Роберт Лав

Третье издание

Ядро Linux

Описание процесса разработки

Исчерпывающее руководство
по проектированию и реализации
ядра Linux

БИБЛИОТЕКА РАЗРАБОТЧИКА

Linux Kernel Development

Third Edition

Robert Love

Addison
Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Ядро Linux

Описание процесса разработки

Третье издание

Роберт Лав

Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75
Л13
УДК 681.3.07

Издательский дом “Вильямс”
Зав. редакцией *С.Н. Тригуб*
Перевод с английского и редакция канд. физ.-мат. наук *С.Г. Тригуб*
По общим вопросам обращайтесь в Издательский дом “Вильямс” по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Лав, Роберт.

Л13 Ядро Linux: описание процесса разработки, 3-е изд. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 496 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1779-9 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Роберт Лав

Ядро Linux: описание процесса разработки

3-е издание

Литературный редактор *И.А. Попова*
Верстка *М.А. Удалов*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 29.06.2012. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 39,9. Уч.-изд. л. 32,1.
Тираж 1000 экз. Заказ № 0000.

Первая Академическая типография “Наука”
199034, Санкт-Петербург, 9-я линия, 12/28

ООО “И. Д. Вильямс”, 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1779-9 (рус.)
ISBN 978-0-672-32946-3 (англ.)

Оглавление

Введение 19

Об авторе 23

Глава 1. Понятие о ядре Linux 25

Глава 2. Начальные сведения о ядре Linux 37

Глава 3. Управление процессами 51

Глава 4. Системный планировщик и диспетчеризация процессов 73

Глава 5. Системные функции 103

Глава 6. Структуры данных ядра 119

Глава 7. Прерывания и их обработка 147

Глава 8. Нижняя половина обработчика и отложенные действия 169

Глава 9. Общие сведения о синхронизации кода ядра 201

Глава 10. Средства синхронизации ядра 217

Глава 11. Таймеры и управление временем 253

Глава 12. Управление памятью 279

Глава 13. Виртуальная файловая система 311

Глава 14. Уровень блочного ввода-вывода 341

Глава 15. Адресное пространство процесса 359

Глава 16. Страничный кеш и отложенная запись страниц 379

Глава 17. Устройства и модули 395

Глава 18. Отладка 425

Глава 19. Переносимость 443

Глава 20. Заплаты, хакерство и сообщество 461

Список литературы 475

Предметный указатель 479

Содержание

Предисловие 17

Введение 19

Итак... 20
Версия ядра 20
Читательская аудитория 20
Благодарности 22

Об авторе 23

От издательства 24

Глава 1. Понятие о ядре Linux 25

История систем Unix 25
Потом пришел Линус: введение в Linux 27
Обзор операционных систем и ядер 29
Отличие ядра Linux от классических ядер Unix 31
Версии ядра Linux 34
Сообщество разработчиков ядра Linux 35
Перед тем как начать 36

Глава 2. Начальные сведения о ядре Linux 37

Где взять исходный код ядра 37
- Использование Git 37
- Инсталляция исходного кода ядра 38
- Использование заплат 38

Дерево каталогов исходных кодов ядра 39
Сборка ядра 40
- Конфигурирование ядра 40
- Уменьшение количества выводимых сообщений 42
- Порождение нескольких параллельных задач сборки 42
- Инсталляция нового ядра 42

Отличия от обычных приложений 43
- Отсутствие библиотеки libc и стандартных заголовков 44
- Компилятор GNU C 45
- Отсутствие защиты памяти 47
- Нельзя просто использовать вычисления с плавающей точкой 47
- Системная стек-память небольшого фиксированного размера 47
- Синхронизация и параллельное выполнение 48
- Переносимость — это важно 48

Резюме 49

Глава 3. Управление процессами 51
Понятие процесса 51
Дескриптор процесса и структура task_struct 53
Распределение памяти под дескриптор процесса 53
Сохранение дескриптора процесса 55
Состояние процесса 56
Изменение текущего состояния процесса 58
Контекст процесса 58
Дерево семейства процессов 58
Создание нового процесса 60
Копирование при записи 60
Функция fork() 61
Функция vfork() 62
Реализация потоков в ядре Linux 63
Создание потоков 64
Потоки в ядре 65
Завершение процесса 66
Удаление дескриптора процесса 68
Дилемма “беспризорного” процесса 68
Резюме 70

Глава 4. Системный планировщик и диспетчеризация процессов 73
Мультипрограммный режим работы 73
Системный планировщик Linux 75
Стратегия планирования 76
Процессы, ориентированные на ввод-вывод и на вычисления 76
Приоритет процессов 77
Кванты времени 78
Стратегия планирования в действии 79
Алгоритм работы планировщика системы Linux 80
Классы планировщика 80
Планирование процессов в системах Unix 81
Справедливое планирование задач 83
Реализация планировщика в системе Linux 85
Учет времени 85
Выбор процесса 87
Точка входа в планировщик 91
Замораживание и активизация процессов 92
Вытеснение и переключение контекста 96
Вытеснение пространства пользователя 97
Вытеснение пространства ядра 98
Стратегии планирования в режиме реального времени 99
Системные функции для управления планировщиком 100
Системные функции для изменения стратегии и приоритета 101
Системные функции для изменения привязки к процессору 101
Передача процессорного времени другим задачам 102
Резюме 102

Глава 5. Системные функции 103
Взаимодействие с ядром 103
API, POSIX и библиотека C 104
Системные функции 105
Номера системных функций 106
Быстродействие системных функций 107
Обработчик вызова системных функций 107
Как определить, какую системную функцию вызвать 108
Передача параметров 109
Реализация системных функций 109
Разработка системных функций 109
Проверка параметров 110
Контекст системной функции 113
Завершающие этапы регистрации системной функции 114
Доступ к системным функциям из пользовательских приложений 116
Почему не нужно создавать системные функции 117
Резюме 118

Глава 6. Структуры данных ядра 119
Связанные списки 119
Однонаправленный и двунаправленный связанный список 120
Циклически связанные списки 120
Перемещение по элементам связанного списка 121
Реализация в ядре Linux 122
Работа со связанными списками 124
Обход элементов связанного списка 127
Очереди 130
Система kfifo 131
Создание очереди 132
Постановка в очередь 132
Выборка из очереди 132
Определение размера очереди 133
Очистка и удаление очереди 133
Примеры использования очередей 134
Таблицы отображения 134
Инициализация структуры idr 135
Выделение нового UID 135
Поиск UID 137
Удаление UID 137
Аннулирование idr 137
Двоичные деревья 138
Двоичные деревья поиска 138
Самобалансирующиеся двоичные деревья поиска 139
Красно-черные деревья 139
Реализация в Linux 140
Какие структуры данных следует использовать, если... 142
Алгоритмическая сложность 143
Что такое алгоритм? 143

Понятие большого “O” 144
Понятие большой “тета” 144
Временная сложность алгоритма 145
Резюме 146

Глава 7. Прерывания и их обработка 147

Прерывания 147
Обработчики прерываний 149
Верхняя и нижняя половины 150
Регистрация обработчика прерывания 150
Флаги обработчика прерываний 151
Остальные параметры функции обработки прерывания 152
Пример обработчика прерывания 153
Освобождение обработчика прерывания 153
Написание обработчика прерывания 154
Обработчики общих запросов на прерывание 155
Пример настоящего обработчика прерывания 156
Контекст прерывания 158
Реализация системы обработки прерываний 159
Интерфейс /proc/interrupts 162
Управление прерываниями 163
Запрещение и разрешение прерываний 164
Запрещение заданной линии IRQ 165
Состояние системы обработки прерываний 166
Резюме 167

Глава 8. Нижняя половина обработчика и отложенные действия 169

Нижняя половина 170
Когда используется нижняя половина обработчика 171
Многообразие нижних половин 171
Отложенные прерывания 174
Реализация механизма отложенных прерываний 175
Использование отложенных прерываний 177
Тасклеты 179
Реализация тасклетов 179
Использование тасклетов 182
Демон ksoftirqd 184
Старый механизм BH 186
Очереди отложенных действий 187
Реализация очередей отложенных действий 188
Использование очередей отложенных действий 191
Старый механизм очередей задач 194
Какие механизмы обработчиков нижних половин следует использовать 195
Блокировки между нижними половинами обработчиков 196
Запрещение обработки нижних половин 197
Резюме 199

Глава 9. Общие сведения о синхронизации кода ядра 201
Критические участки и конфликты из-за доступа к системным ресурсам 202
Зачем вообще нужно что-то защищать? 202
Общая переменная 204
Блокировки 205
Причины возникновения параллелизма 207
Что нужно защищать? 209
Взаимоблокировки 210
Конфликт при блокировке и масштабируемость 213
Резюме 215

Глава 10. Средства синхронизации ядра 217
Неделимые операции 217
Неделимые целочисленные операции 218
64-разрядные неделимые операции 222
Неделимые битовые операции 223
Спин-блокировки 225
Функции для спин-блокировки 227
Другие средства работы со спин-блокировками 229
Спин-блокировки и нижние половины обработчиков прерываний 230
Спин-блокировки по чтению-записи 230
Семафоры 233
Счетные и бинарные семафоры 234
Создание и инициализация семафоров 235
Использование семафоров 236
Семафоры для чтения-записи 237
Мьютексы 238
Сравнение семафоров и мьютексов 240
Сравнение спин-блокировок и мьютексов 240
Условные переменные 241
BKL: большая блокировка ядра 242
Последовательные блокировки 243
Отключение мультипрограммного режима работы ядра 245
Порядок выполнения операций и барьеры 247
Резюме 251

Глава 11. Таймеры и управление временем 253
Основная идея учета времени в ядре 254
Частота импульсов таймера: директива HZ 255
Идеальное значение параметра HZ 256
Преимущества больших значений параметра HZ 257
Недостатки больших значений параметра HZ 258
Переменная jiffies 259
Внутреннее представление переменной jiffies 260
Переполнение переменной jiffies 261
Пользовательские программы и параметр HZ 263
Аппаратные часы и таймеры 264
Часы реального времени 264

Системный таймер 264
Обработчик прерываний от таймера 265
Абсолютное время 267
Таймеры 269
Использование таймеров 270
Конфликты из-за доступа к ресурсам при использовании таймеров 272
Реализация таймеров 272
Задержка выполнения 273
Задержка с помощью цикла 273
Короткие задержки 274
Функция schedule_timeout() 276
Резюме 278

Глава 12. Управление памятью 279

Страничная организация памяти 279
Зоны 281
Выделение страниц памяти 284
Выделение обнуленных страниц памяти 284
Освобождение страниц памяти 285
Функция kmalloc() 286
Флаги gfp_mask 287
Функция kfree() 292
Функция vmalloc() 292
Уровень блочного распределения памяти 294
Структура уровня блочного распределения памяти 295
Интерфейс блочного распределителя памяти 298
Пример использования блочного распределителя памяти 300
Статическое выделение памяти в стеке 301
Одностраничные стеки ядра 302
Справедливое использование стека 303
Отображение верхней памяти 303
Постоянное отображение 303
Временное отображение 304
Выделение памяти для конкретного процессора 305
Новый интерфейс percpu 306
Работа с процессорными данными на этапе компиляции 306
Работа с процессорными данными на этапе выполнения 307
Когда лучше использовать данные, связанные с процессорами 308
Выбор способа выделения памяти 309
Резюме 310

Глава 13. Виртуальная файловая система 311

Общий интерфейс файловых систем 312
Абстрактный уровень файловой системы 312
Файловые системы Unix 314
Объекты VFS и их структуры данных 315
Объект суперблок 317
Операции суперблока 318

Объект inode 320
Операции с файловыми индексами 322
Объект элемента каталога (dentry) 325
Состояние элементов каталога 326
Кеш объектов dentry 327
Операции с элементами каталогов 328
Файловый объект 329
Файловые операции 330
Структуры данных, связанные с файловыми системами 335
Структуры данных, связанные с процессом 337
Резюме 339

Глава 14. Уровень блочного ввода-вывода 341

Структура блочного устройства 342
Буферы и их заголовки 343
Структура bio 346
Векторы ввода-вывода 347
Сравнение старой и новой реализаций 348
Очереди запросов 349
Планировщики ввода-вывода 350
Задачи планировщика ввода-вывода 350
Лифт имени Линуса 351
Планировщик ввода-вывода с ограничением по времени 353
Прогнозирующий планировщик ввода-вывода 355
Планировщик ввода-вывода с полностью равноправными очередями 356
Планировщик ввода-вывода с отсутствием операций (Noop) 357
Выбор планировщика ввода-вывода 357
Резюме 358

Глава 15. Адресное пространство процесса 359

Адресные пространства 359
Дескриптор памяти 361
Выделение дескриптора памяти 363
Удаление дескриптора памяти 363
Структура mm_struct и потоки ядра 364
Области виртуальной памяти 364
Флаги областей VMA 365
Операции с областями VMA 367
Списки и деревья областей памяти 368
Области памяти в реальных приложениях 369
Работа с областями памяти 371
Функция find_vma() 371
Функция find_vma_prev() 372
Функция find_VMA_intersection() 372
Функции mmap() и do_mmap(): создание диапазона адресов 373
Функции munmap() и do_munmap(): удаление диапазона адресов 375
Таблицы страниц 375
Резюме 377

Глава 16. Страничный кеш и отложенная запись страниц 379
Методики кеширования 380
Кеширование при записи 380
Вытеснение данных из кеша 381
Реализация страничного кеша в ОС Linux 383
Объект address_space 383
Операции объекта address_space 385
Базисное дерево 387
Старая хеш-таблица страниц 387
Буферный кеш 388
Потоки синхронизатора 388
Режим ноутбука 390
Экскурс в историю: bdflush, kupdated и pdflush 391
Предотвращение перегрузки с помощью нескольких потоков 392
Резюме 393

Глава 17. Устройства и модули 395
Типы устройств 395
Модули 396
Модуль “Hello, World!” 397
Сборка модулей 398
Установка модулей 401
Генерация зависимостей между модулями 401
Загрузка модулей 401
Поддержка параметров конфигурации 402
Параметры модулей 404
Экспортируемые символы 406
Модель представления устройств 407
Объекты kobject 408
Типы ktype 409
Множества объектов kset 410
Взаимосвязь kobject, ktype и kset 410
Управление и работа с объектами kobject 411
Счетчики ссылок 412
Файловая система sysfs 414
Добавление и удаление объектов файловой системы sysfs 417
Добавление файлов в файловую систему sysfs 418
Уровень событий ядра 421
Резюме 423

Глава 18. Отладка 425
Начало работы 425
Ошибки ядра 426
Отладка с помощью вывода диагностических сообщений 427
Устойчивость 427
Уровни вывода сообщений ядра 428
Буфер сообщений ядра 429
Демоны syslogd и klogd 429

Взаимозаменяемость функций printf() и printk() 430
Сообщения Oops 430
Утилита ksymoops 431
Функция kallsyms 432
Параметры конфигурации для отладки ядра 433
Объявление об ошибках и выдача информации 433
“Магическая” клавиша <SysRq> 434
Сага об отладчике ядра 435
Отладчик gdb 436
Отладчик kgdb 436
Исследование и тестирование системы 437
Использование идентификатора UID в качестве условия 437
Использование условных переменных 437
Использование статистики 438
Ограничение частоты следования и общего количества событий при отладке 438
Поиск методом половинного деления изменений, приводящим к ошибкам 439
Поиск с помощью git 440
Если ничто не помогает — обратитесь к сообществу 441
Резюме 441

Глава 19. Переносимость 443

Переносимые операционные системы 443
История переносимости Linux 445
Размер машинного слова и типы данных 446
Скрытые типы данных 449
Специальные типы данных 449
Типы с явным указанием размера 450
Знаковые и беззнаковые типы char 451
Выравнивание данных 451
Как избежать проблем с выравниванием 452
Выравнивание нестандартных типов данных 452
Пустые поля структур 453
Порядок следования байтов 454
Учет времени 456
Размер страницы памяти 457
Порядок выполнения операций процессором 458
Многопроцессорность, мультипрограммирование и верхняя память 458
Резюме 459

Глава 20. Заплаты, хакерство и сообщество 461

Сообщество 461
Стиль написания исходного кода 462
Отступы 462
Оператор switch 463
Пробелы 463
Фигурные скобки 464
Длина строки исходного кода 465
Соглашения о присвоении имен 466

Функции 466
Комментарии 466
Использование директивы typedef 467
Использование того, что уже есть 468
Избегайте директив ifdef в исходном коде 468
Инициализация структур 468
Исправление ранее написанного кода 469
Организация команды разработчиков 469
Отправка сообщений об ошибках 470
Заплаты 470
Генерация заплат 470
Генерирование заплат с помощью программы Git 471
Публикация заплат 472
Резюме 473

Список литературы 475

Книги по основам построения операционных систем 475
Книги о ядре Unix 476
Книги о ядре Linux 476
Книги о ядрах других операционных систем 476
Книги по API Unix 477
Книги по программированию на языке C 477
Другие работы 477
Веб-сайты 478

Предметный указатель 479

♦

Посвящается Дорис (Doris) и Элен (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. Часть из этих разработчиков руководствуется исключительно собственным интересом, часть — работает в компаниях, которые занимаются операционной системой Linux, часть — работает на производителей компьютерных аппаратных средств, часть — занята 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 “Лучше меньше, да лучше” и т.д. Такое решение предложено Линусом Торвальдсом (Linus Torvalds).

Подход, который предлагаю я, состоит в использовании большего числа комментариев в исходном коде, что поможет читателю понять, чего хотел достичь программист. (Процесс выявления расхождений между поставленной программистом целью и полученной в результате реализацией называется *отладкой*. 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 отправной точки такого понимания.

Вклад Роберта Лава (Robert Love) состоит в предоставлении возможности, благодаря которой опытные разработчики смогут получить полную информацию о том, какие функции должны выполнять различные подсистемы ядра и каким образом предполагается выполнение этих функций. 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 поможет начинающим разработчикам перейти 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 и даже сотням миллионов людей.

Эндрю Мортон (Andrew Morton)
Open Source Development Labs

Введение

Сделав первую попытку превратить свой опыт работы с ядром Linux в текст книги, я понял, что не знаю, куда двигаться дальше. Не хотелось просто писать еще одну книгу о ядре операционной системы. Конечно, на эту тему написано *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 о структуре и практической реализации ядра операционной системы Linux. Информация в ней представлена так, чтобы получить достаточно знаний для решения реальных практических задач и решать эти задачи 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 тех правил (писанных и неписаных), которые действуют в ядре операционной системы. Надеюсь также, 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, третьем по счету, издании представлено несколько больше информации по сравнению с двумя остальными: материал серьезно пересмотрен и доработан, появились новые разделы и главы. С момента выхода второго издания в ядро были внесены изменения. Но, что более важно, сообщество разработчиков ядра Linux приняло решение[1] в ближайшем будущем не начинать разработку ядра версии 2.7. Было решено заняться стабилизацией ядра версии 2.6. Стабилизация включает в себя много моментов, тем не менее есть один важный, касающийся данной книги, — книга, которая посвящена ядру серии 2.6, остается актуальной до сих пор. Поскольку изменения происходят не слишком быстро, существует большой шанс, что “моментальный снимок” ядра останется актуальным и в будущем. Эта книга, по сути, стала канонической документацией по ядру, в которой отражены как история, так и взгляд в будущее.

---

[1] Это решение было принято на саммите разработчиков ядра Linux (Linux Kernel Development Summit), который состоялся летом 2004 года в Оттаве, Канада. Меня пригласили туда в качестве участника.

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! Добавляйте новые функциональные возможности в ядро, даже если они на первый взгляд кажутся весьма простыми. Находите слабые места и закрывайте их! У вас все получится, если вы будете сами *писать* программный код.

Версия ядра

В этой книге рассмотрено ядро Linux версии 2.6. Я не стал описывать устаревшие версии ядра, за исключением случаев, представляющих чисто исторический интерес. Например, мы рассмотрим, как были реализованы определенные подсистемы в ядре версии 2.4, поскольку тогда они были намного проще, и сможем легче во всем разобраться. Что касается данной книги, то она была написана на момент, когда актуальным являлось ядро Linux версии 2.6.34. 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.

Несмотря на то что в этой книге описано ядро версии 2.6.34, я постарался сделать так, чтобы в ней также были максимально корректно отражены сведения по ядру версии 2.6.32. Дело в том, что эта устаревшая версия была официально одобрена в качестве "корпоративного" ядра, которое продолжает использоваться во многих дистрибутивах Linux. А это значит, что мы еще долго будем встречать его во многих производственных системах и оно еще много лет будет находиться в состоянии активной разработки. (Существовавшие ранее ядра версий 2.6.9, 2.6.18 и 2.6.27 — аналогичные примеры "долгожителей".)

Читательская аудитория

Эта книга предназначена для разработчиков программного обеспечения, которые хотят понять, как устроено ядро операционной системы Linux. Тем не менее она *не* является сборником построчных комментариев, извлеченных из исходного кода ядра. Ее также нельзя считать руководством по разработке драйверов или справочником по программному интерфейсу (API) ядра. Целью книги является предоставление достаточной информации о структуре и реализации ядра, 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 об-

ращена 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. 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. Думаю, что эти вопросы важны и достойны обсуждения. Хороший пример — глава 8, "Нижняя половина обработчика и отложенные действия", посвященная компонентам драйверов устройств, называемых *нижними половинами* (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 нижних половин (это может быть интересно разработчикам драйверов устройств, а также другим программистам-профессионалам). На самом деле мне кажется, что обе эти стороны обсуждения будут интересны всем группам 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 использовать ее. 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 одной или другой группе разработчиков, а в надежде, что *весь материал* книги будет полезен всем разработчикам.

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 студентам, изучающим 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.

Благодарности

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, хочу поблагодарить всех своих друзей и коллег за помощь, поддержку и конструктивную критику.

В первую очередь хотел бы высказать благодарность моим коллегам из издательства Addison–Wesley и компании Pearson, которые долго и упорно трудились над тем, чтобы сделать эту книгу лучше. В особенности хочу поблагодарить Марка Табера (Mark Taber) за руководство, благодаря которому третье издание книги превратилось из идеи в конечный продукт, а также редактора-консультанта (development editor) Майкла Тарстона (Michael Thurston) и редактора проектов (project editor) Тоню Симпсон (Tonya Simpson).

Особые благодарности я хотел бы выразить техническому редактору данного издания Роберту П. Дж. Дею (Robert P. J. Day). Его проницательность, опыт и правки помогли сделать эту книгу неизмеримо лучше. Тем не менее, если, несмотря на все его титанические усилия, некоторые ошибки все же остались в книге, то это — вина автора! Такое же огромное спасибо Адаму Билаю (Adam Belay), Заку Брауну (Zak Brown), Мартину Пулу (Martin Pool) и Крису Ривера (Rivera), которые славно потрудились над техническим редактированием первого и второго издания этой книги и не ударили в грязь лицом при работе над третьим изданием.

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 Кроа-Хартман (Greg Kroah-Hartman), Дейв Миллер (Dave Miller), Патрик Мочел (Patrick Mochel), Эндрю Мортон (Andrew Morton), Ник Пиггин (Nick Piggin) и Линус Торвальдс (Linus Torvalds).

Огромное спасибо хочу сказать моим коллегам из Google, самой творческой и интеллектуальной группе людей, с которыми мне когда-либо приходилось сталкиваться и иметь удовольствие работать. Если я здесь начну перечислять их всех, мне не хватит объема книги. Поэтому я хочу выделить Алана Блоунта (Alan Blount), Джея Крима (Jay Crim), Криса Дэниса (Chris Danis), Криса Дибона (Chris DiBona), Эрика Флэтта (Eric Flatt), Майка Локвуда (Mike Lockwood), Сана Мехата (San Mehat), Брайана Рогана (Brian Rogan), Брайана Свитланда (Brian Swetland), Джона Троубриджа (Jon Trowbridge) и Стива Винтера (Steve Vinter) за их дружелюбие, эрудицию и помощь.

Я хочу выразить свою любовь и признательность Полу Амичи (Paul Amichi), Мики Бэббиту (Mikey Babbitt), Киту Бэрбегу (Keith Barbag), Джейкобу Беркману (Jacob Berkman), Нату Фридману (Nat Friedman), Дастину Холлу (Dustin Hall), Джойсу Хокинсу (Joyce Hawkins), Мигуэлю де Иказа (Miguel de Icaza), Джимми Крелу (Jimmy Krehl), Дорис Лав (Doris Love), Линде Лав (Linda Love), Бретти Лаку (Brette Luck), Рэнди О'Дауду (Randy O'Dowd), Сальваторэ Рибаудо (Salvatore Ribaudo) и его чудесной маме, Крису Ривера (Chris Rivera), Кэролайн Родон (Carolyn Rodon), Джой Шо (Joey Shaw), Саре Стьюарт (Sarah Stewart), Джереми ВанДорену (Jeremy VanDoren) и его семье, Льюису Вилле (Luis Villa), Стиву Вейсбергу (Steve Weisberg) и его семье, а также Хелен Винснант (Helen Whinsnant).

И в заключение хочу поблагодарить моих родителей за то, что по наследству от них мне достались хорошо сложенные ушные раковины. Желаю им большого хакерского счастья!

Роберт Лав,
Бостон.

Об авторе

Роберт Лав — программист сообщества с открытым исходным кодом, лектор и автор, который использует систему Linux и содействует ее развитию на протяжении вот уже более пятнадцати лет. В настоящее время Роберт работает старшим разработчиком программного обеспечения в корпорации Google, где он входит в группу разработчиков ядра для мобильной платформы Android. До работы в Google он был главным инженером группы по разработке Linux Desktop в компании Novell. До этого он работал инженером по разработке ядра компании MontaVista Software и Ximian.

Проекты по разработке ядра, которыми занимался Роберт, включают: ядро с приоритетным мультипрограммным режимом (preemptive kernel), планировщик выполнения процессов, уровень событий ядра, улучшение поддержки виртуальной памяти (VM) и несколько драйверов устройств.

Роберт выступал на многочисленных конференциях и написал большое количество статей на тему ядра Linux. Он регулярно получает предложения редактировать статьи в *Linux Journal*. Роберт также является автором двух других книг: *Linux System Programming* и *Linux in a Nutshell*.

Автор получил степень бакалавра по математике и вычислительной технике в университете штата Флорида. Сейчас он проживает в Бостоне.

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 веб-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 электронные адреса:

E-mail: info@williamspublishing.com

WWW: http://www.williamspublishing.com

Наши почтовые адреса:

из России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

из Украины: 03150, Киев, а/я 152

1
Понятие о ядре Linux

В этой главе мы ознакомимся с операционной системой Linux и ее ядром, а также рассмотрим ее место в историческом контексте операционных систем Unix. В наши дни Unix представляет собой семейство операционных систем, в которых реализован *сходный интерфейс прикладных программ* (API) и в основе которых лежат *общие проектные решения*. Однако Unix также является весьма специфической операционной системой, первые версии которой были выпущены вот уже более 40 лет тому назад. Чтобы понять, что такое Linux, вначале мы должны рассмотреть первую версию операционной системы Unix.

История систем Unix

Даже после четырех 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.

В основу операционной системы Unix положена система Multics — многопользовательская операционная система, проект по созданию которой провалился и в котором принимали участие сотрудники Bell Laboratories. После прекращения работ над проектом Multics сотрудники научно-исследовательского центра по информатике Bell Laboratories (Bell Laboratories' Computer Sciences Research) фактически оказались без действующей версии диалоговой операционной системы. Летом 1969 года программисты корпорации Bell Labs разработали проект файловой системы, которая в конечном итоге была включена в операционную систему Unix. В процессе тестирования проекта Томпсон реализовал новую операционную систему на простаивающем компьютере 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, однако этот шаг заложил основу для будущей переносимости всех систем. Первая версия операционной системы Unix, которая использовалась вне стен Bell Labs, называлась Unix System версии 6 (обычно ее 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 компания 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].

Простая структура 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 Беркли (University of California at Berkeley). Варианты операционной системы Unix из Беркли стали называться Berkeley Software Distributions (BSD), или программный дистрибутив Berkeley. Первая версия операционной системы Unix 1BSD, разработанная в Беркли в 1977 году, представляла собой набор заплат и дополнительных программ, созданных для системы Unix компании Bell Labs. Во второй версии 2BSD, выпущенной в 1978 году, принятое ранее направление разработки было продолжено. В ней появились утилиты `csh` и `vi`, которые сохранились в системах Unix и по сей день. Первая самостоятельная версия Berkeley Unix была разработана в 1979 году и называлась 3BSD. В ней, кроме внушительного списка нововведений, была реализована поддержка виртуальной памяти (Virtual memory, VM). Следом за ней появились выпуски серии 4BSD: 4.0BSD, 4.1BSD, 4.2BSD и 4.3BSD. В этих версиях операционной системы Unix были добавлены средства управления заданиями (job control), подкачка страниц по требованию (demand paging) и стек протоколов TCP/IP. В 1994 году университет в Беркли выпустил последнюю официальную версию ОС Unix 4.4BSD, в которой система управления виртуальной памятью была полностью переписана. В наши дни, благодаря разрешающей лицензии BSD, разработка этого направления ОС продолжается в операционных системах Darwin, FreeBSD, NetBSD и OpenBSD.

В 1980–1990-х годах многие компании-разработчики рабочих станций и серверов предложили свои коммерческие версии операционной системы Unix. Обычно за основу этих операционных систем брались выпуски ОС Unix AT&T или Беркли, в которые добавлялись дополнительные возможности, обеспечиваемые соответствующей аппаратной платформой. Среди таких систем были Tru64 компании Digital, HP-UX компании Hewlett Packard, AIX компании IBM, DYNIX/ptx компании Sequent, IRIX компании SGI, а также Solaris и SunOS компании Sun.

Первоначальная продуманная структура операционной системы Unix в сочетании с многолетними нововведениями и улучшениями, которые последовали за ними, сделали Unix мощной, устойчивой и стабильной ОС. В основу устойчивости ОС Unix положено несколько факторов. Во-первых, операционная система Unix проста. В то время как в некоторых операционных системах реализованы тысячи системных вызовов и эти системы имеют недостаточно ясные черты дизайна, Unix-подобные операционные системы обычно имеют только несколько сотен системных вызовов и достаточно четкий, я бы даже сказал, простой 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()`, `lseek()` и `close()`. В-третьих, ядро и системные утилиты операционной

[1] А куда же делась версия System IV? Это была внутренняя экспериментальная версия.

[2] Конечно же, не все, но многое! Самым заметным исключением являются сокеты. Однако существуют современные операционные системы, например наследник Unix компании Bell Labs — система Plan9, в которой практически все аспекты представлены в виде файлов.

системы Unix написаны на языке программирования C. Эта особенность позволяет очень просто перенести систему на различные аппаратные платформы и сделать ее доступной для широкого круга разработчиков. В-четвертых, для ОС Unix характерны очень малое время создания нового процесса и уникальный системный вызов `fork()`. И наконец, операционная система Unix предоставляет простые и в то же время надежные средства межпроцессного взаимодействия (interprocess communication, IPC). В сочетании с быстрым созданием процессов они позволяют создавать простые утилиты, которые *умеют выполнять всего одну функцию, но делают это хорошо* и могут быть связаны вместе для выполнения более сложных функций. В результате в Unix создается полностью иерархическая система, в которой строго разделяются алгоритмы и механизмы их реализации.

Сегодня Unix — современная операционная система, которая поддерживает приоритетный мультипрограммный режим, многопоточность, виртуальную память, подкачку страниц по требованию, совместно используемые библиотеки, загружаемые по требованию, и стек протоколов TCP/IP. Существуют варианты операционной системы Unix, поддерживающие сотни процессоров, в то время как другие варианты ОС Unix работают на миниатюрных устройствах в качестве встраиваемых систем. Хотя разработка Unix больше не является научно-исследовательским проектом, все же продолжаются разработки (с целью получить дополнительные преимущества) с использованием возможностей 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 операционной системе Unix возможность бескомпромиссного развития.

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 и самом передовом 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. Правившая бал в то время операционная система DOS, продукт корпорации Microsoft, была для Торвальдса полезна только лишь для того, 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 в исходном коде (это запрещалось лицензией ОС Minix), а также технические решения, которые использовал автор ОС Minix.

Поставленный перед такой проблемой, Линус сделал именно то, что и должен был сделать любой нормальный студент университета, — он решил написать собственную операционную систему. Начал он с написания простого эмулятора терминала, с помощью которого он подключался к большим Unix-системам в университете. На протяжении учебного года его эмулятор терминала рос, развивался и улучшался. Постепенно у Линуса появилась еще не совсем зрелая, но полноценная Unix-система, и в 1991 году он опубликовал в Интернете ее первую версию.

Использование операционной системы Linux и количество пользователей ее ранних версий 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, в который были вовлечены многие люди.

А теперь вернемся в настоящее. Сейчас Linux — развитая операционная система, работающая на аппаратных платформах Alpha, ARM, PowerPC, SPARC, AMD x86-64 и многих других. Она работает в различных системах, как размером с часы, так и на больших суперкомпьютерных кластерах. Под управлением Linux работают как небольшие портативные устройства, так и огромные центры обработки информации (датацентры). Сегодня коммерческий интерес к операционной системе Linux достаточно высок. Как новые корпорации, ориентирующиеся исключительно на Linux, такие как Red Hat, так и старые монстры наподобие IBM предлагают решения на основе этой ОС для встраиваемых систем, мобильных устройств, рабочих станций и серверов.

Операционная система Linux является клоном Unix, но ОС Linux — это не Unix. Хотя в ОС Linux позаимствовано много идей от Unix и в ней реализован API ОС Unix (как это определено в стандарте POSIX и спецификации Single Unix), все же система Linux не является производной от исходного кода Unix, как это имеет место для других Unix-систем. Там, где это уместно, 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.

Одной из наиболее интересных особенностей операционной системы Linux является то, что это не коммерческий продукт, а, наоборот, 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 любой может внести свой вклад в развитие операционной системы 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 вы пользовались 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 (kernel), библиотека функций языка C (C library), компилятор, набор необходимых инструментальных средств (toolchain), основные системные утилиты, такие как программа для входа в систему (login) и командная оболочка (shell). В операционную систему Linux может быть включена современная реализация системы X Windows, включая полнофункциональную среду для запуска настольных приложений

[3] Для тех, кому интересно, дискуссия по поводу отличия свободно распространяемого (free) кода от открытого (open) доступна в Интернете по адресам `http://www.fsf.org` и `http://www.opensource.org`.

[4] Вероятно, вам нужно прочесть лицензию GNU GPL, если вы еще не сделали этого. В файле `COPYING` в исходном коде ядра есть копия этой лицензии. В Интернете лицензия доступна по адресу `http://www.fsf.org`. Обратите внимание на то, что последней версией GNU GPL является версия 3.0. Однако разработчики ядра решили оставаться в рамках лицензии версии 2.0.

(desktop environment), такую, как, например, GNOME. Для ОС Linux существуют тысячи бесплатных 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 *операционная система* стало несколько размытым. Многие пользователи считают, что то, что они видят на экране, — и есть операционная система. Строго говоря, в этой книге тоже под *операционной системой* понимается часть компьютерной системы, которая отвечает за основные функции использования и администрирования. В это понятие входят ядро и драйверы устройств, системный загрузчик (boot loader), командная оболочка или любой другой интерфейс пользователя, а также базовая файловая система и системные утилиты. В общем, только *необходимые* компоненты, к которым не относится веб-браузер или медиаплеер. Термин *система*, напротив, 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 — внутренняя. В своей основе ядро — это программное обеспечение, которое предоставляет базовые функции для всех остальных частей операционной системы, управляет аппаратным обеспечением и распределяет системные ресурсы. Ядро часто называют *супервизором* (*supervisor*), *основной частью* (*core*) или *внутренностями* (*internals*) операционной системы. Типичные компоненты ядра — обработчики прерываний, которые обслуживают запросы на прерывания, поступающие от различных устройств, планировщик, распределяющий 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. Системные переменные (system state) и область памяти, в которой находится ядро, вместе называются *пространством ядра* (kernel-space), или привилегированным режимом. Соответственно, пользовательские программы выполняются в *пространствах задач* (user-space), или в пользовательском режиме. Пользовательским программам доступно лишь некоторое подмножество машинных ресурсов, они не могут выполнять некоторые системные функции, напрямую работать с аппаратурой, обращаться к системной памяти (за пределами адресного пространства, выделенного пользовательской программе ядром) и делать другие недозволенные вещи. При выполнении программного кода ядра система переходит в пространство ядра, или переключается в привилегированный режим. При запуске обычных процессов система переключается в пользовательский, или непривилегированный режим.

Прикладные программы, работающие в системе, взаимодействуют с ядром с помощью *системных вызовов* (*system call*) (рис. 1.1). Прикладная программа обычно вызывает функции различных библиотек, например *библиотеки функций* языка C, которые в свою очередь обращаются к системным вызовам для того, чтобы отдать приказ ядру выполнить определенные действия от их имени. Некоторые библиотечные вызовы выполняют

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()`, которая обеспечивает форматирование и буферизацию выходных данных и лишь после этого один раз обращается к системному вызову `write()` для вывода данных на консоль. Некоторые библиотечные функции один к одному соответствуют функциям ядра. Например, библиотечная функция `open()` практически не делает ничего, кроме выполнения системного вызова `open()`. В то же время некоторые библиотечные функции, как, например, `strcpy()`, надо полагать, вообще не обращаются к ядру. Когда прикладная программа выполняет системный вызов, 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 основным способом выполнения задач.

