полезность вначале в качестве замены файлов, связанных с устройствами, которые раньше экспортировались через файловую систему /proc, а позже в качестве мощного инструмента просмотра информации о системной иерархии объектов. Вначале, до появления объектов kobject, файловая система sysfs называлась driverfs. Позже стало ясно — новая объектная модель была бы очень кстати, и в результате этого появилась концепция объектов kobject. Сегодня каждая система, на которой работает ядро 2.6, имеет поддержку файловой системы sysfs, и практически во всех случаях эта файловая система монтируется.

Основная идея работы файловой системы sysfs — это привязка объектов kobject к структуре каталогов с помощью поля dentry, которое есть в структуре kobject. Вспомните из материала главы 12, "Виртуальная файловая система", что структура dentry используется для представления элементов каталогов. Связывание объектов с элементами каталогов проявляется в том, что каждый объект просто видится как каталог файловой системы. Экспортирование объектов kobject в виде файловой системы выполняется путем построения дерева элементов каталогов в оперативной памяти. Но обратите внимание, объекты kobject уже образуют древовидную структуру — нашу модель устройств! Поэтому простое назначение каждому объекту иерархии, которые уже образуют дерево в памяти, соответствующего элемента каталога позволяет легко построить файловую систему sysfs.

На рис. 17.2 показан частичный вид файловой системы sysfs, которая смонтирована на каталог /sys.

Корневой каталог файлопой системы sysfs содержит семь подкаталогов: block, bus, class, devices, firmware, module и power. В каталоге block содержатся каталоги для каждого зарегистрированного в системе устройства блочного ввода-вывода.

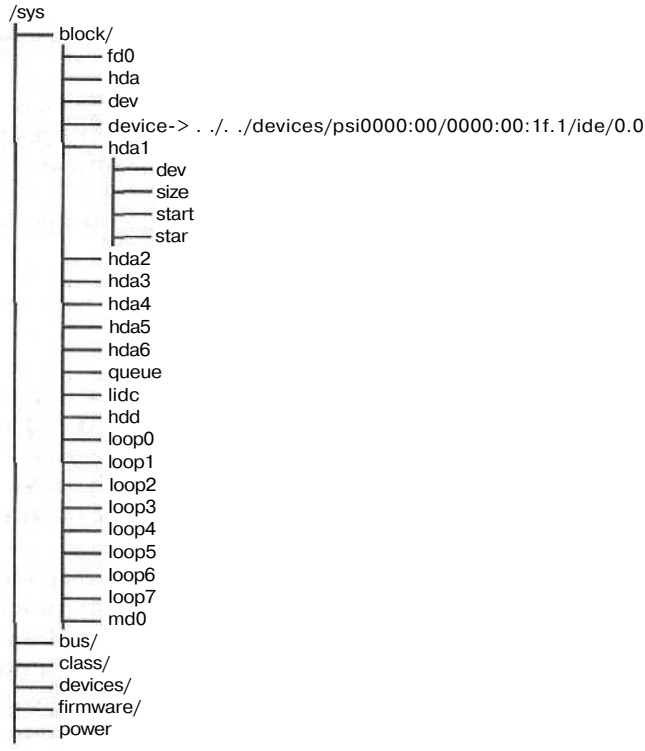


Рис. 17.2. *Содержимое части каталога /sys*

Каждый из каталогов в свою очередь содержит подкаталоги, соответствующие разделам блочного устройства. Каталог bus позволяет просматривать информацию о системных шинах. В каталоге class представлена информация о системных устройствах, которая организована в соответствии с высокоуровневыми функциями этих устройств. Каталог devices содержит информацию о топологии устройств в системе. Она отображается непосредственно на иерархию структур устройств ядра. Каталог firmware содержит специфичное для данной системы дерево низкоуровневых подсистем, таких как ACPI, EDD, EFT и т.д. В каталоге power содержатся данные по управлению электропитанием всех устройств системы.

Наиболее важным является каталог devices, который экспортирует модель устройств ядра во внешний мир. Структура каталога соответствует топологии устройств в системе. Большинство информации, которая содержится в других каталогах, — это просто другое представление данных каталога devices. Например, в каталоге /sys/class/net/ информация представлена в соответствии с высокоуровневым представлением зарегистрированных сетевых устройств. В этом каталоге может содержаться подкаталог eth0, который содержит символьную ссылку device на соответствующее устройство каталога devices.

Посмотрите на содержимое каталога /sys той системы Linux, к которой вы имеете доступ. Такое представление системных устройств является очень четким и ясным. Оно показывает взаимосвязь между высокоуровневым представлением информации в каталоге class, низкоуровневым представлением в каталоге devices

и драйверами устройств — в каталоге bus. Такое представление взаимосвязи между устройствами очень информативно. Оно становится еще более ценным, если осознать, что все эти данные свободно доступны и описывают все то, что происходит внутри ядра[1].

Добавление и удаление объектов на файловой системе sysfs

Инициализированные объекты kobject автоматически не экспортируются через файловую систему sysfs. Для того чтобы сделать объект видимым через sysfs, необходимо использовать функцию kobject_add().

```
int kobject_add(struct kobject *kobj);
```

Положение объекта на файловой системе sysfs зависит от его положения в объектной иерархии. Если установлен указатель parent объекта, то объект будет отображен внутри каталога, соответствующего объекту, на который указывает указатель parent. Если указатель parent не установлен, то объект будет отображен в каталоге, соответствующем значению переменной kset->kobj. Если для некоторого объекта не установлены ни значение поля parent, ни значение поля kset, то считается, что данный объект не имеет родительского и будет отображаться в корневом каталоге файловой системы sysfs. Такое поведение практически всегда соответствует тому, что нужно. Поэтому одно из полей parent или kset (или оба) должно быть установлено правильным образом перед вызовом функции kobject_add() . Имя каталога, который представляет объект kobject в файловой системе sysfs, будет определяться значением поля kobj->name.

Вместо того чтобы последовательно вызывать функции kobject_init() и kobject_add(), можно вызвать функцию kobject_register().

```
int kobject_register(struct kobject *kobj);
```

Удаление объекта из файловой системы sysfs выполняется с помощью функции kobject_del().

```
void kobject_del(struct kobject *kobj);
```

Функция kobjectunregister() сочетает в себе выполнение функций kobject_del() и kobject_put().

```
void kobject_unregister(struct kobject * kobj);
```

Все эти четыре функции определены в файле lib/kobject.c и объявлены в файле <linux/kobject.h>.

[1] Если вас заинтересовала информация о файловой системе sysfs, то, вероятно, вам будет интересно также ознакомиться с HAL, hardware abstraction layer (уровень абстракции аппаратного обеспечения), информация о котором доступна по адресу http://hal.freedesktop.org/. Подсистема HAL позволяет создать в оперативной памяти базу данных на основании информации файловой системы sysfs, объединяя вместе понятия классов, устройств и драйверов. На основании этих данных уровень HAL предоставляет API, которое позволяет разрабатывать более интеллектуальные программы.

Добавление файлов на файловой системе sysfs

Объекты kobject отображаются на каталоги, и такое отображение выполняется естественным образом. А как насчет создания файлов? Файловая система sysfs — это не что иное, как дерево каталогов без файлов.

Атрибуты, используемые по умолчанию

Набор файлов, которые создаются в каталоге по умолчанию, определяется с помощью поля ktype объектов kobject и множеств kset. Следовательно, все объекты kobject одного типа имеют один и тот же набор файлов в каталогах, которые этим объектам соответствуют. Структура kobject_type содержит поле default_attrs, которое представляет собой массив структур attribute. Атрибуты отображают данные ядра на файлы в файловой системе sysfs.

Структура attributes определена в файле <linux/sysfs.h>.

```
/* структура attribute - атрибуты позволяют отобразить данные ядра
   на файлы файловой системы sysfs */
struct attribute {
       char              *name;    /* имя атрибута */
       struct module     *owner;   /* модуль, если есть, которому
                                      принадлежат данные */
       mode_t             mode;    /* права доступа к файлу */
       };
```

Поле name содержит имя атрибута. Такое же имя будет иметь и соответствующий файл на файловой системе sysfs. Поле owner — это указатель на структуру module, которая представляет загружаемый модуль, содержащий соответствующие данные. Если такого модуля не существует, то значение поля равно NULL. Поле mode имеет тип mode_t и указывает права доступа к файлу на файловой системе sysfs. Если атрибут предназначен для чтения всеми, то флаг прав доступа доллсен быть установлен в значение S_IRUGO, если атрибут имеет право на чтение только для владельца, то права доступа устанавливаются в значение S_IRUSR. Атрибуты с правом на запись, скорее всего, будут иметь права доступа S_IRUGO | S_IWUSR. Все файлы и каталоги на файловой системе sysfs принадлежат пользователю с идентификаторами пользователя и группы равными нулю.

Структура attribute используется для представления атрибутов, а структура sysfs_ops описывает, как эти атрибуты использовать. Поле sysfs_ops — это указатель на одноименную структуру, которая определена в файле <linux/sysfs.h> следующим образом.

```
struct sysfs_ops {
       /* метод вызывается при чтении файла на файловой системе sysfs */
       ssize_t (*show)  (struct kobject *kobj,
                         struct attribute *attr,
                         char *buffer);
       /* метод вызывается при записи файла на файловой системе sysfs */
       ssize_t (*store)  (struct kobject *kobj,
                          struct attribute *attr,
                          const char *buffer,
                          size_t size);
};
```

Метод show() вызывается при чтении файла. Он должен выполнить копирование значения атрибута, который передается в качестве параметра attr, в буфер, на который указывает параметр buffer. Размер буфера равен PAGE_SIZE байт. Для аппаратной платформы значение PAGE_SIZE равно 4096 байтов. Функция должна возвратить количество байтов данных, которые записаны в буфер в случае успешного завершения, и отрицательный код ошибки, если такая ошибка возникает.

Метод store() вызывается при записи. Он должен скопировать size байт данных из буфера buffer в атрибут attr. Размер буфера всегда равен PAGE_SIZE или меньше. Функция должна возвратить количество байтов данных, которые прочитаны из буфера при успешном выполнении, и отрицательный код ошибки в случае неудачного завершения.

Так как этот набор функций должен выполнять операции ввода-вывода для *всех* атрибутов, то необходимо выполнить некоторые дополнительные действия, чтобы вызвать обработчик, специфичный для каждого атрибута.

Создание нового атрибута

Обычно атрибутов, которые используются по умолчанию и предоставляются типом ktype, связанным с объектом kobject, оказывается достаточно. Действительно, все объекты kobject одного типа должны быть чём-то похожи друг на друга или даже быть идентичными по своей природе. Например, для всех разделов жестких дисков один и тот же набор атрибутов должен подходить для всех объектов kobject. Это не просто упрощает жизнь, но и позволяет упорядочить код и получить одинаковый способ доступа ко всем каталогам файловой системы sysfs, связанным с родственными объектами.

Тем не менее иногда требуется, чтобы определенный экземпляр объекта kobject имел некоторые специфические свойства. Для таких объектоп может оказаться желательным (или необходимым) создать атрибут, которого нет у общего типа данного объекта. Для такого случая ядро предоставляет функцию sysf s_create_file () для добавления атрибута к существующему объекту.

```
int sysfs_create_file(struct kobject *kobj, const struct attribute *attr);
```

Эта функция позволяет привязать структуру attribute, на которую указывает параметр attr, к объекту kobject, на который указывает параметр kobj. Перед тем как вызвать эту функцию, необходимо установить значение атрибута (заполнить поля структуры). Эта функция возвращает значение нуль в случае успеха и отрицательное значение в случае ошибки.

Обратите внимание, что для обработки указанного атрибута используется структура sysfs_ops, соответствующая типу ktype объекта. Иными словами, существующие функции show () и store {), которые используются для объекта по умолчанию, должны иметь возможность обработать вновь созданный атрибут.

Кроме того, существует возможность создавать символьные ссылки. Создать символьную ссылку на файловой системе sysfs можно с помощью вызова следующей функции.

```
int sysfs_create_link(struct kobject *kobj,
                      struct kobject *target, char *name);
```

Эта функция создает символьную ссылку с именем name в каталоге объекта, соответствующего параметру kobj, на каталог, соответствующий параметру target. Эта функция возвращает нулевое значение в случае успеха и отрицательный код ошибки в противном случае.

Удаление созданного атрибута

Удаляется атрибут с помощью вызова функции sysfs_remove_file().

```
void sysfs_remove_file(struct kobject *kobj, const struct attribute *attr);
```

После возврата из этой функции указанный атрибут больше не отображается в каталоге объекта.

Символьная ссылка, созданная с помощью функции sysfs_create_link(), может быть удалена с помощью функции sysfs_remove_link().

```
void sysfs_remove_link (struct kobject *kobj, char *name);
```

После возврата из функции символьная ссылка с именем name удаляется из каталога, на который отображается объект kobj.

Все эти четыре функции объявлены в файле <linux/kobject.h>. Функции sysfs_create_file() и sysfs_remove_file() определены в файле fs/sysfs/file.c, а функции sysfs_create_link() и sysfs_remove_link() — в файле fs/sysfs/symlink.c.

Соглашения по файловой системе sysfs

Файловая система sysfs — это место, где должна реализовываться функциональность, для которой раньше использовался системный вызов ioctl() для специальных файлов устройств, или файловая система procfs. Сегодня модно выполнять такие вещи через атрибуты файловой системы sysfs в соответствующем каталоге. Например, вместо того чтобы реализовать новую директиву ioctl() для специального файла устройства, лучше добавить соответствующий атрибут в каталоге файловой системы sysfs, который относится к этому устройству. Такой подход позволяет избежать использования небезопасных, из-за отсутствия проверки типов аргументов, директив ioctl(), а также файловой системы /proc с ее бессистемным расположением файлов и каталогов.

Однако чтобы файловая система sysfs оставалась четко организованной и интуитивно понятной, разработчики должны придерживаться определенных соглашений.

Во-первых, каждый атрибут sysfs должен экспортировать значение одной переменной на файл. Значения должны быть в текстовом формате и соответствовать простым типам языка программирования С. Целью такого представления является необходимость избежать чрезвычайно запутанного и плохо структурированного представления информации, которое мы сегодня имеем на файловой системе /proc. Использование одной переменной на файл позволяет легко считывать и записывать данные из командной строки, а также просто работать через файловую систему sysfs с данными ядра в программах, написанных на языке С. В случаях, когда одно значение на файл приводит к неэффективному представлению информации, допустимо использование файлов, в которых хранится несколько значений одного типа. Эти данные необходимо четко разделять. Наиболее предпочтительным разделителем является символ пробела. При разработке кода ядра необходимо всегда помнить, что

файлы файловой системы sysfs являются представлениями переменных ядра, и ориентироваться на доступ к ним из пространства пользователя, в частности из командной строки.

Во-вторых, данные файловой системы sysfs должны быть организованы в виде четкой иерархии. Для этого необходимо правильно разрабатывать связи "родитель-потомок" объектов kobject. Связывать атрибуты с объектами kobject необходимо с учетом того, что эта иерархия объектов существует не только в ядре, но и экспортируется в пространство пользователя. Структуру файловой системы sysfs необходимо поддерживать в четком виде!

Наконец, необходимо помнить, что файловая система sysfs является службой ядра и в некотором роде интерфейсом ядра к прикладным программам (Application Binary Interface, ABT). Пользовательские программы должны разрабатываться в соответствии с наличием, положением, содержимым и поведением каталогов и файлов на файловой системе sysfs. Изменение положения существующих файлов крайне не рекомендуется, а изменение поведения атрибутов, без изменения их имени или положения, может привести к серьезным проблемам.

Эти простые соглашения позволяют с помощью файловой системы sysfs обеспечить в пространстве пользователя интерфейс ядра с широкими возможностями. При правильном использовании файловой системы sysfs разработчики прикладных программ не будут вас ругать и будут вам благодарны за хороший код.

Уровень событий ядра

Уровень событий ядра (kernel event layer) — это подсистема, которая позволяет передавать информацию о различных событиях из ядра в пространство пользователя и реализована, как вы уже, наверное, догадываетесь, на базе объектов kobject. После выпуска ядра версии 2.6.0 стало ясно, что необходим механизм для отправления сообщений из ядра в пространство пользователя, в частности для настольных рабочих компьютеров, что позволит сделать такие системы более функциональными, а также лучше использовать асинхронную обработку. Идея состояла в том, что ядро будет помещать возникающие события в стек. Например, "Жесткий диск переполнен!", "Процессор перегрелся!", "Раздел диска смонтирован!", "На горизонте появился пиратский корабль!" (последнее, конечно, шутка).

Первые реализации подсистемы событий ядра появились незадолго до того, как эта подсистема стала тесно связанной с объектами kobject и файловой системой sysfs. В результате такой связи реализация получилась достаточно красивой. В модели уровня событий ядра, события представляются в виде *сигналов,* которые посылаются объектами, в частности объектами типа kobject. Так как объекты отображаются на элементы каталогов файловой системы sysfs, то *источниками* событий являются определенные элементы пути на файловой системе sysfs. Например, если поступившее событие связано с первым жестким диском, то адресом источника события является каталог /sys/block/hda. Внутри же ядра источником события является соответствующий объект kobject.

Каждому событию присваивается определенная строка символов, которая представляет сигнал и называется *командой (verb)* или *действием (action).* Эта строка символов содержит в себе информацию о том, *что именно* произошло, например *изменение (modified)* или *размонтирование (unmounted).*

Каждое событие может нести в себе некоторую дополнительную информацию (дополнительную нагрузку, payload). Вместо того чтобы передавать в пространство пользователя строку, которая содержит эту полезную информацию, данная дополнительная информация представляется с помощью атрибутов, отображаемых на файловой системе sysfs.

События ядра поступают из пространства ядра в пространство пользователя через интерфейс netlink. Интерфейс netlink - это специальный тип высокоскоростного сетевого сокета групповой передачи (multicast), который используется для передачи сообщений, связанных с сетевой подсистемой. Использование интерфейса netlink позволяет выполнить обработку событий ядра с помощью простых блокирующих вызовов функций для чтения информации из сокетов. Задача пространства пользователя— реализовать системный процесс-демон, который выполняет прослушивание сокета, считывает информацию о всех приходящих событиях, обрабатывает их и отправляет полученные сообщения в системный стек пространства пользователя. Одна из возможных реализаций такого демона, работающего в пространстве пользователя, — это D-BUS[2]. который также реализует и системную шину сообщений. Таким образом, ядро может подавать сигналы так же, как это делают все остальные компоненты системы.

Для отправки события в пространство пользователя код ядра должен вызвать функцию kobject_uevent().

```
int kobject_uevent(struct kobject *kobj,
                   enum kobject_action action,
                   struct attribute *attr);
```

Первый параметр указывает объект kobject, который является источником сигнала. Соответствующее событие ядра будет содержать элемент пути на файловой системе sysfs, связанный с объектом, сгенерировавшим сигнал.

Второй параметр позволяет указать *команду* или *событие,* которое описывает сигнал. Сгенерированное событие ядра будет содержать строку, которая соответствует номеру, передаваемому в качестве значения параметра enum kobject_action. Вместо того чтобы непосредственно передать строку, здесь используется ее номер, который имеет тип перечисления (enum). Это дает возможность более строго выполнить проверку типов, изменить соответствие между номером строки и самой строкой в будущем, а также уменьшить количество ошибок и опечаток. Перечисления определены в файле <linux/kobject_uevent.h> и имеют имена в формате KOBJ_foo. На момент написания книги были определены следующие события: KOBJ_MOUNT, KOBJ_UNMOUNT, KOBJ_ADD, KOBJ_REMOVE и KOBJ_CHANGE. Эти значения отображаются на строки "mount" (монтирование), "unmount" (размонтироваыие), "add" (добавление), "remove" (удаление) и "change" (изменение) соответственно. Допускается добавление новых значений событий, если существующих значений недостаточно.

Последний параметр — опциональный указатель на структуру attribute. Этот параметр можно трактовать как дополнительную информацию (payload) о событии. Если только одного значения события недостаточно, то событие может предоставить информацию о том, в каком файле файловой системы sysfs содержатся дополнительные данные.

[2]Более подробную информацию о демоне D-BUS можно найти на сайте http://dbus.freedesktop.org/.

Рассмотренная функция использует динамическое выделение памяти и поэтому может переходить в состояние ожидания. Существует атомарная версия рассмотренной функции, которая идентична ей по всем, кроме того что при выделении использует флаг GFP_ATOMIC.

```
int kobject_uevent_atomic (struct kobject *kobj,
                           enum kobject_action action,
                           struct attribute *attr);
```

По возможности необходимо использовать стандартный интерфейс без атомарного выделения памяти. Параметры этих функций и их смысл — идентичны.

Использование объектов kobject и их атрибутов не только дают возможность описать события в терминах файловой системы sysfs, но и стимулируют создание новых объектов и их атрибутов, которые еще не представлены через файловую систему sysfs.

Обе рассмотренные функции определены в файле lib/kobject_uevent.c и объявлены в файле <linux/kobject_uevent.h>.

Кратко об объектах kobject и файловой системе sysfs

В этой главе рассматривается модель представления устройств, файловая система sysfs, объекты kobject и уровень событий ядра. Описание материала главы было бы невозможно без рассмотрения родственных вещей: были также описаны множества kset, подсистемы, атрибуты, типы ktype и счетчики ссылок kref. Эти структуры предназначены для использования разными людьми в разных местах. Разработчикам драйверов необходимо только ознакомление с внешними интерфейсами. Большинство подсистем драйверов эффективно скрывают внутренние механизмы использования объектов kobject и других, близких к ним структур. Понимание основных принципов работы и знание основного назначения интерфейсов, таких как sysfs_create_file() , является достаточным для разработчиков драйверов. Однако для разработчиков, которые занимаются разработкой основного кода ядра, может потребоваться более детальное понимание принципов функционирования объектов kobject. Объекты kobject могут оказаться еще более важными, так как их могут использовать и те разработчики, которые вообще не занимаются разработкой подсистем драйверов!!!

Эта глава — последняя из тех, которые посвящены подсистемам ядра. В следующих главах будут рассмотрены некоторые общие вопросы, которые также могут оказаться важными для разработчиков ядра. Например, основные рекомендации по отладке кода!

18

Отладка

Один из самых существенных факторов, который отличает разработку ядра от разработки пользовательских приложений, — это сложность отладки. Отлаживать код ядра сложно, но крайней мере по сравнению с кодом пространства пользователя. Еще больше усугубляет ситуацию тот факт, что ошибка в ядре может привести к катастрофическим последствиям для всей системы.

Успех в освоении приемов отладки ядра и, в конце концов, в разработке ядра вообще, в основном, зависит от опыта и понимания принципов работы операционной системы в целом. Понятно также, что, для того чтобы успешно выполнять отладку ядра, необходимо понимать, как ядро работает. Тем не менее когда-то нужно начать, и в этой главе будут рассмотрены подходы к отладке ядра.

С чего необходимо начать

Итак, готовы ли вы начать охоту за ошибками? Этот путь может оказаться длинным и полным разочарований. Некоторые ошибки ставили в тупик все сообщество разработчиков ядра на несколько месяцев. К счастью, на каждую из таких злостных ошибок находятся простые, которые легко исправить. Если вам повезет, то все проблемы, с которыми вы столкнетесь, будут простыми и тривиальными. Однако чтобы это проверить, необходимо начать исследования. Для этого понадобится следующее.

- Сама проблема. Может звучать глупо, но дефект должен быть конкретным и хорошо определенным. Очень помогает, если его хотя бы кто-нибудь может устойчиво воспроизвести. Однако, к сожалению, дефекты обычно ведут себя не так хорошо, как хотелось бы, и не всегда могут быть хорошо определены.
- Версия ядра, в которой существует дефект (обычно это последняя версия, хотя кто может это гарантировать?). Еще лучше, если известна версия ядра, в которой проблема впервые появилась. Мы рассмотрим, как это установить, если нет такой информации.
- Немного удачи, опыта и их комбинации.

Если дефект нельзя воспроизвести, то многие из приведенных ниже подходов становятся бесполезными. Очень важно, чтобы проблему можно было повторить. Если этого не удается сделать, то исправление дефекта становится возможным только путем визуального анализа кода для того, чтобы найти в нем ошибку. На самом

деле так случается достаточно часто (например, с разработчиками ядра), но очевидно, что шансы добиться успеха становятся более весомыми, если появляется возможность воспроизвести проблему.

Может также показаться странным, что существуют дефекты, которые кто-то не может воспроизвести. Дело в том, что в пользовательских программах дефекты чаще всего проявляются очень просто, например *вызов функции* foo *приводит к созданию файла* core. В ядре все совсем по-другому. Взаимодействия между ядром, пространством пользователя и аппаратурой могут быть достаточно тонкими. Состояния конкуренции за ресурсы могут возникать с вероятностью одно на миллион итераций алгоритма. Плохо спроектированный или даже не правильно скомпилированный код может обеспечивать удовлетворительную производительность на одной системе, но неудовлетворительную на другой, Очень часто происходит так, что на какой-то случайной машине, при очень специфическом характере загрузке, начинают проявляться дефекты, которые больше нигде не проявляются. Чем больше доступно дополнительной информации при локализации дефекта, тем лучше. Во многих случаях, как только удалось устойчиво воспроизвести проблему, можно считать, что большая половина работы сделана.