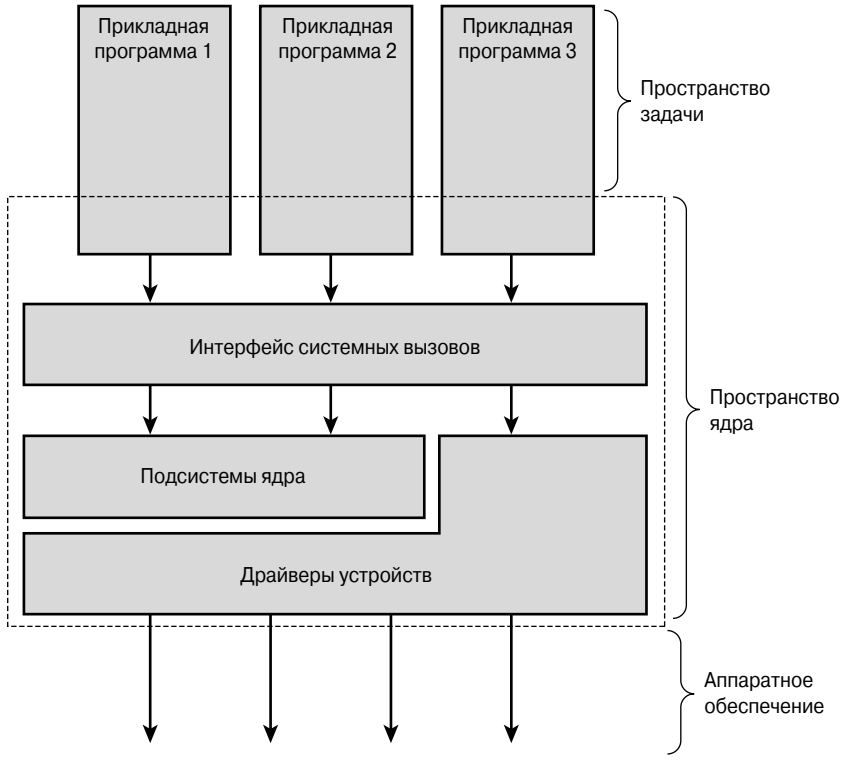


Рис. 1.1. Взаимодействие прикладных программ, ядра и аппаратного обеспечения

В функции ядра входит также управление системным аппаратным обеспечением. Практически все платформы, включая те, на которых работает операционная система Linux, используют механизм *прерываний* (*interrupt*). Когда аппаратному устройству необходимо как-то взаимодействовать с системой, оно выставляет на шину специальный сигнал, который, в буквальном смысле слова, *прерывает* работу процессора. Последний, в свою очередь, запускает специальную программу в ядре, причем все это происходит в реальном масштабе времени. Как правило, каждому типу прерывания назначается

определенное число, или *номер прерывания*. Ядро использует этот номер для запуска 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 запускает соответствующий обработчик прерывания. Обработчик прерывания обрабатывает данные, поступившие с клавиатуры, и дает знать контроллеру клавиатуры о том, что ядро готово для приема новых данных. Для обеспечения синхронизации выполнения программы ядро может запрещать прерывания: или сразу все прерывания, или только прерывание с определенным номером. Во многих операционных системах, включая Linux, обработчики прерываний выполняются не в контексте процессов, а в специальном *контексте прерывания* (*interrupt context*), который не связан ни с одним процессом. Этот специальный контекст существует только для того, чтобы дать обработчику прерывания возможность быстро отреагировать на прерывание и закончить работу.

Описанные выше контексты выполнения программ полностью определяют всю широту возможных действий ядра. Фактически можно заключить, что в операционной системе Linux процессор в любой момент времени выполняет один из трех перечисленных ниже типов действий.

- Выполнение кода пользовательской программы в непривилегированном режиме.
- Выполнение действий в контексте процесса в привилегированном режиме от имени определенного процесса.
- Обработка прерывания в привилегированном режиме в контексте прерывания, не связанном с процессами.

Этот список является исчерпывающим. Даже в исключительных случаях работу процессора можно охарактеризовать одним из перечисленных выше типов действий. Например, в случае простоя система запускает в ядре в контексте процесса специальный *холостой процесс* (*idle process*)

Отличие ядра Linux от классических ядер Unix

Благодаря общему происхождению и одинаковому API современные ядра Unix имеют некоторые общие характерные черты. (Мои любимые книги по структуре классических ядер Unix перечислены в библиографии.) За небольшими исключениями ядра Unix представляют собой монолитные статические бинарные файлы. Это означает, что они существуют в виде больших исполняемых образов, которые выполняются в одном адресном пространстве. Для работы операционной системы Unix обычно требуется система с контроллером управления *страничной организацией памяти* (*page memory management unit, MMU*). Это аппаратное обеспечение позволяет защитить память в системе и предоставить каждому процессу уникальное виртуальное адресное пространство. Исторически так сложилось, что для работы Linux нужен MMU. Однако существуют специальные реализации, для которых MMU не требуется. Это довольно редкая особенность, которая позволяет запускать систему Linux на миниатюрных встроенных системах, у которых нет блока MMU. Тем не менее такая возможность представляет больше теоретический, чем практический интерес. Дело в том, что даже самые простые современные встроенные системы оснащены всеми передовыми устройствами, включая блок MMU. Поэтому в данной книге мы будем рассматривать только системы с MMU.

Что лучше — монолитное ядро или микроядро?

Ядра операционных систем, 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 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 находятся и выполняются в одном большом адресном пространстве ядра. Взаимодействия в ядре осуществляются очень просто, потому что все, что выполняется в режиме ядра, выполняется в одном адресном пространстве. Ядро может вызывать функции непосредственно, как это делают пользовательские приложения. Сторонники такой модели обычно указывают на простоту и высокую производительность монолитных ядер. Большинство систем Unix имеют монолитное ядро.

При реализации микроядра обычно отказываются от 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 пользовательском (непривилегированном) режиме. Тем не менее при таком подходе все серверы изолированы и независимы друг от друга и запускаются каждый в своем адресном пространстве. Следовательно, прямой вызов функций, как в случае монолитного ядра, невозможен. Поэтому все взаимодействия в микроядре выполняются с помощью *передачи сообщений*. Механизм межпроцессного взаимодействия (Interprocess Communication, 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. Более того, модульный принцип построения системы позволяет по мере необходимости одному серверу выгрузить из памяти другой сервер.

Поскольку привлечение механизма IPC приводит к росту дополнительных издержек по сравнению с обычным вызовом функций и при этом может потребоваться переключение контекста из пространства пользователя в пространство ядра, и наоборот, при передаче сообщений возникают небольшие задержки, что приводит к падению производительности системы по сравнению с монолитными ядрами, в которых используются обычные вызовы функций. Поэтому в современных операционных системах с микроядром большинство серверов выполняется в пространстве ядра. Это позволяет избавиться от издержек, связанных с переключением контекста, а кроме того, дает потенциальную возможность прямого вызова функций. Ядро операционной системы Windows NT (оно положено в основу ядер таких операционных систем, как Windows XP, Vista и 7), а также ядро Mach (на котором базируется часть операционной системы Mac OS X) — это все примеры архитектур с микроядром. В последних версиях ядер как Windows NT, так и Mac OS X, все серверы выполняются только в пространстве ядра, что является отклонением от первоначальной идеи микроядра.

Ядро ОС Linux монолитное, т.е. оно выполняется в одном адресном пространстве, в режиме ядра. Тем не менее ядро Linux позаимствовало некоторые хорошие решения из микроядерной модели: в нем используется модульный принцип построения, возможность приоритетного планирования самого себя (его называют ядром с приоритетным мультипрограммным режимом, или *kernel preemption*), поддерживается многопоточный режим, а также 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, с приоритетным планированием самого ядра.

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 с учетом исторических корней Unix (и, что более важно, с учетом сложившегося API ОС Unix). Поскольку в основу операционной системы Linux не была положена никакая из существующих версий ОС Unix, Линус и компания имели возможность найти и выбрать наилучшие решения для любой проблемы и даже со временем придумать 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 может по необходимости динамически загрузить и выгрузить свой исполняемый код.
- Ядро Linux поддерживает симметричную многопроцессорную обработку (SMP). Хотя в большинстве коммерческих вариантов операционной системы Unix сейчас также поддерживается SMP, в большинстве традиционных реализаций ОС Unix такая поддержка не предусмотрена.
- В ядре Linux поддерживается приоритетное планирование. В отличие от традиционных версий ОС Unix, ядро Linux в состоянии прервать выполнение текущего задания, даже если оно работает в режиме ядра. Среди коммерческих реализаций ОС Unix приоритетное планирование в ядре поддерживается только в операционных системах Solaris и IRIX. В большинстве ядер других версий Unix такая возможность не поддерживается.
- 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 Linux принята объектно-ориентированная модель устройств, в которой поддерживаются классы устройств, события, возникающие при горячем подключении устройств, и файловая система в пространстве пользователя sysfs (user-space device filesystem).
- В ядре Linux отсутствуют некоторые функции ОС Unix, которые разработчики посчитали плохо спроектированными, как, например, поддержка интерфейса STREAMS, или поддержка стандартов, которые невозможно аккуратно реализовать.
- 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 плохо проработанной, то разработчики ядра не обязаны ее реализовывать. 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, которые в большей степени являются маркетинговой “фишкой” или используются редко, такие, как страничная организация памяти ядра, не были реализованы.

Несмотря на имеющиеся различия, Linux остается 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 предоставления новых драйверов устройств. Разрабатываемые версии ядра, наоборот, подвержены быстрым изменениям, где может происходить практически все, что угодно. По мере того как разработчики экспериментируют с новыми решениями, в исходный код ядра довольно часто вносятся радикальные изменения.

Ядра Linux стабильных и разрабатываемых версий можно отличить друг от друга с помощью простой схемы нумерации (рис. 1.2). Версию ядра определяют три или четыре числа, которые разделяются точкой. Первое число определяет номер основной (major) версии, второе — номер младшей (minor) версии, а третье число указывает на номер выпуска (revision). Необязательное четвертое число определяет номер стабильной версии. Значение младшей версии также определяет, является ли ядро стабильным или разрабатываемым; если это значение четное, значит, ядро стабильное, а если нечетное — разрабатываемое. Так, например, версия 2.6.30.1 определяет стабильное ядро. Ядро имеет основную версию 2, младшую версию 6, номер выпуска 30 и первую стабильную версию. Первые два числа также определяют серию ядер, в данном случае серия ядер — 2.6.

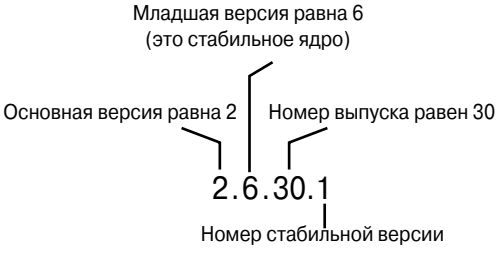


Рис. 1.2. Соглашение о нумерации версий ядра

Процесс разработки ядра включает несколько фаз. Вначале разработчики ядра работают над новыми функциями, что напоминает хаос. Через определенное время ядро оказывается сформировавшимся, и в конце концов объявляется о замораживании функций. Начиная с этого момента Линус не принимает никаких запросов на включение новых функций в ядро. Однако работа над существующими функциями может быть продолжена. После того как Линус решит, что ядро достигло практически стабильного уровня, исходный код замораживается. В этом случае допускается только исправление ошибок. Вскоре после этого (можно надеяться) Линус выпускает первую версию ядра новой, стабильной серии. Например, разрабатываемая серия 1.3 стабилизировалась в версии 2.0, а при стабилизации серии ядер 2.5 получилась серия 2.6.

В рамках одной серии Линус регулярно выполняет новые выпуски ядра. При этом к номеру серии добавляется через точку номер нового выпуска. Например, первой версией ядра серии 2.6 была 2.6.0. Следующей была версия 2.6.1. В ней были исправлены ошибки, замеченные в предыдущем выпуске, добавлены новые драйверы устройств и новые возможности. Однако различия между двумя выпусками, как, например, между 2.6.3 и 2.6.4, незначительны.

Приведенное выше описание процесса разработки ядра технически правильное, и все так и происходило вплоть до 2004 года. Тем не менее летом 2004 года на ежегодной конференции для приглашенных разработчиков ядра Linux было принято решение продолжить разработку ядра Linux серии 2.6 и в ближайшем будущем не приступать к разработке ядра серии 2.7. Такое решение было принято потому, что ядро 2.6 получилось очень хорошим; в основном оно было стабильно, и не предвиделось никаких новых функций, которые требовали бы внесения серьезных изменений в ядро. Кроме того, ядро признали достаточно зрелым продуктом, вносить в который новые дестабилизирующие изменения было нецелесообразно.

Правильность такого курса была неоднократно подтверждена временем, поскольку на протяжении ряда последующих лет ядро серии 2.6 показало себя достаточно зрелым и многофункциональным продуктом. На момент написания данной книги разработка ядра серии 2.7 даже не стояла на повестке дня и выглядела маловероятной. Вместо этого цикл разработки ядра серии 2.6 был существенно удлинен. Теперь каждый выпуск был, по сути, серией мини-разработок. В результате Эндрю Мортон (Andrew Morton) (второй после Линуса человек в команде разработчиков) изменил предназначение своего дерева разработки ядра серии 2.6. Раньше оно использовалось для тестирования фундаментальных изменений, относящихся к процессу управления памятью, теперь же стало тестовым стендом общего назначения. Таким образом, дестабилизирующие изменения стали вноситься прямо в дерево разработки ядра серии 2.6 и со временем — в одну из мини-серий. Поэтому на протяжении последних нескольких лет каждый выпуск серии 2.6, например 2.6.29, занимал по времени несколько месяцев, причем в него вносились существенные изменения по сравнению с предшественником. Эти “серии разработки в миниатюре” оказались довольно успешными. Они обеспечивали высокий уровень стабильности ядра, в которое продолжали вноситься новые возможности и изменение которых в ближайшем будущем выглядело маловероятным. В самом деле, достижение согласия между разработчиками ядра в процессе выпуска новых версий, по-видимому, будет продолжаться бесконечно долго.

Для того чтобы компенсировать уменьшение частоты выпуска новых версий ядра, разработчики ввели упомянутое выше понятие *стабильного выпуска* (*stable release*). В них (например, в версии 8 из 2.6.32.8) исправлялись критические ошибки, которые часто привносились из разрабатываемой версии ядра (в нашем примере 2.6.33). Таким образом, предыдущий выпуск всегда находится под пристальным вниманием разработчиков в плане улучшения стабильности работы.

Сообщество разработчиков ядра Linux

Когда вы начинаете разрабатывать код ядра Linux, вы становитесь частью глобального сообщества разработчиков. Главный форум этого сообщества — *список рассылки разработчиков ядра Linux* (Linux-kernel mailing list, сокращенно *lkml*). Информация по поводу подписки на этот форум доступна по адресу `http://vger.kernel.org`. Следует заметить, что это достаточно перегруженный сообщениями список рассылки (количество сообщений составляет 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 тестировщиков, получить экспертную оценку и задать вопросы.

В последующих главах будет приведен обзор процесса разработки ядра и более полно описано, как успешно принимать участие в деятельности сообщества разработчиков ядра. Кроме того, пассивное чтение списка рассылки *Lkml* — это хороший источник дополнительных сведений, которых вы не найдете в этой книге.

Перед тем как начать

Эта книга посвящена ядру Linux и целям, поставленным перед разработчиками, проектным решениям, которые были направлены на достижение этих целей, и реализациям, которые позволят осуществить эти проектные решения. В книге описаны практические вопросы. При объяснении сложных понятий как и что работает в ней используется подход на основании золотой середины между теорией и практикой. Моя цель — дать читателю информацию из первых рук и помочь разобраться в проектных решениях и реализациях, положенных в основу ядра Linux. Такой интересный подход в сочетании с личным практическим опытом автора и советами по хакерским приемам позволяет быть уверенным в том, что книга станет хорошим подручным средством для тех, кто собирается разрабатывать код ядра, новый драйвер устройства или просто хочет лучше понять принципы работы ОС Linux.

Во время чтения книги у вас должен быть доступ к системе Linux и исходным кодам ее ядра. В идеале предполагается, что читатель — 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 мы рассмотрим отличия между ядром и пользовательскими программам, а также общие программные конструкции, которые используются в ядре. Хотя ядро Linux, вне всякого сомнения, является уникальным во многих отношениях, в конечном итоге оно мало чем отличается от любого другого большого современного программного проекта.

Где взять исходный код ядра

Исходный программный код последней версии ядра Linux всегда доступен как в виде полного архива в формате `.tar` (т.е. в виде архивного файла, созданного утилитой `tar`), так и в виде инкрементальной заплаты по адресу `http://www.kernel.org`.

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

Использование Git

Несколько лет назад ведущие разработчики ядра Linux под руководством Линуса Торвальдса начали использовать новую систему контроля версий для управления исходным кодом ядра. При создании этой системы, которую Линус назвал *Git*, во главу угла ставилась скорость работы. В отличие от традиционных систем контроля версий, таких как *CVS*, Git является распределенной системой. По этой причине рабочий процесс на ее основе и алгоритм использования оказались совершенно незнакомы большинству разработчиков. Тем не менее я настоятельно рекомендую использовать Git для загрузки исходного кода ядра Linux и управления им.

Чтобы получить с помощью Git копию последней зафиксированной версии дерева ядра Linux, введите приведенную ниже команду.

```
$ git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git
```

После загрузки исходного кода нужно обновить дерево до состояния последнего обновления, которое сделал Линус. Для этого введите следующее:

```
$ git pull
```

С помощью этих двух команд вы сможете развернуть у себя официальное дерево разработки ядра Linux и всегда поддерживать его в актуальном состоянии. О том, как вносить в него собственные изменения и фиксировать их в хранилище, речь пойдет в главе 20, "Заплаты, хакерство и сообщество". Полное описание возможностей Git выходит за рамки этой книги, однако в Интернете вы найдете большое количество великолепных руководств и инструкций по использованию.

Инсталляция исходного кода ядра

Архив `.tar` исходного кода ядра распространяется в сжатых форматах GNU zip (gzip) и bzip2. Формат bzip2 предпочтительнее, поскольку он 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`* — номер соответствующей версии исходного кода ядра. После загрузки файла его нужно распаковать с помощью простой команды. Если `.tar`-архив сжат с помощью утилиты `bzip2`, введите такую команду:

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

Если сжатие выполнено с помощью `gzip`, то команда должна иметь следующий вид:

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

Обе эти команды позволяют разархивировать и развернуть дерево исходных кодов ядра в каталог `linux-`*`x.y.z`*. При использовании Git для получения исходного кода ядра и управления им вам не нужно загружать архив в формате `.tar`. Просто введите команду `git clone`, как было описано в предыдущем разделе, и она сама загрузит и распакует самую последнюю версию дерева исходного кода ядра.

> **Где лучше всего инсталлировать и изменять исходный код**
>
> Исходный код ядра обычно инсталлируется в каталог `/usr/src/linux`. Заметим, что это дерево исходного кода нельзя использовать для собственных разработок. Версия ядра, с которой была скомпилирована ваша *библиотека C*, обычно привязана к этому дереву каталогов. Кроме того, чтобы вносить изменения в ядро, не обязательно иметь права пользователя `root`, вместо этого лучше работать в собственном рабочем каталоге и использовать права пользователя `root` только для инсталляции ядра. Даже при инсталляции нового ядра каталог `/usr/src/linux` должен оставаться без изменений.

Использование заплат

В сообществе разработчиков ядра Linux *заплаты* (patch) — это основной *язык общения*. Дело в том, что разработчики обмениваются друг с другом внесенными ими изменениями в исходном коде ядра в виде заплат. При этом наиболее важными являются *инкрементальные заплаты* (incremental patch), которые позволяют перейти от одного дерева ядра к другому. Вместо того чтобы загружать большой архив ядра, можно просто применить инкрементальную заплату и перейти от имеющейся версии дерева ядра к следующей. Это позволяет сэкономить время и не нагружать лишний раз каналы связи. Для

того чтобы применить инкрементную заплату, перейдите к дереву каталогов исходных кодов ядра и введите приведенную ниже команду.

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

Как правило, заплата, предназначенная для перехода на некоторую версию ядра, должна применяться к предыдущей версии ядра. Процесс генерации и применения заплат будет подробнее описан в следующих главах.

Дерево каталогов 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.

Таблица 2.1. Дерево исходных кодов ядра

Каталог	Описание
arch	Исходный код, специфичный для аппаратной платформы
block	Слой блочных операций ввода-вывода
crypto	Криптографический API
Documentation	Документация исходного кода ядра
drivers	Драйверы устройств
firmware	Микропрограммы устройств, необходимые для использования некоторых драйверов
fs	Подсистема VFS и отдельные файловые системы
include	Заголовочные файлы ядра
init	Коды для загрузки и инициализации ядра
ipc	Код межпроцессного взаимодействия
kernel	Основные подсистемы ядра, такие как планировщик
lib	Вспомогательные подпрограммы
mm	Подсистема управления памятью и поддержка виртуальной памяти
net	Сетевая подсистема
samples	Примеры и демонстрационный код
scripts	Сценарии компиляции и построения ядра
security	Модуль безопасности Linux
sound	Звуковая подсистема
tools	Вспомогательные утилиты для разработки Linux
usr	Код инициализации пространства пользователя (initramfs)
virt	Инфраструктура виртуализации

Ряд файлов, находящихся в корне дерева исходных кодов, также заслуживают отдельного 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 файл построения ядра.

Сборка ядра

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

Конфигурирование ядра

Поскольку исходный код ядра Linux доступен, то это означает, что есть возможность сконфигурировать ядро перед компиляцией. В результате у вас появляется возможность скомпилировать в ядро поддержку только необходимых драйверов и функций. Конфигурирование ядра — необходимый этап перед его компиляцией. Поскольку в ядре существует бесчисленное количество функций и вариантов поддерживаемого аппаратного обеспечения, возможностей по конфигурации, мягко говоря, *много*. Процесс конфигурации выполняется с помощью указания ряда параметров конфигурации в виде `CONFIG_ВОЗМОЖНОСТЬ`. Например, поддержка симметричной многопроцессорной обработки (Symmetric multiprocessing, SMP) устанавливается с помощью параметра `CONFIG_SMP`. Если этот параметр установлен, то поддержка функций SMP включена. Если не установлен, то функции поддержки SMP отключены. Параметры конфигурации используются как для определения того, какие файлы должны быть скомпилированы во время сборки ядра, так и для управления процессом компиляции через директивы препроцессора.

Параметры конфигурации, которые управляют процессом сборки ядра, бывают двух типов: логические (*boolean*) и с тремя состояниями (*tristate*). Логические параметры могут принимать значение `yes` или `no`. Как правило, параметры конфигурации ядра наподобие `CONFIG_PREEMPT` являются логическими. Параметры конфигурации с тремя состояниями могут принимать значение `yes`, `no` или `module`. Значение `module` означает, что данный параметр конфигурации установлен, но соответствующий ему код должен быть скомпилирован в виде модуля (т.е. как отдельный объект, который загружается динамически). В случае параметров конфигурации с тремя состояниями значение `yes` явно указывает на то, что соответствующий код должен быть скомпилирован в основной файл образа ядра, а не в виде модуля. Драйверы устройств обычно определяются параметрами конфигурации с тремя состояниями.

Параметры конфигурации могут также задаваться в виде строк символов или чисел. Эти параметры не управляют процессом сборки ядра, а позволяют указать значения, которые встраиваются в исходный код с помощью директив препроцессора. Например, с помощью параметра конфигурации можно указать размер статически размещаемого массива.

Ядра, которые включаются в поставки ОС Linux такими производителями, как Canonical (для Ubuntu) или Red Hat (для Fedora), 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 спектра оборудования в виде дополнительных модулей. Однако в любом случае разработчикам ядра нужно компилировать собственные ядра и самим решать, какие модули включать, а какие нет.

К счастью, в ядре поддерживается несколько средств, с помощью которых можно выполнить конфигурацию. Самое простое средство — 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 графическую утилиту на основе интерфейса *ncurses*:

```
$ make menuconfig
```

Имеется также и более удобная графическая утилита, использующая библиотеку *gtk+*:

```
$ make gconfig
```

В этих трех утилитах все параметры конфигурации делятся на категории наподобие Processor Type and Features (Типы и свойства процессора). При этом пользователь имеет возможность перемещаться по категориям, просматривать параметры конфигурации ядра и, разумеется, изменять их значения.

Приведенная ниже команда позволяет создать файл конфигурации, содержащий стандартные значения параметров для текущей аппаратной платформы.

```
$ make defconfig
```

И хотя значения этих параметров установлены достаточно произвольно (ходят слухи, что для аппаратной платформы i386 используется конфигурация Линуса), они являются хорошей отправной точкой, если ранее вам никогда не приходилось заниматься конфигурацией ядра. Чтобы ускорить процесс конфигурации, запустите эту команду, а потом проверьте, включена ли поддержка всех нужных вам аппаратных средств.

Параметры конфигурации сохраняются в файле `.config`, который находится в корне дерева каталогов исходного кода ядра. Большинство разработчиков считают, что гораздо проще непосредственно отредактировать этот файл конфигурации. В самом деле, с помощью любого текстового редактора достаточно легко выполнить поиск нужного параметра в этом файле и изменить его значение. После внесения изменений в файл конфигурации или при использовании существующего файла конфигурации для нового дерева каталогов исходного кода ядра его необходимо ратифицировать и обновить конфигурацию с помощью команды

```
$ make oldconfig
```

Эту команду вам также нужно запустить перед сборкой ядра.

Параметр конфигурации `CONFIG_IKCONFIG_PROC` позволяет сохранить все текущие параметры конфигурации ядра в виде сжатого файла `config.gz` и поместить его в каталог `/proc`. Это позволит вам легко создать клон текущей конфигурации при сборке нового ядра. Если этот параметр конфигурации установлен в вашем текущем ядре, то просто скопируйте файл `config.gz` из каталога `/proc` и воспользуйтесь им для создания нового ядра, как показано ниже.

```
$ zcat /proc/config.gz > .config
$ make oldconfig
```

После того как будет сконфигурировано ядро (как бы там ни было, вы должны были сделать это!), можно выполнить сборку с помощью команды

```
$ make
```

В отличие от предыдущих серий ядер, в серии 2.6 больше нет необходимости перед сборкой ядра выполнять команду `make dep`, так как дерево зависимостей создается автоматически. Также больше не нужно указывать конкретный тип сборки, например *bzImage*,

или отдельно собирать модули, как делалось в более ранних версиях. Стандартное правило, записанное в файле `Makefile`, в состоянии обработать все!

Уменьшение количества выводимых сообщений

Существует один прием, который поможет уменьшить количество сообщений, выводимых на экран во время сборки ядра, и в то же время позволит увидеть все предупреждения компилятора и сообщения об ошибках. Он заключается в перенаправлении стандартного устройства вывода команды `make` в файл, как показано ниже.

```
$ make > ../detritus
```

Если вам понадобится проанализировать выводимые сообщения, откройте этот файл в любом текстовом редакторе. Поскольку предупреждения и сообщения об ошибках по умолчанию перенаправляются на стандартное устройство вывода ошибок и отображаются на экране терминала, в этой процедуре нет особого смысла. Поэтому я выполняю приведенную ниже команду.

```
$ make > /dev/null
```

В результате весь огромный объем бессмысленных сообщений будет перенаправлен в "большую сточную канаву" `/dev/null`.

Порождение нескольких параллельных задач сборки

Программа `make` позволяет разбить процесс сборки ядра на несколько параллельно выполняющихся *заданий*. Каждое из этих заданий выполняется независимо от остальных и одновременно с остальными, что существенно ускоряет процесс сборки ядра на многопроцессорных системах. Кроме того, параллельная сборка позволяет также более эффективно использовать центральный процессор компьютера, поскольку при компиляции большого дерева исходного кода он значительное время простаивает, ожидая завершения операций ввода-вывода.

По умолчанию утилита `make` запускает только одну задачу, так как часто файлы сборки пользователей содержат некорректную информацию о зависимостях. При этом несколько заданий могут начать "наступать друг другу на пятки", что приведет к ошибкам в процессе построения. Разумеется, в файле сборки ядра `Makefile` таких ошибок нет, поэтому порождение нескольких задач не приведет к ошибкам построения. Для построения ядра с использованием нескольких параллельных задач сборки выполните команду

```
$ make -jn
```

где число *n* определяет количество порождаемых заданий. Как правило, на один процессор запускается одно или два задания. Например, на 16-процессорной машине можно воспользоваться приведенной ниже командой.

```
$ make -j32 > /dev/null
```

Используя такие великолепные утилиты, как `distcc` и `ccache`, можно еще больше ускорить время сборки ядра.

Инсталляция нового ядра

После завершения процесса сборки ядра его нужно инсталлировать в системе. Процесс инсталляции существенно зависит от используемой аппаратной платформы и типа системного загрузчика. Для того чтобы узнать, в какой каталог должен быть скопирован

образ ядра и как сделать его загружаемым, обратитесь к руководству по используемому системному загрузчику. При этом всегда сохраняйте копии одного-двух старых ядер, которые гарантированно работоспособны! Это на случай, если новое ядро не будет нормально работать или вовсе откажется загружаться.

Например, для платформы x86 при использовании системного загрузчика `grub` скопируйте загружаемый образ ядра из файла `arch/i386/boot/bzImage` в каталог `/boot` и переименуйте его в, скажем, `vmlinuz-`*`версия`*. После этого отредактируйте файл `/boot/grub/grub.conf`, добавив в него новую запись, которая соответствует новому ядру. В системах, где используется загрузчик LILO, необходимо соответственно отредактировать файл `/etc/lilo.conf` и запустить утилиту `lilo`.

К счастью, инсталляция модулей ядра автоматизирована и не зависит от аппаратной платформы. Для этого просто запустите следующую команду с правами пользователя `root`:

```
% make modules_install
```

В результате все скомпилированные модули будут инсталлированы в их корректные домашние каталоги, находящиеся в иерархии каталога `/lib/modules`.

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

Отличия от обычных приложений

Ядро имеет некоторые уникальные отличительные особенности по сравнению с обычными пользовательскими приложениями. И хотя эти отличия не всегда приводят к усложнениям при разработке кода ядра по сравнению с пользовательскими приложениями, все же они делают сам процесс разработки немного *другим*.

Эти отличительные особенности делают ядро совершенно непохожим ни на что. Некоторые из привычных вещей в нем немного видоизменены, другие — полностью обновлены. Несмотря на то что 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 и к стандартным заголовкам языка C.
- Код ядра написан с использованием компилятора GNU C.
- В ядре нет такой защиты памяти, как в режиме пользователя.
- В ядре нельзя также просто использовать вычисления с плавающей точкой, как в пользовательских приложениях.
- Ядро использует стек небольшого фиксированного размера для каждого процесса.
- Поскольку обработка прерываний в ядре выполняется асинхронно, в нем реализован режим приоритетного планирования. Поддержка симметричной многопроцессорной обработки (SMP) привела к тому, что в ядре необходимо учитывать наличие параллелизма и использовать синхронизацию.
- Переносимость кода ядра очень важна.

Рассмотрим более детально перечисленные выше пункты, поскольку все разработчики ядра должны постоянно иметь их в виду.

Отсутствие библиотеки `libc` и стандартных заголовков

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

Однако этот факт не должен приводить вас в замешательство, поскольку многие из функций библиотеки языка C реализованы непосредственно в ядре. Например, обычные функции работы со строками находятся в файле `lib/string.c`. Чтобы воспользоваться ими, нужно лишь включить в исходный код заголовочный файл `<linux/string.h>`.

> **Заголовочные файлы**
>
> Обратите внимание на то, что под термином *заголовочные файлы*, используемом в этой книге, имеются в виду заголовочные файлы ядра, принадлежащие его дереву исходного кода. В файлы исходного кода ядра нельзя включать заголовочные файлы, расположенные вне этого дерева каталогов, а также использовать внешние библиотеки.
>
> Основные файлы находятся в подкаталоге `include/`, расположенном в корне дерева исходного кода ядра. Например, заголовочный файл `<linux/inotify.h>` расположен в подкаталоге `include/linux/inotify.h` дерева исходных кодов.
>
> Набор заголовочных файлов, зависимых от аппаратной платформы, расположен в подкаталоге `arch/<`*`платформа`*`>/include/asm` дерева исходных кодов ядра. Например, при сборке ядра для платформы x86 все аппаратно-зависимые заголовочные файлы находятся в подкаталоге `arch/x86/include/asm`. Для включения этих файлов в исходный код используется префикс `asm/`, например `<asm/ioctl.h>`.

Что касается отсутствующих функций, то самая известная из них — `printf()`. Ядро не имеет доступа к функции `printf()`, однако в нем реализована функция `printk()`, которая работает ничем не хуже своего аналога. Функция `printk()` копирует отформатированную строку в буфер системных сообщений ядра (kernel log buffer), который обычно считывается с помощью программы `syslog`. Использование этой функции аналогично использованию `printf()`:

```
printk("Привет, всем! Строка '%s' и целое число '%d'\n", str, i);
```

Одно важное отличие между функциями `printf()` и `printk()` состоит в том, что в функции `printk()` можно использовать флаг важности вывода (priority flag). Этот флаг используется программой `syslog` для того, чтобы определить, где именно нужно отображать сообщения ядра. Ниже приведен пример использования этого флага.

```
printk(KERN_ERR "this is an error!\n");
```

Обратите внимание на то, что между `KERN_ERR` и выводимым сообщением отсутствует запятая. Это сделано преднамеренно. Флаг важности вывода определен с помощью директивы препроцессора как строковый литерал, который во время компиляции автоматически добавляется к выводимому сообщению. Функция `printk()` будет использоваться на протяжении всей книги.

Компилятор GNU C

Как и все "уважающие себя" ядра Unix, ядро Linux написано на языке C. Может быть, это покажется странным, но ядро Linux написано не на чистом языке C стандарта ANSI C. Там, где это только возможно, разработчики ядра используют различные расширения языка, реализованные в компиляторе *gcc* (GNU Compiler Collection — это набор компиляторов GNU, в котором содержится компилятор C, используемый для компиляции ядра, а также всего остального программного обеспечения, написанного на языке C для системы Linux).

Разработчики ядра используют как расширения языка ISO C99[1], так и расширения GNU C. Все это привело к тому, что ядро Linux было привязано к компилятору `gcc`. Несмотря на это, для компиляции ядра Linux можно также использовать любой другой современный компилятор, например Intel C, в котором достаточно хорошо реализована поддержка функций компилятора `gcc`. Самой старой поддерживаемой версией компилятора `gcc` является 3.2, однако настоятельно рекомендуется использовать версию 4.4 или более позднюю. В исходном коде ядра Linux не используются какие-либо особенные расширения стандарта C99. Кроме того, поскольку стандарт C99 является официальной редакцией языка C, эти расширения постепенно начинают использоваться и в других частях кода. Более интересные и, возможно, менее знакомые отклонения от стандарта языка ANSI C связаны с расширениями GNU C. Рассмотрим некоторые наиболее интересные расширения, которые могут встретиться в исходном коде ядра. Именно они отличают исходный код ядра от исходного кода других программных проектов, в которых, возможно, вам приходилось участвовать.

Встраиваемые функции

В стандарте C99 определены так называемые *встраиваемые функции* (inline functions), которые поддерживает компилятор GNU C. Как следует из названия, исполняемый код функции автоматически помещается компилятором во все места программы, где указан вызов этой функции. Это позволяет избежать дополнительных потерь времени процессора, связанных с вызовом функции и возвратом из нее (сохранение и восстановление регистров), а также позволяет потенциально повысить уровень оптимизации программы, поскольку компилятор теперь может одновременно оптимизировать коды и вызывающей, и вызываемой функции. Обратной стороной такой подстановки (ничто в этой жизни не дается даром!) является увеличение объема кода, увеличение используемой памяти и уменьшение эффективности использования процессорной кеш-памяти команд. Поэтому разработчики ядра используют подстановку для небольших функций, критичных ко времени выполнения. Настоятельно не рекомендуется использовать эту же возможность для больших функций, особенно когда они вызываются больше одного раза или не слишком критичны ко времени выполнения.

Встраиваемые функции объявляются с помощью ключевых слов `static` и `inline`, которые указываются при определении функции, как показано ниже.

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

[1] Стандарт ISO C99 — это последняя основная версия стандарта ISO C. По сравнению с предыдущей редакцией стандарта ISO C90 стандарт C99 был существенно расширен и дополнен. Так, в нем появились *выделенные инициализаторы* (*designated initializers*), позволяющие выполнять поименную инициализацию отдельных полей структур, массивы переменной длины, комментарии в стиле языка C++, типы `long long` и `complex`. Тем не менее при написании ядра Linux используется только ограниченный набор возможностей стандарта C99.

Объявление функции должно предшествовать ее первому вызову, иначе компилятор не сможет выполнить подстановку. Обычно встраиваемые функции определяются в заголовочных файлах. Поскольку функция объявляется как статическая (`static`), экспортируемая функция при этом не создается. Если встраиваемая функция используется только в одном файле, то она может быть размещена в верхней части этого файла.

При написании кода ядра следует отдавать преимущество встраиваемым функциям, а не использовать вместо них сложные макросы. Это требование связано с читабельностью кода и его надежностью.

Встроенный ассемблер

Компилятор `gcc` 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()`. Например, приведенная ниже встроенная директива ассемблера выполняет команду `rdtsc` процессора x86, которая возвращает значение счетчика времени (TSC — Time Stamp Counter) работы процессора, прошедшего с момента последнего сброса.

```
unsigned int low, high;
asm volatile("rdtsc" : "=a" (low), "=d" (high));
/* Переменные low и high теперь содержат значение старших и младших
   32-разрядов 64-разрядного регистра tsc */
```

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

Комментирование ветвлений

Компилятор `gcc` имеет встроенные директивы, позволяющие оптимизировать различные ветви условных операторов, выполнение которых наиболее или наименее вероятно. Компилятор использует эти директивы для оптимизации кода соответствующих веток. В ядре эти директивы заключены в макросы `likely()` и `unlikely()`, которые легко использовать.