Дефекты ядра

Дефекты в ядре могут быть такими же разнообразными, как и дефекты в пользовательских программах. Они возникают по различным причинам и проявляются в разнообразных формах. Дефекты занимают диапазон от явно неправильного кода (например, запись правильного значения в неправильное место) до ошибок синхронизации (например, если не правильно блокируется совместно используемая переменная). Эти дефекты проявляются в любой форме; от плохой производительности до неправильного функционирования и даже до потери данных.

Часто, между тем моментом, когда в ядре возникла ошибка и тем моментом, когда пользователь ее заметил происходит большая цепь событий. Например, разделяемая структура данных, у которой нет счетчика использования может привести к возникновению состояния конкуренции за ресурс (race condition). Если не принять необходимых мер, то один процесс может освободить память, в которой хранится структура, в то время, как другой процесс может эту структуру все еще использовать. Спустя некоторое время второй процесс может обратиться к этим данным, что в свою очередь может привести к попытке разыменования указателя со значением NULL, если будут считаны случайные данные ("мусор"), или вообще не привести ни к чему плохому (если данные в соответствующей области памяти еще не были перезаписаны). Разыменование указателя со значением NULL приводит к выводу сообщения "oops", в то время, как случайный "мусор" может привести к потере данных (и соответственно к неправильному функционированию, или опять же к выводу сообщения "oops", но уже по другом поводу). Пользователь же заметит только неправильное функционирование или сообщение "oops". Разработчик ядра при этом должен пойти но обратному пути: исходя из ошибки определить, что к данным было обращение после того, как память с этими данными была освобождена, что это произошло в результате возникновения конкуренции за ресурс и исправить ошибку путем правильного учета количества ссылок на совместно используемую структуру данных. Для этого также вероятно потребуется применение блокировок.

Отладка ядра может показаться сложным делом, тем не менее, ядро не особо отличается от других больших программных проектов. У ядра есть свои уникальные особенности, такие как ограничения связанные со временем выполнения участков кода, возможности возникновения состояний конкуренции (race) — как результат параллельного выполнения множества потоков в ядре. Можно дать стопроцентную гарантию, что если приложить некоторые усилия и понимание, то проблемы ядра можно с успехом находить и решать (и даже, возможно, получать удовольствие от успешного преодоления трудностей).

Функция printk()

Функция форматированного вывода сообщений printk() работает аналогично библиотечной функции printf() языка С. Действительно в этой книге до этого момента мы не видели никаких существенных отличий в ее использовании. Для большинства задач это именно так: функция printk() — это просто функция ядра, выполняющая форматированный вывод сообщений. Однако, некоторые различия все же имеются.

Устойчивость функции printk()

Одно из проверенных и часто используемых свойств функции printk() — это ее устойчивость. Функцию printk() можно вызывать практически *в любое время* и *в любом месте* ядра. Её можно вызывать из контекста прерывания и из контекста процесса. Её можно вызывать во время удержания блокировки. Её можно вызывать одновременно на нескольких процессорах и она не требует при этом удерживать какие-нибудь блокировки.

Эта функция очень устойчива, и это очень важно, потому что полезность функции printk() базируется на том факте, что она всегда доступна и всегда работает.

Неустойчивость функции printk()

Слабое место у функции printk() в плане устойчивости все же существует. Её нельзя использовать до некоторого момента при загрузки ядра, пока консоль еще не инициализирована. Действительно, если нет консоли, то куда будут выводится сообщения?

Обычно это не проблема, если не нужно выполнять отладку кода, который выполняется на очень ранних стадиях процесса загрузки (например, функции setup_arch(), которая выполняет инициализацию специфичную для аппаратной платформы). Отладка такого рода— настоящая задача: отсутствие каких-либо способов вывода сообщений, а только проблема в полном составе.

В таких ситуациях тоже есть некоторые обнадеживающие моменты, но их не много. Настоящие хакеры, которые работают с аппаратурой на таком низком уровне, для связи с внешним миром используют аппаратное обеспечение соответствующей платформы, которое всегда работает (например, последовательный порт). Поверьте, что у большинства людей такая работа не вызовет радости. Для одних аппаратных платформ такое решение работает, для других платформ (включая платформу i386) существуют заплаты кода, которые тоже позволяют сэкономить время.

Одно из решений проблемы — вариант функции printk() , который может выводить информацию па консоль на очень ранних стадиях процесса загрузки - early_printk(). Поведение этой функции аналогично функции printk(), за исключением имени и возможности работать на очень ранних стадиях загрузки. Однако, такое решение не переносимо, потому что не для всех поддерживаемых аппаратных платформ этот метод работы реализован. Если же он есть, то может сослужить хорошую службу.

Кроме ситуаций, когда необходимо выводить на консоль информацию на очень ранних стадиях загрузки системы, можно положиться на функцию printk() , которая работает практически всегда.

Уровни вывода сообщений ядра

Главное отличие между функциями printk() и printf() — это возможность в первой указывать *уровень вывода сообщений ядра* (*loglevel).* Ядро использует уровень вывода сообщений для принятия решения о том, выводить сообщение на консоль или нет. Ядро выводит на консоль все сообщение с уровнями меньшими, или равными, соответствующему значению для консоли (console loglevel). Уровень вывода сообщений можно указывать следующим образом.

```
printk(KERN_WARNTNG "Это предупреждение!\n");
printk(KERN_DEBUG "Это отладочное сообщение!\n");
printk("Мыне указали значения loqlevel!\n" );
```

Строки KERN_WARNING и KERN_DEBUG определены через препроцессор в заголовочном файле <linux/kernel.h>. Эти макросы раскрываются в строки, соответстпенно "<4>" и "<7>", которые объединяются со строкой формата в самом начале сообщения, выводимого функцией printk(). После этого на основании уровня выпода сообщения и уровня вывода консоли (значение переменной console_loglevel) ядро принимает решение выводить информацию на консоль или нет. В табл. 18.1 приведен полный список возможных значений уровня вывода сообщений.

Таблица 18.1. Доступные значения уровня вывода сообщений ядра (loglevel)

Значение loglevel	Описание
KERN_EMERG	Аварийная ситуация
KERN_ALERT	Проблема, на которую требуется немедленно обратить внимание
KERN_CRIT	Критическаяситуация
KERN_ERR	Ошибка
KERN_WARNING	Предупреждение
KERN_NOTICE	Обычная ситуация, но на которую следует обратить внимание
KERN_INFO	Информационное сообщение
KERN_DEBUG	Отладочное сообщение — обычно избыточная информация

Если уровень вывода сообщений ядра не указан, то его значение по умолчанию равно DEFAULT_MESSAGE_LOGLEVEL, который в данный момент равен KERN_ WARNING. Так как это значение может измениться, то для своих сообщений необходимо всегда указывать уровень вывода.

Наиболее важный уровень вывода— KERN_EMERG определен как "<0>", а наименее важный — KERN_DEBUG, как "<7>". Например, после обработки препроцессором кода из предыдущего примера получается следующее.

```
printk("<4>Это предупреждение!\n");
printk("<7>Это отладочное сообщение!\n");
printk("<4>Мы не указали значения loglevel!\n");
```

Как вы будете использовать функцию printk() зависит только от вас. Конечно, обычные сообщения, которые должны быть видимы, должны иметь соответствующий уровень вывода. Отладочные сообщения, которые в большом количестве встраиваются в самые разные места кода с целью разобраться с проблемой — "допустим ошибка здесь", "пробуем", "работает" - могут иметь любой уровень вывода. Один вариант — оставить уровень при котором сообщения выводятся на консоль равным значению этого параметра по умолчанию, а уровень вывода ваших сообщений установить в значение KERN_CRIT, или что-то около этого. Можно поступить и наоборот— для отладочных сообщений установить уровень KERN_DEBUG и поднять уровень при котором сообщения выводятся на консоль. Каждый из вариантов имеет свои положительные и отрицательные стороны — вам решать.

Уровни вывода сообщений определены в файле <linux/kernel.h>.

Буфер сообщений ядра

Сообщения ядра хранятся в кольцевом буфере (log buffer) размером LOG_BUF_LEN. Этот размер можно изменять во время компиляции с помощью параметра CONFIG_ LOG_BUF_SHIFT. Для однопроцессорной машины это значение по умолчанию равно 16 Кбайт. Другими словами в ядре может хранится до 16 Кбайт системных сообщений. Если общий размер всех сообщений ядра достигает этого максимального значения и приходит новое сообщение, то оно переписывается поверх самого старого из хранящихся в буфере сообщений. Буфер сообщений ядра называется *кольцевым,* потому что запись и считывание сообщений выполняется по круговой схеме.

Использование кольцевого буфера предоставляет определенные преимущества. Так как одновременные операции чтения и записи в кольцевом буфере выполняются достаточно просто, то функцию printk() можно использовать даже из контекста прерывания. Более того, это позволяет просто организовать управление системными сообщениями. Если сообщений оказывается очень много, то новые сообщения просто затирают старые. Если возникает проблема, которая проявляется в генерации большого количества сообщений, то буфер сообщений просто начинает переписывать себя вместо того, чтобы бесконтрольно занимать память. Единственный недостаток кольцевого буфера — возможность потерять сообщения, что не такая уж и большая плата за ту устойчивость, которую такое решение предоставляет.

Демоны syslogd и klogd

В стандартной системе Linux для извлечения сообщений ядра из буфера используется специальный демон пространства пользователя klogd, который направляет эти сообщения в файл журнала системных сообщений. Для чтения системных сообщений программа klogd может считывать данные из файла /proc/kmsg, или использовать системный вызов syslog(). По умолчанию используется подход на

основе файловой системы /proc. Если сообщений нет, то демон klogd блокируется на операции чтения, пока не поступит новое сообщение. Когда приходит новое сообщение, демон возвращается к выполнению, считывает сообщения и обрабатывает их. По умолчанию сообщения отправляются демону syslogd.

Демон syslogd добавляет полученные сообщения в конец файла журнала, по умолчанию — /var/log/messages. Имя соответствующего файла можно настроить в конфигурационном файле /etc/syslog.conf.

Изменить уровень вывода сообщений на консоль (console loglevel) можно при старте демона klogd с помощью флага -c.

Замечание относительно функции printk() и разработки ядра

Когда впервые начинают разрабатывать код ядра, то скорее всего очень часто приходится заменять функцию printf() на функцию printk(). Это нормально, потому что нельзя не принимать во внимание многолетний опыт по написанию пользовательских программ и использовании функции printf(). Следует надеяться, что повторение таких ошибок не будет продолжаться долго, потому что повторяющиеся ошибки компоновщика начнут быстро надоедать.

Однажды вдруг окажется, что вы поймали себя на том, что начали использовать функцию printk() вместо функции printf() в пользовательских программах. Когда для вас этот день наконец наступит, то можно сказать, что вы стали настоящим хакером и специалистом по разработке кода ядра.

Сообщения Oops

Сообщения *oops* — обычный для ядра способ сообщить пользователю, что произошло что-то нехорошее. Так как ядро управляет всей системой, то оно не может само себя исправить, или завершить, как это возможно для программ пространства пользователя, когда они делают что-то не так. Вместо этого, ядро выводит сообщение oops. Такое сообщение включает вывод информации об ошибке на консоль, вывод дампа содержимого всех регистров и вывод обратной трассировки вызовов функций (back trace). Сбои в работе ядра трудно обработать, поэтому ядро должно "пролезть" через многие дыры, чтобы вывести сообщение oops и выполнить за собой все необходимые действия по очистке. Часто после выдачи сообщения oops ядро находится в несогласованном состоянии. Например, в момент возникновения ситуации, в которой выдается сообщение oops, ядро может находится в процессе обработки важных данных. В этот момент может удерживаться блокировка, или выполняться сеанс взаимодействия с оборудованием. Ядро должно аккуратно отойти от текущего состояния и попытаться восстановить контроль над системой. Во многих случаях это невозможно. Если ситуация, в которой выдается сообщение oops, возникает в контексте прерывания, то ядро не может продолжать работу и переходит в состояние паники. Состояние паники проявляется в полной остановке системы. Если oops возникает в холостой задаче (idle task, идентификатор pid равен нулю), или при выполнении процесса init (идентификатор pid равен единице), то ядро также переходит в состояние паники, потому что ядро не может продолжать выполнение

без этих важных процессов. Однако, если oops возникает при выполнении любого другого процесса, то ядро завершает этот процесс и продолжает работу.

Сообщение oops может выдаваться по многим причинам, включая недопустимый доступ к памяти (memory access violation) и выполнение недопустимой машинной команды. Как разработчику ядра, вам придется иметь дело с сообщениями oops и далее, несомненно, быть причиной их появления.

Ниже показано сообщение oops для машины аппаратной платформы PPC, которое возникло и обработчике таймера для сетевого интерфейсного адаптера tulip.

```
Oops: Exception in kernel mode, sig: 4
Unable to handle kernel NULL pointer dereference at virtual address 00000001

NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905dl0 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME: 1 IR/DR: 11
TASK = c0712530[0]  swapper Last syscall: 120
GPROO: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012AO 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace:
 [c013ab30] tulip_timer+0xl28/0xlc4
 [c0020744] run_timer_softirq+0xl0c/0xl64
 [c001b864] do_softirq+0x88/0xl04
 [c0007e80] timer_mterrupt+0x284/0x298
 [c00033c4] ret_from_except+0x0/0x34
 [c0007b84] default_idle+0x20/0x60
 [c0007bf8] cpu_idle+0x34/0x38
 [c0003ae8] rest_init+0x24/0x34
```

У пользователей ПК может вызвать удивление количество регистров процессора (32 - огромное число!). Сообщение oops для аппаратной платформы x86, которые возможно вам более знакомы, имеют несколько более простой вид. Тем не менее, важная информация идентична для всех аппаратных платформ: содержимое всех регистров и обратная трассировка.

Обратная трассировка показывает точную последовательность вызовов функций, которая привела к проблеме. В данном случае можно точно определить, что случилось: машина выполняла холостое задание - холостой цикл: вызов функции `cpu_idle()`, из которой циклически вызывается функция `default_idle()`. Поступило прерывание от системного таймера, в котором вызываются обработчики таймеров ядра. Среди них вызывается обработчик таймера — функция `tulip_timer()`, в которой выполнено разыменование указателя со значением NULL. Можно даже воспользоваться значением смещения (числа вроде *0x128/0x1c4,* которые указаны справа от имени функции) для точного нахождения команды, в которой возникла ошибка.

Содержимое регистров точно также полезно, хотя и используется не так часто. Вместе с дизассемблированным кодом функции содержимое регистров может помочь восстановить точную последовательность событий, которая привела к проблеме. Если значение в некотором регистре не соответствует ожидаемому, то это может пролить некоторый свет на корень проблемы. В данном случае можно проверить, какие регистры содержат значение NULL (все разряды нулевые) и определить, какая

из переменных функции содержит не то значение. В ситуациях, похожих на данную, скорее всего причина — конкуренция за ресурс (race) и скорее всего между таймером и другой частью сетевого адаптера. Отладка состояний конкуренции за ресурсы — всегда серьезная задача.

Утилита ksymoops

Только что рассмотренное сообщение oops имеет так называемый *декодированный* вид, потому что адреса памяти транслированы в имена функций, которые им соответствуют. Не декодированный вид предыдущего сообщения выглядит следующим образом.

```
NIP: C013A7F0 LR: C013A7F0 SP: C0685E00 REGS: c0905d10 TRAP: 0700
Not tainted
MSR: 00089037 EE: 1 PR: 0 FP: 0 ME 1 IR/DR: 11
TASK = c0712530[0] 'swapper' Last syscall: 120
GPR00: C013A7C0 C0295E00 C0231530 0000002F 00000001 C0380CB8 C0291B80 C02D0000
GPR08: 000012A0 00000000 00000000 C0292AA0 4020A088 00000000 00000000 00000000
GPR16: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
GPR24: 00000000 00000005 00000000 00001032 C3F7C000 00000032 FFFFFFFF C3F7C1C0
Call trace: [c013ab30] [c0020744] [c001b864] [c0007e80] [c00061c4]
 [c0007b84] [c0007bf8] [c0003ae8]
```

Адреса обратной трассировки должны быть переведены в символические имена функций. Это можно сделать с помощью команды ksymoops при наличии файла System.map, который сгенерирован во время компиляции данного ядра. Если используются загружаемые модули ядра, то необходима также информация о модулях. Утилита ksymoops пытается самостоятельно определить всю необходимую информацию, поэтому обычно ее можно просто вызывать следующим образом.

```
ksymoops saved_oops.txt
```

Программа выводит декодированную информацию сообщения oops. Если информация, которая используется по умолчанию, недоступна, или есть необходимость указать альтернативное положение соответствующих информационных файлов, то на такой случай программа принимает различные параметры. Страницы руководства по данной программе, которые необходимо прочитать перед использованием, содержат всю необходимую информацию.

Программа ksymoops включена в большинство поставок операционной системы Linux.

Функция kallsyms

К счастью, больше нет необходимости использовать программу ksymoops. Это очень полезно, потому что, хотя, у разработчиков обычно нет проблем с ее использованием, пользователи часто указывают неправильный файл System.map, или неправильно декодируют сообщение oops.

В разрабатываемой серии ядра 2.5 была введено новая возможность kallsyms, которая включается с помощью конфигурационного параметра CONFIG_KALLSYMS. Эта функция включает в исполняемый образ ядра информацию для отображения адресов памяти в соответствующие имена функций ядра, что дает возможность ядру

самостоятельно декодировать информацию обратной трассировки. Следовательно, декодирование сообщений oops больше не требует файла System.map, или утилиты kallsyms. Как недостаток такого подхода следует отметить некоторое увеличение объема памяти, используемой ядром, так как таблица перевода адресов памяти в имена функций загружается в постоянно отображаемую память ядра. На такое увеличение объемов используемой памяти стоит пойти, по крайней мере, на этапе разработки ядра.

Конфигурационные параметры отладки ядра

Существует несколько конфигурационных параметров, которые помогают в отладке и тестировании кода ядра и которые включаются во премя компиляции. Эти параметры доступны в пункте Kernel hacking меню редактора конфигурации ядра. Все эти параметры зависят от параметра CONFIG_DEBUG_KERNEL. Для разработки кода ядра следует включать только те параметры, которые необходимы.

Некоторые из этих параметров достаточно полезны, такие как отладка работы со слябовым распределителем памяти (slab layer debugging), отладка работы с верхней памятью (high memory debugging), отладка работы с отображаемым на память вводом-выводом (I/O mapping debugging), отладка работы со спин-блокировками (spinlock debugging) и проверка переполнения стека (stack overflow checking). Однако, один из самых полезных параметров — это *проверка перехода в состояние ожидания при захваченной спин-блокировке (sleep-inside-spinlock checking),* которая на самом деле выполняет значительно больше работы.

Отладка атомарных операций

Начиная с серии 2.5 в ядре появилась отличная инфраструктура для определения всех типов нарушения атомарности. Вспомните из главы 8, "Введение в синхронизацию выполнения кода ядра", что атомарность означает неделимое выполнение, то есть код выполняется без перерыва до завершения, или не завершается вообще. Код, который удерживает спин-блокировку, или выполняется при запрещенной преемптивности ядра, является атомарным. Во время атомарного выполнения нельзя переходить в состояние ожидания. Ожидание при удерживаемой спин-блокировке — один из вариантов взаимоблокировки.

Благодаря свойствам преемптивности, ядро имеет глобальный счетчик преемптивности. Ядро может быть настроено так, что, если выполняется переход в состояние ожидания, или даже выполняется код, который потенциально может переходить в состояние ожидания при выполнении атомарной операции, то ядро выводит предупреждающее сообщение и обратную трассировку. Потенциальные ошибки, которые детектируются таким образом, включают вызов функции schedule () при удерживаемой блокировке, выполнение блокирующего выделения памяти при удерживаемой блокировке, или переход в состояние ожидания при удерживаемой ссылке на данные, связанные с процессором. Эта отладочная инфраструктура может обнаружить очень много ошибок и ее очень рекомендуется использовать.

Следующие конфигурационные параметры позволяют полностью использовать данную возможность.

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_SPINLOCK_SLEEP=y
```

Генерация ошибок и выдача информации

Существует несколько подпрограмм ядра, которые позволяют легко сигнализировать о наличии дефектов кода, обеспечивать объявления об ошибках и выводить необходимую информацию. Две наиболее часто используемые — это BUG() и BUG_ON(). При вызове эти функции создают ситуацию oops, которая проявляется в выводе обратной трассировки стека ядра и сообщения об ошибке. Каким образом эти вызовы генерируют ситуацию oops зависит от аппаратной платформы. Для большинства аппаратных платформ вызовы BUG() и BUG_ON() определяются как некоторая недопустимая машинная команда, которая приводит к выводу желаемого сообщения oops.

Обычно эти вызовы используются в качестве объявления о наличие ошибки (assertion), чтобы сигнализировать о ситуации, которая не должна произойти.

```
if (bad_thing)
        BUG();
```

Или даже так.

```
BUG_ON(bad_thing);
```

О более критичной ошибке можно сигнализировать с помощью функции panic (). Функция panic () печатает сообщение об ошибке и останавливает ядро. Ясно, что эту функцию следует использовать только в самой плохой ситуации.

```
if (terrible_thing)
        panic("foo is %ld!\n", foo);
```

Иногда необходимо просто вывести на консоль трассировку стека, чтобы облегчить отладку. В этих случаях используется функция dump_stack(). Эта функция отображает на консоль содержимое регистров процессора и обратную трассировку вызовов функций.

```
if (!debug_check) {
        printk(KERN_DEBUG "выдать некоторую информацию...\n");
        dump_stack();
}
```

Магическая клавиша SysRq

Использование магической клавиши SysRq, которую можно активизировать с помощью конфигурационного параметра CONFIG_MAGIC_SYSRQ на этапе компиляции, часто позволяет значительно облегчить жизнь. Клавиша SysRq является стандартной на многих клавиатурах. Для аппаратных платформ i386 и PPC ей соответствует комбинация клавиш ALT-PrintScreen. Если указанный конфигурационный параметр активизирован, то специальные комбинации клавиш позволяют взаимодейство-

вать с ядром независимо от того, чем ядро в данный момент нанимается. Это в свою очередь позволяет выполнять некоторые полезные операции даже на неработоспособной системе.

В дополнение к конфигурационному параметру существует вызов sysctl для включения и выключения этого свойства.

```
echo 1 > /proc/sys/kernel/sysrq
```

Список возможных комбинаций клавиш можно получить с консоли путем нажатия комбинации клавиш SysRq-h. Комбинация клавиш SysRq-s выполняет синхронизацию не сохраненных буферов файловых систем на диск, комбинация SysRq-u размонтирует все файлопые системы, а SysRq-b — перегружает машину. Последовательное использование этих комбинаций клавиш позволяет более безопасно перегрузить машину, которая зависла, чем простое нажатие кнопки reset.

Если машина заблокирована очень сильно, то она может не отвечать на магические комбинации клавиш SysRq, или соответствующая операция не будет выполнена. Если же повезет, то эти комбинации клавиш смогут помочь при отладке, а также сохранить данные. В табл. 18.2 приведен список поддерживаемых команд SysRq.

Таблица 18.2. Список поддерживаемых команд SysRq

Команда	Описание
SysRq-b	Перегрузить машину (reboot)
SysRq-e	Послать сигнал SIGTERM всем процессам, кроме процесса init
SysRq-h	Отобразить на консоли помощь по использованию комбинаций клавиш SysRq
SysRq-i	Послать сигнал SIGKILL всем процессам, кроме процесса init
SysRq-k	Клавиша безопасного доступа: завершить все процессы, связанные с текущей консолью
SysRq-l	Послать сигнал SIGKILL всем процессам, включая процесс init
SysRq-m	Отобразить на консоли дамп информации по использованию памяти
SysRq-o	Завершить работу машины (shutdown)
SysRq-p	Отобразить на консоли дамп регистров памяти
SysRq-r	Отключить прямой режим работы клавиатуры (raw mode)
SysRq-s	Синхронизировать данные смонтированных файловых систем с дисковыми устройствами
SysRq-t	Отобразить на консоли дамп информации о заданиях
SysRq-u	Размонтировать все смонтированные файловые системы

В файле Documentation/sysrq.txt, который находится в каталоге исходных кодов ядра, приводится более полное описание. Реализация поддержки магической комбинации клавиш находится в файле drivers/char/sysrq.c. Магические комбинации клавиш SysRq — жизненно необходимый инструмент, который помогает в отладке и сохранении "гибнущей" системы, так как предоставляет большие возможности для любого пользователя при работе с консолью. Тем не менее необходимо соблюдать осторожность при его использовании на критичных машинах. Если же машина используется для разработок, то польза от этих команд огромная.

Сага об отладчике ядра

Многие разработчики ядра давно высказываются о необходимости встроенного в ядро отладчика. К сожалению, Линус не желает видеть отладчик ядра в своем дереве исходного кода, Он уверен, что использование программ-отладчиков приводит к плохому исправлению ошибок неправильно информированными разработчиками. Никто не может поспорить с его логикой — исправления ошибок, построенные на основании хорошего понимания кода скорее всего будут верными. Тем не менее большинство разработчиков ядра все же нуждаются в официальном отладчике, встроенном в ядро. Поскольку такая возможность навряд ли появится в ближайшее время, то взамен было разработано несколько заплат, которые добавляют поддержку отладчика в стандартном ядре. Не смотря на то, что это внешние и неофициальные заплаты, они являются мощными инструментами с высокой функциональностью. Перед тем, как обращаться к этим решениям, посмотрим, на сколько нам может помочь стандартный отладчик ОС Linux - gdb.

Использование отладчика gdb

Для того, чтобы мельком заглянуть внутрь работающего ядра можно использовать стандартный отладчик GNU. Запуск отладчика для работы с ядром почти ни чем не отличается от отладки выполняющегося процесса.