Например, рассмотрим приведенный ниже оператор `if`.

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

Для того чтобы отметить эту ветку как крайне маловероятную (т.е. управление которой вряд ли когда-либо будет передано), необходимо указать следующее:

```
/* Мы предполагаем, что переменная 'error' практически всегда
   будет равна нулю... */
if (unlikely(error)) {
    /* ... */
}
```

И наоборот, чтобы отметить приведенную ниже ветку как наиболее вероятную, нужно записать так:

```
/* Мы предполагаем, что переменная 'success' практически
   всегда не будет равна нулю... */
if (likely(success)) {
   /* ... */
}
```

Эти директивы необходимо использовать только в том 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 увеличивают производительность, когда направление ветвления предсказано правильно, однако приводят к *потере* производительности при неправильном предсказании. Чаще всего директивы `unlikely()` и `likely()` используются для проверки ошибок, как показано выше в примерах. Как вы уже могли догадаться, в коде ядра чаще всего используется директива `unlikely()`, поскольку обычно в операторе `if` проверяется условие наступления какого-либо особого случая.

Отсутствие защиты памяти

Когда прикладная программа предпринимает незаконную попытку обращения к памяти, ядро может перехватить эту ошибку, отправить приложению сигнал `SIGSEGV` и аварийно завершить соответствующий пользовательский процесс. Если же ядро предпринимает попытку некорректного обращения к памяти, то результаты могут быть менее контролируемы. В конце концов, кто может контролировать само ядро? Нарушение правил доступа к памяти в режиме ядра приводит к ошибке *oops*, которая является наиболее часто встречающейся 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 имеет страничной организации. Поэтому каждый сэкономленный байт памяти в ядре — это еще один байт доступной физической памяти для пользовательских приложений. Помните об этом всякий раз, когда нужно будет добавить очередную новую и полезную *функцию ядра*.

Нельзя просто использовать вычисления с плавающей точкой

Когда в пользовательской программе используются команды 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 команд с плавающей точкой, зависят от типа аппаратной платформы. Но обычно при этом ядро перехватывает системное прерывание, после чего инициирует переход из режима вычислений с целыми числами в режим работы с плавающей точкой.

В отличие от пользовательского приложения, в режиме ядра нет такой роскоши, как прямое использование команд с плавающей точкой, поскольку ядро не может легко перехватить системное прерывание, возникшее в режиме ядра. При использовании режима вычислений с плавающей точкой в режиме ядра, кроме прочих рутинных операций, требуется вручную сохранять и восстанавливать состояние регистров с плавающей точкой математического сопроцессора. Если говорить коротко, то лучше всего *этого никогда не делать*! За исключением редких особых случаев нельзя выполнять вычисления с плавающей точкой в режиме ядра!

Системная стек-память небольшого 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 вполне законным в пользовательском режиме, поскольку область стека пользовательских программ имеет огромный размер, который

к тому же по необходимости может динамически увеличиваться в размере. Разработчики, которые писали программы под старые и не очень интеллектуальные операционные системы, как, например, MS-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 конфигурируется на этапе компиляции и может быть равен 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.

В следующих главах мы обсудим системный стек более подробно.

Синхронизация и параллельное выполнение

В ядре постоянно возникают конфликты из-за доступа к системным ресурсам. В отличие от однопоточной пользовательской программы, некоторые задачи ядра предполагают одновременный доступ к общим системным ресурсам, из-за чего могут возникать конфликтные ситуации. Для их разрешения используется механизм синхронизации. В частности, возможны следующие ситуации.

- В ядре Linux поддерживается приоритетный мультипрограммный режим работы. Алгоритм планирования процессов выполняется особым модулем ядра, который называется системным планировщиком (scheduler). Поэтому ядро должно осуществлять синхронное выполнение этих задач.
- В ядре Linux поддерживается симметричная многопроцессорная обработка. Поэтому, без всякого предупреждения, код ядра, выполняемый одновременно на двух или более процессорах, может одновременно обратиться к одному и тому же системному ресурсу.
- Прерывания возникают асинхронно по отношению к исполняемому коду. Поэтому, без всякого предупреждения, прерывания могут возникнуть во время обращения к ресурсу общего доступа, кроме того, обработчик прерывания также может обратиться к этому же ресурсу.
- Ядро Linux работает в мультипрограммном режиме. Поэтому, без всякого предупреждения, планировщик может прервать текущую исполняемую задачу ядра и переключиться на выполнение любой другой задачи ядра, которая также может обратиться к некоторому общему ресурсу.

Стандартное решение для предотвращения конфликтных ситуаций за ресурсы состоит в использовании спин-блокировок и семафоров. Более подробное обсуждение вопросов синхронизации и параллелизма приведено в следующих главах.

Переносимость — это важно

При разработке пользовательских программ переносимость *не всегда* ставится во главу угла, однако операционная система Linux является переносимой и должна оставаться таковой. Это означает, что аппаратно-независимый код, написанный на языке C, должен компилироваться без ошибок и правильно выполняться на большом количестве

систем. С другой стороны, аппаратно-зависимый код должен быть корректно разнесен по соответствующим каталогам дерева исходных кодов ядра.

Несколько правил, таких как не создавать зависимости от порядка следования байтов, обеспечивать возможность использования кода на 64-разрядных системах, не привязываться к размеру страницы памяти или машинного слова и другие, имеют большое значение. Вопросы переносимости более подробно рассматриваются в одной из следующих глав.

Резюме

Как вы уже поняли, ядро Linux обладает уникальными качествами. В нем принят ряд особых правил и исключений, которые являются очень важными и влияют на работу всей системы. Выше мы уже говорили, что сложность кода ядра и необходимый уровень подготовки программистов для его разработки принципиально ничем не отличаются от любого другого большого программного проекта. Самым важным шагом на пути к разработке ядра Linux является осознание того, что ядро не так уж и страшно, как оно кажется на первый взгляд. Незнакомо? Конечно! Непреодолимо? Не для всех!

Вводный материал, который был представлен в первой главе, и базовые моменты, описанные в текущей, надеюсь, станут хорошим фундаментом для тех знаний, которые будут получены при прочтении всей книги. В каждой из последующих глав будут описаны конкретные подсистемы ядра и принципы их работы. Помимо всего прочего, очень важно, чтобы вы самостоятельно смогли разобраться в исходном коде ядра и внести в него изменения. Только в процессе реального знакомства с исходным кодом и выполнения экспериментов с ним вы сможете все понять. В конце концов, исходный код доступен совершенно свободно! Так пользуйтесь этой возможностью!

3
Управление процессами

В этой главе мы ознакомимся с концепцией процессов (process) — одной из основополагающих абстракций операционной системы Unix. Здесь будет дано определение понятию *процесс*, а также ряду связанных с ним понятий, таким как потоки (threads). После этого мы обсудим методы управления каждым процессом, выполняемые в ядре: способы их нумерации, создания и завершения. Поскольку конечной целью создания операционной системы является возможность запуска пользовательских программ, управление процессами — это ключевая часть любой операционной системы, включая Linux.

Понятие процесса

Процесс — это программа (т.е. объектный код, хранящийся на каком-либо носителе информации), которая находится в состоянии выполнения. Однако процесс — это не только исполняемый программный код, который в операционной системе Unix часто называют *сегментом кода* (text section). В процесс также включается набор ресурсов, таких как открытые файлы и сигналы, ожидающие обработки, внутренние данные ядра, состояние процессора, адресное пространство памяти, в которое отображается один или несколько объектов, один или несколько *потоков выполнения кода* (threads of execution), а также *сегмент данных* программы (data section), содержащий глобальные переменные. По сути, процессы — это живой результат выполнения программного кода. Поэтому ядро должно управлять всеми этими элементами эффективно и прозрачно для пользовательского приложения.

Потоки выполнения кода, которые часто для краткости называют *потоками* (threads), представляют собой объекты, выполняющие определенные действия внутри процесса. В каждом из потоков содержатся уникальный *счетчик команд* (program counter), стек процесса и набор регистров процессора. Планировщик ядра управляет выполнением отдельных потоков, а не процессов. В традиционных Unix-подобных операционных системах каждый процесс содержал только один поток. Однако в современных системах многопоточные программы (т.е. программы, в которых запускается более одного потока выполнения команд) используются очень широко. Как будет показано ниже, в операционной системе Linux используется уникальная реализация

потоков, в которой 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*. Благодаря виртуальному процессору для пользовательских процессов создается иллюзия, что они монопольно используют всю компьютерную систему, несмотря на то, что физическим процессором могут одновременно пользоваться десятки других процессов. Понятие виртуального процессора более подробно обсуждается в главе 4, "Системный планировщик и диспетчеризация процессов". Виртуальная память позволяет процессу распределять оперативную память компьютера и управлять ею так, как будто он один владеет всей памятью в системе. Виртуальная память будет описана в главе 12, "Управление памятью". Интересно отметить, что потоки располагаются в общей виртуальной памяти процесса, но каждый из них выполняется на собственном виртуальном процессоре.

Следует подчеркнуть, что сама по себе программа процессом не является; процесс — это *выполняющаяся* программа и набор соответствующих ресурсов. Разумеется, может существовать несколько процессов, которые выполняют *одну и ту же* программу. По сути, может даже существовать несколько 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, и не удивительно. В операционной системе Linux это происходит в результате вызова системной функции `fork()` (буквально — ветвление или вилка), которая создает новый процесс путем полного копирования уже существующего. Процесс, который вызвал функцию `fork()`, называется *порождающим*, или *родительским* (parent), а новый процесс именуют *порожденным*, или *дочерним* (child). Родительский процесс после этого продолжает свое выполнение, а дочерний начинает выполняться с одного и того же места — с момента возврата из системной функции `fork()`. В результате возврат из системной функции `fork()` выполняется из ядра дважды: один раз в родительском процессе, а второй раз — в порожденном.

Часто после разветвления в одном из процессов желательно выполнить какую-нибудь другую программу. Для этого существует семейство функций `exec()`, которое позволяет создать новое адресное пространство и загрузить в него новую программу. В современных ядрах Linux функция `fork()` на самом деле реализована через вызов системной функции `clone()`, которая будет рассмотрена в следующем разделе.

В конечном счете выход из программы осуществляется с помощью вызова системной функции `exit()`, которая завершает выполнение процесса и освобождает все занятые им ресурсы. Родительский процесс может выдать запрос о состоянии завершенных дочерних процессов с помощью вызова системной функции `wait4()`[1], которая позволяет одному процессу подождать завершения другого указанного процесса. После завершения процесса он переводится 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()`.

[1] В ядре реализована системная функция `wait4()`. Однако обычно в операционной системе Linux можно пользоваться следующими функциями библиотеки языка C: `wait()`, `waitpid()`, `wait3()` и `wait4()`. Все они возвращают информацию о состоянии завершившегося процесса, хотя и в несколько разной семантике.

На заметку

Существует и другое название для процесса — *задача* (task). Процессы, порожденные в ядре Linux, называют задачами. В этой книге оба этих понятия взаимозаменяемы, хотя я буду стараться называть работающие в ядре программы *задачами*, а выполняющиеся пользовательские программы — *процессами*.

Дескриптор процесса и структура `task_struct`

Список процессов хранится в ядре в виде циклического двухсвязного списка, который называется *списком задач* (task list).[2] Каждый элемент этого списка описывает один запущенный процесс и называется *дескриптором процесса*. Дескриптор процесса имеет тип `task_struct`, структура которого описана в файле `<linux/sched.h>`. Дескриптор процесса содержит всю информацию об определенном процессе.

Структура `task_struct` достаточно большая. На 32-разрядном компьютере она занимает порядка 1,7 Кбайт памяти. Однако этот размер не такой уж большой, учитывая, что в данной структуре содержится вся информация о процессе, которая необходима ядру. В дескрипторе процесса содержатся данные, которые описывают выполняющуюся программу, — открытые файлы, адресное пространство процесса, сигналы, ожидающие обработки, состояние процесса и многое другое (рис. 3.1).

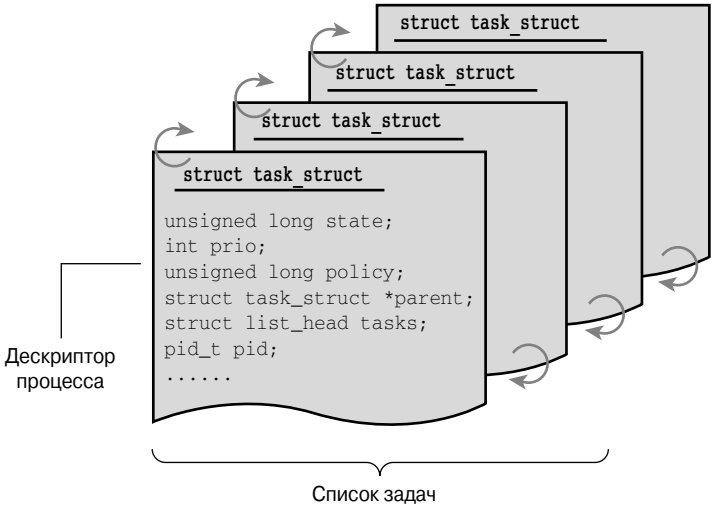


Рис. 3.1. Дескриптор процесса и список задач

Распределение памяти под дескриптор процесса

Память под структуру `task_struct` выделяется с помощью специальной *программы распределения блочного типа* (*slab allocator*), обеспечивающей эффективное повторное использование объектов и *раскрашивание стека* (cache coloring) (подробнее об этом —

[2] Иногда в литературе, посвященной проектированию операционных систем, этот список называется *массивом задач* (task array). Поскольку при реализации в ядре Linux использовался именно связанный список, а не статический массив, мы будем называть его списком задач.

в главе 12, “Управление памятью”). В ядрах до версии 2.6 структура `task_struct` размещалась в конце системного стека (стека ядра), выделяемого для каждого процесса. Это позволяло достаточно эффективно вычислять адрес дескриптора процесса по значению *указателя стека* (stack pointer) и не требовало отдельного регистра процессора для хранения этого адреса. Такой подход был принят в связи с тем, что на популярных аппаратных платформах, в частности, таких как x86, в центральном процессоре имелось всего несколько регистров общего назначения. Поскольку теперь дескриптор процесса создается динамически с помощью специальной программы распределения блочного типа, была введена новая структура под названием `thread_info`, которая опять же размещается 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)[3] (рис. 3.2.).

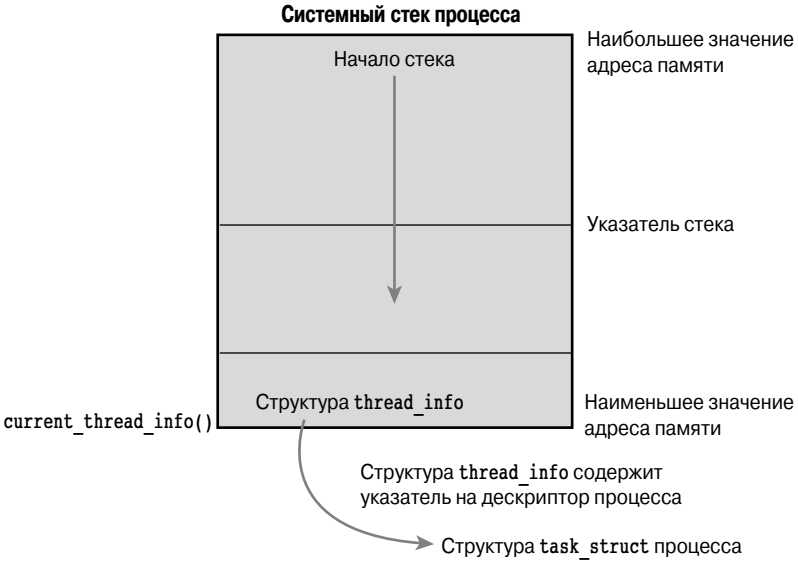


Рис. 3.2. Дескриптор процесса и системный стек

Определение структуры `thread_info` для платформы x86 находится в файле `<asm/thread_info.h>` и имеет приведенный ниже вид.

```
struct thread_info {
       struct task_struct        *task;
       struct exec_domain        *exec_domain;
       __u32                      flags;
       __u32                      status;
       __u32                      cpu;
       int                        preempt_count;
       mm_segment_t               addr_limit;
       struct restart_block       restart_block;
```

[3] Причиной создания структуры `thread_info` было не только наличие аппаратных платформ с малым количеством регистров процессора общего назначения, но и тот факт, что положение этой структуры позволяет достаточно просто рассчитывать смещения адресов ее полей при использовании языка ассемблера.

```
        void                    *sysenter_return;
        int                     uaccess_err;
};
```

Структура `thread_info` размещается в конце системного стека, выделенного для каждой задачи. В ней есть элемент `task`, который указывает на реальную структуру дескриптора задачи `task_struct`.

Сохранение дескриптора процесса

Система идентифицирует процессы по значению уникального *идентификатора процесса* (process identification, PID). Идентификатор `PID` — это целое число, представленное с помощью скрытого типа `pid_t`[4], который обычно соответствует целому типу со знаком `int`. Однако для обратной совместимости со старыми версиями ОС Unix и Linux максимальное значение этого параметра по умолчанию ограничено значением `32768` (что соответствует типу данных `short int`). По желанию его можно увеличить до 4 млн. Для этого нужно отредактировать файл `<linux/threads.h>`. Идентификатор процесса сохраняется ядром в поле `pid` каждого дескриптора процесса.

Это максимальное значение является важным, потому что оно определяет максимальное количество процессов, которые одновременно могут существовать в системе. Хотя значения 32768 вполне достаточно для офисного компьютера, для больших серверов может потребоваться запустить значительно больше процессов. Кроме того, 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 обратная совместимость со старыми приложениями не нужна, то администратор может увеличить максимальное значение идентификатора процесса прямо во время работы системы, отредактировав файл `/proc/sys/kernel/pid_max`.

Обычно в ядре для идентификации задачи используется прямая ссылка на соответствующую структуру `task_struct`. По сути, в бо́льшей части кода ядра, работающего с процессами, выполняется обращение прямо к полям структуры `task_struct`. Следовательно, нам нужно иметь возможность быстро находить дескриптор процесса, выполняемого в данный момент. Специально для этого создан макрос `current`. Конкретная реализация этого макроса зависит от используемой аппаратной платформы. В некоторых аппаратных платформах указатель на структуру `task_struct`, соответствующую выполняемому в настоящий момент процессу, сохраняется в специальном регистре процессора, что намного ускоряет доступ к ней. В ряде других архитектур, в частности x86, с ограниченным набором регистров процессора общего назначения используется тот факт, что структура `thread_info` располагается в стеке. Это позволяет очень эффективно вычислить ее адрес и, соответственно, определить адрес структуры `task_struct`.

Чтобы получить адрес структуры `thread_info` на платформе x86 и определить значение параметра `current`, нужно замаскировать младшие 13 бит указателя стека. Для этой цели предусмотрена функция `current_thread_info()`. Соответствующий код на языке ассемблера приведен ниже.

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

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

В этом фрагменте кода подразумевается, что размер стека составляет 8 Кбайт. Если размер стека составляет 4 Кбайт, вместо 8192 нужно использовать число 4096.

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

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

Для сравнения посмотрим, как определяется адрес дескриптора процесса на платформе PowerPC, оснащенной современным RISC-процессором компании IBM. В ней адрес структуры `task_struct` хранится в одном из регистров общего назначения процессора. Поэтому все, что нужно сделать макросу `current`, — это извлечь значение, хранящееся в регистре `r2`. На платформе PowerPC такой подход можно использовать, так как, в отличие от платформы x86, здесь регистры процессора доступны в изобилии[5]. Поскольку доступ к дескриптору процесса — очень частая и важная операция, разработчики ядра для платформы PowerPC сочли возможным пожертвовать одним регистром общего назначения для этой цели.

Состояние процесса

Текущее состояние процесса описывается в поле `state` дескриптора процесса. Каждый процесс в системе гарантированно находится в одном из пяти различных состояний (рис. 3.3). Эти состояния представляются значением одного из пяти возможных флагов, описанных ниже.

- **`TASK_RUNNING`**. Процесс готов к выполнению (runnable). Иными словами, процесс либо выполняется в данный момент, либо находится в очереди процессов, которые ожидают выполнения (эти очереди, `runqueue`, обсуждаются в главе 4, “Системный планировщик и диспетчеризация процессов”). Для пользовательского процесса данное состояние является единственно возможным. Активно выполняющийся системный процесс также может находиться в этом состоянии.
- **`TASK_INTERRUPTIBLE`**. Процесс приостановлен (находится в состоянии ожидания, *sleeping*), т.е. заморожен до наступления некоторого события. При наступлении данного события ядро переводит процесс в состояние `TASK_RUNNING`. Процесс также можно досрочно разморозить и перевести в исполняемое состояние, послав специальный сигнал.
- **`TASK_UNINTERRUPTIBLE`**. Это состояние аналогично `TASK_INTERRUPTIBLE`, за исключением того, что процесс *нельзя* разморозить и перевести в исполняемое состояние с помощью специального сигнала. Данное состояние используется в случае, когда процесс, находящийся в состоянии ожидания, нельзя пре-

[5] Видимо, разработчики ядра для платформы x86 не знают, что начиная с процессора 80286 компания Intel на аппаратном уровне реализовала довольно мощные средства управления задачами, которые были расширены в процессоре 80386 с учетом появившегося 32-разрядного режима и в поздних моделях Pentium 4 для поддержки 64-разрядного режима работы. При этом предусмотрен специальный регистр `TR` (task register), в котором хранится дескриптор контекста текущей выполняемой задачи, а переключение задач можно выполнить всего одной ассемблерной командой `LTR`. Однако из года в год эти средства упорно игнорируются разработчиками из-за их сложности и сравнительно большого времени выполнения этой команды (порядка 250–270 машинных тактов). — *Примеч. ред.*

рывать или когда ожидается, что некоторое событие может возникать достаточно часто. Так как задача, находящаяся в этом состоянии, не реагирует на посланные ей сигналы, TASK_UNINTERRUPTIBLE используется гораздо реже, чем TASK_INTERRUPTIBLE[6].

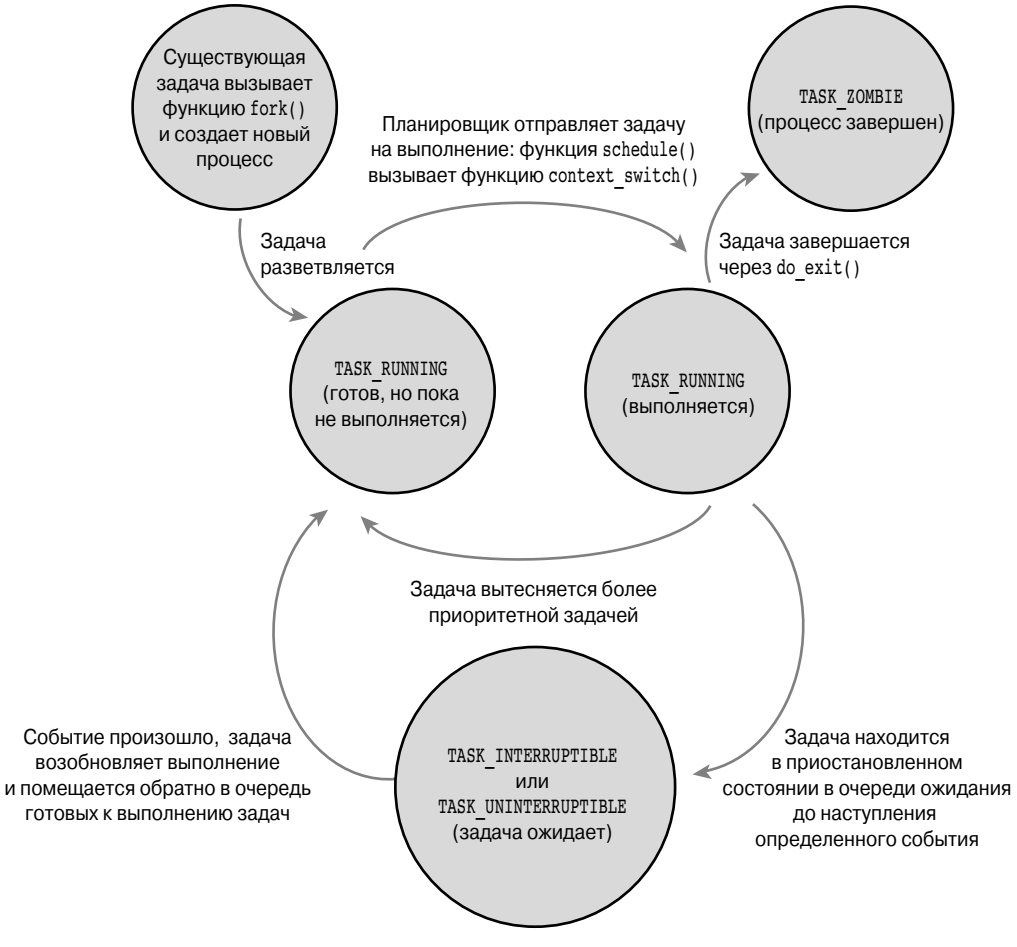


Рис. 3.3. Диаграмма состояний процесса

- **__TASK_TRACED**. Выполняется *трассировка* процесса другим процессом, например отладчиком с помощью системной функции ptrace().
- **__TASK_STOPPED**. Выполнение процесса остановлено. Задача не выполняется и не имеет права выполняться. Такое может случиться, если задача получает

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

какой-либо из сигналов (SIGSTOP, SIGTSTP, SIGTTIN или SIGTTOU), а также если любой из сигналов приходит в тот момент, когда процесс находится в состоянии отладки.

Изменение текущего состояния процесса

В коде ядра часто необходимо изменять состояние процесса. Предпочтительнее всего для этой цели использовать функцию

```
set_task_state(task, state); /* Перевести задачу 'task' в состояние
'state' */
```

которая устанавливает указанное состояние для указанной задачи. По мере необходимости эта функция также предоставляет *барьер памяти* (*memory barrier*), чтобы гарантированно изменить состояние задачи на других процессорах (необходимо только при использовании SMP-систем). В остальных случаях это эквивалентно выражению

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

Существует и другая функция, `set_current_state(state)`, вызов которой эквивалентен `set_task_state(current, state)`. Чтобы ознакомиться с подробностями реализации этой и других связанных с ней функций, обратитесь к файлу `<linux/sched.h>`.

Контекст процесса

Одной из самых важных частей процесса является исполняемый программный код, который считывается из *исполняемого файла* (*executable*) и выполняется в адресном пространстве процесса. Обычно выполнение программы осуществляется в *пространстве пользователя*. Когда в программе выполняется вызов системной функции (см. главу 5, "Системные функции") или возникает исключительная ситуация, программа входит в *пространство ядра*. С этого момента говорят, что ядро "выполняется от имени процесса" и все это происходит в *контексте процесса*. В контексте процесса макрос `current` работает корректно[7]. 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 управление передается планировщику, который и выбирает для выполнения наиболее приоритетный процесс.

Системные функции и обработчики исключительных ситуаций имеют в ядре строго определенный интерфейс. 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 Unix существует четкая иерархия процессов, и Linux здесь не исключение. Все процессы являются потомками процесса `init`, значение идентификатора `PID` для которого равно `1`. Ядро запускает процесс `init` на последнем шаге процедуры начальной загрузки системы. Процесс `init`, в свою очередь, читает системные

[7] Кроме контекста процесса, существует еще контекст прерывания, описанный в главе 7, "Прерывания и их обработка". В контексте прерывания система не работает от имени процесса, поскольку при этом выполняется код обработчика прерывания. С обработчиком прерывания не связан ни один процесс, поэтому и контекст процесса отсутствует.

файлы сценариев начальной загрузки (initscripts) и запускает другие программы, в результате чего процедура начальной загрузки завершается.

Каждый процесс в системе имеет только одного родителя. Кроме того, каждый процесс может иметь одного или нескольких потомков (порожденных процессов) либо не иметь их вовсе. Процессы, которые порождены одним и тем же родительским процессом, называются *родственными* (*siblings*). Информация о взаимосвязи процессов хранится в дескрипторе процесса. В структуре `task_struct` каждого процесса есть указатель на структуру `task_struct` родительского процесса, который называется `parent`, а также указатель на список порожденных процессов, который называется `children`. Следовательно, если известен текущий процесс (`current`), то для него можно определить дескриптор родительского процесса с помощью приведенного ниже выражения.

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

Аналогично можно пройтись в цикле по всем дочерним процессам, порожденным от текущего процесса, с помощью приведенного ниже кода.

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

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

Дескриптор процесса `init` — это структура данных с именем `init_task`, выделяемая статически. В приведенном ниже коде, который всегда выполняется успешно, показан хороший пример использования связей между всеми процессами.

```
for (task = current; task != &init_task; task = task->parent)
        ;
/* переменная task теперь указывает на процесс init */
```

По сути, проходя по иерархии процессов, можно перейти от одного любого процесса системы к *любому* другому. Иногда, однако, желательно выполнить цикл по *всем* процессам системы. Это сделать совсем несложно, поскольку список задач представляет собой циклический двухсвязный список. Имея корректный указатель на дескриптор любого процесса, с помощью приведенного ниже кода можно получить указатель на следующую задачу в списке.

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

По аналогии указатель на предыдущую задачу в списке можно получить так:

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

Два приведенных выше действия можно выполнить с помощью макросов `next_task(task)` и `prev_task(task)` соответственно. Наконец, чтобы перебрать все задачи в списке, предусмотрен макрос `for_each_process(task)`. На каждом шаге цикла переменная `task` указывает на следующую задачу из списка, как показано ниже.

```
struct task_struct *task;

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

Внимание!

Организация цикла по всем задачам системы, в которой выполняется много процессов, может привести к сильной потере производительности системы. 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 используется *механизм порождения процессов* (spawn mechanism), который создает для процесса новое адресное пространство, загружает в него исполняющийся код и передает ему управление. В системе Unix применяется довольно необычный подход, при котором указанные выше операции разделены на две самостоятельные функции: `fork()` и `exec()`.[8] Первая функция — `fork()` — создает порожденный процесс, который является копией текущей задачи. Он отличается от родительского процесса только значением идентификатора `PID` (который должен быть уникальным в системе), значением параметра `PPID` (идентификатор `PID` родительского процесса, для которого устанавливается значение `PID` первоначального процесса), некоторыми ресурсами, такими как сигналы, ожидающие обработки (которые не наследуются), а также статистикой использования ресурсов. Вторая функция — `exec()` — загружает исполняемый файл в адресное пространство процесса и передает ему управление. Комбинация функций `fork()` и `exec()` аналогична той одной функции создания процесса, которая реализована в большинстве других операционных систем.

Копирование при записи

Традиционно при выполнении функции `fork()` делался дубликат всех ресурсов родительского процесса и передавался порожденному. Такой подход достаточно простой и неэффективный, поскольку при этом копируется большое количество данных, которые по идее должны быть общими. Хуже того, если новый процесс сразу же запустит другой исполняемый файл, все скопированные для него ресурсы просто-напросто будут потеряны. Поэтому в операционной системе Linux в системной функции `fork()` используется так называемый механизм *копирования* страниц памяти *при их записи* (copy-on-write, COW). Он позволяет отложить или вообще предотвратить копирование одинаковых страниц данных. Чтобы не создавать дубликат адресного пространства процесса, родительский и дочерний процессы могут совместно использовать одну и ту же копию адресного пространства.

Однако при этом данные помечаются особым образом, и если вдруг один из процессов попытается изменить какие-либо данные, то для него создается их дубликат. В результате каждый из процессов получит свою 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). Описанная методика позволяет отложить копирование каждой страницы памяти до того момента, пока в нее не будет осуществляться запись. В случае, если в страницы памяти

[8] Под `exec()` будем понимать любую функцию из семейства `exec*()`. В ядре реализована системная функция `execve()`, на основе которой реализованы библиотечные функции `execlp()`, `execle()`,`execv()` и `execvp()`.

никогда не делается запись, как, например, при вызове функции exec() сразу после вызова fork(), то эти страницы никогда не будут копироваться.

Единственные издержки, которые возникают при вызове функции fork(), — это дублирование таблиц страниц родительского процесса и создание уникального дескриптора дочернего процесса. В часто встречаемом случае, когда сразу после вызова функции fork() в дочернем процессе запускается новый исполняемый файл, такая оптимизация позволяет предотвратить потерю значительного количества памяти, которая неизбежно возникает при копировании данных (размер адресного пространства легко может достигать нескольких десятков мегабайт). Эта оптимизация очень важна, потому что идеология операционной системы Unix предусматривает быстрое выполнение процессов.

Функция fork()

В операционной системе Linux функция fork() реализована через вызов системной функции clone(). Ей передается в качестве аргументов набор флагов, определяющих, какие ресурсы должны быть общими (если нужно) у родительского и дочернего процессов. Далее, в разделе "Реализация потоков в ядре Linux", об этих флагах рассказано более подробно. Во всех библиотечных функциях fork(),vfork() и __clone() вызывается системная функция clone(), которой передается набор соответствующих флагов. В самой же функции clone() вызывается функция 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_process() и запускает новый процесс на выполнение. Ниже описаны наиболее интересные моменты, выполняемые функцией copy_process().

1. Вызывается функция dup_task_struct(), которая создает новый стек ядра, структуры thread_info и task_struct для нового процесса, причем все значения указанных структур данных идентичны для родительского и дочернего процессов. На этом этапе дескрипторы родительского и дочернего процессов идентичны.
2. Проверяется, не будет ли при создании нового процесса исчерпан лимит на количество процессов для данного пользователя.
3. Теперь необходимо сделать так, чтобы дочерний процесс отличался от родительского. При этом различные поля дескриптора дочернего процесса очищаются или устанавливаются в начальные значения. Не копируются только те поля, в которых содержатся в основном статистические данные использования системных ресурсов. Бо́льшая часть полей структуры task_struct не меняется.
4. Далее дочерний процесс переводится в состояние TASK_UNINTERRUPTIBLE, чтобы он случайно не начал выполняться.
5. 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.
6. Вызывается функция alloc_pid(), которая назначает новое значение идентификатора PID для новой задачи.

7. В зависимости от значения флагов, переданных в функцию `clone()`, в функции `copy_process()` выполняется 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 становятся уникальными и будут копироваться на этом этапе.
8. Наконец, в функции `copy_process()` выполняется окончательная зачистка структур данных и возвращается указатель на новый дочерний процесс.

Далее происходит возврат в функцию `do_fork()`. Если возврат из функции `copy_process()` произошел успешно, то новый дочерний процесс размораживается и запускается. Ядро преднамеренно запускает дочерний процесс на выполнение раньше родительского[9]. В обычной ситуации, когда в дочернем процессе сразу же вызывается функция `exec()`, это позволяет избежать издержек, связанных с использованием механизма копирования при записи. Дело в том, что эти издержки непременно возникают в случае, если родительский процесс начинает выполняться первым и что-то записывать в свое адресное пространство.

Функция `vfork()`

Вызов системной функции `vfork()` позволяет получить тот же эффект, что и вызов функции `fork()`, за исключением того, что записи таблиц страниц родительского процесса не копируются. Вместо этого дочерний процесс запускается в виде отдельного потока в адресном пространстве родительского процесса. При этом родительский процесс замораживается до того момента, пока дочерний процесс не вызовет функцию `exec()` или не завершится. Дочернему процессу *запрещена* запись в адресное пространство. Такая оптимизация была весьма желанной в старые времена 3BSD, когда была введена функция `vfork()`. Дело в том, что в то время при реализации функции `fork()` не использовалась технология копирования при записи. Поскольку на современном этапе эта технология уже реализована и дочерний процесс запускается первым, единственное преимущество функции `vfork()` заключается в том, что она не копирует записи таблиц страниц родительского процесса. Если когда-нибудь в операционной системе Linux будет реализована возможность копирования элементов таблиц страниц при записи[10], то у функции `vfork()` вообще не останется никаких преимуществ. Поскольку семантика функции `vfork()` достаточно запутанна (что, например, произойдет, если вызов функции `exec()` завершится неудачно?), в идеале функция `vfork()` не нужна и в ядре она не должна быть реализована. Вполне возможно реализовать системную функцию `vfork()` через обычную функцию `fork()`, что и было на самом деле сделано в ядрах Linux до версии 2.2.

В настоящее время системная функция `vfork()` реализована через специальный флаг, который указывается при вызове системной функции `clone()`, как показано ниже.

1. При выполнении функции `copy_process()` поле `vfork_done` структуры `task_struct` устанавливается в значение `NULL`.

---

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

[10] Уже сейчас есть заплаты, позволяющие добавить эту возможность в ОС Linux. Со временем, скорее всего, эта возможность будет включена в основную ветку разработки ядра.

2. При выполнении функции `do_fork()`, если соответствующий флаг установлен, поле `vfork_done` устанавливается в ненулевое значение (начинает указывать на определенный адрес).
3. После того как дочерний процесс начнет выполняться первым, родительский процесс не возвращается в основную программу, а ждет специального сигнала от дочернего процесса, который тот посылает через указатель `vfork_done`.
4. В функции `mm_release()`, которая вызывается, как только задача покидает свое адресное пространство, проверяется значение поля `vfork_done`. Если оно не равно `NULL`, родительский процесс получает указанный выше сигнал.
5. При возврате в функцию `do_fork()` родительский процесс размораживается и возвращается к выполнению основной программы.

Если все прошло так, как планировалось, то дочерний процесс начинает выполняться в новом адресном пространстве, а родительский процесс продолжает работать в своем старом адресном пространстве. Хотя попутные издержки уменьшаются, все же реализация такого механизма не очень привлекательна.

Реализация потоков в ядре Linux

В современном программировании потоки (threads) являются довольно популярной программной абстракцией. Благодаря этому появляется возможность выполнения нескольких потоков команд в общем адресном пространстве памяти. Потоки также могут совместно использовать открытые файлы и другие ресурсы. Потоки позволяют реализовать режим *одновременного выполнения программ* (*concurrent programming*) и обеспечить истинный *параллелизм* на многопроцессорных системах.

Реализация потоков в операционной системе Linux уникальна. С точки зрения ядра Linux не существует отдельной *концепции* потоков. В ядре Linux все потоки реализованы в виде стандартных процессов. В нем 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 свою структуру `task_struct` и с точки зрения ядра является обычным процессом, который (так уж случилось!) совместно использует с другими процессами общие ресурсы, такие как адресное пространство.

Этот подход в корне отличается от того, что принято в других операционных системах, таких как Microsoft Windows или Sun Solaris. В них существуют средства для *явной* поддержки потоков в ядре, которые иногда называются *процессами с быстрым переключением контекста* (lightweight process). Само название — процесс с быстрым переключением контекста — свидетельствует о разнице в философии, принятой в Linux и других операционных системах. В этих системах потоки — это абстракция, которая обеспечивает облегченные и более быстрые с точки зрения выполнения блоки кода, по сравнению с обычными тяжеловесными процессами. Для операционной системы Linux потоки — это просто способ совместного использования ресурсов несколькими процессами, которые и так имеют достаточно малое время переключения контекста[11].

[11] В качестве примера можно привести результаты тестирования времени создания процессов (и даже потоков!) в операционной системе Linux и в других ОС. В Linux все происходит быстрее.

В качестве примера предположим, что у нас есть процесс, состоящий из четырех потоков. В операционных системах с явной поддержкой потоков должен существовать дескриптор процесса, в котором хранятся ссылки на дескрипторы четырех различных потоков. В дескрипторе процесса описываются совместно используемые ресурсы, такие как адресное пространство и открытые файлы. В дескрипторах потока описываются ресурсы, которые принадлежат только этим потокам. В ОС Linux, наоборот, существуют просто четыре процесса и, соответственно, четыре обычные структуры `task_struct`. Параметры этих четырех процессов установлены так, чтобы они могли совместно использовать определенные ресурсы. В результате получается красивое и элегантное решение проблемы.

Создание потоков

Потоки создаются так же, как и обычные задачи, за исключением того, что системной функции `clone()` передаются флаги, указывающие на то, какие ресурсы должны использоваться совместно:

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

Результат выполнения этого кода будет таким же, как и при выполнении обычной функции `fork()`, за исключением того, что адресное пространство, ресурсы файловой системы, дескрипторы файлов и обработчики сигналов останутся общими. Другими словами, и новая задача, и родительский процесс являются тем, что в народе популярно называется *потоками*.

Для сравнения вызов обычный функции `fork()` может быть реализован так:

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

Функцию `vfork()` можно реализовать следующим образом:

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

Флаги, которые передаются системной функции `clone()`, указывают особенности поведения нового процесса и конкретизируют, какие именно ресурсы должны быть общими для родительского и дочернего процессов. В табл. 3.1 приведены флаги системной функции `clone()`, описанные в файле `<linux/sched.h>`, и вызываемый ими эффект.

Таблица 3.1. Флаги системной функции `clone()`

Флаг	Описание
`CLONE_FILES`	Родительский и дочерний процессы совместно используют открытые файлы
`CLONE_FS`	Родительский и дочерний процессы совместно используют информацию о файловой системе
`CLONE_IDLETASK`	Установить значение PID в нуль (используется только для бездействующих (idle) задач)
`CLONE_NEWNS`	Создать новое пространство имен для дочерней задачи
`CLONE_PARENT`	Родительский процесс для вызывающего процесса становится родительским и для дочернего
`CLONE_PTRACE`	Продолжить трассировку для дочернего процесса
`CLONE_SETTID`	Возвратить значение идентификатора TID в пространство пользователя

Окончание табл. 3.1

Флаг	Описание
CLONE_SETTLS	Для дочернего процесса создать новую область локальных данных потока (thread local storage, TLS)
CLONE_SIGHAND	У родительского и дочернего процессов будут общие обработчики сигналов и сигналы блокирования
CLONE_SYSVSEM	У родительского и дочернего процессов будет общая семантика обработки флага SEM_UNDO для семафоров System V
CLONE_THREAD	Родительский и дочерний процессы будут принадлежать одной группе потоков
CLONE_VFORK	Использовать vfork(): родительский процесс будет находиться в замороженном состоянии, пока дочерний процесс не возобновит его работу
CLONE_UNTRACED	Запретить трассирующему процессу использование флага CLONE_PTRACE для дочернего процесса
CLONE_STOP	Запустить процесс и перевести в состояние TASK_STOPPED
CLONE_CHILD_CLEARTID	Очистить идентификатор TID для дочернего процесса
CLONE_CHILD_SETTID	Установить идентификатор TID для дочернего процесса
CLONE_PARENT_SETTID	Установить идентификатор TID для родительского процесса
CLONE_VM	У родительского и дочернего процессов будет общее адресное пространство

Потоки в ядре

Часто требуется выполнить в ядре некоторые операции в фоновом режиме. В ядре такая возможность реализована в виде *потоков ядра* (kernel thread) — обычных процессов, которые выполняются исключительно в пространстве ядра. Существенным отличием между потоками ядра и обычными процессами является то, что потоки в ядре не имеют своего 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 ядра планируются и вытесняются так же, как и обычные процессы.

Потоки ядра в Linux выполняют определенные задачи. Самые важные из них — flush и ksoftirqd. Чтобы увидеть список таких потоков в вашей системе Linux, воспользуйтесь командой ps -ef. Их довольно много! Потоки ядра создаются во время начальной загрузки системы другими такими же потоками. Это и понятно, поскольку поток ядра может быть создан только другим потоком ядра. В ядре эти потоки обрабатываются автоматически: все новые потоки ядра разветвляются от специального процесса kthreadd, запущенного в ядре. Интерфейс для порождения нового потока ядра из существующего описан в файле <linux/kthread.h> и приведен ниже.