```
gdb vmlinux /proc/kcore
```

Файл vmlinux — это декомпрессированный исполняемый образ ядра, который хранится в корне каталога исходных кодов, где выполнялась сборка выполняющегося ядра. Сжатые файлы zlmage, или bzlmage использовать нельзя.

Опциональный параметр /proc/kcore исполняет роль файла core, чтобы позволить отладчику читать из памяти выполняющегося ядра. Чтобы иметь возможность читать этот файл, необходимо иметь права пользователя root.

Можно пользоваться практически всеми командами программы gdb для чтения информации. Например, чтобы напечатать значение переменной можно восполбзо - ваться командой.

```
p global_variable
```

Для того, чтобы дизассемблировать код функции можно выполнить следующую команду.

```
disassemble function
```

Если ядро было скомпилировано с указанием флага -g (необходимо добавить -g к значению переменной CFLAGS в файле Makefile ядра), то отладчик gdb сможет выдавать больше информации. Например, можно выводить дампы структур данных и разыменовывать указатели. При этом также получается ядро значительно большего размера, поэтому для обычной работы не следует компилировать ядро с отладочной информацией.

К сожалению, на этом заканчиваются возможности использования отладчика gdb. С его помощью никак нельзя изменять данные ядра. Нет возможности пошагово выполнять код ядра, или устанавливать точки остановки (breakpoint). Невозможность изменять структуры данных ядра — это большой недостаток. Хотя очень полезно

иметь возможность дизассемблировать код функций, еще более полезной была бы возможность изменять структуры данных.

Отладчик kgdb

Отладчик kgdb —это заплата ядра, которая позволяет с помощью отладчика gdb отлаживать ядро по линии последовательной передачи. Для этого требуется два компьютера. На перпом выполняется ядро с заплатой kgdb. Второй компьютер используется для отладки ядра по линии последовательной передачи (нуль-модемный кабель, соединяющий две машины) с помощью gdb. Благодаря отладчику kgdb полностью доступен весь набор функций gdb: чтение и запись любых переменных, установка точек остановки, установка точек слежения (watch points), пошаговое исполнение и др.. Специальные версии kgdb даже позволяют вызывать функции.

Установка kgdb и линии последовательной передачи несколько сложная процедура, но если ее выполнить, то отладка ядра значительно упрощается. Заплата ядра также устанавливает большое количество документации в каталог Documentation/, ее следует прочитать.

Несколько человек выполняют поддержку заплаты kgdb для различных аппаратных платформ и версий ядра. Поиск в Интернет — наилучший способ найти необходимую заплату для заданного ядра.

Отладчик kdb

Альтернативой kgdb является отладчик kdb. В отличие от kgdb отладчик kdb — не удаленный отладчик. Отладчик kdb — это заплата, которая сильно модифицирует ядро и позволяет выполнять отладку прямо на той же машине, где выполняется ядро. Кроме всего прочего поддерживается возможность изменения переменных, установки точек остановки и пошаговое выполнение. Выполнять отладку просто — необходимо нажать на консоли клавишу break. При выводе сообщения oops переход в отладчик выполняется автоматически. Более подробная документация доступна в каталоге Documentation/kdb после применения заплаты. Заплата kdb доступна в Интернет по адресу http://oss.sgi.com/.

Исследование и тестирование системы

По мере того, как вы будете накапливать опыт в отладке ядра, у вас будет появляться все больше маленьких хитростей, которые помогают в исследовании и тестировании ядра для получения ответов на интересующие вопросы. Так как отладка ядра требует больших усилий, то каждый маленький совет, или хитрость может оказаться полезным. Рассмотрим несколько таких хитростей.

Использование идентификатора UID в качестве условия

Если разрабатываемый код связан с контекстом процесса, то иногда появляется возможность выполнить альтернативную реализацию не "ломая" существующий код. Это важно, если необходимо переписать важный системный вызов и при этом необходима полностью функционирующая система, на которой этот вызов нужно отладить. Например, допустим, что нужно переписать алгоритм работы системного вы-

зова fork(), который бы использовал некоторые новые возможности, которые уже существуют в ядре. Если сразу не получится все сделать так как надо, то будет очень тяжело отлаживать ядро, так как неработающий системный вызов fork () скорее всего приведет к неработоспособности системы. Но как и всегда, есть надежда.

Часто безопасным будет сохранить старый алгоритм, а новую реализацию выполнить в другом месте. Этого можно достичь используя идентификатор пользователя (UID) в качестве условия того, какой алгоритм использовать.

```
if (current->uid != 7777) {
        /* старый алгоритм .. */
} else {
        /* новый алгоритм .. */
}
```

lice пользователи, кроме того, у которого идентификатор UID равен 7777 будут использовать старый алгоритм. Для тестирования нового алгоритма можно создать нового пользователя с идентификатором 7777. Это позволяет более просто оттестировать критические участки кода, связанные с выполнением процессов.

Использование условных переменных

Если код, который необходимо протестировать, выполняется не в контексте процесса, или необходим более глобальный метод для контроля новых функций, то можно использовать условные переменные. Этот подход даже более простой, чем использование идентификатора пользователя. Необходимо просто создать глобальную переменную и использовать ее в качестве условия выполнения того, или другого участка кода. Если значение переменной равно нулю, то следует выполнить один участок кода. Если переменная не равна нулю, то выполняется другой участок. Значение переменной может быть установлено с помощью отладчика, или специального экспортируемого интерфейса.

Использование статистики

Иногда необходимо получить представление о том, насколько часто происходит некоторое событие. Иногда требуется сравнить несколько событий и вычислить характеристики для их сравнения. Это очень легко сделать путем введения статистки и механизма для экспортирования соответствующих параметров.

Например, допустим, что необходимо выяснить на сколько часто происходит событие *foo* и событие *bar*. В файле исходного кода, в идеале там, где соответствующие события возникают, вводится две глобальные переменные.

```
unsigned long foo_stat = 0;
unsigned long bar_stat = 0;
```

Как только наступает интересующее событие, значение соответствующей переменной увеличивается на единицу. Эти переменные могут быть экспортированы как угодно. Например, можно создать интерфейс к ним через файловую систему /proc, или написать свой системный вызов. Наиболее просто прочитать их значение с помощью отладчика.

Следует обратить внимание, что такой подход принципиально не безопасен на SMP машине. В идеале необходимо использовать атомарные переменные. Однако,

для временной статистики, которая необходима только для отладки, никакой защиты обычно не требуется.

Ограничение частоты следования событий при отладке

Часто необходимо встроить в код отладочные проверки (с соответствующими функциями вывода информации), чтобы визуально производить мониторинг проблемы. Однако, в ядре некоторые функции вызываются по много раз в секунду. Если в такую функцию будет встроен вызов функции `printk()`, то системная консоль будет перегружена выводом отладочных сообщений и ее будет невозможно использовать.

Для предотвращения такой проблемы существует два сравнительно простых приема. Первый — ограничение частоты следования событий — очень полезен, когда необходимо наблюдать, как развивается событие, но частота возникновения события очень большая. Чтобы ограничить поток отладочных сообщений, эти сообщения выводятся только раз в несколько секунд, как это показано в следующем примере.

```
static unsigned long prev_jiffy = jiffies; /* ограничение частоты */

if (time_after(jiffies, prev_jiffy + 2*HZ)) {
        prev_jiffy = jiffies;
        printk(KERN_ERR "blah blah blah\n");
}
```

В этом примере отладочные сообщения выводятся не чаще, чем один раз в две секунды. Это предотвращает перегрузку консоли сообщениями и системой можно нормально пользоваться. Частота вывода может быть большей, или меньшей, в зависимости от требопаний.

Вторая ситуация имеет место, когда необходимо замечать любые появления события. В отличие от предыдущего примера нет необходимости выполнять мониторинг развития событий. А только получить сообщение о том, что что-то произошло. Вероятно это уведомление необходимо получить один, или два раза. Проблема возникает в том случае, если проверка, которая после того, как сработала один раз, начинает срабатывать постоянно. Решением в данном случае будет не ограничение частоты, а ограничение общего количества повторений.

```
static unsigned long limit = 0;

if (limit < 5) {
        limit++;
        printk(KERN_ERR "blah blah blah\n");
}                                          .  ,
```

В этом примере количество отладочных сообщений ограничено числом пять. После пяти сообщений условие всегда будет ложно.

В обоих примерах переменные должны быть статическими (`static`) и локальными по отношению к той функции, где используются. Это позволяет использовать одинаковые имена переменных в разных функциях.

Ни один из этих примеров не рассчитан на SMP, или преемптивность, хотя очень легко перейти к атомарным операциям и сделать их безопасными для использования и в этих случаях. Однако, честно говоря, это всего лишь отладочный код, поэтому зачем нужны лишние проблемы?

Нахождение исполняемых образов с изменениями приводящими к ошибкам

Обычно полезно знать, в какой версии исходных кодов ядра появился дефект. Если известно, что дефект появился в версии 2.4.18, но его не было в версии 2.4.17, то сразу появляется ясная картина изменений, которые привели к появлению ошибки. Исправление ошибки сводится к обратным изменениям, или другим исправлениям измененного кода.

Однако, чаще оказывается неизвестным в какой версии появился дефект. Известно, что проблема проявляется в *текущей* версии ядра, и кажется, что она *всегда* была в текущей версии. Хотя это и требует некоторой исследовательской работы, но приложив небольшие усилия можно найти изменения, которые привели к ошибкам. Если известны эти изменения, то до исправления ошибки уже недалеко.

Для того, чтобы начать, необходима четко повторяемая проблема. Желательно, чтобы проблема проявлялась сразу же после загрузки системы. Далее необходимо гарантированно работающее ядро. Вероятно, это ядро уже известно. Например, может оказаться, что пару месяцев назад ядро работало нормально, поэтому стоит взять ядро того времени. Если это не помогает, то можно воспользоваться еще более старой версией. Такой поиск ядра без дефекта должен быть не сложным, если, конечно, дефект не существовал всегда.

Далее, необходимо ядро, в котором гарантированно есть дефект. Для облегчения поиска следует воспользоваться наиболее ранней версией ядра, в которой есть дефект. После этого начинается поиск исполняемого образа, в котором появилась ошибка. Рассмотрим пример. Допустим, что последнее ядро, в котором не было ошибки— 2.4.11, а последнее с ошибкой— 2.4.20. Начинаем с версии ядра, которая находится посредине— 2.4.15. Проверяем версию 2.4.15 на наличие ошибки. Если версия 2.4.15 работает, значит ошибка появилась в более поздней версии. Следует попробовать версию между 2.4.15 и 2.4.20, скажем версию 2.4.17. С другой стороны, если версия 2.4.15 не работает, то тогда ясно, что ошибка появилась в более ранней версии, и следует попробовать версию, скажем 2-4.13. И так далее.

В конце концов проблема сужается до двух ядер — одно с дефектом, а другое — без. В таком случае есть ясная картина изменений, которые привели к проблеме.

Такой подход избавляет от необходимости проверять ядра всех версий!

Если ничто не помогает — обратитесь к сообществу

Возможно, вы уже испробовали все, что знали. Вы просидели за клавиатурой несчетное количество часов, и даже дней, а решение все еще не найдено. Если проблема в основном ядре Linux, то всегда можно обратиться за помощью к людям из сообщества разработчиков ядра.

Короткое, но достаточно детальное описание проблемы вместе с вашими находками, посланное в список рассылки разработчиков ядра по электронной почте, может помочь отыскать решение. В конце концов, дефектов никто не любит.

Глава 20, "Заплаты, разработка и сообщество" специально посвящена сообществу разработчиков ядра и его основному форуму — списку рассылки разработчиков ядра Linux (Linux Kernel Mail List, LKML).

19

Переносимость

Linux — переносимая операционная система, которая поддерживает большое количество различных компьютерных аппаратных платформ. *Переносимость*— это свойство, которое указывает, насколько легко можно перенести код, который выполнялся на одной аппаратной платформе, для выполнения на другой аппаратной платформе (если вообще это возможно). Известно, что ОС Linux является переносимой операционной системой, поскольку ее уже *перенесли* (портировали) па большое количество различных аппаратных платформ. Тем не менее переносимость не возникает сама по себе, для выполнения она требует большого количества проектных решений. Сегодня процесс перенесения ОС Linux на другую аппаратную платформу достаточно прост (в относительном смысле, конечно). В этой главе рассказывается о том, как писать переносимый код, — вопрос, о котором всегда необходимо помнить при написании нового кода ядра или драйверов устройств.

Некоторые операционные системы специально разрабатываются с учетом требований переносимости как главного свойства. По возможности минимальное количество кода выполняется зависимым от аппаратуры. Разработка на языке ассемблера сводится к минимуму, а интерфейсы и свойства выполняются принципиально общими и абстрактными, чтобы иметь возможность работать на различных аппаратных платформах. Очевидным преимуществом в этом случае является легкость поддержки новой аппаратной платформы. В некоторых случаях простые операционные системы с высокой переносимостью могут быть нормированы на новую аппаратную платформу только путем изменения нескольких сотен строк специфичного кода. Недостаток такого подхода состоит в том, что не используются специфические свойства аппаратной платформы и код не может быть в ручную оптимизирован под конкретную машину. Переносимость ставится выше оптимальности. Примером операционных систем с высокой переносимостью могут быть Minix, OpenBSD и многие исследовательские системы.

С противоположной стороны можно поставить операционные системы, в которых оптимизация кода выполняется в ущерб переносимости. Код, по возможности, пишется на языке ассемблера или специфические свойства аппаратных платформ используются каким-либо другим образом. Свойства ядра разрабатываются на основании свойств аппаратной платформы. В этом случае перенос операционной системы на другую аппаратную платформу сводится к написанию ядра с нуля. Оптимальность выполняется в ущерб переносимости. Примером таких систем могут быть DOS и Windows 9x. Сегодня таким системам нет необходимости иметь более оптимальный код, чем переносимым операционным системам, однако они предоставляют возможность в максимальной степени использовать ручную оптимизацию кода.

Операционная система Linux в плане переносимости занимает промежуточное положение. Насколько это целесообразно из практических соображений, интерфейсы и код сохраняются независимыми от аппаратной платформы и пишутся на языке С. Однако функции ядра, которые критичны к производительности, выполняются зависимыми от аппаратной платформы. Например, низкоуровневый код и код, который должен выполняться очень быстро, разрабатывается зависимым от аппаратной платформы и обычно на языке ассемблера. Такой подход позволяет сохранить переносимость ОС Linux и при этом воспользоваться оптимизациями.

В случаях, когда переносимость становится помехой производительности, производительность обычно всегда побеждает. В остальных случая сохраняется переносимость кода.

Обычно экспортируемые интерфейсы ядра независимы от аппаратной платформы. Если различные части одной подпрограммы должны быть разными для разных аппаратных платформ (из соображений производительности или по необходимости), то код выполняется в виде нескольких функций, которые вызываются, в нужных местах. Для каждой поддерживаемой аппаратной платформы реализуются свои функции, которые затем компонуются в общий исполняемый образ ядра.

Хороший пример — планртровщик. Большая часть планировщика-написана независимым от аппаратной платформы образом на языке С. Реализация находится в файле `kernel/sched.c`. Некоторые из функций планировщика, такие как переключение состояния процессора или переключение адресного пространства, очень сильно зависят от аппаратной платформы. Следовательно, функция `context_switch ()`, которая переключает выполнение от одного процесса к другому и написана на языке С, вызывает функции `switch_to ()` и Kwitch_ram () для переключения состояния процессора и переключения адресного пространства соответственно.

Код функций `switch_to ()` и `switch_mm ()` выполнен отдельно для каждой аппаратной платформы, которую поддерживает операционная система Linux. Когда операционная система Linux портируется на новую аппаратную платформу, то для новой аппаратной платформы просто необходимо реализовать эти функции.

Файлы, которые относятся к определенной аппаратной платформе, находятся в каталоге arch/<аппаратная платформа>/ и include/asm-<аппаратная платформа>/, где <апппаратная платформа> — это короткое имя, которое представляет аппаратную платформу, поддерживаемую ядром Linux. Например, аппаратной платформе Intel x8б присвоено имя i386. Для этого типа машин файлы находятся в каталогах `arch/i386` и `include/asm-i386`. Ядра серии 2.6 поддерживают следующие аппаратные платформы: `alpha`, `arm`, `cris`, `h8300`, `i38`, `ia64`, `m68k`, `m68knommu`, `mips`, `mips64`, `parisc`, `ppc`, `ppc64`, `s390`, `sh`, `spare`, `sparc64`, `um`, `v850` и `x8б-б4`. Более полное описание приведено в табл. 19.1.

История переносимости Linux

Когда Линус Торвальдс впервые выпустил операционную систему Linux в ничего не подозревающий мир, эта ОС работала только на аппаратной платформе Intel i386. Хотя данная операционная система и была достаточно хорошо обобщена и хорошо написана, переносимость для нее не была основным требованием. Однажды Линус даже говорил, что операционная система Linux не будет работать ни на какой аппаратной платформе, кроме i386! Тем не менее в 1993 году началась работа по пор-

тиропаиию ОС Linux на машины Digital Alpha. Аппаратная платформа Digital Alpha была повой высокопроизводительной RISC-платформой с поддержкой 64-разрядной адресации памяти. Она очень сильно отличалась от аппаратной платформы i386, о которой говорил Линус. Тем не менее, первоначальный перенос на аппаратную платформу Alpha занял около года, и аппаратная платформа Alpha стала первой официально поддерживаемой аппаратной платформой после x8б. Это портировапие было, наверное, самым сложным, потому что — первым. Вместо простого переписывания ядра для поддержки новой аппаратной платформы, части ядра были переписаны с целью введения переносимости[1]. Хотя это и привело к выполнению большого количества работы, в результате получился более ясный для понимания код, и в будущем перенос стало выполнять более просто.

Первые выпуски ОС Linux поддерживали только платформу i386, а серия ядер 1.2 уже поддерживала Digital Alpha, Intel x86, MIPS и SPARC, хотя такая поддержка была отчасти экспериментальной.

С выпуском ядра версии 2.0 была добавлена официальная поддержка платформ Motorola 68k и PowerPC. В дополнение к этому поддержка всех аппаратных платформ, которые ранее поддерживались ядрами серии 1.2, стала официальной и стабильной.

В серию ядер 2.2 была введена поддержка еще большего количества аппаратных платформ: добавлены ARM, IBM S/390 и UltraSPARC. Через несколько лет в серии ядер 2.4 количество поддерживаемых аппаратных платформ было почти удвоено, и их количество стало равным 15. Была добавлена поддержка платформ CRIS, IA-64, 64-разрядная MIPS, HP PA-RISC, 64-разрядная IBM S/390 и Hitachi SH.

В серии 2.6 количество поддерживаемых аппаратных платформ было доведено до 20 за счет добавления платформ Motorola 68k бел устройства MMU, H8/300, IBM POWER, v850, x86-64 и версии ядра, которое работает на виртуальной машине под ОС Linux - Usermode Linux. Поддержка 64-разрядной s390 была объединена с 32-разрядной платформой s390, чтобы избежать дублирования.

Необходимо заметить, что каждая из этих аппаратных платформ поддерживает различные типы машин и микросхем. Некоторые из поддерживаемых аппаратных платформ, такие как ARM и PowerPC, поддерживают очень большое количество типов микросхем и машин. Поэтому, хотя ОС Linux и работает на 20 аппаратных платформах, она работает на гораздо большем количестве типов компьютеров!

Размер машинного слова и типы данных

Машинное слово (word) — это количество данных, которые процессор может обработать за одну операцию. Здесь можно применить аналогию документа, состоящего из *символов (character,* 8 бит) и *страниц* (много слов). Слово— это некоторое количество битов, как правило 16, 32 или 64. Когда говорят о "n-битовой" машине, то чаще всего имеют в виду размер машинного *слова.* Например, когда говорят, что процессор Intel Pentium — это 32-разрядный процессор, то обычно имеют в виду размер машинного слова, равный 32 бит, или 4 байт.

[1] Это нормальная ситуация при разработке ядра. Если что-либо должно быть сделано, то это должно быть сделано хорошо! Разработчики ядра неохотно переписывают большие участки кода даже во имя совершенства.

Размер процессорных регистров общего назначения равен размеру машинного слова этого процессора. Обычно разрядность остальных компонентов этой же аппаратной платформы в точности равна размеру машинного слова. Кроме того, по крайней мере для аппаратных платформ, которые поддерживаются ОС Linux, размер адресного пространства соответствует размеру машинного слова[2]. Следовательно, размер указателя равен размеру машинного слова. В дополнение к этому, размер типа long языка С также равен размеру машинного слова. Например, для аппаратной платформы Alpha размер машинного слова равен 64 бит. Следовательно, регистры, указатели и тип long имеют размер 64 бит. Тип int для этой платформы имеет размер 32 бит. Машины платформы Alpha могут обработать 64 бит— одно слово с помощью одной операции.

Слова, двойные слова и путаница

Для некоторых операционных систем и процессоров стандартную порцию данных не называют машинным *словом.* Вместо этого, *словом* называется некоторая фиксированная порция данных, название которой выбрано случайным образом или имеет исторические корни. Например, в некоторых системах данные могут разбиваться на байты (byte — 8 бит), слова (word — 16 бит), двойные слова (double word— 32 бит) и четверные слова (quad word— 64 бит), несмотря на то что на самом деле система является 32-разрядной. В этой книге и вообще в контексте операционной системы Linux под машинным словом понимают стандартную порцию данных процессора, как обсуждалось ранее.

Для каждой аппаратной платформы, поддерживаемой операционной системой Linux, в файле <asm/types.h> определяется константа BITTS_PER_LONG, которая равна размеру типа long языка С и совпадает с размером машинного слова системы. Полный список всех поддерживаемых аппаратных платформ и их размеры машинного слова приведены в табл. 19.1.

Стандарт языка С явно указывает, что размер памяти, которую занимают переменные стандартных типов данных, зависит от аппаратной реализации[3], при этом также определяется минимально возможный размер типа. Неопределенность размеров стандартных типов языка С для различных аппаратных платформ имеет свои положительные и отрицательные стороны. К плюсам можно отнести то, что для стандартных типов языка С можно пользоваться преимуществами, связанными с размером машинного слова, а также отсутствие необходимости явного указания размера. Для ОС Linux размер типа long гарантированно равен размеру машинного слова. Это не совсем соответствует стандарту ANSI С, однако является стандартной практикой в ОС Linux. Как недостаток можно отметить, что при разработке кода нельзя рассчитывать на то, что данные определенного типа занимают в памяти определенный размер. Более того, нельзя гарантировать, что переменные типа int занимают столько же памяти, сколько и переменные типа long[4].

[2] Размерадресуемой памяти может быть меньше максимального значения машинного слова. Например, для 64-разрядных аппаратных платформ размер указателя ранен 64 бит, однако только 48 бит можно использовать для адресации. В дополнение к этому, общее количество физической памяти может быть больше максимального значения машинного слова, как, например, это имеет место при наличии расширения Intel PAE.

[3] За исключением размера типа char, который всегда равен 8 бит.

[4] Насамом деле, для 64-разрядных аппаратных платформ, которые поддерживаются ОС Linux, размеры типов int и long не совпадают. Размер типа int равен 32 бит, а размер типа long — 64 бит. Для хорошо знакомых 32-paphядных аппаратных платформ оба типа данных имеют размер 32 бит.

Таблица 19.1. Поддерживаемые аппаратные платформы

Аппаратная платформа	Описание	Размер машинного слова
alpha	Digital Alpha	64 бит
arm	ARM и StrongARM	32 бит
cris	CRIS	32 бит
h8300	H8/300	32 бит
I386	Intel x86	32 бит
ia64	IA-64	64 бит
m68k	Motorola 68k	32 бит
m86knommu	m68k без устройства MMU	32 бит
mips	MIPS	32 бит
mips64	64-разрядная MIPS	64 бит
parisc	HP PA-RISC	32 бит, или 64 бит
ppc	PowerPC	32 бит
ppc64	POWER	64 бит
s390	IBM S/390	32 бит, или 64 бит
sh	Hitachi SH	32 бит
spare	SPARC	32 бит
sparc64	UltraSPARC	64 бит
um	Usermode Linux	32 бит, или 64 бит
v850	v850	32 бит
x8_ 64	X86-64	64 бит

Ситуация еще более запутывается тем, что одни и те же типы данных в пространстве пользователя и в пространстве ядра не обязательно должны соответствовать друг другу. Аппаратная платформа sparc64 предоставляет 32-разрядное пространство пользователя, а поэтому указатели, типы `int` и `long` имеют размер 32 бит. Однако в пространстве ядра для аппаратной платформы размер типа `int` равен 32 бит, а размер указателей и типа `long` равен 64 бит. Тем не менее такая ситуация не является обычной.

Всегда необходимо помнить о следующем.

- Как и требует стандарт языка C, размер типа `char` всегда равен 8 бит (1 байт),
- Нет никакой гарантии, что размер типа `int` для всех поддерживаемых аппаратных платформ будет равен 32 бит, хотя сейчас для всех платформ он равен именно этому числу.
- То же касается и типа `short`, который для всех поддерживаемых аппаратных платформ сейчас равен 16 бит.
- Никогда нельзя надеяться, что тип `long` или указатель имеет некоторый заданный размер. Этот размер для поддерживаемых аппаратных платформ может быть равен 32, или 64 бит.

- Так как размер типа long разный для различных аппаратных платформ, никогда нельзя предполагать, что sizeof (int.) == sizeof (long).
- Точно так же нельзя предполагать, что размер типа int и размер указателя совпадают.

Скрытые типы данных

Скрытые (opaque) типы данных — это те типы, для которых не раскрывается их внутренняя структура, или формат. Они похожи на *черный ящик,* насколько это можно реализовать в языке программирования С. В этом языке программирования нет какой-либо особенной поддержки для этих типов. Вместо этого, разработчики определяют новый тип данных через оператор typedef, называют его скрытым и надеются на то, что никто не будет преобразовывать этот тип в стандартный тип данных языка С. Любые использования данных этих типов возможны только через специальные интерфейсы, которые также создаются разработчиком. Примером может быть тип данных pid_t, в котором хранится информация об идентификаторе процесса. Размер этого типа данных не раскрывается, хотя каждый может смошенничать, использовать размер по максимуму и работать с этим типом, как с типом int. Если нигде явно не используется размер скрытого типа данных, то размер этого типа может быть изменен, и это не вызовет никаких проблем. На самом деле так уже однажды случилось: в старых Unix-подобных операционных системах тип pid_t был определен как ahort.

Еще один пример скрытого типа данных — это тип atomic_t. Как уже обсуждалось в главе 9, "Средства синхронизации в ядре", этот тип содержит данные целочисленного типа, с которыми можно выполнять атомарные операции. Хотя этот тип и соответствует типу int, использование скрытого типа данных позволяет гарантировать, что данные этого типа будут использоваться только в специальных функциях, которые выполняют атомарные операции. Скрытые типы позволяют скрыть размер типа данных, который не всегда равен полным 32 разрядам, как в случае платформы SPARC.

Другие примеры скрытых типов данных в ядре — это dev_t, gid _t и uid_t. При работе со скрытыми типами данных необходимо помнить о следующем.

- Нельзя предполагать, что данные скрытого типа имеют некоторый определенный размер в памяти.
- Нельзя преобразовывать скрытый тип обратно в стандартный тип данных.

Разрабатывать код необходимо с учетом того, что размер и внутреннее представление скрытого типа данных могут изменяться.

Специальные типы данных

Некоторые данные в ядре, кроме того, что представляются с помощью скрытых типов, требуют еще и специальных типов данных. Два примера — счетчик импульсов системного таймера jiffies и параметр flags, используемый для обработки прерываний. Для хранения этих данных всегда должен использоваться тип unsigned long.

При хранении и использовании специфических данных всегда необходимо обращать особенное внимание на тот тип данных, который представляет эти данные, и использовать именно его. Часто встречающейся ошибкой является использование другого типа, например типа unsigned int. Хотя для 32-разрядных аппаратных платформ это не приведет к проблемам, на 64-разрядных системах возникнут проблемы.

Типы с явным указанием размера

Часто при программировании необходимы типы данных заданного размера. Обычно это необходимо для удовлетворения некоторых внешних требований, связанных с аппаратным обеспечением, сетью или бинарной совместимостью. Например, звуковой адаптер может иметь 32-разрядный регистр, пакет сетевого протокола — 16-разрядное поле данных, а исполняемый файл - 8 битовый идентификатор cookie. В этих случаях тип, который представляет данные, должен иметь *точно* заданный размер.

В ядре типы данных явно заданного размера определены в файле <asm/types.h>, который включается из файла <linux/types.h>. В табл. 19.2 приведен полный список таких типов данных.

Таблица 19.2. Типы данных явно заданного размера

Тип	Описание
s8	байт со знаком
u8	байт без знака
s16	16-разрядное целое число со знаком
u16	16-разрядное целое число без знака
s32	32-разрядное целое число со знаком
u32	32-разрядное целое число без знака
s64	64-разрядное целое число со знаком
u64	64-разрядное целое число без знака

Варианты со знаком используются редко.

Эти типы данных, с явно заданным размером, просто определены с помощью оператора typedef через стандартные типы данных языка С. Для 64-разрядной машины они могут быть определены следующим образом.

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
typedef unsigned int u32;
typedef signed long s64;
typedef unsigned long u64;
```