```
struct task_struct *kthread_create(int (*threadfn)(void *data),
                                   void *data,
                                   const char namefmt[],
                                  .)
```

Новая задача создается процессом kthread, запущенным в ядре, с помощью вызова системной функции clone(). В новом процессе будет выполняться функция threadfn(),

которой передается аргумент data. Процессу будет присвоено имя, указанное в переменной namefmt, а в списке аргументов переменной длины указываются параметры форматирования в стиле функции printf() языка C. После создания процесс находится в незапущенном состоянии (unrunnable state). Для его запуска нужно явным образом вызвать функцию wake_up_process(). Чтобы создать процесс и сразу же запустить его на выполнение, можно воспользоваться единственной функцией kthread_run(), как показано ниже.

```
struct task_struct *kthread_run(int (*threadfn)(void *data),
                                void *data,
                                const char namefmt[],
                                ...)
```

Описанная выше процедура запуска процесса в ядре реализована в виде макроса, в котором вызываются обе функции, kthread_create() и wake_up_process(), как показано ниже.

```
#define kthread_run(threadfn, data, namefmt, ...)                          \
({                                                                         \
    struct task_struct *k;                                                 \
                                                                           \
    k = kthread_create(threadfn, data, namefmt, ## __VA_ARGS__); \
    if (!IS_ERR(k))                                                        \
        wake_up_process(k);                                                \
    k;                                                                     \
})
```

После запуска поток ядра продолжает выполняться до тех пор, пока в нем не будет вызвана функция do_exit() либо пока в какой-либо другой части ядра не будет вызвана функция kthread_stop(). В качестве параметра этой функции передается адрес структуры типа task_struct, возвращенной ранее функцией kthread_create(), как показано ниже.

```
int kthread_stop(struct task_struct *k)
```

Конкретные примеры потоков ядра будут рассмотрены более детально в следующих главах.

Завершение процесса

Как это ни грустно, но любой процесс в конечном итоге должен завершиться. После завершения процесса ядро должно освободить ресурсы, занятые процессом, и оповестить родительский процесс о том, что порожденный им процесс завершил свою работу.

Как правило, уничтожение процесса инициируется самим процессом. Это происходит, когда в самом процессе вызывается системная функция exit(). Причем это может произойти как явно, когда вся работа программы сделана и нужно завершить ее работу, так и неявно, при выполнении возврата из основной процедуры любой программы с именем main(). Другими словами, компилятор языка C помещает вызов функции exit() в код, который выполняется после возврата из процедуры main(). Процесс также может быть завершен непреднамеренно. Так происходит, когда процесс получает сигнал или возникает исключительная ситуация, которую тот не может обработать или проигнорировать. Независимо от того, каким образом завершается процесс, основную массу работы выполняет функция do_exit(), определенная в файле kernel/exit.c. При этом выполняются перечисленные ниже действия.

1. В поле `flags` структуры `task_struct` устанавливается флаг `PF_EXITING`.
2. Вызывается функция `del_timer_sync()`, которая удаляет все таймеры ядра. После выхода из этой функции гарантируется, что в очереди нет никаких ожидающих обработки событий от таймера и никакой обработчик таймера не выполняется.
3. Если включена возможность учета системных ресурсов, занятых процессами (BSD process accounting), то из функции `do_exit()` вызывается функция `acct_update_integrals()` для записи информации об учете ресурсов, которые использовались процессом.
4. Вызывается функция `exit_mm()`, которая освобождает структуру `mm_struct`, описывающую адресное пространство, занятое процессом. Если никакой другой процесс больше не использует это адресное пространство (другими словами, оно больше не является общим), то ядро его уничтожает совсем.
5. Вызывается функция `exit_sem()`. Если процесс находится в очереди ожидания освобождения семафора подсистемы IPC, то в этой функции процесс удаляется из этой очереди.
6. После этого вызываются функции `exit_files()` и `exit_fs()`, которые уменьшают на единицу значение счетчика использования объектов, соответствующих дескрипторам файлов, и данных файловой системы соответственно. Если счетчик ссылок какого-либо объекта достигает значения, равного нулю, то соответствующий объект больше не используется никаким процессом и удаляется.
7. Устанавливается код завершения задачи, который хранится в поле `exit_code` структуры `task_struct`. Значение этого кода передается в виде аргумента функции `exit()` или устанавливается тем механизмом ядра, который вызвал аварийное завершение процесса. Код завершения сохраняется в поле `exit_code` структуры `task_struct` на тот случай, если он понадобится для анализа родительскому процессу.
8. Вызывается функция `exit_notify()`, которая отправляет сигналы родительскому процессу завершающейся задачи и назначает новый родительский процесс (reparent) для всех порожденных этой задачей процессов. Новым родительским процессом становится один из потоков, принадлежащих группе потоков завершающейся задачи, или процесс `init`. Состояние завершающейся задачи, хранящееся в поле `exit_state` структуры `task_struct`, устанавливается в значение `EXIT_ZOMBIE`.
9. Из функции `do_exit()` вызывается функция `schedule()`, которая переключается на новый процесс (подробности описаны в главе 4, “Системный планировщик и диспетчеризация процессов”). Поскольку завершившийся процесс теперь не является планируемым, код этой функции — последнее, что выполняется в рамках текущей задачи. Из функции `do_exit()` возврата нет.

К этому моменту освобождены все объекты, занятые задачей (если они используются только этой задачей). Задача больше не может выполняться, поскольку у нее уже нет адресного пространства, и, кроме того, она находится в состоянии `EXIT_ZOMBIE`. Единственные области памяти, которые теперь занимает задача, — это системный стек и структуры `thread_info` и `task_struct`. Благодаря этим структурам родительский процесс

может проанализировать нужную ему информацию. После того как он запросит эту информацию или уведомит ядро о том, что она ему больше не нужна, оставшиеся области памяти, закрепленные за завершенным процессом, освобождаются и возвращаются операционной системе для дальнейшего использования.

Удаление дескриптора процесса

После окончания работы функции `do_exit()` дескриптор завершившегося процесса все еще существует в системе, но сам процесс находится в состоянии зомби и не может выполняться. Как уже говорилось выше, это позволяет системе получить информацию о дочернем процессе после его завершения. Следовательно, освобождение ресурсов после завершения процесса и удаление его дескриптора должны происходить в разные моменты времени. После того как родительский процесс получил информацию о завершенном дочернем процессе либо уведомил ядро, что эта информация ему больше не нужна, структура `task_struct` дочернего процесса освобождается.

Семейство функций `wait()` реализовано через единственную (и достаточно сложную) системную функцию `wait4()`. Она выполняет стандартное действие — приостанавливает выполнение вызывающей задачи до завершения одного из ее дочерних процессов. При этом функция возвращает идентификатор `PID` завершенного дочернего процесса. Кроме того, при вызове функции `wait4()` передается указатель на область памяти, которая после возврата из функции будет содержать код завершения дочернего процесса.

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

1. Вызывается функция `__exit_signal()`, в которой затем вызывается функция `__unhash_process()`. Из последней функции вызывается еще одна функция — `detach_pid()`, — которая удаляет процесс из хеш-таблицы идентификаторов процессов `pidhash` и сам процесс из списка задач.
2. В функции `__exit_signal()` освобождаются абсолютно все ресурсы, которые продолжают использоваться в завершившемся процессе, а также подбиваются статистические данные использования системных ресурсов.
3. Если завершившаяся задача была последним членом группы потоков, а ее лидер находится в состоянии зомби, то функция `release_task()` уведомляет родительский процесс лидера группы, находящегося в состоянии зомби.
4. Из функции `release_task()` вызывается функция `put_task_struct()`, в которой освобождаются страницы памяти, содержащие системный стек процесса и структуру `thread_info`, а также освобождается блочная кеш-память, содержащая структуру `task_struct`.

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

Дилемма “беспризорного” процесса

Если родительский процесс завершится до завершения всех его дочерних процессов, то должен существовать какой-нибудь механизм *переназначения* нового родительского процесса всем оставшимся дочерним процессам. Иначе “осиротевшие” процессы навсегда останутся в состоянии зомби и будут зря занимать системную память. Решение этой

проблемы было указано выше: новым родительским процессом становится один из потоков группы завершившегося родительского процесса, или процесс `init`. Из функции `do_exit()` вызывается функция `exit_notify()`, которая в свою очередь вызывает функцию `forget_original_parent()`. Из последней функции вызывается еще одна функция `find_new_reaper()`, в которой и выполняется переназначение родительского процесса, как показано ниже.

```
static struct task_struct *find_new_reaper(struct task_struct *father)
{
   struct pid_namespace *pid_ns = task_active_pid_ns(father);
   struct task_struct *thread;

   thread = father;
   while_each_thread(father, thread) {
      if (thread->flags & PF_EXITING)
         continue;

      if (unlikely(pid_ns->child_reaper == father))
         pid_ns->child_reaper = thread;

       return thread;
   }

   if (unlikely(pid_ns->child_reaper == father)) {
      write_unlock_irq(&tasklist_lock);

      if (unlikely(pid_ns == &init_pid_ns))
         panic("Attempted to kill init!");

      zap_pid_ns_processes(pid_ns);
      write_lock_irq(&tasklist_lock);
      /*
       * Мы не можем очистить ->child_reaper или оставить его как есть.
       * Здесь могут быть скрытые задачи EXIT_DEAD в ->children, поэтому
       * функция forget_original_parent() должна их куда-то переместить.
       */
       pid_ns->child_reaper = init_pid_ns.child_reaper;
   }
   return pid_ns->child_reaper;
}
```

В этом коде выполняется попытка поиска другой задачи, входящей в ту же группу потоков процесса. Если таковая найдена, указатель на ее структуру `task_struct` возвращается в вызывающую программу. Если найденная задача не входит в группу потоков процесса, выполняется поиск структуры `task_struct` процесса `init` и возврат ее указателя в вызывающую программу. После того как для дочерней задачи будет найден подходящий родительский процесс, необходимо изменить ссылку на него во всех дочерних задачах, как показано ниже.

```
reaper = find_new_reaper(father);
list_for_each_entry_safe(p, n, &father->children, sibling) {
   p->real_parent = reaper;
   if (p->parent == father) {
      BUG_ON(p->ptrace);
      p->parent = p->real_parent;
   }

   reparent_thread(p, father);
}
```

После этого вызывается функция `ptrace_exit_finish()`, которая переназначает родителей некоторым дочерним задачам, находящимся в списке *трассируемых процессов* (ptraced), как показано ниже

```
void exit_ptrace(struct task_struct *tracer)
{

   struct task_struct *p, *n;
   LIST_HEAD(ptrace_dead);

   write_lock_irq(&tasklist_lock);
   list_for_each_entry_safe(p, n, &tracer->ptraced, ptrace_entry) {
      if (__ptrace_detach(tracer, p))
         list_add(&p->ptrace_entry, &ptrace_dead);
   }

   write_unlock_irq(&tasklist_lock);

   BUG_ON(!list_empty(&tracer->ptraced));

   list_for_each_entry_safe(p, n, &ptrace_dead, ptrace_entry) {
      list_del_init(&p->ptrace_entry);
      release_task(p);
   }
}
```

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 используются именно два списка, достаточно интересна (эта новая особенность появилась в ядрах серии 2.6). Когда задача находится в состоянии трассировки, для нее временно назначается родительским тот процесс, который осуществляет отладку (debugging). Когда завершается истинный родительский процесс, ссылку на него нужно переназначить для *всех* потомков, включая трассируемую задачу. В ядрах более ранних версий для поиска дочерних процессов нужно было организовать цикл *по всем процессам системы*. Для решения этой проблемы и был создан дополнительный список дочерних процессов, находящихся в состоянии трассировки. В результате поиск был сужен до двух сравнительно небольших списков.

После успешного переназначения родительского процесса больше нет риска, что какой-либо процесс навсегда останется в состоянии зомби. Процесс `init` периодически вызывает функцию `wait()` для всех своих дочерних процессов и соответственно удаляет все зомби-процессы, назначенные ему.

Резюме

В этой главе была рассмотрена очень важная абстракция операционной системы, которая называется *процессом*, описаны общие свойства процессов и их назначение, а также представлено сравнение процессов и потоков. Кроме того, было описано, как операционная система Linux хранит и представляет информацию, которая относится к процессам (структуры `task_struct` и `thread_info`), как создаются процессы (через вызовы системных функций `fork()` и в конечном итоге `clone()`), каким образом новые исполняемые образы загружаются в адресное пространство (через семейство системных функций `exec()`). Мы рассмотрели иерархию процессов, описали, как родительский

процесс получает информацию о своих завершившихся дочерних процессах (семейство системных функций `wait()`) и как в конце концов процесс завершается (непреднамеренно или с помощью вызова системной функции `exit()`). Процесс — это основополагающая и ключевая абстракция, которая используется во всех современных операционных системах и в конечном итоге является тем, ради чего вообще создаются операционные системы (т.е. для запуска пользовательских программ).

Следующая глава посвящена планированию процессов — чрезвычайно интересной и деликатной теме. Вы узнаете, как планировщик ядра решает, какой именно процесс нужно запустить, в какой момент времени и в каком порядке.

4

Системный планировщик и диспетчеризация процессов

В предыдущей главе были рассмотрены процессы — важная абстракция операционной системы, связанная с выполняющимся программным кодом. В этой главе речь пойдет о планировщике процессов — подсистеме ядра, благодаря которой процессы могут выполняться.

Системный планировщик (process scheduler) — это компонент ядра, определяющий, какой из процессов должен выполняться, в какой именно момент и насколько долго. Системный планировщик (или для краткости просто *планировщик*) распределяет конечные ресурсы центрального процессора между всеми запушенными в системе процессами. Планировщик — основной компонент *мультипрограммных*, или, как их сейчас называют, *многозадачных* (multitasking) операционных систем, таких как ОС Linux. Поскольку именно планировщик решает, какой процесс должен быть запущен следующим, от его работы зависит, насколько хорошо используются системные ресурсы. В результате у пользователя создается ощущение, будто несколько процессов выполняется одновременно.

Идея работы планировщика чрезвычайно проста. Чтобы использовать процессорное время наилучшим образом, один из *готовых к выполнению* процессов (runnable) должен всегда выполняться. Если количество готовых к выполнению процессов в системе превышает количество процессоров компьютера, то очевидно, что некоторые из этих процессов не будут выполняться в данный конкретный момент времени. Такие процессы должны *ожидать момента своего выполнения* (waiting to run). Поэтому планировщик должен принять одно из основных решений: какой из процессов, готовых к выполнению, должен быть запущен следующим.

Мультипрограммный режим работы

Мультипрограммной (multitasking) называется такая операционная система, в которой по очереди может одновременно выполняться несколько

процессов. На компьютерах, оснащенных одним процессором, такой режим работы ОС позволяет создать для пользователя иллюзию того, что несколько процессов выполняется параллельно. На многопроцессорных компьютерах благодаря мультипрограммному режиму работы ОС процессы действительно выполняются одновременно и параллельно на разных процессорах. Кроме того, такая ОС позволяет на любом из типов компьютеров *заблокировать* или *заморозить* несколько процессов, которые не должны выполняться до наступления определенного события. Несмотря на то что эти процессы находятся в оперативной памяти компьютера, они *не являются готовыми к выполнению*. Вместо этого они вызывают специальную функцию ядра, которая переводит их в состояние ожидания до наступления определенного события (поступления данных от клавиатуры или сетевой карты, исчерпания заданного интервала времени и т.п.). Таким образом, в современной системе Linux в памяти может находиться множество процессов, но, скажем, только один из них может быть готовым к выполнению.

Мультипрограммные ОС бывают двух типов — с *кооперативной* (cooperative) и *приоритетной* (preemptive), или *вытесняемой*, многозадачностью. В Linux, как и во всех вариантах ОС Unix, а также в большинстве современных операционных систем реализован мультипрограммный режим работы с приоритетной многозадачностью. В таком режиме системный планировщик самостоятельно принимает решение о том, в какой момент времени следует прекратить выполнение текущего процесса и переключиться на выполнение другого процесса. Сам факт временного и принудительного прекращения выполнения процесса называется *приоритетным прерыванием обслуживания* (preemption), или просто *вытеснением*. Период времени, в течение которого процесс выполняется до момента своего вытеснения, как правило, установлен заранее и называется *квантом* (timeslice) процессорного времени. По сути, каждый готовый к выполнению процесс запускается на реальном процессоре ограниченное время, соответствующее кванту. Изменяя значения квантов времени для разных процессов, планировщик глобальным образом влияет на работу системы в целом. При этом, кроме всего прочего, планировщик не допускает монопольное использование центрального процессора одним процессом. Во многих современных операционных системах значение кванта времени вычисляется динамически в зависимости от характера работы процесса и установленных стратегий планирования процессов. Как мы увидим чуть позже, в уникальном и “справедливом” планировщике системы Linux кванты, как таковые, не используются, что приводит к очень интересным побочным эффектам.

В отличие от рассмотренного выше, в мультипрограммных ОС с *кооперативной многозадачностью* выполнение процесса не прерывается принудительно: процесс сам знает, когда нужно прервать свое выполнение. Сам факт добровольного и самостоятельного прерывания выполнения процессом самим себя называется *передачей управления* (yielding). В идеале процессы должны достаточно часто прерывать свое выполнение и позволять другим готовым к выполнению процессам занимать центральный процессор приемлемое количество времени. Однако операционная система управлять этим никак не может. Недостатки такого подхода достаточно очевидны. Планировщик при этом не может принимать глобальных решений по поводу длительности выполнения каждого процесса. В результате процессы могут монополизировать центральный процессор на длительное время, превышающее ожидаемое пользователем время отклика системы. Зависание процесса приводит к тому, что он уже не сможет прерывать свое выполнение, что, в свою очередь, может привести к зависанию всей системы. К счастью, в большинстве операционных систем, разработанных за последние два десятилетия, реализован мульти-

программный режим с приоритетной многозадачностью. Самыми известными и позорными исключениями являются только Mac OS 9 (и более ранние версии), а также Windows 3.1 (и ее более ранние версии). Разумеется, в ОС Unix мультипрограммный режим с приоритетной многозадачностью был реализован с момента ее появления.

Системный планировщик Linux

Начиная с самых первых версий Linux, выпущенных в 1991 году, и вплоть до версий ядра серии 2.4 системный планировщик Linux был простым и в плане дизайна не представлял собой ничего особенного. Логику его работы было достаточно просто понять, но при наличии большого количества запущенных процессов и нескольких центральных процессоров в компьютере он плохо поддавался масштабированию.

Поэтому при разработке серии ядер 2.5 системный планировщик был существенно обновлен. Новый планировщик обычно называют *планировщиком типа O(1)*, поскольку положенный в его основу алгоритм[1] устранил недостатки, которые были присущи предыдущим версиям планировщика Linux, позволил реализовать новые расширенные функциональные возможности и обеспечил более высокие характеристики производительности. После введения нового алгоритма вычисления величины квантов процессора за постоянное время, а также создания отдельных очередей готовых к выполнению задач для каждого процессора были сняты ограничения, заложенные в алгоритм системного планировщика предыдущих версий Linux.

В связи с тем, что современные версии Linux поддерживают довольно мощные компьютерные системы, оснащенные десятками, если не сотнями процессоров, новый планировщик типа O(1) как нельзя лучше подходил для этой цели, поскольку он без проблем масштабировался. Тем не менее со временем стало ясно, что и у него есть ряд существенных недостатков, связанных с планированием приложений, чувствительных к задержкам. Такие приложения называются *интерактивными процессами* (interactive processes). К ним относятся практически все приложения, работающие в диалоговом режиме с пользователем. В результате, несмотря на то, что планировщик типа O(1) идеально подходил для больших серверных платформ, на которых обычно не запускаются никакие интерактивные процессы, качество его работы на компьютерах конечного пользователя, где число интерактивных процессов велико, было ниже среднего. Поэтому, начиная с ранних версий ядра серии 2.6, разработчики стали использовать несколько новых системных планировщиков, призванных улучшить производительность интерактивных процессов планировщика типа O(1). Самый заметный среди них — планировщик типа RSDL (Rotating Staircase Deadline, или циклический ступенчатый граничный планировщик), в котором была впервые реализована концепция *справедливого планирования* (fair scheduling), позаимствованная из теории очередей. Эта концепция привела к тому, что в версии ядра 2.6.23 планировщик типа O(1) был окончательно заменен планировщиком типа *CFS* (Completely Fair Scheduler, или полностью справедливый планировщик).

В этой главе будут рассмотрены основы проектирования планировщиков на примере планировщика CFS, а также цели его создания, принципы проектирования, практическая реализация, алгоритмы работы и соответствующие системные вызовы. Кроме того, мы

[1] *O(1)* — это пример обозначения “большого О”, характеризующее быстродействие алгоритма. Практически эта запись означает, что планировщик может выполнить все свои действия за постоянное время, независимо от объема входных данных. Полное объяснение того, что такое обозначение “большого О”, приведено в главе 6, “Структуры данных ядра”.

также рассмотрим планировщик типа O(1), поскольку его реализация представляет собой модель более “классического” системного планировщика Unix.

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, ориентированные на ввод–вывод и на вычисления

Процессы можно грубо разделить на два вида: интенсивно использующие *операции ввода-вывода* (*I/O-bound*) и интенсивно использующие центральный процессор (*processor-bound*). К первому типу относятся процессы, которые бо́льшую часть своего времени выполнения тратят на отправку запросов на ввод-вывод и ожидание ответов на эти запросы. Следовательно, такие процессы могут выполняться только в течение короткого периода времени, поскольку бо́льшую часть своего времени жизни они находятся в состоянии ожидания завершения операций ввода-вывода. Здесь под операциями ввода-вывода мы подразумеваем не только дисковые операции, но и любой другой тип блокируемого ресурса, такой как ввод с клавиатуры или обмен данными по сети. Например, большинство приложений с графическим интерфейсом (GUI) относятся к процессам, ориентированным на ввод-вывод, даже если они никогда не обращаются к диску. Дело в том, что бо́льшую часть своего времени жизни они ожидают ввода команд пользователя с клавиатуры или с помощью мыши.

Процессы, интенсивно использующие процессор, наоборот, бо́льшую часть времени выполняют программный код. Они работают до того момента, пока планировщик не прервет их выполнение и не переключится на другую задачу. Все дело в том, что в таких процессах запросы на выполнение операций ввода-вывода (и последующий переход в состояние ожидания) встречаются сравнительно редко. Поскольку вычислительные процессы не зависят от частоты операций ввода вывода, для обеспечения нормальной скорости реакции системы не требуется, чтобы планировщик часто передавал им управление. Поэтому в стратегии планирования таких процессов предполагается, что вычислительные процессы должны выполняться реже, но более продолжительный период времени. В предельном случае вычислительный процесс представляет собой программный код, в котором выполняется бесконечный цикл. В более реалистичных случаях в таких процессах выполняется программа, в которой используется большое количество математических вычислений. В качестве примера можно привести утилиту `sshkeygen` или пакет MATLAB.

Приведенные выше типы процессов не являются взаимно исключающими. Процессы могут сочетать в себе оба типа поведения. В качестве примера можно привести сервер системы X Window — процесс, который одновременно интенсивно загружает процессор и интенсивно выполняет операции ввода-вывода. В некоторых процессах может интенсивно использоваться ввод-вывод, но в определенные моменты времени — центральный процессор. Хорошим примером может служить текстовый процессор, который обычно ожидает момента нажатия клавиши пользователем, но время от времени может сильно загружать процессор, выполняя проверку орфографии или макрос.

Стратегия планирования операционной системы должна стремиться к удовлетворению двух несовместимых условий: обеспечения высокой скорости реакции процессов (т.е. малого времени отклика системы) и максимального использования системных ресурсов (т.е. высокой производительности). Для удовлетворения этим требованиям в планировщиках часто применяются сложные алгоритмы определения самого 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, интенсивно использующие ввод-вывод, имели больший приоритет. Это приводит к уменьшению времени отклика системы. В операционной системе Linux для обеспечения хорошей скорости реакции интерактивных программ и производительности настольных приложений применяется оптимизация по времени отклика (т.е. обеспечение малого времени задержки). Таким образом, процессы, интенсивно использующие ввод-вывод, имеют более высокий приоритет перед вычислительными процессами. Как будет показано ниже, при этом используется творческий подход, при котором учитываются также и вычислительные процессы.

Приоритет процессов

Самым распространенным типом алгоритмов планирования является планирование *на основе приоритетов* (priority-based). Цель такого планирования состоит в том, чтобы упорядочить процессы в соответствии с их важностью и необходимостью использования процессорного времени. Основная идея, которая не совсем точно реализована в Linux, состоит в том, что процессы с более высоким приоритетом должны выполняться раньше тех, которые имеют более низкий приоритет, в то время как процессы с одинаковым приоритетом планируются на *выполнение по кругу* (round-robin), т.е. периодически один за другим. В некоторых операционных системах процессы с более высоким приоритетом получают также и больший квант времени. Готовый к выполнению процесс, у которого еще не закончился квант времени и который имеет наибольший приоритет, будет выполняться всегда. Как пользователь, так и операционная система может влиять на работу системного планировщика путем изменения значения приоритета процесса.

В ядре Linux используются два разных диапазона приоритетов. Первый из них определяет значение параметра `nice` — число в диапазоне от –20 до +19, стандартное значение которого равно нулю. Большему значению параметра `nice` соответствует меньший приоритет — необходимо быть более "тактичным" к другим процессам системы (от англ. *nice* — тактичный, хороший). Процессам с низким значением параметра `nice` (б*о*льшим приоритетом) выделяется б*о*льшая часть процессорного времени по сравнению с процессами, имеющими высокое значение параметра `nice`. Диапазон значений параметра `nice` является стандартным для всех систем Unix. Однако в различных системах Unix его значения интерпретируются по-разному в зависимости от используемых алгоритмов планирования. В некоторых системах Unix, включая Mac OS X, значение параметра `nice` влияет на *абсолютное* значение кванта времени, выделяемого процессу. В системе Linux оно влияет только на размер выделяемой *доли* (proportion) процессорного времени. Чтобы увидеть список процессов в системе и соответствующие им значения параметра `nice` (в колонке *`NI`*), воспользуйтесь командой `ps -el`.

Второй диапазон соответствует *приоритетам реального времени* (real-time priority). Диапазон этих значений можно изменить, однако по умолчанию он охватывает числа от 0 до 99 включительно. В отличие от параметра `nice` б*о*льшим значениям приоритета реального времени соответствует и больший приоритет. Все процессы реального време-

ни имеют более высокий приоритет по сравнению с обычными процессами. Это означает, что значения приоритета реального времени и параметра `nice` принадлежат различным пространствам значений. В системе Linux приоритеты реального времени реализованы в соответствии с важным стандартом Unix POSIX.1b. Во всех современных Unix-системах они реализованы по аналогичной схеме. Чтобы увидеть список процессов в системе и соответствующие им значения приоритета реального времени (в колонке *RTPRIO*), воспользуйтесь приведенной ниже командой.