Для 32-разрядной машины их можно определить, как показано ниже.

```
typedef signed char s8;
typedef unsigned char u8;
typedef signed short s16;
typedef unsigned short u16;
typedef signed int s32;
typedef unsigned int u32;
typedef signed long long s64;
typedef unsigned long long u64;
```

Знак типа данных char

В стандарте языка С сказано, что тип данных char может быть со знаком или без знака. Ответственность за определение того, какой вариант типа данных char использовать по умолчанию, лежит на компиляторе, препроцессоре или на обоих.

Для большинства аппаратных платформ тип char является знаковым, а диапазон значений данных этого типа от -128 до 127. Для небольшого количества аппаратных платформ, таких как ARM, тип char по умолчанию без знака, а возможные значения данных этого типа лежат в диапазоне от 0 до 255.

Например, для систем, на которых тип char без знака, выполнение следующего кода приведет к записи в переменную i числа 255 вместо -1.

```
char i = -1;
```

На других машинах, где тип char является знаковым, этот код выполнится правильно и в переменную i запишется значение -1. Если действительно нужно, чтобы в любом случае было записано значение -1, то предыдущий код должен выглядеть следующим образом.

```
signed char i = -1;
```

Если в вашем коде используется тип данных char, то следует помнить, что этот тип может на самом деле быть как signed char, так и unsigned char. Если необходим строго определенный вариант, то это нужно явно декларировать.

Выравнивание данных

Выравнивание (alignment) соответствует размещению порции данных в памяти. Говорят, что переменная имеет *естественное выравнивание (naturally aligned),* если она находится в памяти по адресу, значение которого кратно размеру этой переменной. Например, переменная 32-разрядного типа данных имеет естественное выравнивание, если она находится в памяти по адресу, кратному 4 байт (т.е. два младших бита адреса равны нулю). Таким образом, структура данные размером 2^n байт должна храниться в памяти по адресу, младшие n битов которого равны нулю.

Для некоторых аппаратных платформ существуют строгие требования относительно выравнивания данных. На некоторых системах, обычно RISC, загрузка неправильно выровненных данных приводит к генерации системного прерывания (trap), ошибки, которую можно обработать. На других системах с неестественно выравниваемыми данными можно работать, но это приводит к уменьшению произ-

водительности. При написании переносимого кода необходимо предотвращать проблемы, связанные с выравниванием, а данные всех типов должны иметь естественное выравнивание.

Как избежать проблем с выравниванием

Компилятор обычно предотвращает проблемы, связанные с выравниванием, путем естественного выравнивания всех типов данных. На самом деле, разработчики ядра обычно не должны заниматься проблемами, связанными с выравниванием, об этом должны заботиться разработчики компилятора gcc. Однако такие проблемы все же могут возникать, когда разработчику приходится выполнять операции с указателями и осуществлять доступ к данным, не учитывая того, как компилятор выполняет операции доступа к данным.

Доступ к адресу памяти, для которого выполнено выравнивание, через преобразованный указатель на тип данных большего размера может привести к проблемам выравнивания (для разных аппаратных платформ это может проявляться по-разному). Следующий код может привести к указанной проблеме.

```
char dog[10];
char *p = &dog[1];
unsigned long 1 = * (unsigned long *)p;
```

В этом примере указатель на данные типа unsigned char используется, как указатель на тип unsigned long, что может привести к тому, что 32-разрядное значение типа unsigned long будет считываться из памяти по адресу, не кратному четырем.

Если вы думаете: *"Зачем мне это может быть нужно?',* то вы, скорее всего, правы. Тем не менее, если мы такое сделали только что, то такое можно сделать и кто-нибудь еще, поэтому необходимо быть внимательными. Примеры, которые встречаются в реальной жизни, не обязательно будут так же очевидны.

Выравнивание нестандартных типов данных

Как уже указывалось, при выравнивании адрес стандартного типа данных должен быть кратным размеру этого типа. Нестандартные (сложные) типы данных подчиняются следующим правилам выравнивания.

- Выравнивание массива выполняется так же, как и выравнивание типа данных первого элемента (все остальные элементы будут корректно выровнены автоматически).
- Выравнивание объединения (union) соответствует выравниванию самого большого, по размеру, типа данных из тех, которые включены в объединение.
- Выравнивание структуры соответствует выравниванию самого большого, по размеру, типа данных среди типов всех полей структуры.

В структурах также могут использоваться различные способы заполнения (padding).

Заполнение структур

Структуры заполняются таким образом, чтобы каждый ее элемент имел естественное выравнивание. Например, рассмотрим следующую структуру данных на 32-разрядной машине.

```
struct animal_struct {
        char dog;               /* 1 байт */
        unsigned long cat;      /* 4 байт */
        unsigned short pig;     /* 2 байт */
        char fox;               /* 1 байт */
};
```

Эта структура данных в памяти выглядит не так, что связано с необходимостью естественного выравнивания. В памяти компилятор создает структуру данных, которая похожа на следующую.

```
struct animal_struct {
        char dog;               /* 1 байт */
        u8 __pad0[3];           /* 3 байт */
        unsigned long cat;      /* 4 байт */
        unsigned short pig;     /* 2 байт */
        char        fox;        /* 1 байт */
        u8        __pad1;       /* 1 байт */
};
```

Переменные заполнения вводятся для того, чтобы обеспечить естественное выравнивание всех элементов структуры. Первая переменная заполнения вводит дополнительные затраты памяти для того, чтобы разместить поле `cat` на границе 4-байтового адреса. Вторая переменная используется для выравнивания размера самой структуры. Дополнительный байт гарантирует, что размер структуры будет кратен четырем байтам и что каждый элемент массива таких структур будет иметь естественное выравнивание.

Следует обратить внимание, что выражение `sizeof(foo_struct)` равно значению 12 для *любого* экземпляра этой структуры на большинстве 32-разрядных аппаратных платформ. Компилятор языка C автоматически добавляет элементы заполнения, чтобы гарантировать необходимое выравнивание.

Часто имеется возможность переставить поля структуры так, чтобы избежать необходимости заполнения. Это позволяет получить правильно выровненные данные без введения дополнительных элементов заполнения и, соответственно, структуру меньшего размера.

```
struct animal struct {
        unsigned long cat;      /* 4 байта */
        unsigned short pig;     /* 2 байта */
        char dog;       /* 1 байт */
        char fox;       /* 1 байт */
};
```

Эта структура данных имеет размер 8 байт. Однако не всегда существует возможность перестановки элементов структуры местами и изменения определения струк-

туры. Например, если структура поставляется как часть стандарта, или уже используется в существующем коде, то порядок следования полей менять нельзя. Иногда, по некоторым причинам, может потребоваться специальный порядок следования полей структуры, например специальное выравнивание переменных для оптимизации попадания в кэш. Заметим, что, согласно стандарту ANSI C, компилятор никогда не должен менять порядок следования полей в структурах[5] данных — этим правом обладает только программист.

Разработчики ядра должны учитывать особенности заполнения при обмене структурами данных: передача структур по сети или непосредственное сохранение на диск, потому что необходимое заполнение может быть разным для различных аппаратных платформ. Это одна из причин, по которой в языке программирования С нет оператора сравнения структур. Память, которая используется для заполнения структур данных, может содержать случайную информацию, что делает невозможным побайтовое сравнение структур. Разработчики языка, программирования С правильно сделали, что оставили решение задачи сравнения структур на усмотрение программиста, который может создавать свои функции сравнения в каждом конкретном случае, чтобы использовать особенности построения конкретных структур.

Порядок следования байтов

Порядок следования байтов (byte ordering) — это порядок, согласно которому байты расположены в машинном слове. Для разных процессоров может использоваться один из двух типов нумерации байтов в машинном слове: наименее значимый (самый младший) байт является либо самым первым (самым левым, left-most), либо самым последним (самым правым, right-most) в слове. Порядок байтов называется *обратным (big-endian),* если наиболее значимый (самый старший) байт хранится первым, а за ним идут байты в порядке убывания значимости. Порядок байтов называется *прямым (little-endian),* если наименее значимый (самый младший) байт хранится первым, а за ним следуют байты в порядке возрастания значимости.

Даже не пытайтесь основываться на каких-либо предположениях о порядке следования байтов при написании кода ядра (конечно, если код не предназначен для какой-либо конкретной аппаратной платформы). Операционная система Linux поддерживает аппаратные платформы с обоими порядками байтов, включая и те машины, на которых используемый порядок байтов можно сконфигурировать на этапе загрузки системы, а общий код должен быть совместим с любым порядком байтов.

На рис. 19.1 показан пример обратного порядка следования байтов, а на рис. 19.2 — прямого порядка следования байтов.

Аппаратная платформа i386 использует прямой (little-endian) порядок байтов. Большинство других аппаратных платформ обычно использует обратный (big-endian) порядок.

[5] Еслибы компилятор имел возможность изменять порядок следования нолей структуры данных, то любой существующий код, который уже использует эту структуру, мог бы испортить данные. В языке программирования С функции вычисляют положение полей просто путем введения смещений от начального адреса структуры в памяти.

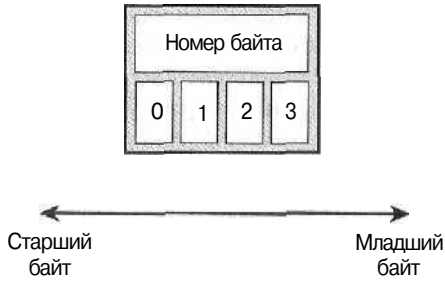


Рис. 19.1. *Обратный (big-endian) порядок следования байтов*

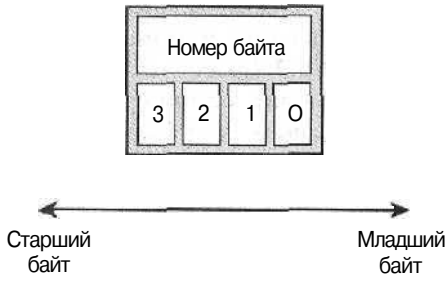


Рис. ***19.2. Прямей*** *(little-endian) порядок следования байтов*

Рассмотрим, что эти типы кодирования обозначают на практике и как выглядит двоичное представление числа 1027, которое хранится в виде четырехбайтового целочисленного типа данных.

```
00000000 00000000 00000100 00000011
```

Внутренние представления этого числа в памяти при использовании прямого и обратного порядка байтов отличаются, как это показано в табл. 19.3.

Таблица 19.3. Расположение данных в памяти для разных порядков следования байтов

Адрес	Обратный порядок	Прямой порядок
0	00000000	00000011
1	00000000	00000100
2	00000100	00000000
3	00000011	00000000

Обратите внимание на то, что для аппаратной платформы с обратным порядком байтов самый старший байт записывается в самый минимальный адрес памяти.

И наконец, еще один пример — фрагмент кода, который позволяет определить порядок байтов для той аппаратной платформы, на которой он выполняется.

```
int x = 1;
if (*(char *)&x == 1)
        /* прямой порядок */
else
        /* обратный порядок */
```

Этот пример работает как в ядре, так и в пространстве пользователя.

История терминов big-endian и little-endian

Термины big-endian и little-endian заимствованы из сатирического романа Джонатана Свифта "Путешествие Гулливера", который был издан в 1726 году. В этом романс наиболее важной политической проблемой народа лилипутов была проблема, с какого конца следует разбивать яйцо: с тупого (big) или острого (little). Тех, кто предпочитал тупой конец называли "тупоконечниками" (big-endian), тех же, кто предпочитал острый конец, называли "остроконечниками" (little-endian).

Аналогия между дебатами лилипутов и спорами о том, какой порядок байтов лучше, говорит о том, что это вопрос больше политический, чем технический.

Порядок байтов в ядре

Для каждой аппаратной платформы, которая поддерживается ядром Linux, в файле <asm/byteorder.h> определена одна из двух констант __BIG_ENDIAN или __LITTLE_ENDIAN, в соответствии с используемым порядком байтов.

В этот заголовочный файл также включаются макросы из каталога include/linux/byteorder/, которые помогают конвертировать один порядок байтов в другой. Ниже показаны наиболее часто используемые макросы.

```
u23__cpu_to_be32(u32); /* преобразовать порядок байтов текущего
                          процессора в порядок big-endian */
u32__cpu_to_le32(u32); /* преобразовать порядок байтов текущего
                          процессора в порядок little-endian */
u32__be32_to_cpu(u32);  /* преобразовать порядок байтов big-endian в
                           порядок байтов текущего процессора */
u32__le32_to_cpus(u32); /* преобразовать порядок байтов little-endian
                           в порядок байтов текущего процессора */
```

Эти макросы выполняют преобразование одного порядка байтов в другой. В случае когда порядки байтов, между которыми выполняется преобразование, одинаковы (например, если выполняется преобразование в обратный порядоку байтов и процессор тоже использует такой же порядок), то эти макросы не делают ничего. В противном случае возвращается преобразованное значение.

Таймер

Никогда нельзя привязываться к какой-либо конкретной частоте генерации прерывания системного таймера и, соответственно, к тому, сколько раз в секунду изменяется переменная jiffies. Всегда необходимо использовать константу HZ, чтобы корректно определять интервалы времени. Это очень важно, потому что значение частоты системного таймера может отличаться не только для разных аппаратных платформ, но и для одной аппаратной платформы при использовании разных версий ядра.

Например, константа HZ для аппаратной платформы x86 сейчас равна 1000. Это значит, что прерывание таймера возникает 1000 раз в секунду, или каждую миллисекунду. Однако до серии ядер 2.6 для аппаратной платформы x86 значение константы HZ было равно 100. Для разных аппаратных платформ эти значения отличаются: для аппаратной платформы alpha константа HZ равна 1024, а для платформы ARM — 100.

Никогда нельзя сравнивать значение переменной jiffies с числом, таким как 1000, и думать, что это всегда будет означать одно и то же. Для получения интервалов времени необходимо всегда умножать или делить на константу HZ, как в следующем примере.

```
HZ            /* одна секунда */
(2*HZ)        /* две секунды */
(HZ/2)        /* полсекунды */
(HZ/100)      /* 10 мс */
(2*H2/100)   /*  20 мс */
```

Константа HZ определена п файле <asm/param.h>. Об этом подробно рассказано в главе 10, "Таймеры и упрадление временем".

Размер страницы памяти

При работе со страницами памяти никогда нельзя привязываться к конкретному размеру страницы. Программисты, которые разрабатывают для аппаратной платформы x86, часто делают ошибку, считая, что размер страницы всегда равен 4 Кбайта. Хотя это справедливо для платформы x86, для других аппаратных платформ размер станицы может быть другим. Некоторые аппаратные платформы поддерживают несколько размеров страниц! В табл. 19-1 приведен список размеров страниц памяти для всех поддерживаемых аппаратных платформ.

Таблица 19.4. Размеры страниц памяти для разных аппаратных платформ

Аппаратная платформа	**Значение** PAGE_SHIFT	**Значение** PAGE_SIZE
alpha	13	8 Кбайт
arm	12, 14, 15	4 Кбайт, 16 Кбайт, 32 Кбайт
cris	13	8 Кбайт
h8300	12	4 Кбайт
i386	12	4 Кбайт
ia64	12, 13, 14, 16	4 Кбайт, 8 Кбайт, 32 Кбайт, 64 Кбайт
m68k	12, 13	4 Кбайт, 8 Кбайт
m86knommu	12	4 Кбайт
mips	12	4 Кбайт
mips64	12	4 Кбайт
parisc	12	4 Кбайт
ppc	12	4 Кбайт
ppc64	12	4 Кбайт
S390	12	4 Кбайт
sh	12	4 Кбайт
spare	12,13	4 Кбайт, 8 Кбайт
sparc64	13	8 Кбайт
v850	12	4 Кбайт
x86_64	12	4 Кбайт

При работе со страницами памяти необходимо использовать константу PAGE_SIZE, которая содержит размер страницы памяти в байтах.

Значение макроса PAGE_SHIFT — это количество битов, на которое необходимо сдвинуть влево значение адреса, чтобы получить номер соответствующей страницы памяти. Например, для аппаратной платформы x86, для которой размер страницы равен 4 Кбайт, макрос PAGE_SIZE равен 4096, а макрос PAGE_SHIFT— 12. Эти значения содержатся в заголовочном файле <asm/page.h>.

Порядок выполнения операций процессором

Вспомните из материала главы 9, "Средства синхронизации в ядре", что для различных аппаратных платформ процессоры в разной степени изменяют порядок выполнения программных инструкций. Для некоторых процессоров порядок выполнения операций строго соблюдается, запись данных в память и считывание данных из памяти выполняются в строго указанном в программе порядке. Другие процессоры имеют ослабленные требования к порядку выполнения операций считывания и записи данных и могут изменять порядок выполнения этих операций с целью оптимизации.

Если код зависит от порядка выполнения операций чтения-записи данных, то необходимо гарантировать, что даже процессор с самыми слабыми ограничениями на порядок выполнения чтения-записи будет выполнять эти операции в правильном порядке. Это делается с помощью соответствующих барьеров, таких как rmb() и wmb(). Более подробная информация приведена в главе 9, "Средства синхронизации в ядре".

Многопроцессорность, преемптивность и верхняя память

Может показаться неправильным включать поддержку симметричной многопроцессорности, возможность вытеснения процессов в режиме ядра и работу с верхней памятью в вопросы переносимости. В конце концов, это не особенности аппаратной платформы, которые влияют на операционную систему, а функции ядра Linux, которые по многом не зависят от аппаратной платформы. Тем не менее для этих функций существуют важные конфигурационные параметры, которые необходимо учитывать при разработке кода. Программировать всегда необходимо под SMP, с поддержкой преемптивности и с использованием верхней памяти, чтобы код был безопасным всегда, при любых конфигурациях. Необходимо всегда соблюдать следующие правила.

- Всегда необходимо учитывать, что код может выполняться на SMP-системе и использовать соответствующие блокировки.
- Всегда необходимо учитывать, что код может выполняться при включенной преемптивности ядра, поэтому необходимо всегда использовать необходимые блокировки и операции для управления преемптивностью.
- Всегда необходимо учитывать, что код может выполняться на системе с поддержкой верхней памяти (непостоянно отображаемая память) и при необходимости использовать функцию kmap ().

Пару слов о переносимости

Если говорить коротко, то написание переносимого, ясного и красивого кода подразумевает следующие два момента.

- Код необходимо разрабатывать с учетом самого общего сценария: следует предполагать, что все, что может случиться, обязательно случится, и принять на этот счет все возможные меры.
- Всегда необходимо все подводить под наибольший общий знаменатель: нельзя полагаться на то, что будут доступны все возможности ядра, следует опираться только на минимум возможностей, которые доступны всем аппаратным платформам.

Написание переносимого кода требует строгого учета многих факторов: размер машинного слова, размеры типов данных, выравнивание в памяти, порядок байтов, размер страницы, изменение порядка операций процессора и т.д. В большинстве случаев при программировании ядра следует гарантировать, что типы данных используются правильно. Тем не менее время от времени все равно всплывают проблемы, связанные с особенностью той или другой аппаратной платформы. Важно понимать проблемы, связанные с переносимостью, и всегда писать четкий и переносимый код ядра.

20

Заплаты, разработка и сообщество

Одно из самых больших преимуществ операционной системы Linux — это связанное с ней большое сообщество пользователей и разработчиков. Сообщество предоставляет множество глаз для проверки кода и множество пользователей для тестирования и отправки сообщений об ошибках. Наконец, сообщество решает, какой код включать в основное ядро. Поэтому важно понимать, как это все происходит.

Сообщество

Если говорить о том, где физически существует сообщество разработчиков ядра Linux, то можно сослаться на *список рассылки разработчиков ядра Linux (Linux Kernel Mail List,* или, сокращенно, *LKML).* Список разработчиков ядра Linux— это то место, где происходит большинство дискуссий, дебатов и флеймов вокруг ядра Linux. Здесь обсуждаются новые возможности, и большая часть кода отправляется в этот список рассылки перед тем, как этот код для чего-нибудь начинает использоваться. В списке рассылки насчитывается до 300 сообщений в день — количество не для слабонервных. Подписаться на этот список (или но крайней мере читать его обзор) рекомендуется всем, кто серьезно занимается разработкой ядра. Даже только наблюдая за работой специалистов, можно узнать достаточно МНОГО.

Подписаться на данный список рассылки можно, отправив сообщение

```
subscribe linux-kernel <your@email.address>
```

в виде обычного текста на адрес majordomo@vger.kernel.org. Больше информации доступно по Интернет-адресу http://vger.kernel.org/, а список часто задаваемых вопросов (FAQ) — по адресу http://www.tux.org/lkml/.

Много других WWW-сайтов и списков рассылки посвящены как ядру, так и вообще операционной системе Linux. Отличный ресурс для начинающих хакеров — http://www.kernelnewbies.org/, сайт, который сможет удовлетворить желания всех, кто, стачивая зубы, грызет основы разработки ядра. Два других отличных источника ин-

формации — это сайт http://www.lwn.net/, Linux Weekly News, на котором есть большая колонка новостей ядра, и сайт http://www.kerneltraffic.org, Kernel Traffic, который содержит сводку сообщений из списка рассылки разработчиков ядра Linux с комментариями.

Стиль написания исходного кода

Как и для любого большого программного проекта, для ядра Linux определен стиль написания исходного кода, который определяет форматирование и размещение кода. Это сделано не потому, что стиль написания, который принят для Linux, лучше других (хотя очень может быть), и не потому, что все программисты пишут неразборчиво (хотя тоже бывает), а потому, что *одинаковость* стиля является важным моментом для обеспечения *производительности* разработки. Часто говорят, что стиль написания исходного кода не важен, потому что он не влияет на скомпилированный объектный код. Однако для большого программного проекта, в котором задействовано большое количество разработчиков, такого как ядро, важна слаженность стиля. Слаженность включает в себя одинаковость восприятия, что ведет к упрощению чтения, к избежанию путаницы и вселяет надежду на то, что и в будущем стиль останется одинаковым. К тому же, это приводит к увеличению количества разработчиков, которые смогут нормально читать ваш код, и увеличивает количество кода, который вы сможете нормально читать. Для проектов с открытым исходным кодом чем больше будет глаз, тем лучше.

Пока стиль еще не выбран и широко не используется, не так важно, *какой именно* стиль выбрать. К счастью, еще очень давно Линус представил на рассмотрение стиль, который необходимо использовать, и при написании большей части кода сейчас стараются придерживаться именно этого стиля. Подробное описание стиля приведено в файле Documentation/CodingStyle со свойственным Линусу юмором.

Отступы

Для выравнивания текста и введения отступов используются символы табуляции. Размер одного символа табуляции при отображении соответствует восьми позициям. Это не означает, что для структурирования можно использовать восемь или четыре символа "пробел" либо что-нибудь еще. Это означает, что каждый уровень отступа равен одному символу табуляции от предыдущего и что при отображении длина символа табуляции равна восьми символам. По непонятным причинам, это правило почти всегда нарушается, несмотря на то что оно очень сильно влияет на читабельность. Восьмисимвольная табуляция позволяет очень легко визуально различать отдельные блоки кода даже после нескольких часов работы.

Если табуляция в восемь символов кажется очень большой, то не нужно делать так много вложений кода. Почему ваши функции имеют пять уровней вложенности? Необходимо исправлять код, а не отступы.

Фигурные скобки

Как располагать фигурные скобки, это личное дело каждого, и практически нет никаких принципиальных причин, по которым одно соглашение было бы лучше другого, но какое-нибудь соглашение все-таки должно быть. Принятое соглашение при

разработке ядра — это размещать открывающую скобку в первой строке, сразу за соответствующим оператором. Закрывающая скобка помещается в первой позиции с новой строки, как в следующем примере.

```
if (fox) {
        dog();
        cat();
}
```

В случае, когда за закрывающей скобкой продолжается то же самое выражение, то продолжение выражения записывается в той же строке, что и закрывающая скобка, как показано ниже

```
if (fox) {
        ant();
        pig();
} else {
        dog();
        cat();
}
```

или следующим образом.

```
do {
        dog();
        cat();
} while (fox);
```

Для функций это правило не действует, потому что внутри одной функции тело другой функции описывать нельзя.

```
unsigned long func(void)
{
/* .. */
}
```

И наконец, для выражений, в которых фигурные скобки не обязательны, эти скобки можно опустить.

```
if (foo)
  bar();
```

Логика всего этого базируется на K&R[1].

Длинные строки

При написании кода ядра необходимо стараться, насколько это возможно, чтобы длина строки была не больше 80 символов. Это позволяет строкам, при отображении на терминале размером 80x24 символа, вмещаться в одну строу терминала.

[1] Брайан У. Керниган, Деннис М. Ритчи, *Язык программирования С, 2-е изд.* Пер. с англ. — М.: Издат. дом "Вильяме", 2005.

Не существует стандартного правила, что делать, если длина строки кода обязательно должна быть больше 80 символов. Некоторые разработчики просто пишут длинные строки, возлагая ответственность за удобочитаемое отображение строк на программу текстового редактора. Другие разработчики разбивают такие строки на части и вручную вставляют символы конца строки в тех местах, которые кажутся им наиболее подходящими для этого, и отделяют продолжения разбитой строки от ее начала двумя символами табуляции.

Некоторые разработчики помещают параметры функции друг под другом, если параметры не помещаются в одной строке, как в следующем примере.

```
static void get_pirate_parrot(const char *name,
                              unsigned long disposition,
                              unsigned long feather_quality)
```

Другие разработчики разбивают длинную строку на части, но не располагают параметры функций друг под другом, а просто используют два символа табуляции для отделения продолжений длинной строки от ее начала, как показано ниже.

```
int find_pirate_flag_by_color(const char *color,
            const char *name, int  len)
```

Поскольку на этот счет нет определенного правила, выбор остается за разработчиками, то есть за вами.

Имена

В именах нельзя использовать символы разных регистров. Назвать переменную именем idx, или даже i — это очень хорошо, но при условии, что будет понятно назначение этой переменной. Слишком хитрые имена, такие как theLoopIndex, недопустимы. Так называемая "венгерская запись" (Hungarian notation), когда тип переменной кодируется в ее имени, В данном случае — признак плохого тона. Это C, а не Java и Unix, а не Windows.

Тем не менее, глобальные переменные и функции должны иметь наглядные имена. Если глобальной функции присвоить имя `atty()`, то это может привести к путанице. Более подходящим будет имя `gstactivetty()`. Это все-таки Linux, а не BSD.

Функции

Существует мнемоническое правило: функции не должны по объему кода превышать двух экранов текста и иметь больше *десяти* локальных переменных. Каждая функция должна выполнять одно действие, но делать это хорошо. Не вредно разбить функцию на последовательность более мелких функций. Если возникает беспокойство по поводу накладных расходов за счет вызова функций, то можно использовать подстановку тела — `inline`.

Комментарии

Очень полезно использовать комментарии кода, но делать это нужно правильно. Обычно необходимо описывать, *что* делает код и *для чего* это делается. То, *как* реализован алгоритм, описывать не нужно, это должно быть ясно из кода. Если так сде-

лать не получается, то, возможно, стоит пересмотреть то, что вы написали. Кроме того, комментарии не должны включать информацию о том, кто написал функцию, когда это было сделано, время модификации и др. Такую информацию логично размещать в самом начале файла исходного кода.

В ядре используются комментарии в стиле C, хотя компилятор gcc поддерживает также и комментарии в стиле C++. Обычно комментарии кода ядра должны быть похожи на следующие (только на английском языке, конечно).

```
/*
* get_ship_speed() - возвращает текущее значение скорости
* пиратского корабля
* Необходима для вычисления координат корабля.
* Может переходить в состояние ожидания,
* нельзя вызывать при удерживаемой блокировке.
*/
```

Комментарии внутри функций встречаются редко, и их нужно использовать только в специальных ситуациях, таких как документирование дефектов, или для важных замечаний. Важные замечания часто начинаются со строки "XXX: ", а информация о дефектах— со строки "FIXME: ", как в следующем примере.

```
/*
* FIXME: Считается, что dog == cat.
* В будущем это может быть не так
*/
```

У ядра есть возможность автоматической генерации документации. Она основана на GNOME-doc, но немного модифицирована и называется Kernel-doc. Для создания документации в формате HTML необходимо выполнить следующую команду.

```
make htmldocs
```

Для генерации документации в формате postscript команда должна быть следующей.

```
make psdocs
```

Документировать код можно путем введения комментариев в специальном формате.

```
/**
* find_treasure - нахождение сокровищ, помеченных на карте крестом
* @map - карта сокровищ
* @time - момент времени, когда были зарыты сокровища
*
* Должна вызываться при удерживаемой блокировке pirate_ship_lock.
*/
void find_treasure(int dog, int cat)
{
/* .. */
}
```

Для более подробной информации см. файл Documentation/kernel-doc-nano-HOWTO.txt.

Использование директивы typedef

Разработчики ядра не любят определять новые типы с помощью оператора typedef, и причины этого довольно трудно объяснить. Разумное объяснение может быть следующим.

- Определение нового типа через оператор typedef скрывает истинный вид структур данных.
- Поскольку новый тип получается скрытым, то код более подвержен таким нехорошим вещам, как передача структуры данных в функцию по значению, через стек.
- Использование оператора typedef — признак лени.

Чтобы избежать насмешек, лучше не использовать оператор typedef.

Конечно, существуют ситуации, в которых полезно использовать оператор typedef: сокрытие специфичных для аппаратной платформы деталей реализации или обеспечение совместимости при изменении типа. Нужно хорошо подумать, действительно ли оператор typedef необходим или он используется только для того, чтобы уменьшить количество символов при наборе кода.

Использование того, что уже есть

Не нужно изобретать паровоз. Ядро предоставляет функции работы со строками, подпрограммы для сжатия и декомпрессии данных и интерфейс работы со связанными списками — их необходимо использовать.

Не нужно инкапсулировать стандартные интерфейсы в другие реализации обобщенных интерфейсов. Часто приходится сталкиваться с кодом, который переносится с других операционных систем в систему Linux, при этом на основе существующих интерфейсов реализуется некоторая громоздкая функция, которая служит для связи нового кода с существующим. Такое не нравится никому, поэтому необходимо непосредственно использовать предоставляемые интерфейсы.

Никаких директив ifdef в исходном коде

Использование директив препроцессора ifdef в исходном коде категорически не рекомендуется. Никогда не следует делать чего-нибудь вроде следующего.

```
...
#ifdef config_foo
  foo();
#endif
...
```

Вместо этого, если макрос CONFIG_FOO не определен, необходимо определять функцию foo(), как ту, которая ничего не делает.

```
tifdef CONFIG_FOO
static int foo(void)
{
/* .. */
}
#else
static inline int foo(void) { }
#endif
```

После этого можно вызывать функцию foo() без всяких условий. Пусть компилятор поработает за вас.

Инициализация структур

Структуры необходимо инициализировать, используя метки полей. Это позволяет предотвратить некорректную инициализацию при изменении структур. Это также позволяет выполнять инициализацию не всех полей. К сожалению, в стандарте C99 принят довольно "страшненький" формат меток полей, а в компиляторе gcc ранее использовавшийся формат меток полей в стиле GNU признан устаревшим. Следовательно, для кода ядра необходимо использовать новый формат, согласно стандарту C99, каким бы ужасным он ни был.

```
struct foo my_foo = {
  .a = INITIAL_A,
  .b = INITIAL_B,
};
```

где a и b — это поля структуры struct foo, а параметры INITIAL_A и INITIAL_B — соответственно, их начальные значения. Если поле не указано при инициализации, то оно устанавливается в свое начальное значение, согласно стандарту ANSI C (указателям присваивается значение NULL, целочисленным полям — нулевое значение, а нолям с плавающей точкой— значение 0.0). Например, если структура struct foo также имеет поле int c, то это поле в предыдущем примере будет инициализировано в значение 0.

Да, это ужасно. Но у нас нет другого выбора.

Исправление ранее написанного кода

Если в ваши руки попал код, который даже близко не соответствует стилю написания кода ядра Linux, то все равно не стоит терять надежды. Немного упорства, и утилита indent поможет сделать все как надо. Программа indent — отличная утилита GNU, которая включена во многие поставки ОС Linux и предназначена для форматирования исходного кода в соответствии с заданными правилами. Установки по умолчанию соответствуют стилю форматирования GNU, который выглядит не очень красиво. Для того чтобы утилита выполняла форматирование в соответствии со стилем написания кода ядра Linux, необходимо использовать следующие параметры.

```
indent -kr -i8 -ts8 -sob -180 -ss -bs -psl <файл>
```

Можно также использовать сценарий scripts/Lindent, который вызывает утилиту indent с необходимыми параметрами.

Организация команды разработчиков

Разработчики — это хакеры, которые занимаются развитием ядра Linux. Некоторые делают это за деньги, для некоторых это хобби, но практически все делают это с удовольствием. Разработчики ядра, которые внесли существенный вклад, перечислены в файле CREDITS, который находится в корневом каталоге дерева исходных кодов ядра.

Для различных частей ядра выбираются *ответственные разработчики (mainlainers)*, которые официально выполняют их поддержку. Ответственные разработчики — это один человек или группа людей, которые полностью отвечают за свою часть ядра. Каждая подсистема ядра, например сетевая подсистема, также имеет связанного с ней ответственного. Разработчики, которые отвечают за определенный драйвер или подсистему, обычно перечислены в файле MAINTAINERS. Этот файл тоже находится в корневом каталоге дерева исходных кодов.

Среди ответственных разработчиков существует специальный человек, который отвечает за все ядро в целом (kernel maintainer). Исторически, Линус отвечает за разрабатываемую серию ядер (где наиболее интересно) и за первые выпуски стабильной версии ядра. Вскоре после того как ядро становится стабильным, Линус передает бразды правления одному из ведущих разработчиков. Они продолжают поддержку стабильной серии ядра, а Линус начинает работу над новой разрабатываемой серией. Таким образом, серии ядер 2.0, 2.4 и 2.6 все еще активно поддерживаются.

Несмотря на слухи, здесь нет *никаких* других, в том числе тайных, организаций.

Отправка сообщений об ошибках

Если вы обнаружили ошибку, то наилучшим решением будет исправить ее, сгенерировать соответствующую заплату, оттестировать и отправить, как это будет рассказано в следующих разделах. Конечно, можно и просто сообщить об ошибке, чтобы кто-нибудь исправил ее для вас.

Наиболее важная часть сообщения об ошибке — это полное описание проблемы. Необходимо описать симптомы, системные сообщения и декодированное сообщение oops (если такое есть). Еще более важно, чтобы вы смогли пошагово описать, как устойчиво воспроизвести проблему, и кратко описать особенности вашего аппаратного обеспечения.

Следующий этап — определение того, кому отправить сообщение об ошибке. В файле MAINTAINERS приведен список людей, которые отвечают за каждый драйвер и подсистему. Эти люди должны получать все сообщения об ошибках, которые возникают в том коде, поддержку которого они выполняют. Если вы не смогли найти необходимого разработчика, то отправьте вопрос в список рассылки разработчиков ядра Linux по адресу linux-kernel@vger.kernel.org. Даже если вы нашли нужное ответственное лицо, то никогда не помешает отправить копию сообщения в список рассылки разработчиков.

Больше информации об этом приведено в файлах REPORTING-BUGS и Documentation/oops-tracing.txt.

Генерация заплат

Все изменения исходного кода ядра Linux распространяются в виде заплат (patch). Заплаты представляют собой результат вывода утилиты GNU diff(1) в формате, который может подаваться на вход программы patch(1). Наиболее просто сгенерировать заплату можно в случае, когда имеется два дерева исходных кодов ядра: одно — стандартное, а другое — с вашими изменениями. Обычная схема имен состоит в том, что каталог, в котором находится стандартное ядро, называется linux-x.y.z (каталог, в который разворачивается архив дерева исходного кода в формате tar), а

имя модифицированного ядра — linux. Для генерации заплаты на основе двух каталогов *с* исходным кодом необходимо выполнить следующую команду из каталога, в котором находятся два рассмотренных дерева исходного кода.

```
diff -urN linux-x.y.z/linux/ > my-patch
```

Обычно это делается где-нибудь в домашнем каталоге, а не в каталоге /usr/src/linux, поэтому нет необходимости иметь права пользователя root. Флаг -u указывает, что необходимо использовать унифицированный формат вывода команды diff. Без этого флага внешний вид заплаты получается неудобочитаемым. Флаг -r указывает на необходимость рекурсивного анализа каталогов, а флаг -N указывает, что новые файлы, которые появились в измененном каталоге, должны быть включены в результат вывода команды diff. Если же необходимо получить только изменения одного файла, то можно выполнить следующую команду.

```
diff -u linux-x.y.z/some/file_linux/some/file > my-patch
```

Обратите внимание на то, что вычислять изменения необходимо всегда, находясь в одном текущем каталоге, а именно в том, где находятся оба дерева исходного кода. При этом получается заплата, которую легко могут использовать все, даже в случаях, когда имена каталогов исходного кода отличаются от тех, которые использовались при генерации заплаты. Для того чтобы применить заплату, которая сгенерирована таким образом, необходимо выполнить следующую команду из корневого каталога дерева исходного кода.

```
patch -p1 < ../my-patch
```

В этом примере имя файла, который содержит заплату, - my-patch, а находится он в родительском каталоге по отношению к каталогу, в котором хранится дерево исходного кода ядра. Флаг -p1 означает, что необходимо игнорировать (strip) имя первого каталога в путях всех файлов, в которые будут вноситься изменения. Это позволяет применить заплату независимо от того, какие имена каталогов кода ядра были на той машине, где создавалась заплата.

Полезная утилита diffstat позволяет сгенерировать гистограмму изменений, к которым приведет применение заплаты (удаление и добавление строк). Для того чтобы получить эту информацию для какой-либо заплаты, необходимо выполнить следующую команду.

```
diffstat -p1 my-patch
```

Обычно полезно включить результат выполнения этой команды при отправлении заплаты п список рассылки lkml. Так как программа patch (1) игнорирует все строки до того момента, пока не будет обнаружен формат diff, то вначале заплаты можно включить короткое описание.

Представление заплат

Заплата должна быть сгенерирована так, как описано в предыдущем разделе. Если заплата касается определенного драйвера или подсистемы, то заплату нужно отправить соответствующему ответственному разработчику, одному из тех, которые перечислены в файле MAINTAINERS. Другой вариант — это отправить сообщение в список рассылки разработчиков ядра по адресу linux-kernel@vger.kernel.org.

Обычно тема (subject) письма, в котором содержится заплата, должна быть похожа на следующую " [PATCH] короткое описание. ". В теле письма должны быть описаны основные технические детали изменений, которые вносятся заплатой, а также обоснования необходимости этих изменений. Описание должно быть максимально конкретным. Также необходимо указать, на какую версию ядра рассчитана заплата.

Большинство разработчиков ядра будут просматривать заплату прямо в теле письма и при необходимости записывать все письмо в файл. Следовательно, лучше всего будет вставить заплату прямо в тело письма, в самом конце сообщения. Будьте внимательны, потому что некоторые злобные почтовые клиенты вводят в сообщения дополнительное форматирование. Это испортит заплату и будет надоедать разработчикам. Если ваш почтовый клиент делает такие вещи, то необходимо поискать возможность включения текста без изменений ("Insert Inline") или что-нибудь аналогичное. Нормально работает также присоединение (attachment) заплаты в виде обычного текста, без перекодировки.

Если заплата большая или содержит *несколько* принципиальных изменений, то лучше разбить ее на логические части. Например, если вы предложили новый API и изменили несколько драйверов с целью его использования, то эти изменения можно разбить на две заплаты (новый API и изменения драйверов) и выслать их двумя письмами. Если в каком-либо случае необходимо вначале применить предыдущую заплату, то это необходимо специально указать.

После отправки наберитесь терпения и подождите ответа. Не нужно обижаться на негативный ответ — в конце концов это тоже ответ! Обдумайте проблему и вышлите обновленную версию заплаты. Если ответа нет, то попытайтесь разобраться, что было сделано не так, и решить проблему. Спросите у ответственного разработчика и в списке рассылки по поводу комментариев. Если повезет, то ваши изменения будут включены в новую версию ядра!

Заключение

Наиболее важными качествами любого хакера являются желание и умение работать — нужно искать себе проблемы и решать их. В этой книге приведено описание основных частей ядра, рассказано об интерфейсах, структурах данных, алгоритмах и принципах работы. Книга предоставляет вид ядра изнутри и делает это в практической форме. Она предназначена для того, чтобы удовлетворить ваше любопытство и стать отправной точкой в разработке ядра.

Тем не менее, как уже было сказано, единственный способ начать разрабатывать ядро — это начать *читать* и *писать* исходный код. Операционная система Linux предоставляет возможность работать в сообществе, которое не только позволяет это делать, но и активно побуждает к указанным действиям. Если есть желание действовать — вперед!

A

Связанные списки

Связанный список — это структура хранения информации (контейнер), которая может содержать переменное количество элементов данных, часто называемых *узлами,* и позволяет манипулировать этими данными. В отличие от статического массива, элементы связанного списка можно создавать динамически. Это дает возможность создавать переменное количество элементов списка, причем указанное количество может быть неизвестно на этапе компиляции. Так как элементы связанных списков создаются в разные моменты времени, они не обязательно будут находиться в смежных областях оперативной памяти. Поэтому элементы списка должны быть *связаны* друг с другом таким образом, чтобы каждый элемент содержал указатель на *следующий* за ним элемент (next). Вставка или удаление элементов списка выполняется простым измепением указателей на следующий элемент. Структура связанного списка показана на рис. А.1.

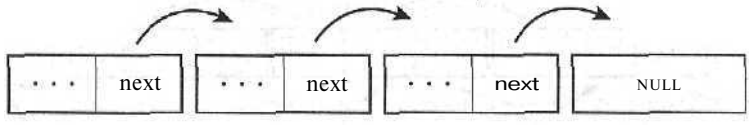


Рис. А.I. Однос&язный список

В. некоторых связанных списках содержится указатель не только на следующий, но и на *предыдущий* элемент (prev). Эти списки называются *двухсвязными (doubly linked),* потому что они связаны как вперед, так и назад. Связанные списки, аналогичные тем, что показаны на рис.А.1, называются *односвязиыми (singly linked).* Двухсвязный список показан на рис. А.2.

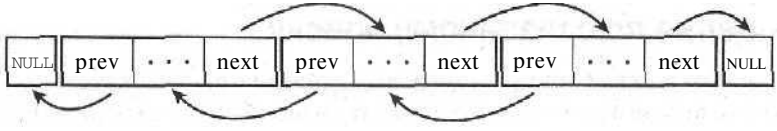


Рис. А.2. Двухсвязный список

Кольцевые связанные списки

Последний элемент связанного списка не имеет следующего за ним элемента, и значение указателя next последнего элемента обычно устанавливается равным специальному значению, обычно NULL, чтобы показать, что этот элемент списка является последним. в определенных случаях последний элемент списка не указывает на специальное значение, а указывает на первый элемент этого же списка. Такой список называется *кольцевым связанным списком (circular linked list),* поскольку связи образуют топологию кольца. Кольцевые связанные списки могут быть как односвязными, так и двухсвязными. В двухсвязных кольцевых списках указатель prev первого элемента указывает на последний элемент списка. На рис. А.3 и А.4 показаны соответственно односвязные и двухсвязные кольцевые списки.