```
ps -eo state,uid,pid,ppid,rtprio,time,comm
```

Отметка `"-"` в выводе этой команды соответствует процессам, не принадлежащим реальному времени.

Кванты времени

Квант времени[2] (timeslice) — это числовое значение, определяющее, как долго может выполняться процесс до того момента, пока он не будет вытеснен. В стратегии планирования должно устанавливаться стандартное значение кванта времени, определение которого является непростой задачей. Слишком большое значение кванта времени приведет к ухудшению интерактивной производительности системы. При этом теряется ощущение, что процессы выполняются одновременно. Слишком малое значение кванта времени приведет к возрастанию накладных расходов на переключение между процессами. Дело в том, что при этом система будет тратить больший процент процессорного времени на переключение с одного процесса с малым квантом времени на другой процесс с малым квантом времени. Более того, снова обостряются противоречивые требования к процессам, интенсивно использующим ввод-вывод и центральный процессор. Для процессов, интенсивно использующих ввод-вывод, не требуется выделять большой квант времени, хотя они и должны запускаться достаточно часто. С другой стороны, для процессов, интенсивно использующих центральный процессор, настоятельно рекомендуется задавать продолжительный квант времени для эффективного использования кеш-памяти процессора.

На основе этих аргументов можно сделать вывод: *любое* большое значение кванта времени приведет к ухудшению интерактивной производительности. При реализации большинства операционных систем такой вывод принимается близко к сердцу и стандартное значение кванта времени устанавливается достаточно малым, например 10 мс. Однако в планировщике CFS системы Linux значения квантов времени напрямую не назначаются процессам. Вместо этого используется обновленный подход, при котором процессам выделяется *доля* (proportion) процессорного времени. Поэтому в системах Linux длительность использования центрального процессора определенным процессом зависит от загрузки системы. Кроме того, на величину доли влияет значение параметра `nice`, установленного для процесса. Оно является своего рода весовым коэффициентом, который изменяет величину этой доли, выделяемой каждому процессу. Процессам с бо́льшим значением параметра `nice` (меньшим приоритетом) назначается малый вес, что приводит к выделению небольшой порции процессорного времени. Процессам с меньшим значением параметра `nice` (бо́льшим приоритетом) назначается большой вес, что приводит к выделению большей порции процессорного времени

[2] Вместо термина *квант времени* (timeslice) в некоторых операционных системах также используются термины *квант* (quantum) или *квант процессора* (processor slice). В ОС Linux применяется термин *квант времени*, его мы и будем использовать.

Как уже упоминалось выше, Linux — это операционная система с поддержкой приоритетного мультипрограммного режима. Как только процесс переходит в состояние готовности к выполнению, он может быть запущен. В большинстве операционных систем сам факт, будет ли процесс запущен немедленно, вытеснив текущую выполняемую задачу, зависит от его приоритета и величины доступного кванта времени. В системе Linux с новым планировщиком CFS такое решение зависит от размера доли процессорного времени, которую новый процесс готов поглотить. Если размер этой доли меньше, чем у текущего выполняемого процесса, выполнение последнего прерывается и управление немедленно передается новому процессу. Если нет, то запуск нового процесса откладывается.

Стратегия планирования в действии

Рассмотрим систему с двумя готовыми к выполнению задачами: текстовым редактором и программой перекодировки видео. Текстовый редактор является процессом с интенсивным использованием операций ввода-вывода, поскольку он практически все время ожидает, пока пользователь нажмет клавишу на клавиатуре. Здесь не имеет особого значения, с какой скоростью пользователь вводит данные с клавиатуры, поскольку по сравнению со скоростью работы процессора это происходит *очень* медленно. Несмотря ни на что, при нажатии клавиши пользователь хочет, чтобы текстовый редактор отреагировал *сразу же*.

В противоположность этому программа перекодировки видео интенсивно использует центральный процессор. Если не считать того, что иногда ей требуется прочитать с диска необработанный поток данных и через некоторое время записать на диск перекодированные видеоданные, она практически все время выполняет код видеокодека, обрабатывающего исходные данные. В результате степень использования центрального процессора легко может доходить до 100%. Для программы перекодировки видео нет строгих ограничений на время выполнения: пользователю не важно, запустится она на полсекунды раньше или на полсекунды позже. Конечно, чем раньше она завершит работу, тем лучше, но задержка перед запуском здесь особого значения не имеет.

В рассматриваемом нами случае было бы просто замечательно, если бы планировщик выделял текстовому редактору бо́льшую часть доступного процессорного времени, чем программе обработки видео, поскольку текстовый редактор является интерактивной программой. Для текстового редактора мы должны учитывать два момента. Во-первых, мы хотим выделить для него бо́льшую часть процессорного времени не потому, что он действительно в этом нуждается (как раз наоборот!). Просто нам нужно, чтобы, когда в этом возникнет необходимость, у текстового редактора было достаточно процессорного времени для обработки интерактивной команды. Во-вторых, мы хотим, чтобы текстовый редактор вытеснял процесс по обработке видео в моменты своей активности (например, после того, как пользователь нажмет клавишу). В результате мы добьемся высокой *интерактивной производительности* текстового редактора и сократим время реакции системы на команды пользователя.

В большинстве операционных систем приведенные выше требования выполняются (если вообще выполняются!) за счет того, что текстовому редактору назначается больший приоритет и больший квант времени, чем программе обработки видео. В продвинутых операционных системах все это происходит автоматически путем определения того, что текстовый редактор работает в интерактивном режиме. В системе Linux эти требования также удовлетворяются, но разными средствами. Вместо назначения текстовому редактору определенного приоритета и кванта времени, в системе Linux гарантируется, что

текстовый редактор всегда будет иметь определенную долю процессорного времени. Если в системе существуют только два запущенных процесса (перекодировщик видео и текстовый редактор) и обоим назначено одинаковое значение параметра `nice`, то эта доля будет составлять 50%. Другими словами, каждому процессу будет гарантированно выделено по половине доступного процессорного времени. Поскольку текстовый редактор практически все время находится в состоянии ожидания нажатия клавиши, он использует гораздо меньше выделенных ему 50% процессорного времени. Следовательно, ничто не мешает программе обработки видео использовать центральный процессор сверх установленного в 50% лимита и быстро закончить свою работу.

Из приведенного описания становится ясно, что самым важным является момент перехода текстового редактора в активное состояние. Наша основная задача — сделать так, чтобы текстовый редактор запускался сразу же после нажатия пользователем клавиши клавиатуры. В данном случае в момент перехода текстового редактора в активное состояние планировщик CFS обнаружит, что из выделенных 50% процессорного времени задача использовала значительно меньше. В особенности планировщик определит, что текстовый редактор выполнялся гораздо меньше времени, чем программа обработки видео. Поскольку планировщик стремится *справедливо распределять* время центрального процессора между всеми процессами системы, он тотчас же прервет выполнение программы обработки видео и передаст управление текстовому редактору. После запуска текстовый редактор быстро обработает факт нажатия клавиши пользователем и снова перейдет в состояние ожидания нажатия следующей клавиши. Поскольку текстовый редактор никогда не будет потреблять выделенные ему 50% времени процессора, описанный выше процесс будет продолжаться снова и снова. При этом планировщик CFS будет по первому же требованию запускать текстовый редактор, а все оставшееся время предоставлять программе обработки видео.

Алгоритм работы планировщика системы Linux

В предыдущих разделах была рассмотрена в самых общих чертах теория работы планировщика процессов в операционной системе Linux. При этом мы только вскользь упомянули о том, как эта теория реализована на практике. Теперь, когда мы разобрались с основами, можно более детально рассмотреть работу планировщика ОС Linux.

Классы планировщика

Планировщик системы Linux имеет модульную конструкцию, что позволяет применять различные алгоритмы для планирования разных типов процессов. Этот модульный принцип построения называется *классами планировщика* (scheduler classes). Благодаря этому в одном планировщике могут одновременно сосуществовать несколько разных встроенных алгоритмов, предназначенных для планирования процессов только определенного типа. Каждый класс планировщика имеет свой приоритет. В основном коде планировщика, который находится в файле `kernel/sched.c`, выполняется итерация по каждому классу согласно с его приоритетом. Решение о том, какой процесс будет запущен следующим, принимает класс планировщика, имеющий наивысший приоритет и соответствующий типу готового к выполнению процесса.

Для обычных процессов, которые в системе Linux называются `SCHED_NORMAL`, а в POSIX — `SCHED_OTHER`, зарегистрирован класс полностью справедливого планировщика CFS. Он определен в файле `kernel/sched_fair.c`. В оставшейся части этого

раздела рассматривается алгоритм CFS и его реализация для любого ядра Linux начиная с версии 2.6.23. Класс планировщика, предназначенный для процессов реального времени, будет рассмотрен в следующем разделе.

Планирование процессов в системах Unix

Перед рассмотрением алгоритма справедливого планирования сначала мы должны обсудить вопрос о том, как планируются процессы в традиционных системах Unix. Как уже отмечалось в предыдущих разделах, при проектировании современных системных планировщиков используются две концепции: приоритет процесса и квант времени. Последний определяет, как долго будет выполняться процесс до того, как он будет вытеснен другим процессом. После создания процесса ему назначается некоторый стандартный квант времени. Процессы с бо́льшим приоритетом будут запускаться планировщиком чаще и (в большинстве систем) выполняться дольше. В системах Unix концепция приоритетов переносится в пространство пользователя в виде значений параметра `nice`. На первый взгляд ничего сложного в этом нет, но на практике это приводит к нескольким патологическим проблемам, которые мы сейчас и обсудим.

Первая проблема состоит в преобразовании значений параметра `nice` в размер кванта времени, что требует принятия решения о том, какие абсолютные значения квантов времени должны соответствовать значениям `nice`. Это приводит к тому, что процесс переключения задач будет выполняться квазиоптимальным образом. Например, предположим, что процессам со стандартным значением параметра `nice` (нуль) назначен квант времени в 100 мс, а процессам с наибольшим значением параметра `nice` (+20, самый низкий приоритет) — квант времени в 5 мс. Далее предположим, что один из этих процессов готов к выполнению. Тогда нашему процессу со стандартным приоритетом будет выделено 20/21 (т.е. 100 из 105 мс) долей процессорного времени, а процессу с низким приоритетом — 1/21 (или 5 из 105 мс). Для данного примера можно использовать любые числа, но поскольку мы выбрали именно их, будем считать, что наш выбор оптимален. А теперь подумаем, что произойдет, если будет запущено только два низкоприоритетных процесса? Мы можем предположить, что каждому из них будет выделено по 50% процессорного времени, что и произойдет. Однако каждый из них будет использовать центральный процессор только 5 мс (5 из 10 мс на каждый процесс)! В результате, вместо того, чтобы переключать контекст дважды за 105 мс, нам придется это делать дважды за 10 мс. И наоборот, если два процесса будут иметь обычный приоритет, каждому из них корректно будет выделено по 50% процессорного времени, правда, квантами по 100 мс. Очевидно, что каждая из приведенных выше схем выделения квантов времени не является оптимальной. По сути, каждая из них является побочным продуктом при сопоставлении значению `nice` величины кванта времени, дополненная смесью приоритетов процессов, готовых к выполнению. В самом деле, предположим, что процесс с высоким значением параметра `nice` (с низким приоритетом) работает в фоновом режиме и начинает интенсивно использовать центральный процессор, а процесс с обычным приоритетом работает в диалоговом режиме с пользователем. В таком случае схема назначения размеров квантов времени должна быть в точности *обратной* от описанного здесь идеала!

Вторая проблема связана с относительными значениями параметра `nice` и опять же с их преобразованием в значение квантов времени. Предположим, у нас есть два процесса, отличающиеся значением параметра `nice` на единицу. Для конкретности пусть значения этого параметра равны 0 и 1. Эти значения должны быть преобразованы (и в планировщике типа O(1) так и происходит на самом деле) в кванты времени размером 100

и 95 мс соответственно. Поскольку значения параметра `nice` очень близки, в данном случае различие приоритета в одну единицу параметра `nice` несущественно. А теперь рассмотрим два других процесса, имеющие значения параметра `nice` 18 и 19. Они соответствуют квантам времени в 10 и 5 мс соответственно. При этом первый процесс будет использовать центральный процессор в два раза больше, чем второй! Поскольку параметр `nice` чаще всего задается в виде относительного значения (при вызове системной функции ей передается разность значений параметра `nice`, а не его абсолютное значение), то при этом возникает проблема, когда изменение значения параметра `nice` на единицу приводит к абсолютно разным эффектам в зависимости от начального значения этого параметра.

Третья проблема связана с тем, что при преобразовании значения параметра `nice` в кванты времени у нас должна быть возможность устанавливать абсолютные значения этих квантов. Причем единицы измерения этих абсолютных значений будут зависеть от архитектуры ядра. В большинстве операционных систем значение кванта времени задается в виде некоторого целого числа, соответствующего отсчетам системного таймера. (Подробнее работа со временем будет описана в главе 11, “Таймеры и управление временем”.) Такая постановка вопроса приводит к целому ряду проблем. Во-первых, минимальное значение кванта времени будет привязано к значению периода колебаний системного таймера, который может составлять как 10 мс, так и 1 мс. Во-вторых, использование системного таймера накладывает ограничения на минимальную разницу значений между двумя квантами времени. В результате последовательные значения параметра `nice` могут быть преобразованы в кванты времени с шагом как 10 мс, так и 1 мс. И наконец, в зависимости от периода колебаний системного таймера одни и те же значения кванта времени будут соответствовать разным временны*м* интервалам. Если вы не понимаете, о чем идет речь в этом абзаце, перечитайте его еще раз после знакомства с главой 11. Это единственный непонятный момент, лежащий в основе работы алгоритма CFS.

Четвертая проблема связана с выводом процесса из состояния ожидания при использовании системного планировщика, учитывающего приоритеты, который стремится оптимизировать работу интерактивных приложений. В таких случаях задаче, переходящей в активное состояние, необходимо повысить приоритет, чтобы она запустилась немедленно, даже если ее квант времени уже был исчерпан. Несмотря на то что такой подход позволяет повысить интерактивную производительность приложений в большинстве (если не во всех!) случаев, тем не менее он открывает потенциальную возможность для определенных задач, часто переходящих в состояние ожидания и выходящих из него, несправедливо много использовать процессорное время за счет остальных задач в системе.

Большинство перечисленных выше проблем можно решить, внеся существенные изменения в алгоритм старого планировщика системы Unix, которые, тем не менее, не нарушают основной идеи его построения. Например, для решения второй проблемы нужно вместо аддитивных значений параметра `nice` использовать их разность. А для решения третьей проблемы необходимо при преобразовании значений `nice` в кванты времени каким-то образом абстрагироваться от периода колебаний системного таймера. Однако эти решения вызывают настоящую проблему, когда назначение задачам абсолютных значений квантов времени приводит к тому, что частота их переключения будет постоянной, а степень справедливости использования центрального процессора — разной. Поэтому подход, принятый в планировщике CFS, состоит в радикальном (для системных планировщиков) переосмыслении процесса выделения квантов времени. Вместо назначения процессу определенного

кванта времени ему выделяется определенная доля процессорного времени. В результате в планировщике CFS степень справедливости использования центрального процессора будет постоянной, а частота переключения задач — разной.

Справедливое планирование задач

В основу работы планировщика CFS положен простой принцип: моделирование планирования процессов таким образом, как если бы в системе был установлен идеальный и полностью многозадачный процессор. В подобных системах каждому процессу выделяется $1/n$ процессорного времени, где n — это количество готовых к выполнению процессов. При этом процессы запускаются на бесконечно малое время. В результате в течение произвольного периода времени планировщик должен запустить все n процессов, которые должны выполняться одинаковое количество времени.

В качестве примера предположим, что у нас есть два процесса. При использовании стандартной модели планирования, принятой в системах Unix, мы должны сначала запустить первый процесс на 5 мс, а затем второй процесс на те же 5 мс. При выполнении каждому процессу будет выделено 100% процессорного времени. Однако при использовании идеального и полностью многозадачного процессора оба процесса должны выполняться *одновременно* на протяжении 10 мс. При этом каждому из них должно быть выделено ровно по половине мощности процессора. Вот эта модель планирования и называется *идеальным мультипрограммным режимом работы*, или *идеальной многозадачностью*.

Разумеется, такая модель также непрактична, поскольку на одном процессоре невозможно запустить одновременно несколько процессов в *буквальном* смысле этого слова. Более того, запускать процессы на бесконечно малое время крайне неэффективно, так как при этом *растут издержки* на переключение задач (т.е. замещение одного процесса другим) и падает эффективность использования кеш-памяти процессора. Таким образом, несмотря на то, что нужно стремиться запускать процессы на очень малое время, планировщик CFS еще должен учитывать возникающие при этом издержки и падение производительности из-за неэффективного использования кеш-памяти процессора. Поэтому планировщик CFS запускает каждый процесс на некоторое время, а затем циклически выбирает следующий подходящий для запуска процесс. Вместо назначения каждому процессу определенного кванта времени планировщик CFS вычисляет длительность запуска процесса в зависимости от числа готовых к выполнению процессов. Вместо значения `nice` для вычисления значения кванта времени планировщик CFS использует это значение в качестве *весового коэффициента*, определяющее долю процессорного времени, выделяемого процессу. Процессам с бо́льшим значением параметра `nice` (меньшим приоритетом) назначается меньший вес относительно стандартного значения параметра `nice`. И наоборот, процессам с меньшим значением параметра `nice` (бо́льшим приоритетом) назначается больший вес.

Затем каждому процессу назначается квант времени, пропорциональный его весу, деленный на общий вес всех готовых к выполнению процессов. Для вычисления значения реального кванта времени в планировщике CFS рассчитывается приближенное значение "бесконечно малой" планируемой длительности выполнения задачи в идеальной мультипрограммной среде. Оно называется *планируемой задержкой* (targeted latency). Уменьшение планируемой задержки приводит к улучшению интерактивности процесса и тесному приближению к идеальной мультипрограммной среде. Однако при этом возрастают накладные расходы на переключение процессов и снижается общая производительность системы за счет неэффективного использования кеш-памяти процессора. Предположим,

что планируемая задержка равна 20 мс и что у нас имеется два готовых к выполнению процесса с одинаковыми приоритетами. *Независимо от значения приоритетов этих процессов* каждый из них будет выполняться по 10 мс, после чего будет вытеснен другим процессом. При наличии четырех задач с одинаковыми приоритетами каждая из них будет выполняться только 5 мс. Если таких задач 20, то каждая из них будет выполняться в течение всего 1 мс.

Обратите внимание на то, что при стремлении числа готовых к выполнению задач к бесконечности выделенная доля процессорного времени и размер кванта стремятся к нулю. В конечном счете все это приводит к неприемлемому росту накладных расходов, связанному с переключением задач. Поэтому в планировщике CFS наложено минимальное ограничение на размер кванта времени, выделяемого каждому процессу, которое называется *минимальной гранулярностью* (minimum granularity). По умолчанию ее размер равен 1 мс. Таким образом, даже если число готовых к выполнению процессов в системе стремится к бесконечности, каждый из них все равно будет выполняться 1 мс. Такой подход предотвращает рост накладных расходов на переключение задач. Проницательный читатель уже, наверное, заметил, что планировщик CFS становится не совсем справедливым в случае, когда число готовых к выполнению процессов настолько велико, что рассчитанный квант времени становится меньше минимальной гранулярности. И это правда! Несмотря на то что в теории очередей существуют определенные модификации алгоритма, призванные улучшить сложившееся положение вещей, разработчики планировщика CFS пошли на этот компромисс. Дело в том, что при нормальных условиях, когда в системе существует разумное количество готовых к выполнению процессов, планировщик CFS действительно справедливый!

А теперь снова рассмотрим случай с двумя готовыми к выполнению процессами, но на этот раз они будут отличаться значением параметра `nice`. Пусть для определенности у одного процесса этот параметр равен стандартному значению (нулю), а у другого — 5. Эти значения параметра `nice` соответствуют разным весовым коэффициентам, в результате нашим двум процессам будут назначены разные доли процессорного времени. В данном случае весовые коэффициенты будут уменьшать примерно в 1,3 раза долю процессора низкоприоритетного процесса (со значением `nice`, равным 5). Если планируемая задержка, как и раньше, равна 20 мс, то нашим двум процессам будет выделено по 15 и 5 мс процессорного времени соответственно. Предположим, наши два процесса имеют значение `nice` 10 и 15. Что при этом произойдет со значениями выделяемых квантов времени? Ничего! Будут выделены те же 15 и 5 мс. Абсолютные значения параметра `nice` больше не учитываются планировщиком при принятии решений. На выделяемую долю процессорного времени влияет только относительное значение параметра `nice`.

Обобщая все вышесказанное, отметим, что доля процессорного времени, выделяемая каждому процессу, зависит только от значения относительной разности параметра `nice` между текущим и всеми остальными готовыми к выполнению процессами в системе. Абсолютные значения параметра `nice` не оказывают прямого влияния на размер выделяемого кванта времени, поскольку учитывается только величина их разности. Выделяемые процессам абсолютные значения квантов времени, на которые влияет значение параметра `nice`, теперь не являются абсолютными числами. Они влияют только на размер доли процессорного времени. Планировщик CFS потому и назван *справедливым*, что он предоставляет каждому процессу долю процессорного времени. Как уже отмечалось выше, планировщик CFS не является идеально справедливым, поскольку принятая в нем модель планирования — это всего лишь приближение к идеальной мультипрограммной

среде. Однако для *n* готовых к выполнению процессов в нем *можно* установить минимальное значение планируемой задержки, когда планирование процессов будет еще справедливым.

Реализация планировщика в системе Linux

После обсуждения причин, побудивших к введению в ядро планировщика CFS и логики его работы, мы приступим к исследованию реальной реализации этого планировщика, которая находится в файле `kernel/sched_fair.c`. В частности, будут рассмотрены следующие компоненты планировщика CFS:

- учет времени;
- выбор процесса;
- точка входа в планировщик;
- замораживание и активизация процессов.

Учет времени

Во всех системных планировщиках должно каким-то образом учитываться время выполнения процесса. В большинстве систем Unix это происходит так, как было описано выше, — каждому процессу назначается определенный квант времени. После поступления очередного прерывания от системного таймера значение кванта времени уменьшается на единицу, что соответствует одному периоду колебаний таймера. Как только значение кванта времени становится равным нулю, выполнение текущего процесса прерывается планировщиком и управление передается другому готовому к выполнению процессу, у которого значение кванта больше нуля.

Структура объекта планировщика

Несмотря на то что в планировщике CFS понятие кванта времени не используется, тем не менее он каким-то образом должен учитывать время выполнения каждого процесса. Дело в том, что планировщик CFS должен обеспечить справедливое использование центрального процессора каждым процессом в системе. Для этой цели в нем используется *структура объекта планировщика* (scheduler entity structure), которая называется `sched_entity` и определена в файле `<linux/sched.h>`, как показано ниже.

```
struct sched_entity {
   struct    load_weight          load;
   struct    rb_node              run_node;
   struct    list_head            group_node;
   unsigned  int                  on_rq;
   u64                            exec_start;
   u64                            sum_exec_runtime;
   u64                            vruntime;
   u64                            prev_sum_exec_runtime;
   u64                            last_wakeup;
   u64                            avg_overlap;
   u64                            nr_migrations;
   u64                            start_runtime;
   u64                            avg_wakeup;
/* Множество переменных stat опущено, они появляются, только
   если установлен параметр конфигурации CONFIG_SCHEDSTATS */
};
```

Структура объекта планировщика встроена в *дескриптор процесса* `task_stuct` в виде элемента `se`. (Дескриптор процесса был рассмотрен в главе 3, “Управление процессами”.)

Виртуальное время выполнения

В переменной `vruntime` хранится значение *виртуального времени выполнения* (virtual runtime) процесса, которое соответствует реальному времени выполнения процесса (т.е. количеству времени использования центрального процессора), приведенному (с учетом весовых коэффициентов) к числу готовых к выполнению процессов в системе. Виртуальное время выполнения измеряется в наносекундах и никак не связано с периодом колебаний системного таймера. Оно используется для аппроксимации модели “идеального многозадачного процессора”, принятой в планировщике CFS. В случае подобного идеального процессора нам была бы не нужна переменная `vruntime`, поскольку все готовые к выполнению процессы тогда выполнялись бы в идеальной мультипрограммной среде. Это означает, что при использовании идеального процессора виртуальное время выполнения всех процессов с одинаковым приоритетом было бы одинаковым, т.е. всем задачам была бы выделена равная и справедливая доля реального процессора. Поскольку реальные процессоры очень далеки от идеала, нам приходится запускать каждый процесс по очереди. Поэтому в планировщике CFS используется переменная `vruntime`, предназначенная для учета длительности выполнения процесса на реальном процессоре и принятия решения о том, на сколько времени следует запустить процесс в будущем.

Этот учет выполняется в функции `update_curr()`, которая определена в файле `kernel/sched_fair.c` и приведена ниже.

```
static void update_curr(struct cfs_rq *cfs_rq)
{
   struct sched_entity *curr = cfs_rq->curr;
   u64    now = rq_of(cfs_rq)->clock;
   unsigned long delta_exec;

   if (unlikely(!curr))
      return;

   /*
    * Определим количество времени, в течение которого
    * выполнялась текущая задача с момента последнего
    * изменения нагрузки (это значение не может выходить
    * за пределы 32 битов):
    */
   delta_exec = (unsigned long)(now - curr->exec_start);
   if (!delta_exec)
      return;

   __update_curr(cfs_rq, curr, delta_exec);
   curr->exec_start = now;

   if (entity_is_task(curr)) {
      struct task_struct *curtask = task_of(curr);

      trace_sched_stat_runtime(curtask, delta_exec, curr->vruntime);
      cpuacct_charge(curtask, delta_exec);
      account_group_exec_runtime(curtask, delta_exec);
   }
}
```

В функции update_curr() вычисляется время выполнения текущего процесса, и это значение сохраняется в переменной delta_exec. Затем оно передается в функцию __update_curr(), в которой оно приводится к числу готовых к выполнению процессов в системе. После этого текущее значение переменной vruntime процесса увеличивается на значение весового коэффициента, как показано ниже.

```
/*
* Обновляет статистику времени выполнения текущей задачи.
* Текущая задача пропускается, если она не относится
* к нашему классу планирования.
*/
static inline void
__update_curr(struct cfs_rq *cfs_rq, struct sched_entity *curr,
              unsigned long delta_exec)
{

   unsigned long delta_exec_weighted;

   schedstat_set(curr->exec_max, max((u64)delta_exec, curr->exec_max));

   curr->sum_exec_runtime += delta_exec;
   schedstat_add(cfs_rq, exec_clock, delta_exec);
   delta_exec_weighted = calc_delta_fair(delta_exec, curr);

   curr->vruntime += delta_exec_weighted;
   update_min_vruntime(cfs_rq);
}
```

Функция update_curr() периодически вызывается по сигналу системного таймера, а также в случае, когда процесс готов к выполнению либо переходит в состояние ожидания (т.е. становится не готовым к выполнению). Таким образом, в переменной vruntime точно отслеживается время выполнения текущего процесса. Кроме того, на основе ее значения принимается решение о том, какой процесс должен быть запущен следующим.

Выбор процесса

В предыдущем разделе мы уже говорили о том, что при использовании идеального и полностью многозадачного процессора значение переменной vruntime будет одинаковым для всех готовых к выполнению процессов, имеющих одинаковый приоритет. В реальности идеальной мультипрограммной среды не существует. Поэтому в планировщике CFS предпринята попытка выравнивания виртуального времени выполнения процесса на основе простого правила. При принятии решения о том, какой процесс должен быть запущен следующим, планировщик CFS выбирает процесс с минимальным значением переменной vruntime. Собственно в этом и состоит основной алгоритм справедливого планирования — выбирается задача с наименьшим виртуальным временем выполнения. И это все! Оставшийся материал текущего раздела будет посвящен реализации алгоритма выбора процесса с минимальным значением переменной vruntime.

В планировщике CFS используется *красно-черное дерево* (red-black tree) для отслеживания списка готовых к выполнению процессов и эффективного поиска процесса с наименьшим значением vruntime. Красно-черное дерево, которое в системе Linux называется *rbtree*, является *самобалансирующимся двоичным деревом поиска* (self-balancing binary search tree). Такие деревья в общем и красно-черные деревья в частности будут рассмотрены в главе 6, “Структуры данных ядра”. А пока что тем, кто не знаком с теори-

ей двоичных деревьев, нужно знать, что красно-черные деревья — это структура данных, в элементах которой хранятся произвольные данные, идентифицируемые специальными *ключами* (key). Причем поиск данных по значению некоторого ключа выполняется очень быстро. В частности, следует заметить, что время выборки данных, идентифицируемых некоторым ключом, зависит логарифмически от общего числа элементов дерева.

Выбор следующего процесса

Для начала предположим, что у нас есть красно-черное дерево, элементы которого соответствуют всем готовым к выполнению процессам в системе. При этом ключом для каждого элемента служит значение виртуального времени выполнения процесса. Чуть позже мы рассмотрим способ построения такого дерева, а пока что будем считать, что оно у нас просто есть. В этом дереве процессу, который будет запущен планировщиком CFS следующим (т.е. процессу с минимальным значением `vruntime`), соответствует самый левый узел. Другими словами, если мы будем двигаться по этому дереву от корня вниз и все время проходить через левых потомков, то, достигнув последнего элемента дерева (т.е. максимально удалившись влево), обнаружим процесс с минимальным значением `vruntime`. Опять же, если вы не знакомы с теорией деревьев двоичного поиска, не беспокойтесь. Вам нужно просто знать, что процесс поиска элемента дерева выполняется очень быстро. Таким образом, алгоритм выбора следующего для запуска процесса в планировщике CFS можно сформулировать так: нужно запустить процесс, который соответствует самому левому узлу красно-черного двоичного дерева. Функция, выполняющая этот выбор, называется `__pick_next_entity()`, определена в файле `kernel/sched_fair.c` и приведена ниже.

```
static struct sched_entity *__pick_next_entity(struct cfs_rq *cfs_rq)
{
   struct rb_node *left = cfs_rq->rb_leftmost;

   if (!left)
      return NULL;

   return rb_entry(left, struct sched_entity, run_node);
}
```

Обратите внимание на то, что в функции `__pick_next_entity()` не выполняется реальный обход дерева с целью поиска самого левого узла, поскольку его значение уже сохранено в переменной `rb_leftmost`. Несмотря на то что процесс обхода дерева выполняется очень быстро (время зависит от высоты дерева и в случае сбалансированного дерева подчиняется закону O(logN), где N — количество узлов в дереве), намного проще сохранить его значение в переменной. Данная функция возвращает значение процесса, который планировщик CFS должен запустить следующим. Если функция возвращает значение `NULL`, то самый левый узел в дереве не найден, а это означает, что в дереве вообще нет узлов. В данном случае в системе нет готовых к выполнению процессов, поэтому планировщик CFS должен запустить холостой процесс.

Добавление процесса в дерево

А теперь рассмотрим, как планировщик CFS добавляет процесс в красно-черное дерево и сохраняет в переменной значение самого левого узла. Это происходит в момент перехода процесса в состояние готовности к выполнению (активизации) или при создании первого процесса с помощью функции `fork()`, как описано в главе 3. Добавление процессов в дерево выполняется с помощью функции `enqueue_entity()`, как показано ниже.

```
static void
enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int flags)
{
   /*
    * Обновим нормализованное значение vruntime перед обновлением
min_vruntime
    * с помощью вызова функции update_curr().
    */
    if (!(flags & ENQUEUE_WAKEUP) || (flags & ENQUEUE_MIGRATE))
       se->vruntime += cfs_rq->min_vruntime;

    /*
     * Обновим статистику времени выполнения текущего процесса.
     */
    update_curr(cfs_rq);
    account_entity_enqueue(cfs_rq, se);

    if (flags & ENQUEUE_WAKEUP) {
       place_entity(cfs_rq, se, 0);
       enqueue_sleeper(cfs_rq, se);
    }

    update_stats_enqueue(cfs_rq, se);
    check_spread(cfs_rq, se);
    if (se != cfs_rq->curr)
       __enqueue_entity(cfs_rq, se);
}
```

В этой функции обновляются статистические данные, связанные с временем выполнения процесса, а затем вызывается функция `__enqueue_entity()`. В ней и выполняется вся трудоемкая работа, связанная с добавлением элемента в красно-черное дерево, как показано ниже.

```
/*
 * Добавляет элемент в красно-черное дерево
 */
static void __enqueue_entity(struct cfs_rq *cfs_rq, struct sched_entity
*se)
{
    struct   rb_node   **link = &cfs_rq->tasks_timeline.rb_node;
    struct   rb_node    *parent = NULL;
    struct   sched_entity *entry;
    s64 key = entity_key(cfs_rq, se);
    int leftmost = 1;

    /*
     * Поищем подходящее место в дереве
     */
    while (*link) {
        parent = *link;
        entry = rb_entry(parent, struct sched_entity, run_node);
        /*
         * Не обращаем внимания на конфликты. Узлы с одинаковыми
         * ключами будут храниться вместе.
         */
         if (key < entity_key(cfs_rq, entry)) {
            link = &parent->rb_left;
         } else {
            link = &parent->rb_right;
            leftmost = 0;
         }
}
```

```
/*
 * Запомним адрес самого левого узла дерева, поскольку он часто
 * используется
 */
    if (leftmost)
       cfs_rq->rb_leftmost = &se->run_node;

    rb_link_node(&se->run_node, parent, link);
    rb_insert_color(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Проанализируем эту функцию. Обход дерева в поисках нужного ключа, которым служит значение переменной (vruntime) процесса, выполняется в теле цикла while(). Согласно правилу построения сбалансированного дерева, нужно выбрать дочерний элемент из левой ветки, если значение искомого ключа меньше значения ключа текущего элемента, и из правой ветки — если больше. Если хотя бы один раз мы перешли к правой ветке, значит, помещаемый в дерево процесс никак не может находиться в крайнем левом узле, поэтому значение переменной leftmost обнуляется. Если мы все время перемещались по дереву только влево, значение переменной leftmost остается равным единице, поэтому в конце цикла мы получим новое значение самого левого узла. Теперь можно обновить значение переменной rb_leftmost и занести в нее процесс, вставляемый в дерево. Цикл завершается, когда сравнивается значение ключа у узла, который не имеет дочерних узлов в направлении нашего движения. При этом значение переменной link становится равным NULL и цикл завершается. После выхода из цикла вызывается функция rb_link_node() для родительского узла, которая делает из нашего вставляемого в дерево процесса новый дочерний узел. В функции rb_insert_color() обновляются самобалансирующиеся свойства дерева. Процесс раскрашивания деревьев будет рассмотрен в главе 6, “Структуры данных ядра”.

Удаление процесса из дерева

Теперь рассмотрим, как планировщик CFS удаляет процессы из красно-черного дерева. Это происходит в момент блокировки процесса (когда процесс становится не готовым к выполнению) или его завершения (т.е. процесс перестает существовать).

```
static void
dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity *se, int sleep)
{
    /*
     * Обновим статистику времени выполнения текущего процесса.
     */
    update_curr(cfs_rq);

    update_stats_dequeue(cfs_rq, se);
    clear_buddies(cfs_rq, se);

    if (se != cfs_rq->curr)
       __dequeue_entity(cfs_rq, se);

    account_entity_dequeue(cfs_rq, se);
    update_min_vruntime(cfs_rq);

    /*
     * Нормализуем объект после обновления min_vruntime, поскольку оно
     * может повлиять на элемент ->curr. Нам нужно отразить это изменение
     * в нашем нормализованном положении.
     */
```

```
    if (!sleep)
       se->vruntime -= cfs_rq->min_vruntime;
}
```

Как и при добавлении процесса к красно-черному дереву, основная работа выполняется во вспомогательной функции `__dequeue_entity()`, как показано ниже.

```
static void __dequeue_entity(struct cfs_rq *cfs_rq, struct sched_entity
*se)
{
   if (cfs_rq->rb_leftmost == &se->run_node) {
      struct rb_node *next_node;

      next_node = rb_next(&se->run_node);
      cfs_rq->rb_leftmost = next_node;
   }

   rb_erase(&se->run_node, &cfs_rq->tasks_timeline);
}
```

Удаление процесса из дерева намного проще, чем вставка, поскольку при реализации красно-черного дерева предусмотрена функция `rb_erase()`, которая и выполняет всю работу. В конце этой функции обновляется значение переменной `rb_leftmost`. Если удаляемый процесс относился к самому левому узлу дерева, вызывается функция `rb_next()`, которая выполняет поиск следующего узла в порядке обхода дерева. После удаления текущего самого левого узла этот узел становится самым левым узлом дерева.

Точка входа в планировщик

Основной точкой входа в системный планировщик является функция `schedule()`, определенная в файле `kernel/sched.c`. Это как раз та функция, которая вызывается из других частей ядра для активизации системного планировщика, который и решает, какой процесс должен быть запущен, а затем запускает его. По сравнению с классами планировщика функция `schedule()` является очень обобщенной. Другими словами, в ней выполняется поиск класса с самым высоким приоритетом, связанного с процессом, готовым к выполнению, и запрашивается у него, какой процесс должен быть запущен следующим. Судя по этому описанию, для вас не должно быть неожиданностью, что функция `schedule()` очень проста. В ней есть только одна важная часть, которая малоинтересна для читателей, чтобы приводить ее на страницах этой книги, и связанная с вызовом функции `pick_next_task()`. Эта функция также определена в файле `kernel/sched.c`. В ней выполняется обход каждого класса планировщика, начиная с класса с самым высоким приоритетом, и выбирается самый высокоприоритетный процесс в самом высокоприоритетном классе, как показано ниже.

```
/*
 * Выбор самой высокоприоритетной задачи
 */
static inline struct task_struct *
pick_next_task(struct rq *rq)
{
   const struct sched_class *class;
   struct task_struct *p;

   /*
    * Оптимизация: нам известно, что если все задачи принадлежат
    * классу fair, эту функцию можно вызвать напрямую
    */
```

```
    if (likely(rq->nr_running == rq->cfs.nr_running)) {
        p = fair_sched_class.pick_next_task(rq);
        if (likely(p))
            return p;
    }

    class = sched_class_highest;
    for ( ; ; ) {
        p = class->pick_next_task(rq);
        if (p)
            return p;
        /*
         * Значение никогда не будет равно NULL, поскольку класс idle всегда
         * возвращает значение не-NULL для p.
         */
        class = class->next;
    }
}
```

Обратите внимание на оптимизацию, выполненную в начале функции. Поскольку CFS является классом планировщика для обычных процессов, а в большинстве систем запускаются именно обычные процессы, существует небольшой обходной маневр, позволяющий ускорить выбор следующего процесса в классе CFS в случае, если общее количество готовых к выполнению процессов в системе равно количеству готовых к выполнению процессов в классе CFS. В таком случае все готовые к выполнению процессы будут относиться к классу CFS.

Основу рассматриваемой функции составляет цикл `for()`, в котором выполняется обход каждого класса планировщика согласно с его приоритетом, начиная с самого высокого. В каждом классе реализована функция `pick_next_task()`, которая возвращает указатель на следующий готовый к выполнению процесс или, если таковых нет, значение `NULL`. Следующий готовый к выполнению процесс выбирается из класса, который первым вернул значение не-`NULL`. В реализации функции `pick_next_task()` для класса CFS вызывается функция `pick_next_entity()`, в которой в свою очередь вызывается функция `__pick_next_entity()`, которую мы обсуждали в предыдущем разделе.

Замораживание и активизация процессов

Замороженные (заблокировнные) задачи находятся в специальном состоянии *не готовности к выполнению* (nonrunnable state). Этот момент очень важен, поскольку без введения такого состояния системный планировщик мог бы выбрать и запустить на выполнение задачу, которая сама того не ожидает либо, хуже того, в которой режим ожидания наступления определенного события реализован в виде бесконечного цикла, впустую расходующего процессорное время. Задача может быть заморожена по нескольким причинам, однако почти всегда это связано с ожиданием наступления определенного события. Таким событием может быть истечение определенного интервала времени, поступление порции данных при чтении файла либо прерывание от какого-либо аппаратного устройства. Кроме того, задача может быть непреднамеренно заморожена при попытке захватить семафор в ядре (это описывается в главе 9, “Общие сведения о синхронизации кода ядра”). Чаще всего задача замораживается при выполнении файловых операций ввода-вывода. Например, при выполнении функции `read()` системе отправляется запрос на чтение данных из файла, при котором они должны быть прочитаны с диска. Другой пример — ожидание ввода данных с клавиатуры. В любом случае реакция ядра будет одинаковой: задача сама себя помечает как замороженную, переводит сама себя в очередь

ожидающих процессов, удаляет сама себя из красно-черного дерева готовых к выполнению процессов, после чего вызывает функцию `schedule()` для выбора нового процесса, которому будет передано управление. При активизации задачи все происходит наоборот: задача переходит в состояние готовности к выполнению, удаляется из очереди ожидания и добавляется к красно-черному дереву.

Как уже отмечалось в предыдущей главе, при замораживании задача может находиться в одном из двух состояний: `TASK_INTERRUPTIBLE` и `TASK_UNINTERRUPTIBLE`. Они отличаются только тем, что в состоянии `TASK_UNINTERRUPTIBLE` задача игнорирует поступившие сигналы, в то время как в состоянии `TASK_INTERRUPTIBLE` — принудительно активизируется и обрабатывает пришедший сигнал. Оба типа замороженных задач помещаются в очередь ожидающих процессов, ждут наступления определенного события и не являются готовыми к выполнению.

Очереди ожидания

Замораживание задачи обрабатывается с помощью *очередей ожидания* (wait queue). Очередь ожидания представляет собой обычный список процессов, которые ожидают наступления определенного события. Очереди ожидания в ядре представляются с помощью типа данных `wait_queue_head_t`. Они могут быть созданы статически с помощью макроса `DECLARE_WAITQUEUE()` или выделены динамически с помощью функции `init_waitqueue_head()`. Процессы помещают сами себя в очередь ожидания и помечают себя как не готовые к выполнению. При поступлении события, связанного с очередью ожидания, процессы, находящиеся в этой очереди, активизируются. Очень важно, чтобы процессы замораживания и активизации задачи были реализованы корректно, чтобы не возникал конфликт из-за доступа к ресурсу[3] (race conditions).