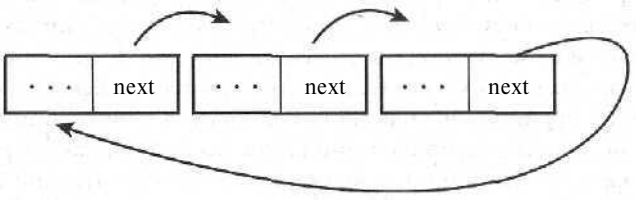


Рис. А3. Односвязный кольцевой список

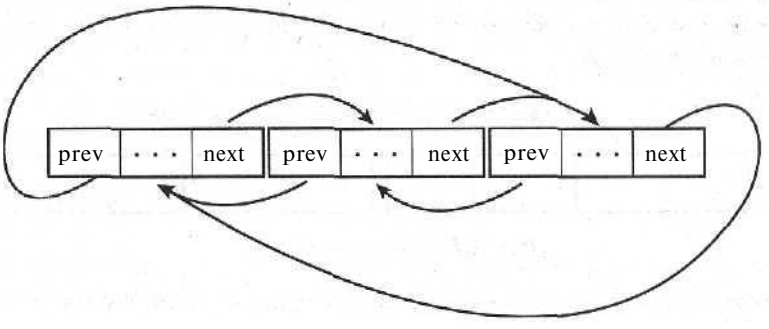


***Рис.** А.4. Двухсвязный кольцевой список*

Стандартной реализацией связанных списков в ядре Linux является *двухсвязный кольцевой список.* Такие связанные списки обеспечивают наибольшую гибкость работы.

Перемещение по связанному списку

Перемещение по связанному списку выполняется последовательно (линейно). После того как просмотрен текущий элемент, выполнятся разыменование его указателя next, что позволяет обратиться к следующему за ним элементу и т.д. Это самый простой и наиболее подходящий метод перемещения но связанному списку. Если важна возможность произвольного доступа к любому элементу контейнера, то связанные списки не используются. Связанные списки используются, когда важна возможность динамического добавления и удаления элементов, а также возможность последовательного прохождения по всем элементам списка.

Часто первый элемент списка представлен с помощью специального указателя, который называется *головным элементом* или *головой (head),* что дает возможность быстро и легко обращаться к первому элементу. В некольцевом связанном списке последний элемент отличается тем, что его указатель равен значению NULL. В кольцевом связанном списке последний элемент отличается тем, что указывает на головной элемент. Таким образом прохождение списка можно выполнить линейно, начиная с первого элемента и заканчивая последним. В двухсвязном списке прохождение можно также выполнить и в противоположном направлении, начиная с последнего и заканчивая первым элементом. Конечно, если задан определенный элемент списка, то можно перейти по списку вперед и назад на заданное количество элементов. При этом нет необходимости проходить весь список.

Реализация связанных списков в ядре Linux

В ядре Linux для прохождения по связанным спискам используется унифицированный подход. При прохождении связанного списка, если не важен порядок прохода, эту операцию не обязательно начинать с головного элемента, на самом деле вообще не важно, с какого элемента списка начинать прохождение! Важно только, чтобы при таком прохождении были пройдены все узлы. В большинстве случаев нет необходимости вводить концепции первого и последнего элементов. Если в кольцевом связанном списке содержится коллекция несортированных данных, то *любой* элемент можно назвать головным. Для прохождения всего связанного списка необходимо взять любой элемент и следовать за указателями, пока снова не вернемся к тому элементу, с которого начали обход списка. Это избавляет от необходимости вводить специальный головной элемент. Кроме того, упрощаются процедуры работы со связанными списками. Каждая подпрограмма должна просто принимать указатель на один элемент — *любой* элемент списка. Разработчики ядра даже немножко гордятся такой остроумной реализацией.

Связанные списки в ядре, так же как и в любой сложной программе, встречаются часто. Например, в ядре связанный список используется для хранения списка задач (структура `task_struct` каждого процесса является элементом связанного списка).

Структура элемента списка

Раньше в ядре было несколько реализаций связанных списков. Тем не менее в таких случаях необходима единая реализация с целью убрать разный код, который выполняет одинаковые действия. Во время разработки серии ядер 2.1 была предложена единая реализация связанных списков в ядре. Сегодня во всех подсистемах ядра используется официальная реализация. Для новых разработок необходимо использовать только существующий интерфейс и не нужно изобретать велосипед.

Код работы со связанными списками определен в файле `<linux/list.h>`, а основная структура данных имеет очень простой вид.

```
struct list_head {
        struct list_head *next, *prev;
};
```

Обратите внимание на характерное имя структуры `listhead`. Такое название выбрано, чтобы подчеркнуть, что списку не нужно головного элемента. Наоборот,

обход списка можно начинать с любого элемента, и каждый элемент может быть головным. В связи с этим все элементы списка называются головными (list head). Указатель next указывает на следующий элемент списка, а указатель prev— на предыдущий элемент списка. Если текущий элемент списка является последним, то его указатель next указывает на первый узел. Если же текущим элементом является первый, то его указатель prev указывает па последний узел списка. Благодаря элегантной реализации связанных списков без концепции головного элемента, можно вообще не вводить понятия *первого* и *последнего* элементов.

Структура `listhead` сама по себе бесполезна. Ее необходимо включать в другие структуры данных.

```
struct my_struct {
        struct list_head list;
        unsigned long dog;
        void *cat;
};
```

Перед использованием элементы связанных списков должны быть инициализированы. Так как элементы связанных списков обычно создаются динамически (иначе, вероятно, не нужно использовать связанный список), то эти элементы, как правило, инициализируются во время выполнения.

```
struct my_struct *p;
/* выделить память для структуры my_struct .. */
p->dog = 0;
p->cat = NULL;
INIT_LIST_HEAD(&p->list);
```

Если структура данных создается статически во время компиляции и на нее есть непосредственная ссылка, то инициализацию можно выполнить следующим образом.

```
struct my_struct mine = {
        .list = LIST_HEAD_INIT(mine.list),
        .dog = 0,
        .cat = NULL
};
```

Для того чтобы создать и инициализировать связанный список, можно использовать следующее объявление.

```
static LIST_HEAD(fox);
```

Эта команда позволяет статически создать связанный список с именем fox.

Нет необходимости явно выполнять какие-либо операции со служебными полями элементов связанного списка. Вместо этого необходимо просто включить структуру узла связанного списка в вашу структуру данных, чтобы можно было легко манипулировать данными и выполнять прохождение по связанному списку.

Работа со связанными списками

Для работы со связанными списками ядро предоставляет семейство функций. Все они принимают указатели на одну или более структур list head. Все функции выполнены как функции с подстановкой тела (inline) на языке C, и их все можно найти в файле <linux/list.h>.

Интересно, что время выполнения всех этих функций масштабируется как O (1)[1] Это значит, что они выполняются в течение *постоянного интервала времени* независимо от количества элементов списка, для которого они вызываются. Например, время добавления элемента в связанный список из 3 и 3000 элементов будет одинаковым. Это, возможно, и не вызывает удивления, но тем не менее, приятно.

Для добавления элемента в связанный список можно использовать следующую функцию.

```
list_add(struct list_head *new, struct list head *head)
```

Эта функция добавляет узел new в заданный связанный список сразу после элемента head. Поскольку связанный список является кольцевым и для него не существует понятий *первого* и *последнего* узлов, в качестве параметра head можно использовать указатель на любой элемент списка. Если в качестве рассмотренного параметра всегда передавать указатель на последний элемент, то эту функцию можно использовать для организации стека.

Для добавления элемента в конец связанного списка служит следующая функция.

```
list_add_tail (struct list_head *new, struct list_head *head)
```

Эта функция добавляет новый элемент new в связанный список сразу перед элементом, на который указывает параметр head. Поскольку связанный список является кольцевым, то, как и в случае функции list_add (), в качестве параметра head можно передавать указатель на любой элемент списка. Эту функцию можно использовать для реализации очереди, если передавать указатель на первый элемент.

Для удаления узла списка служит следующая функция.

```
list_del (struct list_head *entry)
```

Эта функция позволяет удалить из списка элемент, на который указывает параметр entry. Обратите внимание, что эта функция не освобождает память, выделенную под структуру данных, содержащую узел списка, на который указывает параметр entry. Данная функция просто удаляет узел из списка. После вызова этой функции обычно необходимо удалить структуру данных, в которой находится узел list_head.

Для удаления узла из списка и повторной инициализации этого узла служит следующая функция.

```
list_del_init(struct list head *entry)
```

[1] Вопросы сложности алгоритмов рассматриваются в приложении B.

Эта. функция аналогична функции list_del(), за исключением того, что она дополнительно инициализирует указанную структуру listhead из тех соображений, что эта структура данных больше не нужна в качестве элемента текущего списка и ее повторно можно использовать.

Для перемещения узла из одного списка в другой предназначена следующая функция.

```
list_move(struct list_head *list, struct list_head *head)
```

Эта функция удаляет элемент list из одного связанного списка и добавляет его в другой связанный список после элемента head.

Для перемещения элемента из одного связанного списка в конец другого служит следующая функция.

```
list_move_tail (struct list_head *list, struct list_head *head)
```

Эта функция выполняет то же самое, что и функция list_rnove (), но вставляет элемент перед элементом head.

Для проверки того, пуст ли список, служит функция.

```
list_empty (struct list_head *head)
```

Эта функция возвращает ненулевое значение, если связанный список пуст, и нулевое значение в противном случае.

Для объединения двух не перекрывающихся связанных списков служит следующая функция.

```
list_splice(struct list_head *list, struct list_head *head)
```

Эта функция вставляет список, на который указывает параметр list, в другой список после параметра head.

Для объединения двух не перекрывающихся списков и повторной инициализации старого головного элемента служит следующая функция.

```
list splice init(struct list head *list, struct list head *head)
```

Эта функция аналогична функции listsplice(), за исключением того, что параметр list, представляющий список, из которого удаляются элементы, повторно инициализируется.

Как избежать двух лишних разыменований

Если вам уже доступны указатели next и prev, то можно сэкономить пару процессорных тактов (в частности, время выполнения операций разыменования указателей) путем вызова внутренних функций работы со связанными списками. Все ранее рассмотренные функции в сущности не делают ничего, кроме получения указателей next и prev и вызовов внутренних функций. Внутренние функции имеют те же имена, что и их оболочки, но перед именем используется два символа подчеркивания. Вместо того чтобы вызвать функцию list_del(list), можно вызвать функцию __list_del(prev, next). Это имеет смысл, только когда указанные указатели уже известны. В противном случае просто получится некрасивый код. Для подробной информации об этих интерфейсах можно обратиться к файлу <linux/list.h>.

Перемещение по связанным спискам

Теперь мы уже знаем, как объявлять, инициализировать и работать со связанными списками в ядре. Это все хорошо, но не имеет никакого смысла, если нет возможности работать С данными, которые хранятся в списках! Связанный список — это просто контейнер, в котором хранятся важные данные. Необходимо иметь способ перемещения по списку и доступа к данным. К счастью, ядро предоставляет набор полезных интерфейсов для перемещения по связанным спискам и обращения к структурам данных, которые хранятся в этих списках.

Обратите внимание, что, в отличие от подпрограмм управления списками, операции перебора элементов списка из н узлов масштабируются как O (n).

Наиболее простой способ выполнять итерации по элементам связанного списка — это использовать макрос list_for_each (). Этот макрос принимает два параметра— указатели на структуры list_head. Первый параметр указывает на текущий элемент списка, а второй — на любой элемент списка, для которого необходимо обойти все узлы. На каждой итерации цикла первый параметр макроса указывает на текущий элемент списка, пока не будут пройдены все элементы, как в следующем примере.

```
struct list_head *p;
list_for_each(p, list) {
        /* p указывает на каждый элемент списка list */
}
```

Это пока все еще бесполезно! Указатель на структуру узла списка — это не то, что нам нужно. Нам нужен указатель на структуру данных, в которой содержится структура узла. В показанном ранее примере структуры данных my_struct необходимо получить указатель на каждый экземпляр структуры my_struct, а не на их поля list. Макрос listentry() возвращает структуру данных, которая содержит соответствующий элемент list_head. Этот макрос принимает три параметра: указатель на текущий узел, тип структуры данных, в которую включен узел списка, и имя поля структуры данных, в которой хранится этот узел.

```
struct list_head *p;
struct my_struct *my;

list_for_each(p, mine->list) {
        my = list_entry(p, struct my_struct, list);
        /*
        * указатель my указывает на все структуры данных,
        * в которые включено поле list
        */
}
```

Макрос list_for_each () раскрывается в обычный цикл for. Предыдущий пример раскрывается следующим образом,

```
for (p = mine->list->next; p != mine->list; p = p->next)
```

Кроме этого, макрос list_for_each() также выполняет предварительную загрузку (prefetch) данных в память, если процессор поддерживает такую возможность,

чтобы все данные следующих элементов списка гарантированно находились в памяти. Когда нет необходимости выполнять предварительную загрузку, можно использовать макрос list_for_each(), который работает в точности, как цикл for. Если нет гарантии, что список содержит очень мало элементов или пустой, то всегда необходимо использовать версию с предварительной загрузкой. Никогда нельзя программировать цикл вручную, необходимо всегда использовать макрос.

Если необходимо выполнить прохождение по спискам в обратном порядке, то следует использовать макрос list_for_each_prev (), который использует для прохождения указатель prev, а не указатель next.

Обратите внимание, что при прохождении связанного списка ничто не мешает удалять элементы из этого списка. Обычно, чтобы предотвратить конкурентный доступ, следует использовать блокировки. Макрос list_for_each_safe() использует временные переменные, чтобы сделать прохождение списка безопасным при одновременном удалении элементов.

```
struct list_head *p, *n;
struct my_struct *my;

list_for_each_safe(p, n, &mine->list) {
        my = list_entry(p, struct my_struct, list);
        /*
        * указатель my указывает на каждый экземпляр
        * структуры my_struct в списке
        */
}
```

Обратите внимание, что этот макрос защищен *только* от операций удаления узлов списка. Для защиты отдельных элементов списка от конкурентного доступа необходимо использовать блокировки.

Б

Генератор случайных чисел ядра

В ядре Linux реализован генератор случайных чисел, который теоретически может генерировать *истинно случайные числа*. Генератор случайных чисел собирает в *пул энтропии* шумы внешней среды, которые поступают из драйверов устройств. Этот пул доступен как в ядре, так и для пользовательских процессов в качестве источника данных, которые не только случайны внутри системы, но и недетерминированы для внешних источников атак. Такие случайные числа используются различными внешними приложениями, особенно для целей криптографии.

Истинно случайные числа отличаются от псевдослучайных чисел, которые генерируются библиотечными функциями языка С. Псевдослучайные числа создаются с помощью детерминированных функций. Хотя такие функции и могут генерировать последовательности чисел, которые обладают некоторыми свойствами истинно случайных чисел, тем не менее такие числа только статистически случайны. Псевдослучайные числа являются детерминированными, потому что если известно хотя бы одно число последовательности, то можно определить и все остальные. Если известно так называемое *порождающее число* последовательности *(seed),* то обычно по нему определяется и вся последовательность. Для приложений, которые требуют истинно случайных чисел, как, например, криптография, псевдослучайные числа обычно не подходят.

В отличие от псевдослучайных чисел, истинно случайные числа не зависят от той функции, которая используется для их генерации. Более того, если известен некоторый член последовательности истинно случайных чисел, то внешний наблюдатель не сможет определить, какие числа будет выдавать генератор в будущем, т.е. такой генератор - недетерминированный.

Физический термин *энтропия*— это мера беспорядка и случайности в любой системе. Энтропия измеряется в единицах энергии на единицу температуры (Джоуль на градус Кельвина). Когда Клод Шеннон (Claude Shennon)[1], создатель информационной теории, искал термин для представления случайности информации, великий

[1] Клод Шеннон (30 апреля 1916-24 февраля 2001) работал инженером в компании Bell Labs. В его наиболее известной работе Математическая теории связи, опубликованной в 1948 году, впервые были разработаны основы информационной теории и было введено понятие энтропии Шеннона. Шеннон также любил кататься на одноколесном велосипеде.

математик Джон фон Нейман (John von Neumann)[2] предложил ему использовать термин энтропия, потому что никто толком не понимает, что за этим понятием кроется. Шеннон согласился, и сегодня это звучит как *энтропия Шеннона*. Некоторые ученые считают, что такое двойное название только вносит путаницу, и когда речь идет об информации, то используют термин *неопределенность*. Разработчики ядра, наоборот, считают, что "энтропия"- это "круто", и поддерживают использование данного термина.

При рассмотрении генераторов случайных чисел понятие энтропии Шеннона является очень важным. Эта характеристика измеряется в битах на символ. Высокое значение энтропии означает, что в последовательности символов мало полезной (точнее, предсказуемой) информации и много случайного "мусора". Ядро поддерживает пул энтропии, который пополняется данными, возникающими в результате недетерминированных событий, связанных с аппаратными устройствами. В идеале, этот пул содержит полностью случайные данные. Для того чтобы иметь представление о значении энтропии пула, ядро постоянно вычисляет меру неопределенности данных в пуле. По мере того как ядро добавляет данные в пул, оно оценивает меру случайности добавляемых данных. И наоборот, по мере того как данные извлекаются из пула, ядро уменьшает значение оценки энтропии. Соответствующая количественная характеристика называется *оценкой энтропии*. Если значение оценки энтропии становится равным нулю, то ядро может отказаться выполнять запрос по считыванию данных из пула.

Генератор случайных чисел ядра был предложен в версии 1.3.30 и находится в файле drivers/char/random.c.

Принцип работы и реализация

Компьютеры— это предсказуемые устройства. Действительно, трудно найти случайное поведение в системе, поведение которой можно практически полностью программировать. Однако окружающая среда, где находится машина, полна различных шумов, которые недетерминированы и которые можно измерить. Источники таких шумов включают моменты времени, в которые возникают события, связанные с аппаратными устройствами, а также события, связанные с взаимодействием пользователей и компьютера. Например, интервалы времени между нажатиями клавиш, перемещения мыши, интервалы времени между некоторыми типами прерываний и время выполнения запроса блочного ввода-вывода являются недетерминированными, и, кроме того, их не может измерить внешний злоумышленник. Случайная информация, которая получается из этих событий, записывается в пул энтропии. Пул растет и заполняется случайными и непредсказуемыми шумовыми данными. По мере добавления данных в пул вычисляется оценка энтропии, и итоговое значение запоминается. Это позволяет всегда иметь информацию о значении энтропии в пуле. На рис. Б.1 показана диаграмма прохождения потока энтропии в пул и из пула.

[2] Джон фон Нейман (28 декабря 1903-8 февраля 1957) работал в Институте специальных исследований в Принстоне (Institute for Advanced Study; Princeton). Он внес большой вклад в математические, экономические и компьютерные науки. Среди наиболее значительных его разработок- теория игр, фон-неймановские алгебры и фон-неймановская проблема узких мест.

Для доступа к пулу энтропии, как из пространства ядра, так и из пространства пользователя, ядро предоставляет набор интерфейсов. Когда выполняется обращение к этим интерфейсам, ядро вначале вычисляет хеш-значение *SHA* данных из пула. Алгоритм SHA (Secure Hash Algorithm, алгоритм вычисления безопасного хеш-значения) — это алгоритм вычисления дайджеста сообщения (профиля сообщения, message digest), который был разработан Агентством национальной безопасности (National Security Agency, NSA) и утвержден в качестве федерального стандарта США Национальным институтом стандартов и технологий (NIST) (федеральный стандарт по обработке информации, FIPS 186). Вычисление дайджеста сообщения выполняется с помощью специального алгоритма, который принимает сообщение переменного размера (большое или маленькое) и выдает на выходе хеш-значение фиксированного размера (обычно размером 128 или 160 байт). Это фиксированное значение и представляет собой дайджест. Входное сообщение не может быть реконструировано по его хеш-значению. Более того, простое изменение входного сообщения (например, изменение одного символа) приведет к радикальному изменению хеш-значения. Алгоритмы вычисления дайджестов сообщений могут использоваться по-разному, включая проверку подлинности данных и дактилоскопию. Другие алгоритмы вычисления дайджестов — это MD4 и MD5. Пользователю возвращается хеш-значение SHA пула, к содержимому пула энтропии непосредственно обращаться нельзя. Считается, что по хеш-значению невозможно получить никакую информацию о состоянии пула. Поэтому если известно несколько значений из пула, то это не дает никакой информации о прошлых и будущих значениях. Ядро может использовать оценку энтропии и отказаться выполнить запрос на считывание данных из пула, если значение энтропии равно нулю. По мере того как из пула считываются данные, оценка энтропии уменьшается. Это реакция на то, что о пуле становится известно больше информации.

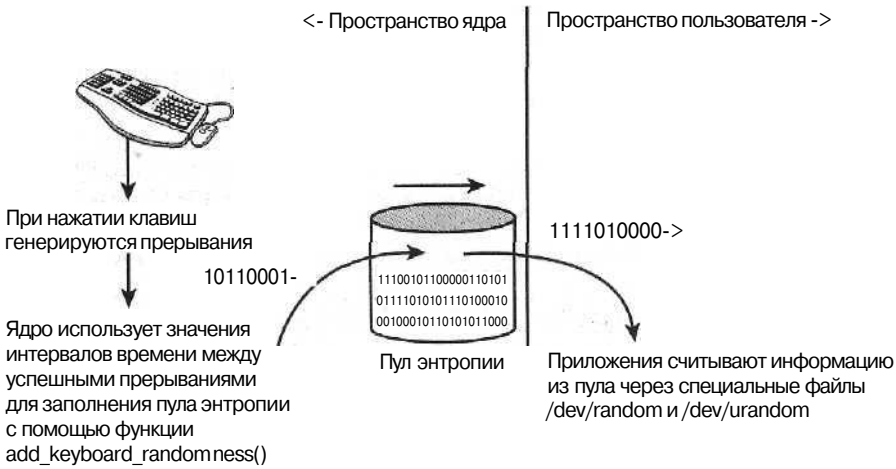


Рис. Б.1. *Прохождение энтропии через пул энтропии ядра*

Когда значение оценки энтропии достигает нуля, то ядро все равно может возвращать случайные числа. Однако в этом случае теоретически появляется возможность того, что злоумышленник сможет предугадать результат вывода. Для этого требуются все результаты вывода из пула энтропии, а также чтобы злоумышленник смог

выполнить криптографический анализ алгоритма SHA. Так как алгоритм SHA считается безопасным, то это невозможно. Для высоконадежной криптографии оценка энтропии позволяет гарантировать устойчивость случайных чисел. Для большинства пользователей такая дополнительная гарантия не нужна.

Почему это реализовано в ядре?

Критерием того, что какую-либо возможность необходимо реализовать в ядре, является сложность реализации этой возможности в пространстве пользователя. Недопустимо вводить что-либо в ядро только потому, что мы это можем сделать. Может показаться, что генератору случайных чисел и пулу энтропии не место в ядре. Однако существует, по крайней мере, три причины, по которым они должны быть в ядре. Во первых, генератору необходим доступ к системным событиям, таким как прерывания и ввод данных пользователями. Для обеспечения доступа к информации об этих событиях из пространства пользователя необходимо экспортировать специальные интерфейсы, чтобы информировать пространство пользователя о том, что эти события произошли. Даже если эти данные будут экспортироваться, то доступ к ним будет не простым и не быстрым. Во-вторых, генератор случайных чисел должен быть безопасным. Хотя такая система и может выполняться с правами пользователя root, тем не менее ядро является значительно более безопасным местом для пула энтропии. И наконец, самому ядру также необходимы случайные числа. Получать информацию о случайных числах, которая необходима ядру, из пространства пользователя — это не практично. В связи с этим генератор случайных чисел работает в ядре.

Проблема с загрузкой системы

Когда ядро загружается, оно выполняет последовательность действий, которые гарантированно можно предугадать. Следовательно, злоумышленник может предсказать состояние пула энтропии на этапе загрузки. Еще хуже то, что каждая загрузка очень похожа на все остальные и пул инициализируется при каждой загрузке в очень близкие значения. Это уменьшает точность оценки энтропии, потому что нет никакого способа обеспечить, чтобы энтропия, которая добавляется на этапе загрузки, была менее предсказуема, чем энтропия, которая добавляется при такой же загрузке в другое время.

Для решения проблемы большинство Linux-систем сохраняет на диске содержимое части пула энтропии между перегрузками системы. При старте системы сохраненные данные считываются и записываются в пул энтропии. Загрузка предыдущего содержимого пула в текущий пул позволяет обойтись без увеличения оценки энтропии.

Таким образом, злоумышленник не может предугадать состояние пула энтропии, не зная одновременно *предыдущего* и текущего состояний системы.

Интерфейсы для ввода энтропии

Ядро экспортирует следующее семейство интерфейсов, которые могут использоваться драйверами и системами для ввода данных в пул энтропии.

```
void add_interrupt_randomness(int irq)
void add_keyboard_randomness (unsigned char scancode)
void add_mouse_randomness(__u32 mouse_data)
```

Функция add_interrupt_randomness () вызывается системой обработки прерываний, когда приходит прерывание, обработчик которого зарегистрирован с флагом SA_SAMPLE_RANDOM. Параметр irq — это номер прерывания. Генератор случайных чисел использует интервалы времени между прерываниями, как источник шума. Следует помнить, что не все устройства для этого подходят. Если устройства генерируют прерывания детерминированным образом (например, прерывания таймера) или на них может воздействовать внешний злоумышленник (например, сетевые устройства), то такие устройства нельзя использовать для ввода информации в пул. Подходящее устройство — жесткий диск, который генерирует прерывания с непредсказуемой частотой.