Существуют простые интерфейсы для замораживания задачи, и они широко используются. Однако при этом может возникнуть конфликт, когда задача замораживается уже *после* того, как соответствующее событие произошло. В подобном случае задача может быть заморожена навсегда. Поэтому в ядре Linux рекомендован более сложный метод замораживания задач, как показано ниже.

```
/* 'q' - это очередь ожидания, в которую мы хотим поместить
 * наш процесс
 */
DEFINE_WAIT(wait);

add_wait_queue(q, &wait);
while (!condition) { /* condition - это событие, которого мы ожидаем */
   prepare_to_wait(&q, &wait, TASK_INTERRUPTIBLE);
   if (signal_pending(current))
      /* Обработка сигнала */
   schedule();
}
finish_wait(&q, &wait);
```

[3] Такое крайне нежелательное состояние возникает в случае, когда некоторый драйвер устройства и процесс в системе пытаются выполнить несколько операций одновременно. Однако в силу конструкции устройства или системы эти операции должны быть выполнены в определенном порядке, иначе происходит потеря данных. Типичный пример — одновременная выдача команд на чтение и запись больших массивов данных. При этом данные, которые читаются, могут быть перезаписаны. В результате операционная система может аварийно завершить свою работу из-за выполнения некорректной операции, вызванной перезаписью данных раньше времени. — *Примеч. ред.*

Чтобы поместить себя в очередь ожидания, задача должна выполнить перечисленные ниже действия.

1. Создать элемент очереди ожидания с помощью макроса `DEFINE_WAIT()`.
2. Добавить себя в очередь ожидания, вызвав функцию `add_wait_queue()`. С помощью этой очереди процесс будет активизирован при наступлении события, которого он ожидает. Разумеется, для этого где-то в другом месте системы должен быть код, в котором вызывается функция `wake_up()` для данной очереди, при наступлении соответствующего события.
3. Вызвать функцию `prepare_to_wait()` для изменения состояния процесса на один из `TASK_INTERRUPTIBLE` или `TASK_UNINTERRUPTIBLE`.
4. Если в п. 3 было выбрано состояние `TASK_INTERRUPTIBLE`, при поступлении сигнала процесс будет активизирован. Такая активизация называется *фиктивной* (spurious), поскольку она вызвана сигналом, а не некоторым событием. В результате задача может проверить тип поступившего сигнала и обработать его.
5. При активизации задачи она снова должна проверить выполнение ожидаемого условия. Если условие выполняется, то производится выход из цикла. Если нет, то снова вызывается функция `schedule()` и повторяется проверка условия.
6. Когда условие выполнено, задача может установить свое состояние в значение `TASK_RUNNING` и удалить себя из очереди ожидания с помощью функции `finish_wait()`.

Если условие выполнится перед тем, как задача будет заморожена, то цикл прервется и задача не перейдет в замороженное состояние по ошибке. Следует заметить, что в теле цикла в коде ядра часто могут выполняться и другие задачи. Например, перед выполнением функции `schedule()` может потребоваться снять блокировки с некоторых ресурсов и захватить их снова после возврата из этой функции либо отреагировать на некоторые другие события.

В качестве простого примера кода, в котором используются очереди ожидания, можно привести функцию `inotify_read()`, определенную в файле `fs/notify/inotify/inotify_user.c`, которая считывает данные из файлового дескриптора `inotify`[4].

```
static ssize_t inotify_read(struct file *file, char __user *buf,
                                        size_t count, loff_t *pos)
{
   struct fsnotify_group *group;
   struct fsnotify_event *kevent;
   char __user *start;
   int ret;
   DEFINE_WAIT(wait);

   start = buf;
   group = file->private_data;

   while (1) {
      prepare_to_wait(&group->notification_waitq,
                      &wait,
                      TASK_INTERRUPTIBLE);
```

[4] Подсистема ядра Linux `inotify` позволяет получать уведомления об изменениях в файловой системе. — *Примеч. ред.*

```
        mutex_lock(&group->notification_mutex);
        kevent = get_one_event(group, count);
        mutex_unlock(&group->notification_mutex);

        if (kevent) {
           ret = PTR_ERR(kevent);
           if (IS_ERR(kevent))
              break;
           ret = copy_event_to_user(group, kevent, buf);
           fsnotify_put_event(kevent);
           if (ret < 0)
              break;
           buf += ret;
           count -= ret;
           continue;
        }
        ret = -EAGAIN;

        if (file->f_flags & O_NONBLOCK)
           break;
        ret = -EINTR;

        if (signal_pending(current))
           break;

        if (start != buf)
           break;

        schedule();
     }
     finish_wait(&group->notification_waitq, &wait);

     if (start != buf && ret != -EFAULT)
        ret = buf - start;
     return ret;
}
```

Эта функция соответствует шаблону, описанному выше. Основное отличие заключается в том, что выполнение определенного условия проверяется в теле цикла `while()`, а не в самом операторе `while()`. Дело в том, что в функции `inotify_read()` выполняется сложная проверка условий, для которой требуется блокировка ресурса. Для прекращения цикла используется инструкция `break`.

Активизация задачи

Активизация задачи выполняется с помощью функции `wake_up()`, которая размораживает все задачи, находящиеся в состоянии ожидания в данной очереди. В ней вызывается функция `try_to_wake_up()`, которая устанавливает состояние задачи в `TASK_RUNNING`, затем вызывается функция `enqueue_task()`, добавляющая процесс в красно-черное дерево, и устанавливается флаг `need_resched`, если приоритет размороженного процесса больше, чем приоритет текущего выполняемого процесса. В коде, который отвечает за наступление некоторого события, обычно вызывается функция `wake_up()` 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.

Важно отметить, что задача, находящаяся в состоянии ожидания, может подвергаться фиктивной активизации. Однако сам факт активизации совсем не означает, что требуемое событие произошло. Поэтому переход задачи в состояние ожидания должен всегда

выполняться в цикле, проверяющем, что нужное событие действительно произошло. На рис. 4.1 показана взаимосвязь состояний планировщика.

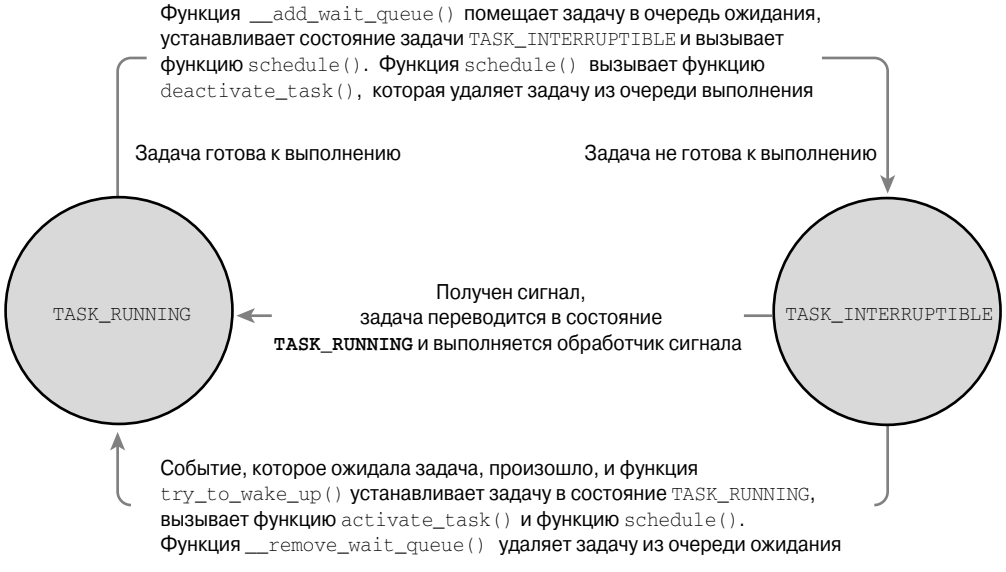


Рис. 4.1. Замораживание и активизация задачи

Вытеснение и переключение контекста

Под переключением контекста будем понимать переход от одной выполняемой задачи к другой, который осуществляется с помощью функции `context_switch()`, определенной в файле `kernel/sched.c`. Эта функция вызывается из функции `schedule()` после выбора нового процесса, который должен быть запущен. В ней выполняются два важных шага, как показано ниже.

- Вызывается функция `switch_mm()`, которая определена в файле `<asm/mmu_context.h>`. Она выполняет переход от виртуальной памяти старого процесса к виртуальной памяти нового процесса.
- Вызывается функция `switch_to()`, определенная в файле `<asm/system.h>`, которая переключает состояние процессора от старой задачи к новой. Эта процедура включает сохранение и восстановление информации стека ядра и регистров процессора, а также любого другого состояния, которое зависит от применяемой аппаратной платформы и должно выполняться для каждого процесса.

Как бы там ни было, ядро должно точно знать, когда следует вызывать функцию `schedule()`. Если бы эта функция явно вызывалась только в коде самого процесса, то пользовательские приложения могли бы выполняться неопределенное время. Поэтому в ядре предусмотрен флаг `need_resched`, сигнализирующий о том, что пришло время перепланировать задачи (табл. 4.1). Этот флаг устанавливается в функции `scheduler_tick()` в момент вытеснения процесса и в функции `try_to_wake_up()`, если активизируемый процесс имеет более высокий приоритет, чем текущий выполняемый процесс. Этот флаг проверяется в коде ядра, и если он установлен, то вызывается функция

schedule() для переключения на новый процесс. Флаг need_resched является своего рода сигналом ядру о том, что нужно как можно скорее запустить системный планировщик, поскольку появился новый процесс, готовый к выполнению.

Таблица 4.1. Функции, использующие флаг need_resched

Функция	Назначение
set_tsk_need_resched()	Установить флаг need_resched для данного процесса
clear_tsk_need_resched()	Сбросить флаг need_resched для данного процесса
need_resched()	Проверить значение флага need_resched для данного процесса. Возвращается значение true, если флаг установлен, и false, если сброшен

Флаг need_resched проверяется во время возврата в пространство пользователя или при выходе из процедуры обработки прерывания. Если он установлен, ядро вызывает системный планировщик перед продолжением своей обычной работы.

Флаг need_resched не является глобальным и устанавливается для каждого процесса. Все дело в том, что гораздо быстрее обратиться к значению, хранящемуся в дескрипторе процесса, чем к глобальной переменной. Как вы помните, адрес дескриптора текущего процесса определяется очень быстро, к тому же велика вероятность, что этот дескриптор будет находиться в кеш-памяти процессора. Исторически так сложилось, что в версиях ядра до 2.2 этот флаг был реализован в виде глобальной переменной. В версиях ядра 2.2–2.4 этот флаг имел тип int и располагался в структуре task_struct. В версии ядра 2.6 он был перенесен в отдельный бит специальной флаговой переменной, находящейся в структуре thread_info.

Вытеснение пространства пользователя

Вытеснение пространства пользователя (user preemption) происходит в тот момент, когда ядро собирается возвратить управление пользовательской программе и при этом установлен флаг need_resched, в результате чего вызывается системный планировщик. При возврате в пользовательскую программу ядро проверяет, находится ли оно в *безопасном статическом состоянии* (safe quiescent state). Другими словами, если оно может безопасно продолжить выполнение текущей задачи, то это означает, что можно безопасно выбрать новую задачу для выполнения. Следовательно, при возврате из ядра в пользовательскую программу, которое происходит в конце обработчика прерывания или в результате вызова системной функции, нужно проверить значение флага need_resched. Если он установлен, то вызывается системный планировщик для выбора нового (более подходящего) процесса, которому будет передано управление. Обе ветки возврата в пользовательское приложение (после обработки прерывания и после вызова системной функции), как правило, зависят от используемой аппаратной платформы и обычно реализуются на языке ассемблера в файле entry.S. В этом файле, помимо кода входа в режим ядра, также содержится и код выхода из режима ядра.

Если говорить вкратце, то вытеснение пространства пользователя может произойти в следующих случаях:

- при возврате в пространство пользователя из системной функции;
- при возврате в пространство пользователя из обработчика прерывания.

Вытеснение пространства ядра

В отличие от большинства вариантов операционной системы Unix и многих других операционных систем, в ядре операционной системы Linux поддерживается *приоритетный мультипрограммный режим* (preemptible). В ядрах, где такой режим не поддерживается, выполнение программного кода ядра не прерывается и продолжается до самого его завершения. Другими словами, в таких ядрах системный планировщик не может перепланировать задачу, если та находится в режиме ядра. В результате планирование выполняется по кооперативному принципу, а не по принципу вытеснения. При этом код ядра будет выполняться либо до его завершения (т.е. возврата в пользовательскую программу), либо пока он не будет заблокирован явно. Поэтому в ядре Linux серии 2.6 стал поддерживаться приоритетный мультипрограммный режим. Теперь планировщик может перепланировать задачу в любой момент при условии, что ядро находится в состоянии, когда это можно безопасно выполнить.

Итак, в каком же состоянии можно безопасно выполнить перепланирование задачи? Ядро способно вытеснить задачу, работающую в пространстве ядра, когда эта задача не удерживает блокировку. Иными словами, блокировки используются в качестве маркеров тех участков кода, в которых задание не может быть вытеснено. Поскольку в ядре поддерживается многопроцессорная работа (SMP-safe), поэтому если блокировка не удерживается, то код ядра является реентерабельным (повторно входимым) и его можно безопасно вытеснить.

Первое изменение, внесенное в ядро и связанное с мультипрограммированием, связано с добавлением счетчика вытеснения `preempt_count` в структуру `thread_info` каждого процесса. Первоначально значение этого счетчика 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, в случае если возврат выполняется в пространство ядра, проверяются 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. В результате будет вызван системный 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 выполняется как обычно, т.е. в текущую выполняющуюся программу. 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 вызывается системный планировщик. Иногда коду ядра необходимо иметь возможность запрещать или разрешать вытеснение задач в режиме ядра. Эта тема будет рассмотрена в главе 9.

Вытеснение пространства ядра также может произойти явно, когда задача блокируется в режиме ядра или явно вызывается функция `schedule()`. Такая форма мультипрограммной работы ядра поддерживалась всегда, так как в этом случае не нужно вводить дополнительную логику для проверки того, что вытеснение проводить безопасно. Предполагается, что если в коде ядра явно вызывается функция `schedule()`, то точно известно, что перепланирование производить безопасно.

Итак, вытеснение пространства ядра может произойти в перечисленных ниже случаях.

- При возврате из обработчика прерывания в пространство ядра.
- Когда код ядра снова становится вытесняемым.

- 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()`).

Стратегии планирования в режиме реального времени

В операционной системе Linux предусмотрены две стратегии планирования в режиме *реального времени* (real-time): `SCHED_FIFO` и `SCHED_RR`. Планирование обычных задач (т.е. не в режиме реального времени), осуществляется с использованием стратегии `SCHED_NORMAL`. В рамках инфраструктуры *классов планирования* (scheduling classes) указанные выше стратегии планирования в режиме реального времени осуществляются не в планировщике CFS, а в специальном планировщике реального времени, который определен в файле `kernel/sched_rt.c`. В оставшейся части этого раздела мы обсудим алгоритмы и стратегии планирования в режиме реального времени.

В стратегии `SCHED_FIFO` реализован простой алгоритм планирования по принципу "первым пришел — первым обслужен" без использования квантов времени. Готовая к выполнению задача со стратегий `SCHED_FIFO` будет всегда планироваться на выполнение перед всеми другими заданиями со стратегией планирования `SCHED_NORMAL`. Когда задача со стратегией `SCHED_FIFO` становится готовой к выполнению, она будет выполняться до тех пор, пока не будет заморожена или пока явно не отдаст управление другой задаче. Поскольку кванты времени при этом не используются, задача может выполняться бесконечно долго. Ее может вытеснить только задача с более высоким приоритетом, которая планируется по стратегии `SCHED_FIFO` или `SCHED_RR`. Если существуют две или более задачи с одинаковым приоритетом, планируемые по стратегии `SCHED_FIFO`, они будут выполняться по кругу, однако передача управления от одной задачи к другой будет выполняться только тогда, когда текущая задача явно уступит центральный процессор другой задаче. Если задача, планируемая по стратегии `SCHED_FIFO`, готова к выполнению, то все другие задачи с более низким приоритетом не могут выполняться до тех пор, пока текущая задача не станет не готовой к выполнению.

Стратегия планирования `SCHED_RR` идентична `SCHED_FIFO` за исключением того, то каждый процесс может выполняться до момента исчерпания выделенного кванта времени процессора. Другими словами, `SCHED_RR` — это `SCHED_FIFO` с квантованием, или *циклический* (round-robin) алгоритм планирования задач реального времени. Как только квант времени, выделенный задаче `SCHED_RR`, будет исчерпан, все остальные задачи реального времени с одинаковым приоритетом планируются по циклическому алгоритму. Здесь квант времени используется только для того, чтобы перепланировать остальные задачи с одинаковым 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 исчерпался квант времени.

В обеих стратегиях планирования в режиме реального времени используются 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 приоритетом.

Рассматриваемые в этом разделе стратегии планирования в операционной системе Linux обеспечивают так называемый *мягкий режим реального времени* (soft real-time). Это означает, что ядро пытается планировать выполнение пользовательских приложений в границах допустимых временны*х* сроков, но не всегда гарантирует, что поставленная цель будет достигнута. В отличие от этого операционные системы с *жестким режимом реального времени* (hard real-time) всегда гарантируют выполнение всех требований по планированию выполнения процессов в заданных временны*х* рамках. Операционная система Linux не может гарантировать возможности планирования задач в реальном времени. Несмотря на то что в ОС Linux не реализована стратегия жесткого планирования задач в реальном времени, тем не менее производительность таких задач вполне приемлема. Ядро серии 2.6 в состоянии работать в очень жестких временны*х* рамках.

Значение приоритетов задач реального времени лежат в диапазоне от 0 до `MAX_RT_PRIO` минус 1. Стандартное значение константы `MAX_RT_PRIO` равно 100, поэтому стандартный диапазон значений приоритетов задач реального времени составляет от 0 до 99. Для задач со стратегией планирования `SCHED_NORMAL` это пространство приоритетов объединяется со значениями параметра `nice`. В результате диапазон приоритетов может лежать от `MAX_RT_PRIO` до `(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 функции для управления планировщиком

В операционной системе Linux предусмотрено семейство системных функций, предназначенных для изменения параметров работы планировщика задач. С их помощью можно изменять приоритет процесса, стратегию планирования, *привязку к процессору* (processor affinity), а также явно *передать* (yield) процессор в использование другим задачам.

Эти функции подробно описаны в различных книгах, а также в справочном руководстве системы Linux. Все они реализованы в библиотеке C без всяких интерфейсных оболочек и поэтому вызываются напрямую как обычная системная функция. Список функций, управляющих системным планировщиком, а также их краткое описание приведено в табл. 4.2. О реализации системных функций в ядре речь пойдет в главе 5, “Системные функции”.

Таблица 4.2. Системные функции для управления планировщиком

Функция	Описание
`nice()`	Задает значение параметра `nice` для процесса
`sched_setscheduler()`	Задает стратегию планирования для процесса
`sched_getscheduler()`	Позволяет узнать стратегию планирования для процесса
`sched_setparam()`	Устанавливает значение приоритета реального времени для процесса
`sched_getparam()`	Позволяет узнать значение приоритета реального времени для процесса
`sched_get_priority_max()`	Позволяет узнать максимальное значение приоритета реального времени для процесса
`sched_get_priority_min()`	Позволяет узнать минимальное значение приоритета реального времени для процесса

Окончание табл. 4.2

Функция	Описание
sched_rr_get_interval()	Позволяет узнать длительность кванта времени для процесса
sched_setaffinity()	Устанавливает привязку процесса к процессору
sched_getaffinity()	Позволяет узнать привязку процесса к процессору
sched_yield()	Позволяет временно передать процессор другим заданиям

Системные функции для изменения стратегии и приоритета

С помощью системных функций sched_setscheduler() и sched_getscheduler() можно соответственно установить и определить стратегию планирования и приоритет реального времени для заданного процесса. При реализации этих функций, как и большинства других системных функций, выполняется большое количество проверок значений аргументов, а также различные действия, связанные с инициализацией и завершением работы функции. Хотя полезная работа этих функций связана только с чтением/записью значений полей policy и rt_priority, находящихся в структуре task_struct.

Системные функции sched_setparam() и sched_getparam() позволяют задать и узнать значение приоритета реального времени для процесса. Они кодируют значение поля rt_priority в специальной структуре sched_param. Функции sched_get_priority_max() и sched_get_priority_min() возвращают соответственно значение максимального и минимального приоритетов для заданной 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 данного процесса на указанную в аргументе величину. Только пользователь с правами 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 соответственно значения полей static_prio и prio структуры task_struct.

Системные функции для изменения привязки к процессору

В планировщике системы Linux поддерживается стратегия жесткой привязки к процессору. Это означает, что, хотя планировщик пытается обеспечивать мягкую или естественную привязку путем удержания процессов на одном и том же процессоре, пользователь может сказать: “Эти задания должны выполняться только на указанных мною процессорах независимо ни от чего”. Значение жесткой привязки хранится в виде битовой маски в поле cpus_allowed структуры task_struct. Каждый бит этой маски соответствует одному из возможных процессоров в системе. По умолчанию все биты маски установлены в единицу, поэтому процесс потенциально может выполняться на любом из процессоров в системе. Однако пользователь с помощью системной функции sched_setaffinity() может задать другую битовую маску, содержащую произвольную комбинацию нулей и единиц. По аналогии функция sched_getaffinity() возвращает текущее значение поля cpus_allowed, содержащего битовую маску.

В ядре жесткая привязка обеспечивается очень простым способом. Во-первых, вновь созданный процесс наследует маску привязки от родительского процесса. Поскольку ро-

дительский процесс выполняется на разрешенном процессоре, то и порожденный процесс также будет выполняться на разрешенном процессоре. Во-вторых, когда привязка процесса изменяется, ядро использует *миграционные потоки* (migration threads) для выталкивания задания на разрешенный процессор. И наконец, *стабилизатор нагрузки* (load balancer) перебрасывает задачу только на разрешенный процессор. Таким образом, процесс всегда будет выполняться только на том процессоре, которому соответствует установленный бит в поле `cpus_allowed` дескриптора процесса.

Передача процессорного времени другим задачам

В ОС Linux предусмотрена системная функция `sched_yield()`, посредством которой процесс может явно уступить процессор другим ожидающим выполнения процессам. Эта функция удаляет текущий процесс из массива активных процессов (где он и находится, поскольку выполняется) и помещает его в массив процессов с истекшим временем выполнения. В результате происходит не только вытеснение процесса и помещение его в конец списка заданий с соответствующим приоритетом, но и гарантируется, что процесс не будет выполняться некоторое время. Поскольку задачи реального времени никогда не могут быть перемещены в массив процессов с истекшим временем выполнения, они представляют собой особый случай. Поэтому они просто перемещаются в конец списка заданий с заданным приоритетом и не помещаются в массив процессов с истекшим временем выполнения. В ранних версиях ОС Linux смысл вызова системной функции `sched_yield()` был совсем другим — в лучшем случае задача просто перемещалась в конец списка заданий с заданным приоритетом. В результате задача уступала процессор другим задачам на непродолжительное время. В современных версиях Linux перед вызовом функции `sched_yield()` пользовательские приложения и даже код ядра должны быть абсолютно уверены в том, что они действительно хотят уступить центральный процессор другим задачам.

В коде ядра для удобства можно вызывать функцию `yield()`, которая проверит, что задача находится в состоянии `TASK_RUNNING`, и только после этого вызовет функцию `sched_yield()`. В пользовательских приложениях используется системная функция `sched_yield()`.

Резюме

Системный планировщик является важным компонентом ядра любой операционной системы, поскольку в запуске процессов состоит основной смысл использования компьютера (по крайней мере, для большинства из нас!). Однако удовлетворение всех требований, предъявляемых к планировщику, — очень не простая задача. Большое количество выполняемых 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 системный планировщик CFS ядра Linux приближается к тому, чтобы удовлетворить всем этим требованиям и обеспечить оптимальное решение для всех случаев, включая отличную масштабируемость и привлекательную реализацию.

В предыдущей главе мы рассмотрели средства управления процессами. В этой главе была описана теория планирования процессов и ее специфические реализации, алгоритмы и интерфейсы, используемые в текущей версии ядра Linux. В следующей главе будет рассмотрен основной интерфейс, который предоставляет ядро выполняющимся процессам. Речь пойдет о системных функциях.

5
Системные функции

В ядре любой современной ОС предусмотрен набор интерфейсов, посредством которых пользовательские приложения могут взаимодействовать с системой. Благодаря им приложения могут иметь контролируемый доступ к системному оборудованию, создавать новые процессы и взаимодействовать с ними, а также запрашивать другие ресурсы операционной системы. Интерфейсы выступают в роли посредника между приложениями и ядром. При этом приложения посредством интерфейсов выдают разнообразные запросы, а ядро выполняет их либо возвращает соответствующий код ошибки. Существование подобных интерфейсов и сам факт, что приложения не могут свободно делать то, что они захотят, является основным моментом при создании стабильной ОС.

Взаимодействие с ядром

Системные функции являются прослойкой между оборудованием компьютера и пользовательскими процессами, которая служит трем основным целям. Во-первых, она обеспечивает абстрактный интерфейс между оборудованием компьютера и пользовательскими приложениями. Например, при чтении данных из файла или записи в файл для приложения не имеет значения, на каком диске или носителе он расположен и какой тип файловой системы имеет. Во-вторых, системные функции гарантируют безопасность и стабильность системы. Так как ядро выступает в качестве посредника между ресурсами системы и пользовательскими приложениями, оно может принимать решение о предоставлении доступа к ресурсам в соответствии с правами пользователей и другими критериями. Например, это позволяет не допустить неправильное использование аппаратных ресурсов программами, воровство каких-либо ресурсов у других программ, а также возможность нанесения вреда системе. И наконец, единственный общий слой между пользовательскими программами и остальной частью системы позволяет осуществить виртуальное представление процессов (см. главу 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 и, конечно же, достичь стабильности работы системы и ее безопасности. В операционной системе Linux вызовы системных

функций являются единственным средством, благодаря которому пользовательские программы могут связываться с ядром; они являются единственной законной точкой входа в ядро, помимо исключений и прерываний. Более того, доступ ко всем остальным интерфейсам системы, таким как файлы устройств, или к системе `/proc` в конечном итоге осуществляется через вызовы системных функций. Интересно, что в ОС Linux реализовано значительно меньше системных функций, чем во многих других операционных системах[1]. В этой главе рассказывается о роли и реализации системных функций в операционной системе Linux.

API, POSIX и библиотека C

Как правило, при разработке прикладных программ прямые вызовы системных функций используются редко, так как для этого существует набор соответствующих функций, входящих в API (Application Programming Interface, или программный интерфейс приложения). Это очень важный момент, поскольку при таком подходе нет прямой связи между интерфейсами, которые используются в приложениях, и интерфейсами, которые предоставляет ядро. В API определен набор программных интерфейсов, которые используются в приложениях. Эти интерфейсы могут быть реализованы с помощью вызова одной или нескольких системных функции, а также вообще без их использования. В действительности может существовать один и тот же программный интерфейс приложений для различных операционных систем, в то время как реализация этих API для разных ОС может существенно отличаться. Взаимосвязь между POSIX API, библиотекой C и системными функциями показана на рис. 5.1.

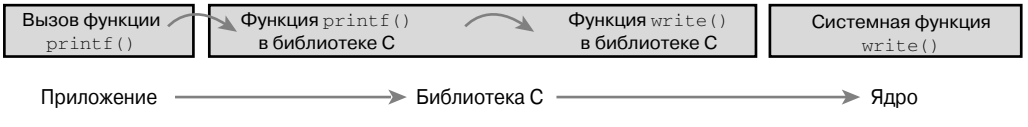


Рис. 5.1. Взаимосвязь между приложениями, библиотекой C и ядром на примере вызова функции `printf()`

Один из самых популярных программных интерфейсов приложений в мире Unix-подобных операционных систем основан на стандарте POSIX. Формально стандарт POSIX состоит из набора стандартов IEEE[2], целью которого является создание переносимого стандарта операционной системы, примерно соответствующей ОС Unix. Разработчики системы Linux стремятся к соответствию стандартам POSIX и SUSv3 там, где это возможно.

---

[1] Для аппаратной платформы x86 существует около 335 системных функций (для каждой аппаратной платформы разрешается определять свои уникальные системные функции). Хотя не для всех операционных систем опубликованы полные спецификации системных функций, но даже по грубым оценкам в некоторых операционных системах таких функций более тысячи. Во время выхода в свет предыдущего издании этой книги для платформы x86 существовало только 250 системных функций.

[2] IEEE (Institute of Electrical and Electronics Engineers — Институт инженеров по электротехнике и электронике) — некоммерческая профессиональная ассоциация, действующая во многих технических областях и отвечающая за многие важные стандарты, такие как стандарт POSIX. За более подробной информацией обратитесь на сайт `http://www.ieee.org`.

Стандарт POSIX является хорошим примером взаимосвязи между интерфейсами API и системными функциями. В большинстве Unix-подобных ОС функции интерфейса API, определенные в стандарте POSIX, имеют очень тесную связь с системными функциями. Более того, стандарт POSIX создавался как раз для того, чтобы сделать те интерфейсы, которые предоставляли ранние версии ОС Unix, похожими между собой. С другой стороны, в некоторых операционных системах, никак не связанных с OS Unix, наподобие Microsoft Windows существуют библиотеки, совместимые со стандартом POSIX.

Частично интерфейс к системным функциям в операционной системе Linux, так же как и в большинстве Unix-систем, обеспечивается за счет библиотеки C. В ней реализован основной API-интерфейс системы Unix, включающий стандартную библиотеку функций языка C и интерфейс для вызова системных функций. Библиотека языка C используется во всех программах, написанных на языке C, а также, учитывая его особенности, может использоваться в программах, написанных на других языках. Кроме того, за счет библиотеки функций языка C обеспечивается также поддержка большей части стандарта POSIX API.

Для прикладного программиста системные функции не имеют особого значения, поскольку все, с чем он работает, — это интерфейс API. С другой стороны, в самом ядре могут вызываться только системные функции. Для ядра не имеет особого значения, какие функции вызываются из библиотеки API и какие системные функции используются в приложении. Тем не менее с точки зрения ядра все-таки важно отслеживать возможное использование системных функций и поддерживать их универсальность и гибкость.

Общий девиз для интерфейсов ОС Unix — это “предоставлять механизм, а не набор правил”. Другими словами, системные функции существуют для того, чтобы обеспечить определенную функциональность в абстрактном смысле этого слова. А то, каким образом используется эта функциональность, ядра не касается.

Системные функции

Системные функции, которые часто в среде Linux называются *syscalls* (от system calls), как правило, вызываются из функций, определенных в библиотеке C. Им может передаваться нуль, один или несколько аргументов (входных данных), в результате обработки которых *могут* возникать один или несколько побочных эффектов[3], как, например, запись данных в файл или копирование участка памяти по указанному адресу. Системные функции также возвращают значение типа `long`[4], свидетельствующее об успешном завершении работы функции либо о возникшей ошибке. Как правило, хотя и не всегда, отрицательное значение свидетельствует об ошибке, а нулевое значение — об успешном завершении. При возникновении ошибки функции библиотеки C записывают ее специальный код в глобальную переменную `errno`. Значение этого кода можно преобразовать в удобную для восприятия человеком форму с помощью одной из библиотечных функций, например `perror()`.

[3] Обратите внимание на слово *могут*. Несмотря на то что вызов почти всех системных функций приводит к тем или иным побочным эффектам (т.е. в результате каким-либо образом изменяется состояние системы), существует небольшое количество системный функций, как, например, `getpid()`, просто возвращающих некоторые данные ядра.

[4] Тип `long` используется для совместимости с 64-разрядными версиями Linux.

И наконец, системные функции имеют строго предопределенное поведение. Например, системная функция `getpid()` запрограммирована на возврат целого значения, содержащего `PID` текущего процесса. Реализация этой функции в ядре очень простая:

```
SYSCALL_DEFINE0(getpid)
{
   return task_tgid_vnr(current); // Возвращает значение current->tgid
}
```

Обратите внимание на то, что в определении ничего не говорится о способе реализации. Необходимые функциональные возможности системной функции должны быть предусмотрены в ядре, но они могут быть реализованы произвольным образом, главное, чтобы результат был правильный. Разумеется, рассматриваемая системная функция в действительности является такой же простой, как показано выше, и существует не так уж много различных вариантов для ее реализации[5].

`SYSCALL_DEFINE0` представляет собой обычный макрос, который определяет системную функцию без параметров, поскольку его имя заканчивается `0`. Получаемый в результате код приведен ниже.

```
asmlinkage long sys_getpid(void)
```

Давайте проанализируем это определение системной функции. Во-первых, обратите внимание на модификатор `asmlinkage` в объявлении функции. Эта директива говорит компилятору о том, что аргументы функции находятся только в стеке. Данный модификатор необходимо использовать для всех системных функций. Во-вторых, функция возвращает значение типа `long`. С целью совместимости 32- и 64-разрядных систем системные функции, возвращающие значение типа `int` в пространстве пользователя, возвращают значение типа `long` в пространстве ядра. В-третьих, обратите внимание на то, что системная функция `getpid()` определена в ядре как `sys_getpid()`. Это соглашение о присвоении имен используется для всех системных функций операционной системы Linux. Так, системная функция `bar()` должна быть реализована в ядре в виде функции `sys_bar()`.

Номера системных функций

В системе Linux каждой системной функции присвоен *специальный номер* (syscall number). Этот уникальный номер используется для обращения к нужной системной функции. Когда пользовательской программе нужно вызвать ту или иную системную функцию, она использует для этой цели соответствующий номер, а не имя системной функции.

Номер системной функции является важным атрибутом. После того как он будет назначен системной функции, его уже нельзя изменять, поскольку это нарушит работу уже скомпилированных прикладных программ. Более того, если системная функция будет удалена из ядра, то соответствующий номер нельзя использовать повторно для другой функции. Все дело в том, что тогда ранее скомпилированные прикладные программы “будут думать”, что вызывают одну функцию, а на самом деле будет вызываться другая

[5] Вам может быть непонятно, почему функция `getpid()` возвращает значение поля `tgid`, которое является идентификатором группы потоков (thread group ID)? Дело в том, что для обычных процессов значение параметра `TGID` равно значению параметра `PID`. При наличии нескольких потоков значение параметра `TGID` одинаково для всех потоков одной группы. Такая реализация дает возможность различным потокам вызывать функцию `getpid()` и получать одинаковое значение параметра `PID`.

функция. В системе Linux предусмотрена так называемая “не реализованная” функция `sys_ni_syscall()`, которая просто возвращает значение `-ENOSYS` — код ошибки, соответствующий некорректной системной функции. Эта функция служит для “затыкания дыр” в случае такого редкого события, как удаление системной функции, или любых других действий, в результате которых становится недоступной определенная функциональность.

Список всех зарегистрированных системных функций хранится в ядре в специальной таблице, адрес которой находится в переменной `sys_call_table`. Формат данной таблицы зависит от используемой аппаратной платформы. Для платформы x86-64 он определен в файле `arch/i386/kernel/syscall_64.c`. С помощью этой таблицы каждому уникальному номеру системной функции сопоставляется адрес реальной системной функции.

Быстродействие системных функций

В ОС Linux системные функции работают быстрее, чем во многих других операционных системах. Это отчасти связано с малым временем переключения контекста. Переход в режим ядра и выход из него выполняются очень просто и эффективно. Другой фактор — это простота самих системных функций и их механизма обработки.

Обработчик вызова системных функций

Код ядра не может непосредственно вызываться из пользовательских приложений. В них нельзя просто так вызвать функцию, которая находится в пространстве ядра, так как ядро расположено в защищенной области памяти. Если бы пользовательские программы могли непосредственно считывать и модифицировать данные, расположенные в адресном пространстве ядра, то всей безопасности и стабильности такой операционной системы была бы грош цена.

Поэтому пользовательские программы должны каким-то образом сигнализировать ядру о том, что им необходимо выполнить системную функцию. В результате ОС должна переключиться в режим ядра, где системная функция может быть вызвана из кода ядра, работающего от имени приложения.

Механизмом, с помощью которого ядру передается такой сигнал, является программное прерывание. При этом в процессоре возникает исключительная ситуация, система переключается в режим ядра и запускается обработчик прерывания. В рассматриваемом нами случае упомянутый обработчик прерывания на самом деле является обработчиком вызова системной функции. Для аппаратной платформы x86 для этой цели зарезервировано прерывание номер 128, которое вызывается с помощью ассемблерной команды `int $0x80`. В результате процессор переключается в привилегированный режим (т.е. входит в режим ядра) и запускает обработчик исключительной ситуации, определяемый вектором номер 128, который и является обработчиком вызова системной функции. Ему соответствует функция с подходящим названием — `system_call()`. Данная функция зависит от типа используемой аппаратной платформы. Для архитектуры x86-64 она реализована на языке ассемблера и определена в файле `entry_64.S`[6].

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

В современных процессорах x86 была введена специальная команда, называемая `sysenter`. Она позволяет ускорить процесс перехода в привилегированный режим и, как следствие, вызов системной функции, причем использует для этого более подходящий способ, а не команду вызова прерывания `int`. Поддержка этой новой команды процессора довольно быстро появилась в ядре Linux. Тем не менее, независимо от того, каким образом управление передается обработчику вызова системной функции, важно отметить, что для перехода в пространство ядра программа пользователя должна вызвать исключительную ситуацию или программное прерывание.