Функция add_keyboard_randomness () использует скан-коды и интервалы времени между нажатиями клавиш для ввода энтропии в пул. Интересно, что эта функция достаточно интеллектуальна и игнорирует повторение символов при постоянном нажатии клавиши, потому что повторяющиеся скан-коды и интервалы времени вносят мало энтропии.

Функция add_mouse_randoraness () использует позицию указателя мыши и интервалы времени между прерываниями для заполнения пула. Параметр mouse_data — это позиция указателя, которая возвращается аппаратным обеспечением.

Эти три функции добавляют передаваемые данные в пул энтропии, вычисляют оценку энтропии добавляемых данных и увеличивают оценку энтропии пула на вычисленное значение.

Все эти экспортируемые интерфейсы используют внутреннюю функцию add_timer_randomness () для ввода данных в пул. Эта функция вычисляет интервалы времени между успешными событиями одного типа и добавляет эти значения в пул. Например, интервалы времени между успешными прерываниями жесткого диска достаточно случайны, особенно если измерять достаточно точно. Самые младшие биты — это обычно электрический шум. После того как эта функция вводит данные в пул, она вычисляет количественную характеристику того, насколько эти данные случайны. Это делается путем вычисления отклонения первого, второго и третьего порядка от предыдущего момента времени и изменения этих отклонений первого, второго и третьего порядка. Наибольшее из этих отклонений, округленное до 12 бит, используется в качестве оценки энтропии.

Интерфейсы для вывода энтропии

Для получения случайных чисел внутри ядра экспортируется один интерфейс.

```
void get_random_bytes(void *buf, int nbytes)
```

Эта функция сохраняет nbytes случайных байтов в буфере памяти, на который указывает параметр buf. Функция возвращает данные, даже если оценка энтропии равна нулю. Для ядра это не так критично, как для пользовательских криптографических программ. Случайные данные мало используются в ядре, в основном они нужны сетевой подсистеме для генерации стартового номера последовательности сегментов при соединении по протоколу TCP.

Код ядра может выполнить следующий код для получения случайных данных размером в одно машинное слово.

```
unsigned long rand;

get_random_bytes(&rand, sizeof(rand));
```

Для программ, которые выполняются в пространстве пользователя, предоставляется два символьных устройства: /dev/random и /dev/urandom. Первое устройство, /dev/random, используется, когда необходимы гарантированно случайные данные для криптографических приложений с высоким уровнем безопасности. Это устройство выдает только то количество битов данных, которое соответствует оценке энтропии в ядре. Когда оценка энтропии становится равной нулю, операция чтения устройства /dev/random блокируется и не возвращает данные, пока значение энтропии не станет существенно положительным. Устройство /dev/urandom не имеет последней возможности, а в остальном работает аналогично. Оба устройства возвращают данные из одного и того же пула.

Чтение из обоих файлов выполняется очень просто. Ниже показана функция пользовательской программы, которая служит для считывания одного машинного слова случайных данных.

```
unsigned long get_random(void)
(
 unsigned long seed = 0;
 int fd;


        fd = open("/dev/urandom", O_RDONLY);
        if (fd == -1) {
             perror("open");
             return 0;
        }
        if (read (fd, &seed, sizeof(seed)) < 0) {
             perror("read");
             seed = 0;
        }
        if (close(fd))
             perror("close");

        return seed;
}
```

Можно также считать $bytes байтов в файл $file, используя программу del.

dd if=/dev/urandom of=$file count=l bs=$bytes

B

Сложность алгоритмов

В компьютерных и связанных с ними дисциплинах полезно выражать сложность, или *масштабируемость,* алгоритмов с помощью количественных значащих характеристик (в отличие от менее наглядных характеристик, таких как быстрый или медленный). Существуют различные методы представления масштабируемости. Один из наиболее часто используемых подходов — это исследование асимптотического *поведения алгоритмов*. Асимптотическое поведение — это поведение алгоритма при достаточно больших значениях входных параметров или, другими словами, при *стремлении* входных параметров *к бесконечности*. Асимптотическое поведение показывает, как масштабируется алгоритм, когда его входные параметры принимают все большие и большие значения. Исследование масштабируемости алгоритмов, т.е. изучение свойств алгоритма при больших значениях входных параметров, позволяет смоделировать поведение алгоритма по отношению к тестовым задачам и лучше понять особенности этого поведения.

Алгоритмы

Алгоритм — это последовательность действий, возможно, с одним входом или более и, в конечном счете, с одним результатом или выходом. Например, подсчет количества людей в комнате представляет собой алгоритм, для которого люди, находящиеся в комнате, являются входными данными, а количество людей в комнате — выходными данными. Операции замещения страниц в ядре Linux или планирование выполнения процессов — это тоже примеры алгоритмов. Математически алгоритм аналогичен функции (или, по крайней мере, может быть смоделирован с помощью функции). Например, если мы обозначим алгоритм подсчета людей в комнате буквой f, а количество людей, которых необходимо посчитать, буквой x, то функцию подсчета количества людей можно записать следующим образом.

$y=f(x)$

В этом выражении буквой y обозначено время подсчета количества людей в комнате.

Множество O

Полезным обозначением асимптотического поведения функции является верхняя граница — функция, значения которой всегда больше значений изучаемой функции. Говорят, что верхняя граница некоторой функции растет быстрее, чем рассматриваемая функция. Специальное обозначение "большого-О" используется для описания этого роста. Это записывается как f (x) $\in$ O (g (x}) и читается так: f принадлежит множеству "О-болыпого" от g. Формальное математическое определение имеет следующий вид.

Если f(x) принадлежит множеству большого O(g(x)), то
$\exists c$ и x', такие что $f(x) \leq c \cdot g(x)$, $\forall x > x'$

Это означает, что время вычисления функции f(x) всегда меньше времени вычисления функции g (x), умноженного на некоторую константу, и это справедливо всегда, для всех значений x, больших некоторого начального значения x'.

Другими словами, мы ищем функцию, которая ведет себя не лучше, чем наш алгоритм в наихудшей ситуации. Можно посмотреть на результаты того, как ведет себя функция при очень больших значениях входных параметров, и понять, как ведет себя алгоритм.

Множество большого-тета

Когда говорят об обозначении большого-О, то чаще всего имеют в виду то, что Дональд Кнут (Donald Knulh) описывал с помощью обозначения "большого-тета". Обозначение "болыпого-О" соответствует верхней границе. Например, число 7 — это верхняя граница числа 6, кроме того, числа 9, 12 и 65— это тоже верхние границы числа 6. Когда рассматривают рост функции, то обычно наиболее интересна *наименьшая верхняя граница* или функция, которая моделирует как верхнюю, так и нижнюю границу'. Профессор Кнут описывает это с помощью обозначения большого-тета следующим образом.

Если f (x) принадлежит множеству большого-тета от g (x), то
g(x) является одновременно и верхней и нижней границей f(x)

Можно также сказать, что функция f (x) *порядка* функции g (x). Порядок, или множество "большого-тета" алгоритма, — один из наиболее важных математических инструментов изучения алгоритмов.

Следовательно, когда говорят об обозначении большого-О, то чаще всего имеют в виду наименьший возможный вариант "большого-О" — "болыное-тета". Об этом не нужно особо волноваться, если, конечно, нет желания доставить удовольствие профессору Кнуту.

[1] Если интересно, то нижняя граница описывается с помощью обозначения большого-омега. Определение аналогично определению множества большого-О, за исключением того, что значения функции g(s) должны быть меньше значений функции f (x) или равны им.

Объединяем все вместе

Вернемся снова к подсчету количества людей в комнате. Допустим, что можно считать по одному человеку за секунду. Следовательно, если в комнате находится 7 человек, то подсчет займет 7 секунд. Очевидно, что если будет n человек, то подсчет всех займет n секунд. Поэтому можно сказать, что этот алгоритм масштабируется, как O(n). Что если задача будет состоять в том, чтобы станцевать перед всеми, кто находится в комнате? Поскольку, независимо от того, сколько человек будет в комнате, это займет одно и то же время, значит, этот алгоритм масштабируется, как O(1). В табл. B.1 показаны другие часто встречающиеся характеристики сложности.

Таблица B.1. Значения масштабируемости алгоритмов во времени

O(g(x))	Название
1	Постоянная (отличная масштабируемость)
log(n)	Логарифмическая
n	Линейная
n^2	Квадратичная
n^3	Кубическая
2^n	Показательная, или экспоненциальная (плохо)
n!	Факториал (очень плохо)

Как масштабируется алгоритм представления всех людей в комнате друг другу? Какая функция может промоделировать этот алгоритм? Для представления одного человека необходимо 30 секунд, сколько времени займет представление 10 человек друг другу? Что будет в случае 100 человек?

Опасность, связанная со сложностью алгоритмов

Очевидно, что будет разумным избегать алгоритмов, которые масштабируются, как O(n!) или $O(2^n)$. Более того, замена алгоритма, который масштабируется, как O(n), алгоритмом, который масштабируется, как O(1), — это обычно серьезное улучшение. Тем не менее это не всегда так, и нельзя принимать решение вслепую, базируясь только на описании "болыпого-О". Вспомните, что в определении множества O(g(x)) фигурирует константа, на которую умножается значение функции g(x). Поэтому есть возможность, что алгоритм, который масштабируется, как O(1), будет выполняться в течение 3 часов. Следовательно, он будет выполняться всегда в течение 3 часов, независимо от количества входных данных, но это может оказаться дольше, по сравнению с алгоритмом, который масштабируется, как O(n), при небольшом количестве входных данных. При сравнении алгоритмов необходимо всегда принимать во внимание количество входных данных. Не стоит слепо оптимизировать для некоторого случайно выбранного варианта.

Г

Библиография и список литературы

Список литературы отсортирован и содержит некоторые из самых интересных и полезных книг, которые близки по теме, или дополняют материал данной книги.

Полезность этих книг проверена временем. Некоторые из них представляют собой "священные писания" но соответствующим темам, в то время как другие просто кажутся автору интересными, глубокими или занимательными. Автор надеется, что читателю они тоже окажутся полезными.

Наилучшая ссылка на "дополнительное чтение", которая лучше всего дополняет материал данной книги — это исходный код ядра. Для работы с ОС Linux у нас есть неограниченный доступ к полному исходному коду ядра современной операционной системы. Не принимайте это как должное! Разберитесь с ним! Читайте код! Пишите код!

Книги по основам построения операционных систем

В этих книгах рассмотрены принципы работы операционных систем в объеме учебных курсов. В них описываются основные понятия, алгоритмы и проблемы, связанные с построением высокофункциональных операционных систем, а также решения указанных проблем. Все эти книги могут быть рекомендованы, но если нужно выделить одну, то это, конечно, книга H. Deitel.

- Deitel H., Deitel P. and Choffnes D. *Operating Systems.* Prentice Hall, 2003. Прекрасная книга по теории операционных систем с отличными примерами из теории и практики. Автор помогал в техническом редактировании этой книги, что, может быть, и является причиной его предвзятого отношения, но все же хочется верить, что от этого книга стала значительно лучше.
- Tanenbaum Andrew. *Operating Systems: Design and Implementation*. Prentice Hall, 1997. Хорошие начальные сведения об оснопах построения, принципах работы и реализации Unix-подобной операционной системы Minix.

- Tanenbaum Andrew. *Modern Operating Systems.* Prentice Hall, 2001. Детальный обзор стандартных проблем разработки операционных систем, а также обсуждение многих концепций, которые используются в современных операционных системах, таких как Unix и Windows.
- Silberschatz A., Galvin P. and Gagne G. *Operating System Concepts.* John Wiley and Sons, 2001. Также известна, как "книга про динозавров", в связи с тем что на обложке нарисованы динозавры, которые не имеют никакого отношения к теме. Хорошее введение в основы построения операционных систем. Книга часто перерабатывается, но все издания должны быть хорошими.

Книги о ядрах Unix

В этих книгах описываются принципы работы и особенности реализации ядер Unix. В первых пяти рассмотрены конкретные варианты Unix, в двух последних — общие моменты всех вариантов Unix.

- Bach Maurice. *The Design of the Unix Operating System.* Prentice Hall, 1986. Обсуждение особенностей построения операционной системы Unix System V, Release 2.
- McKusick M., Bostic K., Karcls M. and Quarterman J. *The Design and Implementation of the 4.4BSD Operating System.* Addison-Wesley, 1996. Описание особенностей построения и реализации операционной системы 4.4BSD от разработчиков этой системы.
- McKusick M. and Neville-Neil G. *The Design and Implementation of the FreeBSD Operating System.* Addison-Wesley, 2004. Основы построения операционной системы FreeBSD 5.
- Mauro J. and McDougall R. *Solaris Internals: Core Kernel Architecture.* Prentice Hall, 2000. Интересное обсуждение основных подсистем и алгоритмов работы ядра ОС Solaris.
- Cooper C. and Moore C. *HP-UX Lli Internals.* Prentice Hall, 2004. Обзор внутреннего устройства операционной системы HP-UX аппаратной платформы PA-RISC.
- Vahalia, Uresh. *Unix Internals:The New Frontiers.* Prentice Hall, 1995. Отличная книга о возможностях современных Unix-подобных операционных систем, включая управление потоками и вытеснением кода в режиме ядра.
- Schimmel Curt. *UNIX Systems for Modern Architectures: Symmetric Multiprocessing and Caching for Kernel Programmers.* Addison-Wesley, 1994. Прекрасная книга о проблемах поддержки современных аппаратных платформ современными Unix-подобными операционными системами.

Книги о ядрах Linux

В этих книгах, как и в текущей, рассказывается о ядрах Linux.

- Rubini A. and Corbet J. *Linux Device Drivers.* O'Reilly and Associates, 2001. Прекрасная книга о том, как писать драйверы устройств для ядер Linux серии 2.4.

- Bovet D. and Cesati M. *Understanding the. Linux Kemel* O'Reilly and Associates, 2002. Обсуждение основных алгоритмов работы ядер Linux серии 2.4. Основное внимание уделено основополагающим принципам функционирования ядра.
- Mosberger D. and Eranian S. *IA-64 Linux Kernel: Design and Implementation.* Prentice Hall, 2002. Отличная книга, посвященная аппаратной платформе Intel Itanium и ядру Linux серии 2.4 для этой аппаратной платформы.

Книги о ядрах других операционных систем

Понимать врагов, точнее не врагов, а конкурентов, — никогда не повредит. В этих книгах обсуждаются основы работы и особенности реализации операционных систем, отличных от операционной системы Linux. Смотрите, что у них хорошо, а что — плохо.

- Kogan M. and Deitel H. *The Design of OS/2.* Addison-Wesley, 1996. Интересный обзор операционной системы OS/2 2.0.
- Solomon D. and Russinovich M. *Inside Windows 2000.* Microsoft Press, 2000. Интересный взгляд на операционную систему, которая чрезвычайно отличается от Unix.
- Richter Jeff. *Advanced Windows.* Microsoft Press, 1997. Описание низкоуровневого и системного программирования под ОС Windows.

Книги по API Unix

Детальное описание системы Unix и API этой операционной системы важно не только для того, чтобы писать мощные прикладные программы, но и для понимания того, что требуется от ядра.

- Stevens W. Richard. *Advanced Programming in the UNIX Environment.* Addison-Wesley, 1992. Отличное, если не самое полное, обсуждение интерфейса системных вызовов Unix.
- Stevens W. Richard. *UNIX Network Programming, Volume 1.* Prentice Hall, 1998. Классический учебник по API сокетов операционной системы Unix.
- Johnson M. and Troan E. *Linux Application Development.* Addison-Wesley, 1998. Общий обзор операционной системы Linux и интерфейсов, которые специфичны для этой операционной системы.

Другие работы

Книги, которые не посвящены операционным системам, но имеют к ним прямое отношение.

- Knuth Donald. *The Art of Computer Programming, Volume 1.* Addison-Wesley, 1997. Бесценный курс по фундаментальным алгоритмам и теории вычислительных систем, который включает лучшие и не самые лучшие алгоритмы управления памятью. (Имеется русский перевод: Кнут Дональд Эрвин. Искусство программирования. Том 1. Основные алгоритмы, 3-е издание. - М: "Вильяме", 2000.)

- Kernighan B. and Ritchie D. *The C Programming Language.* Prentice Hall, 1988. Наилучшая книга по языку программирования С. (Имеется русский перевод: Брайан Керниган, Деннис Ритчи. *Язык программирования С*— М: "Вильяме", 2005 г.)
- Hofstadter Douglas. *Godel, Escher, Bach: An Eternal Golden Braid.* Basic Books, 1999. Глубокий взгляд на человечество через исследование различных предметов, включая компьютерные науки.

Web-сайты

Эти WWW-сайты предоставляют последние новости и другую информацию, связанную с операционной системой Linux и ее ядром.

- *Kernel Traffic.* Отличный обзор сообщений в списке рассылки разработчиков ядра Linux (lkml) за последнюю неделю (очень рекомендуется), http://www.kerneltraffic.org/
- *Linux Weekly News.* Хороший сайт новостей о том, что произошло касательно ядра Linux за последнюю неделю, с прекрасными комментариями (очень рекомендуется), http://www.lwn.net/
- *Kernel Newbies.* Сайт "Kernel Newbies" — это проект сообщества разработчиков с целью предоставления информации и помощи начинающим хакерам, которые стремятся заниматься разработкой ядра. http://www. kernelnewbies -org/
- *Kernel.org.* Официальный архив исходных кодов ядра. Здесь также находятся домашние страницы многих разработчиков основных подсистем ядра с соответствующими заплатами http://www.kernel.org/
- *KernelTrap.* Сайт, посвященный всему, что связано с ядром операционной системы, с большим уклоном в сторону ядра Linux. На сайте много информации о новостях и обзорах из области разработки ядра Linux. Здесь также в большом количестве размещаются интервью с ведущими разработчиками ядра. http://www.kerneltrap.org
- *OS News.* Новости об операционных системах, а также статьи, интервью и обзоры из этой же области. http://www.osnews.com/
- *Сайт, посвященный этой книге.* Новости, сообщения об ошибках и другая информация, которая касается этой замечательной книги. http://tech9.net/rral/kernel_book/

Предметный указатель

A

Application Programing Interface, API, 96

E

Exception, 99; 110

G

gec, 40
  аннотация ветвлений, 41
  встроенный ассемблер, 41
  функции с подстановкой тела, 40
Granularity, 174

L

Linux Kernel Mail List, 405
lkml, 405

P

POSIX, 96

S

SHA, 425
SMP-привязка, 71

T

Task, 46
Translation lookaside buffer, TLB, 329

V

Virtual memory area, 316
VMA, 316

А

Адресное пространство
  плоское, 311
  процесса, SI 1; 313
  сегментированное, 311
  структура
    address_space, 297; 333
    address_space_operations, 334
Алгоритм, 429
Аппаратная платформа, 391
Атомарные операции, 177
  битовые, 181
    change_bit(), 182
    clear_bit(), 182
    set_bit(), 182
    test_and_change_bit(), 182
    test_and_clear_bit(), 182
    test_and_set_bit(), 182
    test_bit(), 182
  отладка, 381
  тип atomic_t, 178
  целочисленные, 178
    atomic_add(), 180
    atomic_add_negative(), 180
    atomic_dec_and_test(), 180
    atomic_inc(), 180
    atomic_inc_and_test(), 180
    atomic_read(), 180
    atomic_set(), 180
    atomic_sub(), 180
    atomic_sub_and_test(), 180

Б

Бинарное дерево, 314
  базисное, 335
  красно-черное, 320
Блок, 294
Блокировки
  advisory, IBS
  dcache_lock, 282
  deadlock, 172
  deadly embrace, 172
  dentry->d_lock, 282
  inode_lock, 273
  lock contention, 174
  mmlist_lock, 314
  page_table_lock, 329
  sell-deadlock, 172
  voluntary, 168
  xtime_lock, 219
  большая блокировка ядра (BKL), 197
    lock_kernel(), 198
    unlock_kernel(), 198
    захват, 198
    освобождение, 198
  взаимоблокировка, 172
    типа ABBA, 172
  в обработчиках нижних половин, 159
  защита данных, 186
  конфликт при захвате, 168; 174
  навязываемые, 168
  необязательные, 168
  обязательные, 168
  порядок захвата, 173
  преемптивность, 184
  применение, 167
  рекомендуемые, 168

рекурсивность, 185
самоблокировка, 172; 189
секвентные, 199; 222
read_seqbegin(), 199
read_seqretry(), 199
write_seqlock(), 199
write_sequnlock(), 199
захват на запись, 200
проверка чтения, 200
семафоры, 184; 190
DECLARE_MUTEX(), 193
DECLARE_RWSEM(), 195
down(), 192; 193
down_interruptible(), 193
down_read(), 195
down_write(), 195
init_MUTEX(), 193
init_rwsem(), 195
sema_init(), 193
up(), 192
up_read{), 195
up_write(), 195
бинарные, 192
захват, 194
захват на запись, 195
захват на чтение, 195
инициализация, 193; 194
инкремент, 192
использование, 196
мьютекс, 192
освобождение, 194
особенности, 191
осовобождение, 195
проверка, 192
реализация, 190
счетчик, 192
чтения-записи, 195
состояние конфликта, 183
спин-блокировки, 183
read_lock(), 188
read_unlock(), 188
spin_is_locked(), 186
spin_lock(), 184
spm_lockbh(), 187
spin_lock_init(), 186
spin_lock_irq{), 186
spin_lock_irqsave(), 185
spin_try_lock(), 186
spin_unlock(), 184
spin_unlock_bh(), 187
spin_unlock_irq{), 186
spin_unlock_irqrestore(), 185
write_lock(), 189
write_unlock(), 189
в обработчиках нижних половин, 187
записи, 188
захват, 187
захват на запись, 190
захват на чтение, 190
инициализация, 187; 190
использование,189; 196
освобождение, 187
отладка, 186; 381
преемптивность, 200
проверка, 190
проверка состояние, 187
чтения,188
структуры
completion, 196
rw_semaphore, 195
semaphore, 193
тупиковая ситуация, 172
условные переменные, 196
complete(), 197
wait_for_completion(), 197
Блочное устройство
RAID, 300
очередь запросов, 301
сегмент, 298
структура
bio, 298
bio_vec, 298
buffer_head, 296; 300
request, 301
request_queue, 301
Буфер, 295
быстрого преобразования адреса (TLB), 329
заголовок, 295
кольцевой,377
сообщений ядра, 377
состояние, 296

В

Ввод-вывод
страничный, 331
Виртуальный ресурс
память, 45; 311
процессор, 45
Возможность использования, 103
CAP_SYS_TIME, 223
Время
абсолютное (wall time), 207; 208; 221
начало эпохи (epoch), 222
операции
gettimeofday(),223
settimeofday(), 223
относительное, 207
time_after(), 216

time_after_eq(), 216
time_before(), 216
time_before_eq(), 216
работы системы (uptime), 208
часовой пояс (time zone), 223
Выравнивание, 396
естественное, 396
массивов, 397
нестандартных типов данных, 397
объединений, 397
переменных, 396
полей структур, 398
проблемы, 397
структур, 397
Вытеснение, 66; 164

Г

Генератор случайных чисел, 423
Головка, 295
Гранулярность, 174
cource, 174
fine, 174
уровень крупных структурных единиц, 174
уровень мелких структурных единиц, 174

Д

Дайджест сообщения, 425
Демон
klogd, 377
kupdated, 339
pdflush, 297; 337; 339
diity_background_ratio, 338
dirty_expire_centisecs, 338
dirty_rado, 338
dirty_writeback_centisecs, 338
laptop_mode, 338
предотвращения зависания, 340
syslogd, 377
Деннис Ритчи, 23
Джон фон Нейман, 424
Дональд Кнут, 430

З

Задание, 46
Задача, 46
Заплата
генерация, 412
представление, 413
применение, 413
утилиты
diff, 412
diffstat, 413
patch, 412
Заполнение структур, 398

И

Инсталляция
модулей, 38; 347
ядра, 38
Исключительная ситуация, 99; 110
Истинно случайные числа, 423

К

Кен Томпсон, 23
Кластер, 295
Клод Шеннон, 423
Код ядра
дерево, 34
заплаты, 34
инсталляция, 33
конфигурация, 35
make config, 36
make defconfig, 36
make gconfig, 36
make menuconfig, 36
make oldconfig, 37
make xconfig, 36
переменные, 35
файл .config, 36
получение, 33
сборка, 34
параллельная, 37
Константа
__BIG_ENDIAN,401
__LITTLE_ENDIAN,401
BITTS_PER_LONG, 392
HZ, 401
PAGE_SHIFT, 403
PAGE_SIZE, 403
Конфигурация
CONFIG_PREEMPT, 171
CONFIG_SMP, 171
Кэш
буферный, 336
дисковый, 331
страничный, 331