Как определить, какую системную функцию вызвать

Для вызова системной функции самого факта перехода в пространство ядра еще недостаточно, поскольку в ядре их существует огромное количество и все из них переходят в пространство ядра одним и тем же образом. Поэтому при переходе в режим ядра нужно каким-то образом передать ему номер вызываемой системной функции. Для аппаратной платформы x86 это выполняется через регистр процессора `rax` (в 64-разрядном режиме) и `eax` (в 32-разрядном режиме). Перед вызовом команды, с помощью которой выполняется переход в режим ядра, программа пользователя помещает в регистр `rax` число, соответствующее номеру системной функции, которую нужно вызвать. Затем значение регистра `rax` считывается в обработчике вызова системной функции. На других аппаратных платформах все происходит примерно так же.

В функции `system_call()` проверяется корректность переданного ей номера системной функции путем сравнения со значением переменной `NR_syscalls`. Если этот номер больше или равен `NR_syscalls`, возвращается значение `-ENOSYS`. В противном случае вызывается указанная системная функция, как показано ниже.

```
call *sys_call_table(,%rax,8)
```

Поскольку размер каждого элемента таблицы системных вызовов равен 64 битам (8 байтам), в коде ядра переданный в регистре номер системной функции умножается на 8, чтобы определить смещение, по которому содержится адрес нужной функции. На платформе x86-32 этот код остается таким же, только число 8 заменяется на 4, а регистр `rax` — на `eax`. Процесс вызова системной функции проиллюстрирован на рис. 5.2.

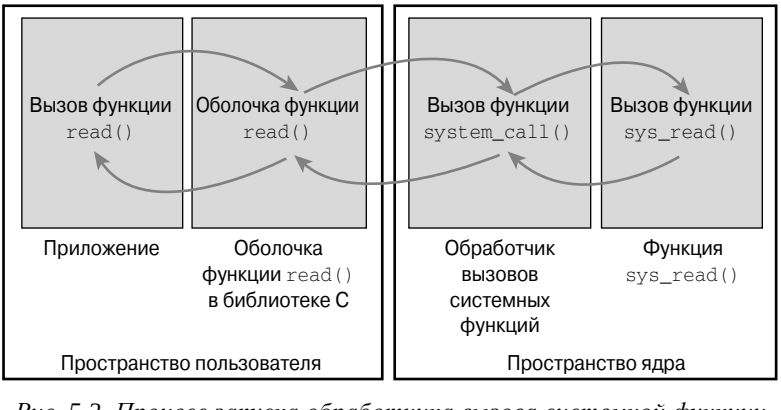


Рис. 5.2. Процесс запуска обработчика вызова системной функции и самой функции

Передача параметров

Кроме передачи ядру номера системной функции, для работы большинства системных функций требуется указать один или несколько параметров. Поэтому тем или иным способом программа пользователя должна передать ядру список параметров перед вызовом программного прерывания. Проще всего это сделать так же, как и при передаче номера системной функции, — загрузить параметры в регистры общего назначения процессора. На платформе x86-32 первые пять параметров загружаются в регистры `ebx`, `ecx`, `edx`, `esi` и `edi` соответственно. В маловероятном случае, когда параметров должно быть больше пяти, в один из регистров загружается адрес блока параметров, расположенных в пространстве пользователя.

Возвращаемое из системной функции значение также передается программе пользователя через регистр. Для аппаратной платформы x86 оно помещается в регистр `eax`.

Реализация системных функций

Фактическая реализация системной функции в ОС Linux никак не связана с алгоритмом работы обработчика ее вызова. Поэтому добавить новую системную функцию в ядро Linux не составляет большого труда. Основная часть работы связана с разработкой алгоритма и реализацией самой системной функции, а регистрация ее в ядре Linux очень проста. Рассмотрим последовательность действий, которые необходимо выполнить, чтобы написать новую системную функцию в операционной системе Linux.

Разработка системных функций

На первом шаге реализации системной функции необходимо четко определить ее назначение, т.е. что она должна делать. Каждая системная функция должна иметь только одно назначение. Многоцелевые системные функции (т.е. когда одна и та же функция выполняет несколько задач в зависимости от значения аргумента) в ОС Linux использовать не рекомендуется. В качестве примера того, как *не нужно* делать, достаточно проанализировать системную функцию `ioctl()`

Далее следует решить, какие аргументы нужно передать системной функции, какие значения и коды ошибок она должна возвращать. Системная функция должна иметь понятный и простой интерфейс, по возможности с минимальным количеством аргументов. Назначение и алгоритм ее работы очень важны и поэтому не должны изменяться, поскольку это может нарушить работу существующих прикладных программ. Старайтесь все продумать заранее, особенно те моменты, которые связаны с возможным изменением системной функции в будущем. Определите, можно ли добавить новые функциональные возможности в существующую системную функцию, или любое их изменение потребует введения совершенно новой функции. Оцените, насколько просто можно исправить ошибки, и не нарушится ли при этом совместимость с предыдущими версиями. С целью обратной совместимости во многих системных функциях предусмотрен специальный аргумент-флаг. Этот флаг используется вовсе не для того, чтобы сделать какую-либо системную функцию многоцелевой, поскольку, как отмечалось выше, такой подход не приветствуется. Он нужен для того, чтобы не вводить новую системную функцию, а просто добавить новые функциональные возможности и параметры к существующей функции и при этом не нарушить ее совместимость с предыдущими версиями.

Важным моментом является разработка интерфейса с прицелом на будущее. Проанализируйте, не ограничили ли вы возможности функции без крайней на то необходимости.

Разрабатываемая вами системная функция должна быть максимально универсальной. Не стоит считать, что она будет использоваться завтра так же, как сегодня. Учтите, что *назначение* системной функции меняться не должно, а ее *использование* — может. Является ли ваша функция переносимой? При ее разработке вы не должны делать каких-либо допущений по поводу разрядности машинного слова или порядка следования байтов. Эти вопросы подробнее обсуждаются в главе 19, "Переносимость". Убедитесь, что никакие ваши неверные допущения не нарушат работу системной функции в будущем, и не забывайте девиз Unix: "Обеспечивать механизм, а не стратегию".

При разработке системной функции не забывайте, что переносимость и устойчивость понадобятся не только сегодня, но и в будущем. Основные системные функции ОС Unix выдержали это испытание временем. Большинство из них так же полезно и применимо сегодня, как и тридцать лет назад!

Проверка параметров

Нужно тщательно проверять все параметры системной функции и убедиться в том, что их значения корректны и законны. Поскольку системные функции выполняются в пространстве ядра, передача им некорректных значений может нарушить стабильность и безопасность работы системы в целом.

Например, в системных функциях, выполняющих операции ввода-вывода данных, обязательно нужно проверить корректность значения файлового дескриптора. В функциях, связанных с управлением процессами, нужно проверить, является ли значение переданного идентификатора `PID` допустимым. Следует проверить каждый параметр не только на предмет допустимости и законности, но и правильности его значения. Прикладные программы не должны посылать ядру запрос на доступ к системным ресурсам, к которым они сами такого доступа не имеют.

Самой важной является проверка корректности указателей, переданных прикладной программой. Представьте себе, что произойдет, если в ядро беспрепятственно будет передан указатель на область памяти, к которой прикладная программа не имеет доступа по чтению! В результате обманным путем она может заставить ядро скопировать данные из этого участка памяти, который принадлежит другому процессу или вовсе не подлежит чтению. Перед использованием указателя, переданного в ядро прикладной программой, ОС должна убедиться в том, что он

- указывает на область памяти, относящуюся к пространству пользователя (пользовательские процессы не должны обманным путем заставлять ядро читать данные, к которым они не имеют доступа);
- указывает на область памяти, принадлежащую адресному пространству процесса (нельзя позволять, чтобы процесс заставил ядро читать данные других процессов);
- не нарушает ограничений (чтение, запись, выполнение), связанных с доступом к указанному участку памяти.

В ядре предусмотрены две функции для выполнения необходимых проверок при копировании данных в пространство пользователя и из него. Не забывайте, что ядро никогда не должно слепо доверять указателю, переданному из прикладной программы! Поэтому одна из перечисленных ниже двух функций должна использоваться всегда.

Для записи данных в пространство пользователя предусмотрена функция `copy_to_user()`. Ей передаются три параметра. Первый — это адрес участка памяти в адресном пространстве процесса, куда нужно выполнить копирование. Второй — указатель на область памяти, находящуюся в пространстве ядра, откуда нужно выполнить копирование. И третий параметр — длина в байтах участка памяти, подлежащего копированию.

Для чтения данных из пространства пользователя используется аналогичная функция `copy_from_user()`. Эта функция копирует данные из области памяти, адрес которой определяется во втором параметре, а длина — в третьем, в область памяти, адрес которой указан в первом параметре.

В случае ошибки обе эти функции возвращают количество байтов, которые они не смогли скопировать. При успешном выполнении операции возвращается нуль. В случае возникновения подобной ошибки ваша системная функция должна отреагировать стандартным образом — возвратить значение `-EFAULT`.

В качестве примера рассмотрим системную функцию, в которой используются обе рассмотренные выше функции: `copy_from_user()` и `copy_to_user()`. Наша функция `silly_copy()` абсолютно бесполезна, поскольку она копирует данные, определяемые первым параметром, в область памяти, определяемую вторым параметром. Используемый нами подход крайне неэффективен, поскольку при этом выполняется никому не нужное промежуточное копирование данных в пространство ядра. Но зато это позволяет проиллюстрировать суть дела.

```
/*
 * silly_copy - крайне бесполезная системная функция,
 * которая копирует 'len' байтов из области памяти,
 * на которую указывает параметр 'src', в область памяти,
 * на которую указывает параметр 'dst', с промежуточным их копированием в
ядро.
 * Предназначена только для иллюстрации процесса копирования
 * данных в пространство ядра и из него.
 */

SYSCALL_DEFINE3(silly_copy,
                unsigned long *, src,
                unsigned long *, dst,
                unsigned long len)
{
   unsigned long buf;

   /* Скопируем участок памяти из 'src', лежащий
    * в пространстве пользователя в 'buf'
    */
   if (copy_from_user(&buf, src, len))
      return -EFAULT;

   /* Скопируем участок памяти из 'buf' в 'dst',
    *  который принадлежит пространству пользователя
    */
   if (copy_to_user(dst, &buf, len))
      return -EFAULT;

   /* Возвратим размер скопированных данных */
   return len;
}
```

В процессе работы обеих функций, `copy_to_user()` и `copy_from_user()`, выполнение процесса может быть приостановлено. Так происходит, например, если какая-либо

из страниц, содержащая пользовательские данные, отсутствует в физической памяти компьютера, поскольку ранее была вытеснена на диск. В таком случае процесс переводится в состояние ожидания до тех пор, пока обработчик исключительной ситуации, вызванной отсутствием страницы, не загрузит ее в память из файла подкачки.

И последняя из возможных проверок — это проверка на соответствие правам доступа. В старых версиях ядра Linux в системных функциях, которым требовались права пользователя `root`, стандартно использовалась функция `suser()`. Эта функция просто проверяла, запущен ли процесс от имени пользователя `root`. Сейчас эту функцию убрали и заменили системой тонкой проверки "возможности" использования ресурса. Новая система позволяет проверить 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 функции `capable(CAP_SYS_NICE)` проверяет, имеет ли вызывающий процесс право изменения значения параметра `nice` других процессов. По умолчанию суперпользователь владеет всеми правами, а пользователь, не являющийся пользователем `root`, не имеет никаких дополнительных прав. В качестве примера ниже приведен исходный код системной функции `reboot()`. Обратите внимание на то, как в самом начале функции выполняется проверка на то, что вызывающий процесс имеет право перезагружать систему (`CAP_SYS_REBOOT`). Если удалить тот единственный условный оператор, то любой процесс сможет перезагрузить систему.

```
SYSCALL_DEFINE4(reboot,
                  int, magic1,
                  int, magic2,
          unsigned int, cmd,
         void __user *, arg)
{
   char buffer[256];

   /* Систему может перегружать только суперпользователь. */
   if (!capable(CAP_SYS_BOOT))
          return -EPERM;

   /* Для усиления безопасности потребуем ряд магических аргументов. */
   if (magic1 != LINUX_REBOOT_MAGIC1  ||
      (magic2 != LINUX_REBOOT_MAGIC2  &&
       magic2 != LINUX_REBOOT_MAGIC2A &&
       magic2 != LINUX_REBOOT_MAGIC2B &&
       magic2 != LINUX_REBOOT_MAGIC2C))
          return -EINVAL;

   /* Если флаг 'pm_power_off' не установлен и поступила команда
    * 'power_off', заменим ее на 'halt'.
    */
   if ((cmd == LINUX_REBOOT_CMD_POWER_OFF) && !pm_power_off)
        cmd = LINUX_REBOOT_CMD_HALT;

   lock_kernel();

   switch (cmd) {
      case LINUX_REBOOT_CMD_RESTART:
         kernel_restart(NULL);
         break;

      case LINUX_REBOOT_CMD_CAD_ON:
         C_A_D = 1;
         break;
```

```
        case LINUX_REBOOT_CMD_CAD_OFF:
            C_A_D = 0;
            break;

        case LINUX_REBOOT_CMD_HALT:
            kernel_halt();
            unlock_kernel();
            do_exit(0);
            break;

        case LINUX_REBOOT_CMD_POWER_OFF:
            kernel_power_off();
            unlock_kernel();
            do_exit(0);
            break;

        case LINUX_REBOOT_CMD_RESTART2:
            if (strncpy_from_user(&buffer[0], arg, sizeof(buffer) - 1) < 0) {
                unlock_kernel();
                return -EFAULT;
            }
            buffer[sizeof(buffer) - 1] = '\0';

            kernel_restart(buffer);
            break;

        default:
            unlock_kernel();
            return -EINVAL;
    }
    unlock_kernel();
    return 0;
}
```

Полный список констант для проверки возможности доступа к ресурсам и требуемые при этом права доступа определены в файле `<linux/capability.h>`.

Контекст системной функции

Как уже упоминалось в главе 3, “Управление процессами”, при выполнении системной функции ядро работает в контексте процесса. При этом в переменной `current` хранится указатель на текущую задачу, которая и является процессом, вызвавшим системную функцию.

В контексте процесса ядро может переходить в состояние ожидания (например, если при вызове системной функции выполнение процесса будет приостановлено или будет явно вызвана функция `schedule()`), а также является полностью вытесняемым. Эти два момента очень важны. Способность перехода в состояние ожидания означает, что в системной функции можно использовать бо́льшую часть функциональных возможностей ядра. Как будет показано в главе 7, “Прерывания и их обработка”, благодаря этому существенно упрощается программирование ядра[7]. Тот факт, что контекст процесса является вытесняемым, означает, что, как и программа пользователя, текущая задача, выполняющая системную функцию, может быть вытеснена другой задачей. Поскольку в новой задаче

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

также может выполняться та же самая системная функция, необходимо тщательно проверить, чтобы системные функции были реентерабельными (повторновходимыми). По сути, данная проблема возникает также и при симметричной мультипроцессорной обработке. Методы синхронизации, обеспечивающие реентерабельность, рассмотрены в главах 9, “Общие сведения о синхронизации кода ядра”, и 10, “Средства синхронизации ядра”.

После возврата из системной функции управление вновь попадает в функцию `system_call()`, которая в конечном итоге переключается в пространство пользователя, и далее выполнение пользовательского процесса продолжается.

Завершающие этапы регистрации системной функции

После того как системная функция будет создана, процедура ее регистрации в качестве официальной системной функции очень проста и состоит в следующем.

1. Добавляется новый элемент в конец таблицы системных функций. Это необходимо сделать для всех аппаратных платформ, на которых поддерживается данная функция (большинство системных функций работает на всех возможных аппаратных платформах). Положение этого элемента в таблице соответствует номеру системной функции, начиная с нуля. Например, десятый элемент таблицы соответствует системной функции номер девять.
2. Для всех поддерживаемых аппаратных платформ номер системной функции должен быть определен в файле `<asm/unistd.h>`.
3. Скомпилируйте системную функцию и поместите ее в образ ядра, а не в самостоятельный модуль. В самом простом случае достаточно поместить исходный код системной функции в один из подходящих файлов, например `sys.c`, находящихся в каталоге `kernel/`. В этом каталоге размещаются файлы с исходным кодом многих системных функций.

Давайте более детально рассмотрим перечисленные выше этапы на примере вымышленной системной функции `foo()`. Прежде всего функция `sys_foo()` должна быть добавлена в таблицу системных функций. Для большинства аппаратных платформ эта таблица находится в файле `entry.S` и выглядит примерно так, как показано ниже.

```
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_eventfd2
        .long sys_epoll_create1
        .long sys_dup3                /* 330 */
        .long sys_pipe2
        .long sys_inotify_init1
        .long sys_preadv
        .long sys_pwritev
        .long sys_rt_tgsigqueueinfo   /* 335 */
        .long sys_perf_event_open
        .long sys_recvmmsg
```

Нам нужно добавить новую системную функцию в конец этого списка:

```
.long sys_foo
```

Нашей системной функции будет назначен следующий по порядку номер (в данном случае 338), хотя это и не указано явно. Для каждой аппаратной платформы, которую мы планируем поддерживать, нужно добавить эту функцию в таблицу системных функций соответствующей аппаратной платформы. При этом номер нашей системной функции для разных аппаратных платформ может отличаться, поскольку он является частью двоичного интерфейса приложений (Application Binary Interface, или ABI), который уникален для каждой аппаратной платформы. Как правило, новая системная функция должна быть доступной для всех аппаратных платформ. Обратите внимание на то, что по договоренности номера системных функций указываются в комментариях через каждые пять записей. Это позволяет быстро найти, какой номер какой системной функции соответствует.

Далее необходимо добавить номер системной функции в заголовочный файл `asm/unistd.h`, который на момент написания книги выглядел примерно так, как показано ниже.

```
/*
* В этом файле определены номера системных функций.
*/

#define __NR_restart_syscall     0
#define __NR_exit                1
#define __NR_fork                2
#define __NR_read                3
#define __NR_write               4
#define __NR_open                5

...

#define __NR_signalfd4         327
#define __NR_eventfd2          328
#define __NR_epoll_create1     329
#define __NR_dup3              330
#define __NR_pipe2             331
#define __NR_inotify_init1     332
#define __NR_preadv            333
#define __NR_pwritev           334
#define __NR_rt_tgsigqueueinfo 335
#define __NR_perf_event_open   336
#define __NR_recvmmsg          337
```

В конец этого списка мы должны добавить следующее:

```
#define __NR_foo               338
```

И в завершение необходимо реализовать саму системную функцию `foo()`. Поскольку для всех вариантов конфигурации наша функция должна быть скомпилирована и помещена в образ ядра, мы определим ее в файле `kernel/sys.c`. Код функции вы должны поместить в наиболее подходящий файл. Например, если системная функция относится к планированию выполнения процессов, то ее код нужно разместить в файле `kernel/sched.c`.

```
#include <asm/page.h>

/*
 * sys_foo - системная функция, доступная для всех платформ.
 *
```

```
 * Возвращает размер стека ядра для процесса.
 */
asmlinkage long sys_foo(void)
{
   return THREAD_SIZE;
}
```

Это все! Загрузите новое ядро. Теперь из пользовательской программы можно вызвать системную функцию `foo()`.

Доступ к системным функциям из пользовательских приложений

Как правило, поддержка системных функций выполняется через библиотеку языка C. При этом для использования системной функции достаточно загрузить в пользовательском приложении соответствующий стандартный заголовочный файл, в котором определены прототипы функции, и скомпоновать программу с библиотекой языка C или другой библиотечной функцией, в которой вызывается ваша системная функция. Однако если вы только что написали новую системную функцию, то, разумеется, она еще не будет поддерживаться в библиотеке `libc`!

К счастью, в ОС Linux предусмотрен набор макросов-оболочек для доступа к системным функциям. Они загружают параметры функции в регистры и вызывают программное прерывание. Эти макросы имеют имя `_syscall`*n*`()`, где *n* — число от 0 до 6, соответствующее количеству параметров вызываемой системной функции. Дело в том, что макросу-оболочке нужно точно знать, сколько параметров передается функции, чтобы загрузить их в соответствующие регистры процессора. В качестве примера рассмотрим системную функцию `open()`, которая определена так, как показано ниже.

```
long open(const char *filename, int flags, int mode)
```

Макрос для вызова этой системной функции без явного обращения к библиотеке будет выглядеть так:

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

После этого в приложении можно просто вызывать функцию `open()`.

Каждый из упомянутых выше макросов-оболочек имеет $2+2\times n$ параметров. Первый параметр соответствует типу возвращаемого значения из системной функции. Во втором параметре задается имя системной функции. Далее указываются тип и имя каждого параметра в том же порядке, что и у системной функции. Константа `__NR_open`, которая определена в файле `<asm/unistd.h>`, определяет номер системной функции. В результате расширения макроса `_syscall3` получается функция на языке C, содержащая вставки на языке ассемблера. В ассемблерном коде и выполняются все описанные в предыдущем разделе действия по вызову системной функции. Они заключаются в загрузке в регистры общего назначения номера системной функции и ее параметров, а также вызова программного прерывания для входа в ядро. Таким образом, чтобы вызвать системную функцию `open()`, достаточно поместить в код приложения приведенный выше макрос.

Давайте напишем макрос, который позволит вызвать нашу замечательную новую системную функцию `foo()` и соответствующий код, с помощью которого ее можно протестировать.

```
#define __NR_foo 338
__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 функции

В предыдущих разделах было показано, что новую системную функцию создать очень легко. Однако данный факт не должен поощрять вас к этим действиям. Безусловно, при создании новых системных функций вы должны проявлять сдержанность и осмотрительность. Вместо создания новой функции часто существуют более удачные альтернативы. Рассмотрим некоторые преимущества и недостатки, а также возможные альтернативные варианты.

При создании нового интерфейса в виде системной функции мы получаем следующие преимущества.

- Системные функции очень просто реализовать и легко использовать.
- В ОС Linux системная функция вызывается очень быстро.

При этом возникают следующие недостатки.

- Необходимо получить номер системной функции, который должен быть официально вам назначен.
- После того как ваша системная функция окажется в стабильной серии ядра, ее код превращается в высеченный на камне манускрипт. Ее интерфейс уже нельзя будет изменить, иначе перестанут работать пользовательские приложения, в которых эта функция используется.
- Для каждой аппаратной платформы необходимо отдельно зарегистрировать системную функцию и осуществлять поддержку ее кода.
- Системную функцию не так то просто вызвать из сценариев командной оболочки, кроме того, ее нельзя напрямую вызвать через файловую систему.
- Поскольку для системной функции нужно назначить номер, за пределами основного дерева ядра ее затруднительно поддерживать и использовать.
- Использование системной функции для простого обмена информацией это сродни стрельбы из пушки по воробьям.

Ниже перечислены возможные альтернативы использованию системной функции.

- Реализовать файл устройства (device node) и использовать функции `read()` и `write()` для обмена данными с этим устройством. Для манипуляции специфическими параметрами или для получения специфической информации от этого устройства можно использовать функцию `ioctl()`.
- Некоторые интерфейсы, например семафоры, можно представить в виде дескрипторов файлов. Управлять ими также можно по аналогии с файлами.
- Добавить информацию в виде файла в соответствующем месте файловой системы `sysfs`.

Для большого числа интерфейсов системные функции — это *правильный* выбор, хотя в ОС Linux пытаются избегать простого добавления системной функции для поддержки каждой новой, вдруг появляющейся абстракции. В результате получился удивительно четкий уровень системных функций, 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). Столь низкая частота появления новых системных функций как раз свидетельствует о том, что Linux стала относительно стабильной операционной системой с полным набором функций.

Резюме

В этой главе было рассмотрено, что такое системные функции и чем они отличаются от библиотечных функций и интерфейса прикладных программ (API). Затем мы рассмотрели процесс реализации системной функции в ядре и цепочку событий, которая происходит при ее вызове: переход в ядро, передача номера функции и всех необходимых параметров соответствующей функции ядра и ее выполнение, возврат результатов работы в пользовательское приложение.

Далее рассказывалось, как добавить новую системную функцию в ядро, и был приведен простой пример вызова системной функции из пользовательского приложения. Весь процесс является достаточно простым! Поскольку добавить новую системную функцию довольно просто, основная работа заключается в ее реализации. В оставшейся части книги рассмотрены основные принципы, а также интерфейсы, которые необходимо использовать при создании хорошо работающих, оптимальных и безопасных системных функций.

В конце главы были рассмотрены некоторые преимущества и недостатки, возникающие при реализации системных функций, и представлен краткий список возможных вариантов, позволяющих избежать добавления новых системных функций.

6
Структуры данных ядра

В этой главе мы ознакомимся с некоторыми внутренними структурами данных, используемыми в исходном коде ядра Linux. Как и в любом другом большом программном проекте, в ядре Linux предусмотрены обобщенные структуры и базовые типы данных, способствующие повторному использованию кода. Разработчики ядра обязаны там, где только возможно, использовать эти структуры данных и не изобретать собственных решений. В последующих разделах этой главы мы изучим перечисленные ниже часто используемые обобщенные структуры данных.

- Связанные списки
- Очереди
- Таблицы отображения
- Двоичные деревья

И завершится глава рассмотрением вопросов оценки сложности алгоритмов — мерой, которая поможет нам решить, какие из алгоритмов и структур данных хорошо приспособлены для работы с огромными массивами входных данных.

Связанные списки

Связанный список — это самая простая и чаще всего используемая структура данных в ядре Linux. Она позволяет сохранять переменное число *элементов*, называемых *узлами* (nodes) списка, и манипулировать ими. В отличие от элементов статического массива, элементы связанного списка можно динамически создавать и помещать их в сам список. Это позволяет на этапе выполнения программы обрабатывать произвольное количество элементов, которые еще не существовали во время компиляции программы, и точное их число было неизвестно. Поскольку элементы создаются в различные периоды времени выполнения программы, они необязательно должны занимать соседние участки памяти компьютера. По этой причине элементы нужно както *связать* друг с другом, т.е. в каждом элементе списка должен храниться указатель на *следующий* элемент. Во время добавления или удаления элементов списка нужно просто скорректировать указатель на следующий узел.

Однонаправленный и двунаправленный связанный список

В простейшем случае структура данных, представляющая связанный список, может выглядеть так, как показано ниже. На рис. 6.1 показано, как представляется такой список в памяти компьютера.

```
/* Элемент связанного списка */
struct list_element {
       void *data;                 /* Данные                            */
       struct list_element *next; /* Указатель на следующий элемент */
};
```

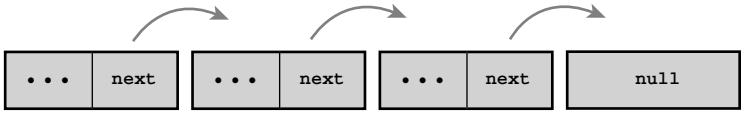


Рис. 6.1. Однонаправленный связанный список

В элементах некоторых связанных списков также хранится указатель на *предыдущий* элемент. Такие списки называются *двунаправленными*, поскольку их элементы связаны как в прямом, так и в обратном направлении. Связанные списки наподобие того, что показан на рис. 6.1, называются *однонаправленными*, поскольку в их элементах отсутствует указатель на предыдущий элемент.

Структура данных, представляющая двунаправленный связанный список, может выглядеть так, как показано ниже. На рис. 6.2 показано, как представляется такой список в памяти компьютера.

```
/* Элемент двунаправленного связанного списка */
struct list_element {
       void *data;                 /* Данные                            */
       struct list_element *next; /* Указатель на следующий элемент  */
       struct list_element *prev; /* Указатель на предыдущий элемент */
};
```

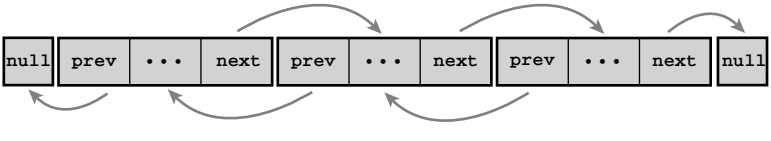


Рис. 6.2. Двунаправленный связанный список

Циклически связанные списки

Обычно из-за того, что последний элемент связанного списка не имеет следующего элемента, в его поле `next` помещается специальное значение, такое как `NULL`, свидетельствующее о том, что это последний элемент в списке. В некоторых типах связанных списков в поле `next` последнего элемента *не* помещается никакое специальное значение, а просто указывается адрес первого элемента списка. Такие списки называются *циклически связанными*, поскольку их элементы связаны по кругу. Циклически связанными могут быть как одно-, так и двунаправленные списки. В последнем случае в поле `prev` первого элемента списка хранится адрес его последнего элемента. На рис. 6.3 и 6.4 показано, как представляются в памяти циклически связанные одно- и двунаправленные списки.

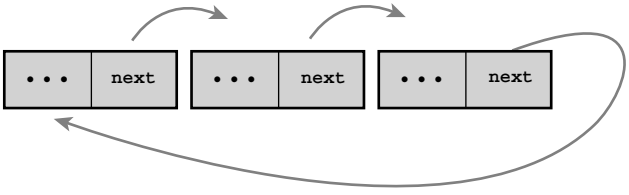


Рис. 6.3. Циклически связанный однонаправленный список

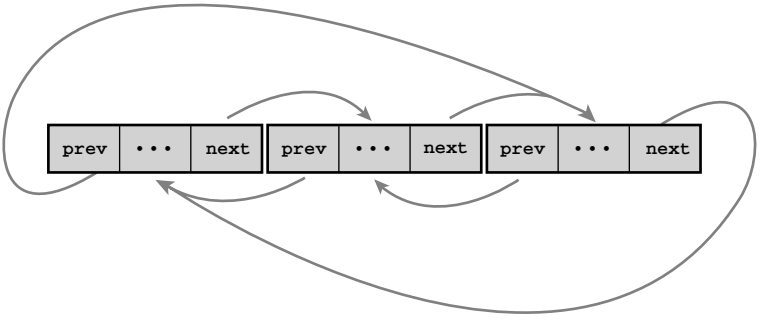


Рис. 6.4. Циклически связанный двунаправленный список

Хотя реализацию связанных списков в ядре Linux можно назвать уникальной, по сути, это не что иное, как *циклически связанные двунаправленные списки*. Использование этого типа связанных списков позволяет достичь большей гибкости.

Перемещение по элементам связанного списка

Перемещение по элементам связанного списка выполняется последовательно, т.е. линейно. Другими словами, после обработки текущего элемента списка нужно прочитать указатель на следующий элемент списка и перейти к нему. И так далее и тому подобное. Это самый простой метод перемещения по элементам связанного списка, к тому же ради него связанные списки, собственно, и были придуманы. Следует отметить, что связанные списки непригодны в случае, когда во главу угла ставится произвольный доступ к данным. Однако связанные списки как нельзя лучше подходят для случаев, когда нужно последовательно перебрать один элемент за другим, а также динамически добавлять и удалять элементы списка.

При реализации связанных списков в специальной переменной сохраняется указатель на первый элемент списка, называемый *головным* (head), или головой списка. Это позволяет быстро получить доступ к началу списка. В нециклических связанных списках их последний элемент четко обозначен, поскольку в его указателе на следующий элемент содержится значение `NULL`. В циклически связанных списках последний элемент обозначен нечетко, поскольку в нем содержится указатель на первый элемент списка. По этой причине обход элементов списка выполняется последовательно, от первого элемента к последнему. В двунаправленных связанных списках возможен также последовательный перебор элементов в обратном порядке — от последнего к первому. Разумеется, в таких списках можно также перемещаться на произвольное количество элементов вперед и назад относительно текущего элемента списка. При этом не требуется перебирать все элементы списка.

Реализация в ядре Linux

По сравнению с реализацией большинства связанных списков, включая тот обобщенный подход, который описан в предыдущем разделе, их реализация в ядре Linux в чем-то уникальна. Выше мы уже говорили о том, что данные (или группа данных, например структуры) можно упорядочить в виде связанного списка, если разместить в них указатели на *следующий* (а возможно, и на *предыдущий*) элемент данных. Например, предположим, что у нас есть структура `fox`[1], описывающая представителя семейства псовых, как показано ниже.

```
struct fox {
     unsigned long tail_length;  /* Высота в см в холке     */
     unsigned long weight;       /* Масса в кг              */
     bool          is_fantastic; /* Это сказочное существо? */
};
```

Для того чтобы преобразовать эту структуру в элемент связанного списка, чаще всего к ней добавляют указатели на следующий и предыдущий элементы, как показано ниже.

```
struct fox {
     unsigned long tail_length;  /* Высота в см в холке             */
     unsigned long weight;       /* Масса в кг                      */
     bool          is_fantastic; /* Это сказочное существо?         */
     struct   fox  *next;        /* Указатель на следующий элемент  *
                                  *  типа fox в списке              */
     struct   fox  *prev;        /* Указатель на предыдущий элемент *
                                  *  типа fox в списке              */
};
```

Однако в ядре Linux применен совершенно иной подход. Вместо того чтобы превращать структуру в элемент связанного списка, в Linux элемент связанного списка (точнее, структура, которая его описывает) *встраивается в саму структуру данных!*

Структура, описывающая элемент связанного списка

В старые добрые времена в ядре Linux существовало несколько реализаций связанных списков. Однако при реализации единой и универсальной концепции связанных списков из ядра был удален дублирующий код. Официальная реализация связанных списков появилась в разрабатываемой серии ядер 2.1. В результате теперь во всех случаях, где нужно использовать связанные списки, применяется только их официальная реализация. Не стоит изобретать велосипед!

Код, описывающий связанный список, находится в заголовочном файле `<linux/list.h>`. Его структура очень проста, как показано ниже.

```
struct list_head {
            struct list_head *next
            struct list_head *prev;
};
```

В переменной `next` хранится указатель на следующий узел списка, а в переменной `prev` — на предыдущий. Очевидно, пока что пользы от этого очень мало. А как же описать огромные связанные списки, в узлах которых содержатся другие связанные списки? Ответ на этот вопрос кроется в том, *как* следует использовать структуру `list_head`.

```
struct fox {
     unsigned long  tail_length;  /* Высота в см в холке        */
```

[1] В переводе с англ. *лисица*.

```
    unsigned long  weight;       /* Масса в кг                */
    bool           is_fantastic; /* Это сказочное существо?   */
    struct list_head list;          /* Список всех структур типа fox */
};
```

При таком подходе в переменной `list.next` структуры `fox` будет находиться указатель на следующий элемент списка, а в переменной `list.prev` — на предыдущий. Теперь наша реализация стала не только лучше, но и полезней. В ядре предусмотрено семейство функций для работы со связанными списками. Например, для добавления нового элемента в существующий связанный список в ядре используется функция `list_add()`. Однако эти функции являются обобщенными, поскольку они работают только со структурами типа `list_head`. Воспользовавшись макросом `container_of()`, мы легко можем найти родительскую структуру, содержащую нужную нам переменную-член. Все дело в том, что в языке C смещение нужной нам переменной относительно начала структуры фиксируется согласно спецификации двоичного интерфейса приложений (ABI) на этапе компиляции.

```
#define container_of(ptr, type, member) ({ \
        const typeof( ((type *)0)->member ) *__mptr = (ptr); \
        (type *)( (char *)__mptr - offsetof(type,member) );})
```

С помощью макроса `container_of()` мы можем определить простую функцию, возвращающую указатель на родительскую структуру, содержащую структуру `list_head`, как показано ниже.

```
#define list_entry(ptr, type, member) \
        container_of(ptr, type, member)
```

Взяв на вооружение функцию `list_entry()`, можно создать в ядре функции для работы со связанными списками, причем во всех из них нам не нужно будет ничего знать о родительских структурах, в которых, собственно, и содержится структура `list_head`.

Определение связанного списка

Как уже отмечалось выше, сама по себе структура `list_head` не представляет никакой ценности, так как обычно она встраивается в другую, определяемую пользователем структуру, как показано ниже.

```
struct fox {
    unsigned long  tail_length;  /* Высота в см в холке       */
    unsigned long  weight;       /* Масса в кг                */
    bool           is_fantastic; /* Это сказочное существо?   */
    struct list_head list;          /* Список всех структур типа fox */
};
```

Перед использованием связанного списка его нужно проинициализировать. Но поскольку большинство его элементов создается динамически (ведь для этого такие списки, собственно, и нужны!), то удобнее всего выполнить инициализацию связанного списка на этапе выполнения программы, как показано ниже.

```
struct fox *red_fox;

red_fox = kmalloc(sizeof(*red_fox), GFP_KERNEL);
red_fox->tail_length = 40;
red_fox->weight = 6;
red_fox->is_fantastic = false;
INIT_LIST_HEAD(&red_fox->list);
```

Если же структура создается статически на этапе компиляции и у вас есть прямая ссылка на нее, то для ее инициализации достаточно будет приведенного ниже кода.

```
struct fox red_fox = {
          .tail_length = 40,
          .weight = 6,
          .list = LIST_HEAD_INIT(red_fox.list),
};
```

Структуры типа `list_head`

В предыдущем разделе было показано, насколько легко можно преобразовать существующую структуру (например, такую, как `fox`) в элемент связанного списка. После внесения простого изменения в код элементы нашей структуры могут обрабатываться с помощью функций ядра, предназначенных для работы со связанными списками. Однако, перед тем как использовать эти функции, нам необходимо создать канонический указатель, с помощью которого можно будет ссылаться на весь список, — т.е. указатель на *головной* элемент.

Одной из положительных сторон реализации связанных списков в ядре является то, что элементы типа `fox` нашего связанного списка становятся неразличимыми. В самом деле, в каждом элементе нашего списка содержится структура `list_head`, поэтому мы можем свободно переходить от одного элемента к другому, пока не будут перебраны все узлы списка. Такой подход достаточно элегантен, но при этом, как правило, требуется отдельный указатель, ссылающийся на весь список, который сам по себе не является элементом этого списка. Любопытно, но по сути этот указатель представляет собой обычный элемент списка типа `list_head`.