Л

Линус Торвальдс, 25

М

Масштабируемость, 71; 174
O(1),419;431
O(n),421;431
алгоритмов, 429
Машинное слово, 391
Многозадачность
preemptive, 66
вытесняющая, 66

кооперативная, 66
преемптивная, 66
Множество
большого-О, 430
болыного-тета, 430
Модуль
загружаемый, 343
depmod, 348
EXPORT_SYMROL(), 353
EXPORT_SYMBOL_GPL(), 353
insmod, 348
Makefile, 346
make modules_install, 347
modprobe, 348
MODULE_AUTHOR(),345
module_exit(), 344
module_init(), 344
MODULE_LICENSE(),345;354
module_param(), 351
module_param_array(), 353
module_param_array_named(), 353
module_param_named(), 352
module_param_string, 352
MODULE_PARM_DESC(),353
rmmod, 348
зависимости, 347
загрузка, 348
инсталляция, 347
конфигурация, 349
параметры, 351
разработка, 343
сборка, 345
экспорт символов, 353

H

Наименьшая верхняя граница, 430
Неатомарные операции
битовые, 182
find_first_bit(), 183
find_first_zero_bit(), 183
sched_find_first_bit(), 75
Нижние половины, 132; 187
bottom half, 134
]ocal_bh_disable(), 160
local_bh_enable(), 160
softirq, 135; 136
запрещение, 160
интерфейс BH, 134; 148
отложенное прерывание, 134; 169; 188
open_softirq(), 140
raise_softirq(), 140
выполнение, 137
вытеснение, 137
генерация,137
демон ksoftirqd, 138; 147
индекс, 139
использование, 139
отдообработчик, 137
регистрация, 140
тасклет, 141
структура
softirq_action, 136
tasklet_struct, 141; 144
тасклет, 134; 139; 169; 187
DECLARE _TASKLET(), 144
DECLARE_TASKLET_DISABLED(), 144
tasklet_disable(), 145
tasklet_disable_nosync(), 145
tasklet_enable(), 145
tasklet_hi_scbedule(), 142
tasklet_init(), 144
tasklet_kill(), 145
tasklet_schedule(), 142; 145
обработчик, 142
планирование, 142; 145
реализация, 141
создание,144

O

Область памяти, 311; 316
деревья, 320
интервал адресов, 317
операции, 319
close(),319
nopage(), 320
open(), 319
populate{),320
права доступа, 317
списки, 320
структура
vm_area_struct, 319
vm_operations_struct, 319
флаги, 317
защиты, 325
Объект
kobject, 356
kobject_get(), 361
kobject_init(), 360
kobject_put(), 361
kobject_set_name(}, 361
декларация, 360
захват, 361
инициализация, 360
освобождение, 361
присвоение имени, 361
счетчик ссылок, 361
Операционная система
Linux, 25
Multics, 23
Unix, 23

AT&T, 23
BSD, 24
особенности, 24
многозадачная,65
определение, 26
с виртуальной памятью, 311
Отладка
BUG(), 382
BUG_ON(),382
dump_stack(),382
атомарные операции, 381
генерация ошибок, 382
исследование и тестирование, 385
конфигурационные параметры, 381
магическая клавиша SysRq, 382
ограничение частоты событий, 387
утилита ksymoops, 380
функция kallsyms, 380
Отладчик
gdb,384
kdb,385
kgdb,385
Отложенное действие, 136
cancel_delayed_work(), 156
create_workqueue(), 156
DECLARE_WORK(), 154
flush_scheduled_work{), 155
flush_workqueue(), 156
INIT_WORK{), 154
queue_work{), 156
run_workqueue(), 152
schedule_delayed_work(), 155
schedule_work{), 155
work_handler(), 154
work queue, 149
демон keventd, 157
использование, 154
ожидание завершения, 155
очереди заданий, 134; 157
планирование, 155
рабочий поток, 150
реализация, 150
создание, 156
структура
cpu_workqueue_struct, 150
work_struct, 151
workqueue_struct, 150
Отображение
анонимное, 312; 325
выполняемого кода, 312
инициализированных переменных, 312
области ввода-вывода, 319
отладка, 381
совместно используемое, 318; 322
страницы памяти, заполненной нулями, 312
файла, 322; 325
частное, 318
Очередь ожидания
add_wait_queue(), 82
DECLARE_WAIT_O_UEUE_HEAD(), 82
remove_wait_queue(), 83

П

Память
MMU, Memory Management Unit, 233
адрес, 311
верхняя
отладка, 381
переносимость, 403
виртуальная, 234; 311
выделение
__alloc_percpu(), 261
__get_free_pages(), 238
alloc_page(),238
alloc_pages, 238
alloc_percpu(), 261
DEFINE_PER_CPU(), 260
get_zeroed_page(), 238
gfp_mask, 238; 241
kmalloc(), 240
kmap(), 257
kmap_atomic(), 258
page_address(), 238
vmalloc(),246
виртуально непрерывный участок, 247
временное отображение, 258
высокоуровневое, 240
контекст процесса, 245
модификаторы зоны, 242
модификаторы операций, 241
нижняя половина, 245
низкоуровневое, 238
обработчик прерывания, 245
отображение верхней памяти, 257
постоянное отображение, 257
связанная с процессором (per-CPU), 259
слябовый распределитель (slab layer, slab allocator), 248
списки свободных ресурсов, 248
стек ядра, 257
физически непрерывный участок, 238; 240; 246
флаги типов, 243
дескриптор, 313
выделение, 315
счетчик использования, 314
удаление, 315
защита, 41
зоны, 235

ZONE_DMA, 235
ZONE_HIGHMEM, 235
ZONE_NORMAL, 235
верхняя память (high memory), 234; 236
нижняя память (low memory), 236
кэширование, 263
область, 311
освобождение
_free_pages(),239
free_page(),239
free_pages(), 239
free_percpu(), 261
kfree(), 245
kunmap(), 258
kunmap_atomic(), 259
vftee(), 247
прямой доступ (ПДП, DMA), 235
связанная с процессором
get_cpu_ptr(), 262
get_cpu_var(), 260
put_cpu_ptr(), 262
put_cpu_var(), 260
слябовый распределитель, 47
kmem_cache_alloc(), 254
kmem_cache_create(), 252
kmem_cache_destroy{), 253
NUMA, 249
дескриптор сляба, 250
кэш, 249
отладка, 381
сляб (slab), 249
стек
проверка переполнения, 381
процесса, 312
ядра, 42; 256
страница, 233
page_count(), 234
виртуальный адрес, 234
вытеснение, 234
гигантская, 318
данные буфера, 297
измененная, 337
копирование при записи, 54
нулевая, 312
размер, 402
флаги, 234
страничный кэш, 235
структура
kmem_cache_s, 250
mm_struct, 313
page, 234;299
slab, 250
vm_area_struct, 316; 319
zone, 237
физическая, 234
Параллелизм
pseudo-concurrency, 169
race condition, 164
безопасность
при SMP-обработке, 170
при прерываниях, 170
защита данных, 170
истинный, 169
конкурентный доступ, 163
критический участок, 164
многопоточность, 57
многопроцессорность, 403
псевдопараллелизм,169
симметричная многопроцессорность, 163; 170
синхронизация, 164
состояние "гонок", 164
состояние конкуренции за ресурс, 42; 164
Переключение контекста, 87
Переносимость, 43; 389
Планирование
передача управления, 66
с динамическим управлением по приоритетам, 68
с управлением по приоритетам, 68
Планировщик, 65
0(1), 66; 71; 175
балансировка нагрузки, 83
битовая маска приоритетов, 74
ввода-вывода, 302
CFQ, 308
deadline, 304
noop, 309
задержки обслуживания, 305
лифтовый алгоритм, 303
объединение, 302
прогнозирующий, 307
слияние, 302
сортировка, 302
с лимитом по времени, 304
с отсутствием операций, 309
с полностью равноправными очередями, 308
массив приоритетов, 74
активный, 76
истекший, 76
очередь выполнения, 72; 174
реального времени, 89
стратегия, 67
SCHED_FIFO, 89
SCHED_RR, 90
структура
prio_array, 74
runqueue, 72
Подсистема, 358

Порядок выполнения, 180
барьер, 403
барьеры, 180; 202
barrier(), 204
mb(), 203
read_barrier_depends(), 203
rmb(), 202
smp_mb(),204
smp_rmb(), 204
smp_wmb(), 204
wmb(), 203
записи памяти, 203
компилятора, 204
чтения памяти, 202
переносимость, 403
Порядок следования
__be32_to_cpu(),401
__cpu_to_be32(),401
__cpu_to_le32(),401
__le32_to_cpus(),401
big-endian, 399
little-endian, 399
байтов,399
определение,400
обратный, 399
прямой,399
Поток, 45; 57; 315
пространства ядра, 59; 316
kernel_thread(), 59
Преемптивность, 164; 169
данные связанные с процессорами, 201
запрещение, 200
preempt_disable(), 201
preerapt_enable{), 201
переносимость, 403
счетчик preempt._count, 89; 160
Прерывание, 27; 109; 169; 427
/proc/interrupts, 123
do_IRQ(), 122
handler, 111
interrupt request line, 110
interrupt service routine, 111
IRQ, 110
ret_from_intr(), 123
верхняя половина, 111; 131
контекст, 111; 119
линия запроса, 110
нижняя половина, 111; 132
обработчик, 111
add_interrupt_randomness(), 122
free_irq(), 114
request_irq(), 112
RTC, 117
shared, 116
быстрый, 113
описание, 115
освобождение, 114
регистрация, 112
реентерабельность, 116
совместно используемый, 113; 116
реализация обработки, 121
управление, 124
cli(), 126
disable_irq(), 126
disable_irq_nosync(), 126
enable_irq(), 126
in_interrupt(), 127
in_irq(), 127
irq_disabled(), 127
local_irq_disable(), 125
local_irq_enable(), 125
local_irq_restore(), 125
local_irq_save(), 125
sti(), 126
synchronize_irq(), 126
запрещение, 125
разрешение, 125
флаг
SA_INTERRUPT, 113; 117
SA_SAMPLE_RANDOM, 113; 427
SA_SHIRQ, 113; 116
Пространство
задачи,27
пользователя,51
ядра, 27; 51
Процесс
I/O-bound, 67
ink, 52
parent, 52
processor-bound, 67
runnable, 65
timeslice, 66
wake_up(),83
адресное пространство, 51; 311
вытеснение, 70
пространства пользователя, 88
пространства ядра, 88
готовый к выполнению, 65
дескриптор, 46; 289
создание, 47
удаление, 61
завершение, 59
идентификатор, 48
иерархия, 52
квант времени, 66; 69; 209
контекст, 52; 104
корневой каталог, 290
макрос current, 49
не готовый к выполнению, 65
ограниченный скоростью ввода-вывода, 67

ограниченный скоростью процессора, 67
операции
wakc_up(), 232
определение, 45
парамтер
nice, 68
переназначение родительского процесса, 61
порожденный, 46
приоритет, 68; 78
пространство имен (namespace), 291 .
родительский, 46
создание, 53
состояние,5()
sel_current_stale(), 51
set_task_state(), 51
sleep, 81
TASK_UNTERRUFTIBLE,50; 81; 230
TASK_RUNNING, 50; 70
TASK_STOPPED, 51
TASK_UNINTERRUPTIBLE, 51; 81; 230
TASK_ZOMBIE, 51
ожидания, 81
заблокированное, 81
ожидания, 170
структура
task_struct, 46
thread_info, 47
текущий каталог. 290
трассировка, 61
флаг
need_resched, 83; 212
Псевдослучайные числа, 423
Пул энтропии, 423
операции
add_inlerrupl_randomness(), 426
add_keyboard_randomness(), 426
add_mouse_randomness(), 426
get_random_bytes{), 427

p

Режим ноутбука, 338

C

Сборка
модулей, 345
Связанный список, 320; 415
головной элемент, 417
двухсвязный, 415
инициализация, 418
кольцевой, 416
однослязный, 415
операции
__list_for_each(), 422
list_add(), 419
list_add_tail(),419
list_del(),419
list_del_init(),419
list_cmpty(), 420
list_cntry(), 421
list_for_each(),421
fist_for_each_prev(), 422
list_for_each_safe(), 422
list_move(), 420
list_move_tail(),420
iist_splice(),420
list_splice_init(), 420
перемещение, 416; 421
структура элемента, 417
Сегмент.
bss,312
данных, 312
кода, 312 ..
Сектор, 294
размер, 294
Система, 26
Системный вызов, 27
errno, 97
errnosys_call_table, 98
getpid(), 97
int $0x80, 99
ioctl(), 101; 368
mmap(), 326
mmap2(),, 326
munmapO, 327
sys_ni_syscall(), 98
syscall, 97
syscall(), 106
доступ из пространства пользователя, 106
модификатор asmlinkage, 98
номер, 98
обработчик, 101
передача параметров, 100
планировщика, 91
nice(),91
sched_get_priority_max{), 91
sched_get_priority_min(), 91; 92
sched_getaffinity(), 91
sched_getcheduler(), 91
sched_getparam(), 91
schcd_getscheduler(), 91 .
sched_rr_get_interval(), 91
sched_setaffinity(), 91
sched_setparam(), 91
sched_setscheduler(), 91
sched_yield(),91;92
производительность, 99
процессорной привязки
schcd_getaffinity(), 92
schcd_sctaffinity(), 92
реализация, 101

регистрация,104
файле entry.S, 98; 105
Сообщение
Oops, 378
уровень вывода, 376
Сообщество разработчиков, 32; 405
maintainers, 412
ответственные разработчики, 412
отправка сообщений об ошибках, 412
список рассылки, 32; 405
Сосредоточенность во времени, 331
Состояние
паники, 378
Средняя загруженность системы, 219
Стиль написания исходного кода, 406
ifdef, 410
indent, 411
typedef, 410
длинные строки, 407
имена, 408
инициализация структур, 411
комментарии, 408
отступы, 406
фигурные скобки, 406
функции,408
Страничный кэш, 331
структура
address_space, 332
address_space_operations, 334
Структура
attribute, 357; 366
cdev, 356
kobj_type, 357
kobject, 356
kref, 362
kref_get(), 362
kref_init(), 362
kref_put(), 362
kset, 358
list_head, 417
subsystem, 358
Структурность, 174

Т

Таблица страниц, 327
PGD, 328
PMD, 328
PTE, 328
глобальный каталог, 328
каталог среднего уровня, 328
управление, 329
уровни,328
Таймер, 139
APIC, 219
временная отметка (tick), 208
гранулярность, 211
декрементный счетчик, 218
динамический, 208; 223
обработчик, 226
задержки, 223
BogoMIPS, 229
mdelay(), 229
udelay(),229
короткие, 229
очередь ожидания, 232
с помощью цикла, 227
импульс (tick), 208
константа
HZ, 209
USER_HZ, 217
операции
add_timer(),225
del_timer(), 225
del_timer_sync(), 226
init_timer(),224
mod_timer(),225
schedule_timeout(), 230
переменная
jiffies, 225
jiffies_64, 214
xtime, 219; 221
jiffies, 213
переносимость, 401
переполнение, 215
прерывание, 207; 208; 219
программируемый интервальный (PIT), 218
системный, 207; 218
обработчик прерывания, 219
срабатывание, 208
структура
timer_list, 224
timespec, 222
счетчик отметок времени (TSC), 219
частота импульсов (tick rate), 208; 209
часы реального времени, 117; 218
ядра, 135; 223
Типы данных
char, 396
sl6, 395
s32, 395
s64, 395
s8, 395
u16, 395
u32, 395
u64, 395
u8,395
Трассировки
вызовов функций, 378

У

Упреждающее чтение, 319
Уровень блочного ввода-вывода, 294
Уровень событий
  kobject_uevent(), 370
  kobjcct_uevcnt_atomic{), 371
Устройство
  блочное, 293
  символьное, 293
  унифицированная модель представления, 355

Ф

Файл
  System.map, 380
Файловая система
  /proc, 363
  sysfs, 352; 363
    HAL, 365
    kobject_add(), 365
    kobject_del(),365
    kobject_init(), 365
    kobject_register(), 365
    kobject_unregister(), 365
    sysfs_create_file(), 367
    sysfs_crcate_link(), 367
    sysfs_remove_file(), 368
    sysfs_remove_link(), 368
    атрибуты, 366
    добавление файлов, 357; 366
    корневой каталог, 363
    операция show(), 367
    операция store(), 367
    соглашения, 368
    структура sysfs_ops, 366
    файлы, 366
  блок, 294
  виртуальная (VFS), 265
  диспетчер логических томов (LVM), 273
  добавление и удаление объектов, 365
  каталог (directory), 268
  метаданные (metadata), 268
  монтирование (mount), 288
    флаги mnt_flags, 289
  обобщенный интерфейс, 266
  объектная ориентированность, 269
  операции
    суперблок
      alloc_inode, 273
      clear_inode, 274
      delete_inode, 273
      destray_inode, 273
      dirty_inode, 273
      drop_inode, 273
      put_inode, 273
      put_super, 273
      read_inode, 273
      remount_fs, 274
      statfs, 273
      sync_fs, 273
      umount_begin, 274
      unlockfs, 273
      write_inode, 273
      write_super, 273
      write_super_lockfs, 273
    файл
      alo_fsync, 286
      aio_read, 285
      aio_write, 285
      check_flags, 287
      fasync, 286
      flush, 286
      fsync, 286
      get_unmapped_area, 287
      ioctl, 286
      llseek, 285
      lock, 286; 287
      mmap, 286
      open,286
      poll, 285
      read, 285
      readdir, 285
      readv, 286
      release, 286
      sendfile, 287
      sendpage, 287
      write, 285
      writev, 287
    файловый индекс
      create, 276
      follow_link, 277
      getattr, 278
      link, 277
      listxattr, 278
      lookup, 276; 278
      mkdir, 277
      mknod, 277
      permission, 278
      put_link, 277
      readlink, 277
      femovexattr, 278
      rename, 277
      rmdir, 277
      setattr, 278
      setxattr, 278
      symlink, 277
      truncate, 277
      unlink, 277
    элемент каталога
      d_compare, 282

d_delete, 282
d_hash, 282
d_iput, 282
d_release, 282
d_revalidate, 282
особенности Unix, 267
пространство имен, 55
пространство имен (namespace), 267; 291
путь (path), 268
системные вызовы, 266
структура
dentry, 279; 356
dentry_operations, 270; 281
file, 283
file_operations, 270; 284
file_struct, 270
file_system_type, 270; 288
files_struct, 289
fs_struct, 270; 289
inode, 274
inode_operations, 270; 276
namespace, 270; 289
super_block, 271
super_operations, 270; 272
vfsmount, 270; 288
суперблок (superblock), 268; 269; 270
управляющий блок (control block), 270
файловый индекс (inode), 268; 269; 274
кэш (icache), 281
файл (file), 267; 269; 283
элемент каталога (dentry), 268; 269; 279
LRU, 280
кэш(dcache),280
состояния, 280
хеш-таблица, 281
Функция
bread(), 332
clone(),54;57
commit_writc(), 335
context_switch(), 390
copy_from_user(), 102
copy_mm(), 315
copy_process(), 55
copy_to_user(), 102
dup_task_struct(), 55
early_printk(),376
exec(), 54
exit(),46;59
exit_mm(), 315
flnd_vma(), 323
find_VMA_intersection(), 324
find_vma_prev(), 324
fork(),46;54;315
madvice(), 319
mmap(),319;325
munmap(), 327
panic(), 382
prepare_wrife(), 335
printk(), 39; 375
readpage(), 335
release_task(), 61
schedule(),76;87;104
SetPageDirty(),335
switch_mm(), 390
switch_to(),390
unhash_process(), 61
vfork(),54;56
vma_link(), 326
wakeup_bdflush(), 337
wb_kupdate(), 338

Х

Хеш-таблица страниц, 336

Ц

Цилиндр, 295

Э

Энтропия, 423
оценка, 424
Шеннона, 424

Я

Ядро
Linux, 30
версии, 31
разрабатываемое, 31
стабильное, 31
схема присваивания имен, 31
tainted, 345
Unix, 29
дефекты, 374
микроядро, 29
модульное, 343
монолитное, 29; 343
особенности,38
отладка, 373
уровень событий, 369
экзоядро,29

Научно-популярное издание

Роберт Лав

Разработка ядра Linux
2-е издание

Литературный редактор *Е.Д. Давидян*
Верстка *Т.Н. Артемеико*
Художественный редактор *СЛ. Чернокозинский*
Корректоры *Л.А. Гордиенко, А.В. Луценко, О.В. Мишутина, В.В. Столяр*

Издательский дом "Вильяме"
101509, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 27.07.2006. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 36,12. Уч.-изд. л. 28,86.
Тираж 3000 экз. Заказ № 2169.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Разработка ядра Linux

Второе издание

Эта книга посвящена основным принципам функционирования и деталям реализации ядра Linux. Материал представлен в форме удобной как для тех, кто занимается разработкой кода ядра, так и для программистов, которые хотят лучше понять особенности работы операционных систем и, соответственно, разрабатывать более эффективные прикладные программы.

В книге детально рассмотрены основные подсистемы и функции ядра Linux, особенности их построения, реализации и соответствующие программные интерфейсы. При этом ядро рассматривается с теоретической и прикладной точек зрения, что может привлечь читателей с различными интересами и потребностями.

Автор книги является разработчиком основных подсистем ядра Linux. В этой книге он делится своим ценными опытом и знаниями по ядрам Linux серии 2.6. Рассмотренные вопросы включают управление процессами, планирование выполнения процессов, управление временем и таймеры ядра, интерфейс системных вызовов, особенности адресации и управления памятью, страничный кэш, подсистему VFS, механизмы синхронизации, проблемы переносимости и особенности отладки. В книге также рассмотрены интересные новшества, которые появились в ядрах серии 2.6, такие как планировщик O(1), преемптивное ядро, уровень блочного ввода-вывода и планировщики ввода-вывода.

Второе издание книги включает...

Обновление информации о большинстве подсистем и функций ядер Linux серии 2.6

- Новые детали о загружаемых модулях ядра
- Расширенное рассмотрение виртуальной памяти и особенностей выделения памяти в режиме ядра
- Дополнительные сведения по отладке кода ядра
- Примеры, касающиеся синхронизации выполнения кода ядра и работы таймеров
- Полезные детали по работе с заплатами и вопросы взаимодействия с сообществом разработчиков

КАТЕГОРИЯ:
Программирование для операционной системы Linux

УРОВЕНЬ:
Для опытных разработчиков и программистов средней квалификации

ОБ АВТОРЕ

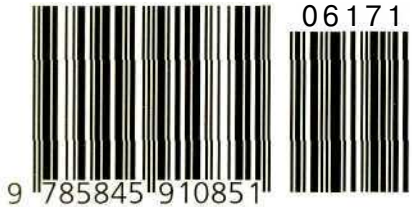
Роберт Лав является активным разработчиком программного обеспечения с открытым исходным кодом и использует операционную систему Linux с первых дней ее существования. Он активно работает как в сообществе разработчиков ядра Linux, так и в сообществе разработчиков графической среды GNOME и сейчас занимает должность главного инженера по разработке ядра группы разработчиков Ximian Desktop корпорации Novell. Проекты по разработке ядра, которыми занимался автор, включают планировщик выполнения процессов, преемптивное ядро, уровень событий ядра, улучшение поддержки виртуальной памяти, улучшение поддержки многопроцессорного оборудования. Роберт является автором утилит schedutils и менеджера томов GNOME. Автор имеет степень бакалавра по математике и вычислительной технике университета штата Флорида.

Вступительное слово Эндрю Мортона (Andrew Morton), ответственного разработчика ядер Linux серии 2.6.

Опубликовано
с разрешения
и при содействии
корпорации Novell, Inc.

www.novellpress.com

ISBN 5-8459-1085-4



www.williamspublishing.com

Novell -зарегистрированнаяторговаямаркав США и других странах. Novell Press и Ximian - торговые марки корпорации Novell, Inc., зарегистрированные в США и других странах. Linux — зарегистрированная торговая марка Линуса Торвальдса.

Этот файл был взят с сайта

http://all-ebooks.com

Данный файл представлен исключительно в ознакомительных целях. После ознакомления с содержанием данного файла Вам следует его незамедлительно удалить. Сохраняя данный файл вы несете ответственность в соответствии с законодательством.
Любое коммерческое и иное использование кроме предварительного ознакомления запрещено.
Публикация данного документа не преследует за собой никакой коммерческой выгоды.
Эта книга способствует профессиональному росту читателей и является рекламой бумажных изданий.
Все авторские права принадлежат их уважаемым владельцам.
Если Вы являетесь автором данной книги и её распространение ущемляет Ваши авторские права или если Вы хотите внести изменения в данный документ или опубликовать новую книгу свяжитесь с нами по email.