```
static LIST_HEAD(fox_list);
```

В приведенном выше фрагменте кода определяется и инициализируется переменная `fox_list` типа `list_head`. Большинству функций, работающих со связанными списками, передается один или два параметра: указатель на головной элемент или (плюс к этому) указатель на элемент списка. Рассмотрим эти функции подробнее.

Работа со связанными списками

Как мы уже говорили, в ядре предусмотрено семейство функций, предназначенных для работы со связанными списками, и каждой из них передается один или два указателя на структуры типа `list_head`. Эти функции реализованы в виде *встраиваемых* (inline) функций на базовом языке C и описаны в файле `<linux/list.h>`.

Любопытно, что все эти функции принадлежат множеству `O(1)`[2]. А это, в свою очередь, означает, что время выполнения всех функций постоянно, независимо от размеров списка или массива исходных данных. Например, независимо от того, сколько элементов находится в списке (3 или 3000), вставка элемента в список или удаление элемента из него будет выполняться за одно и то же время. Возможно, данный факт не будет для вас неожиданностью, но его нужно знать.

Добавление элемента в связанный список

Для добавления узла в связанный список используется функция `list_add()`.

```
list_add(struct list_head *new, struct list_head *head)
```

[2] О том, что это такое, вы узнаете ниже, в разделе “Алгоритмическая сложность”.

Эта функция добавляет новый элемент, определяемый параметром `new`, в существующий список сразу *после* элемента, определяемого параметром `head`. Поскольку список является циклическим, в нем не представляется возможным определить, какой из элементов является *первым*, а какой *последним*. Поэтому в качестве параметра `head` можете передать ссылку на любой нужный элемент списка. Но если вы укажете ссылку на воображаемый "последний" элемент списка, то с помощью данной функции сможете реализовать стек.

Возвращаясь к нашему примеру со структурой `fox`, предположим, что у нас есть новый элемент типа `fox`, который мы хотим добавить в список `fox_list`. Как это сделать, показано ниже.

```
list_add(&f->list, &fox_list);
```

Для добавления элемента в конец связанного списка используется функция `list_add_tail()`.

```
list_add_tail(struct list_head *new, struct list_head *head)
```

Эта функция добавляет новый элемент, определяемый параметром `new`, в существующий список непосредственно *перед* элементом, определяемым параметром `head`. Так же как и в случае с функцией `list_add()`, в качестве параметра `head` можете указать ссылку на *любой* элемент, поскольку списки являются циклическими. С помощью функции `list_add_tail()` можно реализовать очередь, если передать ей ссылку на воображаемый "первый" элемент списка.

Удаление элемента из связанного списка

После добавления узла в связанный список второй по важности операцией является удаление элемента из списка. Для этого используется функция `list_del()`.

```
list_del(struct list_head *entry)
```

Данная функция удаляет элемент, определяемый параметром `entry`, из списка. Обратите внимание на то, что при этом память, принадлежащая элементу `entry` или той структуре данных, в которой он расположен, не освобождается. Эта функция просто удаляет указанный элемент из списка, и все. После вызова функции `list_del()`, как правило, нужно аннулировать удаленную из списка структуру данных и элемент `list_head`, расположенный в ней.

Например, чтобы удалить элемент типа `fox`, который мы добавили к списку `fox_list` в предыдущем разделе, используется следующий фрагмент кода:

```
list_del(&f->list);
```

Заметьте, мы не передаем функции указатель на сам список `fox_list`. Мы передаем ей только указатель на удаляемый элемент, а функция просто корректирует указатели на этот элемент, расположенные в предыдущем и последующем элементах, и таким образом удаляет его из списка. Реализация этой функции говорит сама за себя.

```
static inline void __list_del(struct list_head *prev,
                              struct list_head *next)
{
        next->prev = prev;
        prev->next = next;
}

static inline void list_del(struct list_head *entry)
{
        __list_del(entry->prev, entry->next);
}
```

Для удаления элемента из списка и его повторной инициализации в ядре предусмотрена функция `list_del_init()`.

```
list_del_init(struct list_head *entry)
```

Эта функция работает точно так же, как и `list_del()`, за исключением того, что она выполняет повторную инициализацию элемента `list_head` специальным значением, свидетельствующим о том, что он более не является узлом списка. При этом вы можете продолжать использовать данные самой структуры.

Перемещение и соединение узлов связанных списков

Для перемещения узла из одного списка в другой используется функция `list_move()`.

```
list_move(struct list_head *list, struct list_head *head)
```

Эта функция удаляет элемент, определяемый параметром `list`, из его списка и добавляет данный элемент в новый список *после* элемента, определяемого параметром `head`.

Чтобы переместить элемент из одного списка в конец другого, используется функция `list_move_tail()`.

```
list_move_tail(struct list_head *list, struct list_head *head)
```

Эта функция выполняет те же действия, что и `list_move()`, но помещает новый элемент *перед* элементом, определяемым параметром `head`.

Чтобы проверить, является ли список пустым, используется функция `list_empty()`.

```
list_empty(struct list_head *head)
```

Она возвращает ненулевое значение, если список пуст, и нулевое значение — в противном случае.

Чтобы слить воедино два несвязанных между собой списка, используется функция `list_splice()`.

```
list_splice(struct list_head *list, struct list_head *head)
```

Эта функция соединяет между собой два списка. При этом список, определяемый параметром `list`, помещается *после* элемента `head`.

Чтобы слить воедино два несвязанных между собой списка и повторно инициализировать старый список, используется функция `list_splice_init()`.

```
list_splice_init(struct list_head *list, struct list_head *head)
```

Экономия нескольких косвенных обращений

Если в вашей программе уже где-то сохранены указатели на следующий и предыдущий элементы списка, то вы можете сэкономить несколько операторов (речь идет об операторах разыменования для получения ссылок на элементы списка), вызвав напрямую внутренние функции ядра для работы со списками. Все перечисленные выше функции, по сути, не делают ничего, кроме получения указателей на следующий и предыдущий элементы списка и вызова соответствующих внутренних функций. Имена этих внутренних функций, как правило, совпадают с именами их оболочек, за исключением того, что перед ними добавляются два символа подчеркивания. Например, вместо функции `list_del(list)` можно вызвать функцию `__list_del(prev, next)`. Однако это имеет смысл делать только в случае, если указатели на следующий и предыдущий элементы списка *уже* известны в программе. В противном случае вы просто напишете уродливый код. Список всех интерфейсов можно найти в заголовочном файле `<linux/list.h>`.

Эта функция работает так же, как и `list_splice()`, за исключением того, что она выполняет повторную инициализацию пустого списка, на который указывает параметр `list`.

Обход элементов связанного списка

Теперь вы знаете, как с помощью средств ядра можно объявлять, инициализировать элементы связанного списка и манипулировать ими. Все это просто замечательно, однако если вы не сможете получить доступ к своим данным, то все эти средства просто бесполезны! Связанные списки — обычные контейнеры, содержащие важные для вас данные. Поэтому в вашем распоряжении должны быть средства для работы со списками, позволяющие перемещаться по элементам связанного списка и получать доступ к реальным структурам, содержащим данные. К счастью, в ядре предусмотрен прекрасный набор интерфейсов для обхода элементов связанного списка и обращения к структурам данных, содержащихся в них.

В отличие от функций манипулирования элементами связанного списка, рассмотренных в предыдущих разделах, перебор всех элементов списка очевидно потребует $O(n)$ операций, где n — количество узлов в списке.

Основной метод

Чаще всего для перебора всех элементов списка используется макрос `list_for_each()`, которому передаются два параметра — оба структуры типа `list_head`. Первый параметр — это указатель, который используется для обращения к текущему элементу списка. Он является временной переменной, которую вы, тем не менее, должны указать. Второй параметр — это структура типа `list_head`, соответствующая головному узлу списка, который вы хотите обойти (подробнее см. выше, в разделе "Структуры типа list_head"). На каждой итерации цикла в первом параметре будет находиться указатель на очередной элемент списка, пока не будут перебраны все его элементы, как показано ниже.

```
struct list_head *p;

list_for_each(p, &fox_list) {
   /* p указывает на элемент списка */
}
```

Ну что ж, пока что макрос `list_for_each()` нам полезен мало! Какой прок от указателя на структуру типа `list_head`? Ведь нам нужен указатель на структуру, в которой содержится эта структура `list_head`. Возвращаясь к предыдущему примеру со структурой типа `fox`, нам нужен указатель на структуру типа `fox` каждого элемента списка, а не указатель на переменную `list` этой структуры. Чтобы получить указатель на структуру, содержащую структуру типа `list_head`, мы должны воспользоваться макросом `list_entry()`, описанным выше. Например,

```
struct list_head *p;
struct       fox *f;

list_for_each(p, &fox_list) {
    /* Переменная f содержит указатель на структуру, в которой
       находится переменная list */
    f = list_entry(p, struct fox, list);
}
```

Практический метод

Описанный в предыдущем разделе метод обхода всех элементов списка не способствует созданию интуитивно понятного и элегантного кода, хотя он и продемонстрировал методику работы с узлами типа `list_head`. Поэтому в большей части кода ядра для обхода всех элементов списка используется другой макрос — `list_for_each_entry()`. В нем выполняется работа макроса `list_entry()`, что намного упрощает обход всех элементов списка.

```
list_for_each_entry(pos, head, member)
```

Здесь в переменной `pos` будет находиться указатель на текущий объект, содержащий структуру типа `list_head`. Этот как раз то значение, которое возвращается макросом `list_entry()`. В переменной `head` находится указатель на структуру типа `list_head`, соответствующую головному узлу, с которого вы хотите начать обход списка (в предыдущем примере это `fox_list`). Параметр `member` соответствует имени переменной типа `list_head` в объекте `pos` (в предыдущем примере это `list`). Это описание может легко сбить вас с толку, однако пользоваться макросом `list_for_each_entry()` очень легко. Ниже приведен пример того, как можно переписать приведенный выше код с `list_for_each()`, чтобы обойти все элементы списка типа `fox`.

```
struct fox *f;

list_for_each_entry(f, &fox_list, list) {
   /* На каждой итерации цикла в переменной 'f'
      содержится указатель на текущую структуру типа fox. */
}
```

А теперь рассмотрим реальный пример, взятый из подсистемы ядра Linux `inotify`, которая позволяет получать уведомления об изменениях в файловой системе.

```
static struct inotify_watch *inode_find_handle(struct inode *inode,
                                               struct inotify_handle *ih)

{
    struct inotify_watch *watch;

    list_for_each_entry(watch, &inode->inotify_watches, i_list) {
         if (watch->ih == ih)
              return watch;
    }
    return NULL;
}
```

В этой функции выполняется перебор всех элементов списка, указатель на который содержится в переменной `inode->inotify_watches`. Каждый такой элемент представляет собой структуру типа `inotify_watch`, а элементу типа `list_head` в этой структуре присвоено имя `i_list`. На каждой итерации цикла в переменной `watch` будет находиться указатель на очередной элемент списка. В этой простой функции выполняется поиск элементов в списке `inotify_watches`, находящемся в структуре типа `inode`, которая передана в качестве параметра. Наша цель — найти элемент типа `inotify_watch`, в котором дескриптор `inotify_handle` совпадает с тем, который передан функции в качестве параметра `ih`.

Обход списка в обратном направлении

Макрос `list_for_each_entry_reverse()` работает точно так же, как и `list_for_each_entry()`, за исключением того, что перебор элементов списка выполнятся в обратном порядке. Другими словами, вместо выборки указателя `next` для перехода на *следующий* элемент списка в этом макросе используется указатель `prev` и выполняется переход на *предыдущий* элемент списка. Используется он точно так же, как и макрос `list_for_each_entry()`, как показано ниже.

```
list_for_each_entry_reverse(pos, head, member)
```

Одна из причин, по которой может понадобиться просматривать список в обратном порядке, — это скорость. Если вам известно, что элемент, который нужно найти, вероятнее всего, находится *перед* текущим элементом, с которого начинается поиск, то, просмотрев список в обратном порядке, вы очень скоро его обнаружите. Вторая причина возникает при работе с упорядоченными списками. Например, при использовании связанного списка в виде стека выполняется обход этих элементов из конца в начало, чтобы реализовать стратегию обслуживания *LIFO* (Last-in/First-out — последним пришел, первым обслужен). Если у вас нет видимых причин, по которым может понадобиться выполнить обход списка в обратном направлении, не стоит сильно переживать по этому поводу, просто пользуйтесь макросом `list_for_each_entry()`, и все.

Обход списка с удалением

Стандартные методы перебора элементов списка не работают в случае, если при этом удаляются узлы списка. Все дело в том, что в стандартных методах подразумевается, что элементы списка при выполнении обхода не изменяются. Поэтому если в теле цикла текущий элемент списка будет удален, то на новой итерации программа не сможет перейти на следующий (или предыдущий) элемент. Подобные проблемы нередко возникают при выполнении циклов, поэтому программисты решают их путем сохранения указателя на следующий (или предыдущий) элемент во временной переменной *до* того, как будет удален текущий элемент. На этот случай в ядре Linux предусмотрен специальный макрос:

```
list_for_each_entry_safe(pos, next, head, member)
```

Макрос `list_for_each_entry_safe()` работает точно так же, как и `list_for_each_entry()`, за исключение того, что при его вызове вы должны указать переменную `next`, имеющую тот же тип, что и `pos`. Она используется в самом макросе `list_for_each_entry_safe()` для сохранения указателя на следующий элемент списка, что позволяет без последствий удалить текущий элемент. Рассмотрим пример, снова взятый из подсистемы ядра `inotify`.

```
void inotify_inode_is_dead(struct inode *inode)
{
   struct inotify_watch *watch, *next;

   mutex_lock(&inode->inotify_mutex);
   list_for_each_entry_safe(watch, next, &inode->inotify_watches,
                                          i_list) {
      struct inotify_handle *ih = watch->ih;
      mutex_lock(&ih->mutex);
      inotify_remove_watch_locked(ih, watch); /* Удалим элемент watch */
      mutex_unlock(&ih->mutex);
   }
   mutex_unlock(&inode->inotify_mutex);
}
```

В этой функции выполняется перебор всех элементов списка `inotify_watches` и их удаление. Если бы в коде использовалось стандартное решение на основе макроса `list_for_each_entry()`, то это бы привело к ошибке в программе типа "использование после удаления". Дело в том, что для перемещения на следующий элемент списка нам нужен доступ к текущему элементу, указатель на который содержится в переменной `watch`, а его содержимое к этому моменту уже разрушено.

Если нужно перебрать элементы списка в обратном порядке и при этом удалить некоторые из них, используется макрос `list_for_each_entry_safe_reverse()`, как показано ниже.

```
list_for_each_entry_safe_reverse(pos, n, head, member)
```

Этот макрос используется точно так же, как и описанный выше `list_for_each_entry_safe()`.

> **Не забывайте о блокировке!**
>
> Описанный выше "безопасный" вариант макроса `list_for_each_entry()` ***только*** предотвращает возникновение ошибок *внутри* тела цикла, связанных с удалением текущего элемента списка. Однако, если существует вероятность удаления элементов этого же списка из другого участка кода либо их одновременная обработка, следует заблаговременно позаботиться о выставлении блокировки при доступе к списку.
>
> Синхронное выполнение кода и блокировки описано в главе 9, "Общие сведения о синхронизации кода ядра", и главе 10, "Средства синхронизации ядра".

Другие методы работы со связанными списками

В ядре Linux предусмотрено большое количество функций, предназначенных для работы со связанными списками и позволяющих выполнить над ними практически любую мыслимую операцию. Все они описаны в заголовочном файле `<linux/list.h>`.

Очереди

В ядре любой операционной системы довольно часто используется обобщенный шаблон программирования, называемый *"производитель и потребитель"* (producer and consumer). Согласно этому шаблону, производитель создает данные, например сообщение об ошибке, которое должно быть кем-то прочитано, или сетевой пакет, который должен быть чем-то обработан. В свою очередь, потребитель читает, обрабатывает или каким-либо другим образом *потребляет* данные. Часто так бывает, что проще всего реализовать этот шаблон можно с помощью *очереди*. Производитель помещает данные в очередь, а потребитель выбирает их из нее. При этом данные выбираются потребителем в том же порядке, в котором они были помещены в очередь производителем. Другими словами, если некоторый фрагмент данных был первым помещен в очередь, то он будет первым и удален из нее. В результате реализуется стратегия обслуживания *FIFO* (First-in/First-out — первым пришел, первым обслужен). Пример стандартной очереди приведен на рис. 6.5.

В ядре Linux обобщенная реализация очереди называется *`kfifo`*. Ее код находится в файле `kernel/kfifo.c`, а определения содержатся в файле `<linux/kfifo.h>`. В этом разделе мы рассмотрим API, который был обновлен в ядре версии 2.6.33. Он немного отличается от API, который был в версиях ядра до 2.6.33, поэтому перед написанием кода внимательно ознакомьтесь с содержимым файла `<linux/kfifo.h>`.

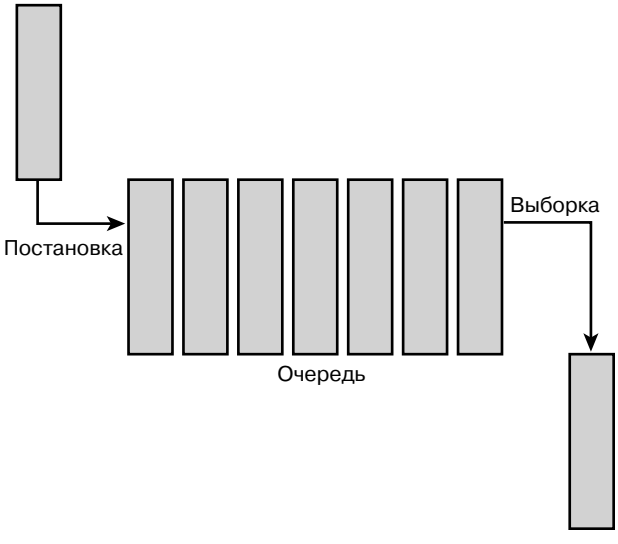


Рис. 6.5. Очередь, реализующая стратегию обслуживания FIFO

Система `kfifo`

Система очередей в Linux работает аналогично всем остальным подобным абстракциям. Она обеспечивает выполнение двух основных операций: *постановку в очередь* (enqueue), которая почему-то названа `in`, и *выборку из очереди* (dequeue), также неудачно названную `out`. В объекте `kfifo` хранятся два смещения относительно начала очереди. Одно из них (входное) предназначено для *постановки* элемента в очередь (in offset), а второе (выходное) — для *выборки* его из очереди (out offset). Входное смещение указывает на место в очереди, куда будет записан следующий элемент при выполнении операции постановки в очередь. Выходное смещение — это место в очереди, откуда будет браться следующий элемент при выполнении операции выборки из очереди. Выходное смещение всегда меньше или равно входному смещению. Дело в том, что нет особого смысла в том, когда выходное смещение будет больше входного. В противном случае это грозит тем, что из очереди может быть извлечен элемент, который туда еще не поставлен.

При выполнении операции постановки в очередь (`in`) данные копируются в участок памяти очереди, на который указывает входное смещение. После завершения операции входное смещение увеличивается на размер скопированных данных. При выполнении операции выборки из очереди (`out`) данные копируются из участка памяти, на который указывает выходное смещение. После завершения операции выходное смещение увеличивается на размер извлеченных из очереди данных. Если выходное значение равно входному — очередь пуста. При этом нельзя выполнить операцию выборки до тех пор, пока новые данные не будут помещены в очередь. Если входное смещение равно размеру очереди, значит, очередь заполнена и в нее нельзя помещать новые данные до выполнения операции *очистки* (reset).

Создание очереди

Перед использованием объекта kfifo его нужно определить и проинициализировать. Как и с большинством объектов ядра, это можно сделать динамически или статически. Чаще всего используется динамический метод, как показано ниже.

```
int kfifo_alloc(struct kfifo *fifo, unsigned int size, gfp_t gfp_mask);
```

Функция kfifo_alloc() создает и инициализирует объект типа kfifo длиной size байт. Параметр gfp_mask используется ядром для выделения памяти под очередь. (Распределение памяти будет рассмотрено в главе 12, "Управление памятью".) При успешном выполнении функция kfifo_alloc() возвращает нулевое значение, а при ошибке — отрицательное число. Ниже приведен пример использования этой функции.

```
struct kfifo fifo;
int ret;

ret = kfifo_alloc(&fifo, PAGE_SIZE, GFP_KERNEL);
if (ret)
    return ret;
/* Теперь объект 'fifo' представляет собой очередь
 длиной PAGE_SIZE... */
```

Если вы выделили память под буфер самостоятельно, то проинициализировать очередь можно так, как показано ниже.

```
void kfifo_init(struct kfifo *fifo, void *buffer, unsigned int size);
```

Функция kfifo_init() создает и инициализирует в памяти объект типа kfifo длиной size байт. При этом адрес буфера, где будет размещаться очередь, указывается в параметре buffer. При вызове обеих функций, kfifo_alloc() и kfifo_init(), значение параметра size должно быть кратно 2^n.

Статическое объявление объекта типа kfifo проще, хотя оно и используется реже.

```
DECLARE_KFIFO(name, size);
INIT_KFIFO(name);
```

В результате будет создан статический объект типа kfifo с именем name и размером очереди size байт. Как и прежде, число size должно быть кратно 2^n.

Постановка в очередь

После того как объект типа kfifo создан и проинициализирован, для записи данных в очередь используется функция kfifo_in().

```
unsigned int kfifo_in(struct kfifo *fifo, const void *from, unsigned int len);
```

Эта функция копирует len байтов, начиная с адреса from, в очередь, представленную переменной fifo. При успешном выполнении функция kfifo_in() возвращает количество байтов, скопированных в очередь. Если в очереди будет свободно меньше, чем len байтов, то функция скопирует только свободное количество байтов. Поэтому реальное число скопированных байтов может быть меньше, чем len, или даже вовсе равно нулю, если ничего скопировать не удалось.

Выборка из очереди

После того как данные были помещены в очередь с помощью функции kfifo_in(), для их извлечения используется функция kfifo_out(), как показано ниже.

```
unsigned int kfifo_out(struct kfifo *fifo, void *to, unsigned int len);
```

Эта функция копирует не более, чем `len` байтов, из очереди, определяемой параметром `fifo`, в буфер, заданный параметром `to`. В случае успешного выполнения функция `kfifo_out()` возвращает число скопированных байтов. Если в очереди осталось меньше, чем `len` байтов, функция скопирует их и возвратит число, меньшее, чем `len`.

После извлечения из очереди данные удаляются из нее навсегда. Это обычная практика использования очередей. Однако, если вам нужно прочитать данные из очереди, не удаляя их, воспользуйтесь функцией `kfifo_out_peek()`, как показано ниже.

```
unsigned int kfifo_out_peek(struct kfifo *fifo, void *to, unsigned int len,
                            unsigned offset);
```

Эта функция работает так же, как и `kfifo_out()`, за исключением того, что после чтения значение выходного смещения не изменяется. В результате данные можно будет извлечь из очереди позже, воспользовавшись функцией `kfifo_out()`. Параметр `offset` определяет индекс в очереди. Чтобы прочитать данные с головы очереди, так как это делает функция `kfifo_out()`, его значение должно быть равно нулю.

Определение размера очереди

Чтобы определить размер буфера в байтах, который используется для размещения очереди типа `kfifo`, вызовите функцию `kfifo_size()`, как показано ниже.

```
static inline unsigned int kfifo_size(struct kfifo *fifo);
```

В еще одном примере неудачного выбора имени функции в ядре используется функция `kfifo_len()` для определения числа байтов, находящихся в очереди типа `kfifo`.

```
static inline unsigned int kfifo_len(struct kfifo *fifo);
```

Чтобы перед записью в очередь типа `kfifo` узнать размер ее свободного буфера, вызовите функцию `kfifo_avail()`, как показано ниже.

```
static inline unsigned int kfifo_avail(struct kfifo *fifo);
```

И наконец, функции `kfifo_is_empty()` и `kfifo_is_full()` возвращают ненулевое значение, если указанная в параметрах очередь типа `kfifo` пуста или заполнена, и нулевое значение — в противном случае.

```
static inline int kfifo_is_empty(struct kfifo *fifo);
static inline int kfifo_is_full( struct kfifo *fifo);
```

Очистка и удаление очереди

Чтобы очистить очередь типа `kfifo` и удалить все ее содержимое, используется функция `kfifo_reset()`, как показано ниже.

```
static inline void kfifo_reset(struct kfifo *fifo);
```

Чтобы удалить очередь типа `kfifo`, размещенную с помощью функции `kfifo_alloc()`, используется функция `kfifo_free()`.

```
void kfifo_free(struct kfifo *fifo);
```

Если очередь типа `kfifo` была создана с помощью функции `kfifo_init()`, то освобождение занимаемого ею буфера лежит целиком на совести программиста. Как это сделать, зависит от того, каким образом этот буфер был распределен. Вопросы распределения и освобождения динамической памяти рассматриваются в главе 12, “Управление памятью”.

Примеры использования очередей

После знакомства с интерфейсами функций рассмотрим примеры использования очередей типа `kfifo`. Предположим, нами была создана очередь типа `kfifo` размером 8 Кбайт и указатель на нее помещен в переменную `fifo`. Теперь мы можем поместить данные в очередь. В данном примере мы будем записывать в очередь обычные целые числа. В вашем коде вы можете помещать в нее более сложные структуры данных, как того требует решение поставленной задачи. Использование целых чисел позволяет нам просто показать, как работают очереди типа `kfifo`.

```
unsigned int i;

/* Запишем в очередь типа kfifo с именем 'fifo' последовательность
    целых чисел [0, 32) */
for (i = 0; i < 32; i++)
     kfifo_in(fifo, &i; sizeof(i));
```

Теперь в очереди типа `kfifo` с именем `fifo` будут находиться числа от 0 до 31 включительно. Мы можем прочитать ее первый элемент и убедиться, что он равен нулю.

```
unsigned int val;
int ret;

ret = kfifo_out_peek(fifo, &val, sizeof(val), 0);
if (ret != sizeof(val))
    return -EINVAL;
printk(KERN_INFO "%u\n", val); /* Должно быть напечатано число 0 */
```

Чтобы извлечь все числа из очереди и вывести их на печать, воспользуемся функцией `kfifo_out()`, как показано ниже.

```
/* Запустим цикл, пока в очереди есть данные... */
while (kfifo_avail(fifo)) {
     unsigned int val;
     int ret;

     /* ... прочитаем одно число за раз */
     ret = kfifo_out(fifo, &val, sizeof(val));
     if (ret != sizeof(val))
          return -EINVAL;
     printk(KERN_INFO "%u\n", val);
}
```

В результате выполнения этого фрагмента кода на печать будут выведены числа от 0 до 31 в возрастающем порядке. (Если окажется, что на печать будут выведены числа с 31 до 0, значит, мы работаем со стеком, а не с очередью.)

Таблицы отображения

Таблица отображения (map), которую часто называют *ассоциативным массивом* (associative array), представляет собой совокупность уникальных ключей, в которой каждому ключу поставлено в соответствие определенное значение. Взаимосвязь между ключом и его значением называется *отображением* (mapping). Таблицы отображения поддерживают по меньшей мере три операции, перечисленные ниже.

- Добавить (ключ, значение)
- Удалить (ключ)
- Значение = Поиск (ключ)

Несмотря на то что хеш-таблица представляет собой особый вид таблицы отображения, далеко не все таблицы отображения реализуются через вычисление хеш-функции. В отличие от хеш-таблиц в таблицах отображения для хранения значений могут использоваться *самобалансирующиеся двоичные деревья поиска* (self-balancing binary search tree). Хотя асимптотический коэффициент сложности для среднего случая (average-case asymptotic complexity) хеш-таблиц лучше (см. раздел "Алгоритмическая сложность"), двоичные деревья поиска предпочтительнее для худших случаев, поскольку в них используется логарифмическая, а не линейная зависимость. В двоичных деревьях поиска также *сохраняется порядок* следования элементов (order preservation), что позволяет очень эффективно выполнять итерации по всем элементам совокупности в порядке их сортировки. И наконец, двоичные деревья поиска не требуют вычисления хеш-функции. В них можно использовать любой тип ключей, при условии, что для них можно определить оператор `<=`.

Несмотря на то что для всех совокупностей общим является отображение ключей на значения, название *таблица отображения* (map) часто используется для обозначения исключительно ассоциативных массивов, реализованных с использованием двоичных деревьев поиска, чтобы отличать их от хеш-таблиц. Например, в стандартной библиотеке шаблонов (STL) C++ контейнер `std::map` реализован на основе самобалансирующегося двоичного дерева поиска или аналогичной структуры данных, поскольку он предоставляет возможность упорядоченного обхода совокупности.

В ядре Linux предусмотрена простая и эффективная реализация структуры данных таблицы отображения. Однако она не является универсальной, поскольку используется всего для одной-единственной цели — отображения уникальных идентификаторов пользователей (UID) на указатели системных структур. Кроме поддержки трех перечисленных выше операций для таблиц отображения, в ядре Linux введена также комбинированная операция *размещения* (allocate), которая является надстройкой над операцией *добавления* (add). Она не только добавляет пары UID/значение в таблицу отображения, но и сама генерирует UID.

Для отображения пользовательских UID, таких как дескрипторы наблюдения `inotify` или идентификаторы POSIX-таймера, на связанные с ними структуры данных ядра, такие как `inotify_watch` или `k_itimer` соответственно, используется структура данных `idr`. Как и для ряда других структур, имя для этой структуры выбрано неудачно и способно сбить вас с толку.

Инициализация структуры `idr`

Процесс инициализации структуры `idr` очень простой. Сначала вам нужно статически или динамически выделить для нее память, а затем вызвать функцию `idr_init()`, как показано ниже.

```
void idr_init(struct idr *idp);
```

Например:

```
struct idr id_huh; /* Статически определяемая структура idr */
idr_init(&id_huh); /* Инициализация структуры idr           */
```

Выделение нового UID

После инициализации структуры `idr` можно выделить новый UID. Этот процесс происходит в два этапа. Сначала вы должны указать системе, что хотите выделить новый

UID для структуры idr. Это может повлечь за собой изменение размера связанного с ней двоичного дерева. Затем следует вызвать еще одну функцию, которая и выделит на самом деле новый UID. Такие сложности связаны с тем, что сначала системе может понадобиться без блокировки изменить исходный размер дерева, а это, в свою очередь, может потребовать выделение дополнительной памяти. Вопросы распределения памяти будут рассмотрены в главе 12, "Управление памятью", а блокировки — в главе 9, "Общие сведения о синхронизации кода ядра", и главе 10, "Средства синхронизации ядра". А пока что рассмотрим использование структуры idr без учета блокировок.

Первая функция, которая может изменить размер связанного дерева, называется idr_pre_get().

```
int idr_pre_get(struct idr *idp, gfp_t gfp_mask);
```

По мере необходимости (т.е. если это нужно для выделения нового UID) данная функция изменяет размер структуры idr, указатель на которую содержится в переменной idp. Если изменение размера структуры необходимо, то при выделении памяти будут использоваться *gfp-флажки*, которые мы рассмотрим в главе 12, "Управление памятью". Вам не нужно заботиться о синхронизации одновременного доступа к этой функции. В отличие от большинства других функций ядра, функция idr_pre_get() возвращает число 1 при успешном выполнении и нуль — в случае ошибки. Поэтому будьте внимательны!

Вторая функция, idr_get_new(), выделяет новый UID и добавляет его в структуру idr.

```
int idr_get_new(struct idr *idp, void *ptr, int *id);
```

Для выделения нового UID и привязки его к указателю ptr в этой функции используется структура idr, указатель на которую содержится в переменной idp. В случае успешного выполнения эта функция возвращает нуль и записывает новый UID в переменную id. В случае возникновения ошибки функция возвращает ненулевой код ошибки -EAGAIN, если вы должны будете снова вызвать функцию idr_pre_get(), и код -ENOSPC, если структура idr заполнена.

Рассмотрим полноценный пример.

```
int id;

do {
    if (!idr_pre_get(&idr_huh, GFP_KERNEL))
        return -ENOSPC;
    ret = idr_get_new(&idr_huh, ptr, &id);
} while (ret == -EAGAIN);
```

В случае успешного выполнения этого фрагмента кода в переменную id будет записан новый UID, который будет связан со структурой ptr (ее мы здесь не определили).

Функция idr_get_new_above() позволяет задать минимальное значение возвращаемого UID.

```
int idr_get_new_above(struct idr *idp, void *ptr, int starting_id, int
*id);
```

Она работает точно так же, как и idr_get_new(), за исключением того, что значение нового UID будет гарантированно больше или равно значению, указанному в переменной starting_id. Данный вариант функции позволяет гарантировать для пользователей idr, что UID не будет использоваться повторно. В свою очередь, это означает, что значение UID будет уникальным не только среди текущих распределенных идент-

фикаторов, но и на протяжении всего сеанса работы системы. Приведенный ниже фрагмент кода похож на предыдущий, за исключением того, что в нем мы требуем, чтобы значения UID строго увеличивались.

```
int id;

do {
    if (!idr_pre_get(&idr_huh, GFP_KERNEL))
        return -ENOSPC;
    ret = idr_get_new_above(&idr_huh, ptr, next_id, &id);
} while (ret == -EAGAIN);

if (!ret)
   next_id = id + 1;
```

Поиск UID

После размещения нескольких номеров UID в `idr` можно выполнить их поиск. При этом в вызывающей программе необходимо указать UID, а функция `idr_find()` возвратит связанный с ним указатель, как показано ниже. Все происходит намного проще, чем при распределении нового UID.

```
void *idr_find(struct idr *idp, int id);
```

В случае успешного выполнения эта функция извлекает из структуры `idr`, на которую указывает переменная `idp`, и возвращает указатель, связанный с UID, номер которого задан в переменной `id`. В случае ошибки функция возвращает значение `NULL`. Обратите внимание на то, что если сопоставить UID значение `NULL` с помощью функции `idr_get_new()` или `idr_get_new_above()`, то функция `idr_find()` будет выполнена без ошибки и возвратит значение `NULL`. В результате вы не сможете понять, была ли ошибка при выполнении функции. Поэтому не стоит сопоставлять UID значения `NULL`.

Ниже приведен простой пример использования функции `idr_find()`.

```
struct my_struct *ptr = idr_find(&idr_huh, id);
if (!ptr)
   return -EINVAL; /* Ошибка */
```

Удаление UID

Для удаления UID из дерева `idr` используется функция `idr_remove()`.

```
void idr_remove(struct idr *idp, int id);
```

В случае успешного выполнения данная функция удаляет UID, указанный в переменной `id`, из дерева `idr`, на которое указывает переменная `idp`. К сожалению, функция `idr_remove()` вообще не возвращает никаких ошибок, например, если значение `id` вообще не входит в дерево `idp`.

Аннулирование `idr`

Аннулирование структуры `idr` выполняется с помощью простого вызова функции `idr_destroy()`.

```
void idr_destroy(struct idr *idp);
```

В случае успешного выполнения функция `idr_destroy()` освобождает только неиспользуемую память, которую занимает структура `idr`, заданная указателем `idp`. При

этом память, занимаемая текущими выделенными UID, не освобождается. Как правило, в коде ядра структуры `idr` аннулируются только в момент завершения работы системы или выгрузки ядра. При этом ядро не может быть выгружено до тех пор, пока не завершит свою работу последний пользователь (т.е. в системе не останется ни одного UID). Однако с помощью функции `idr_remove_all()` можно принудительно удалить все UID, как показано ниже.

```
void idr_remove_all(struct idr *idp);
```

Эту функцию нужно вызвать *перед* функцией `idr_destroy()`, чтобы гарантированно освободить всю память, занимаемую структурой `idr`, на которую указывает переменная `idp`.

Двоичные деревья

Дерево (tree) представляет собой структуру данных, которая позволяет представить данные в иерархической древовидной форме. С точки зрения математики дерево — это *непериодический*, *связанный* и *направленный граф*, каждая вершина которого (называемая *узлом*) имеет нуль или более исходящих ребер и нуль или одно входящее ребро. *Двоичное дерево* (binary tree) — это дерево, каждая вершина которого имеет не более двух исходящих ребер, т.е. дерево, в котором каждый узел имеет нуль, один или два потомка. Пример двоичного дерева приведен на рис. 6.6.

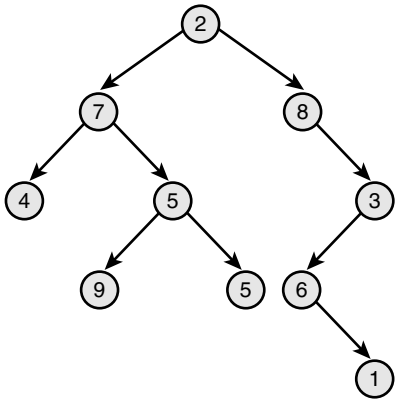


Рис. 6.6. Двоичное дерево

Двоичные деревья поиска

Двоичное дерево поиска (binary search tree, BST) представляет собой обычное двоичное дерево, узлы которого упорядочены специальным образом. Порядок следования вершин часто определяется на основе перечисленных ниже правил.

- Часть дерева, расположенная левее корневого узла, содержит вершины, значения которых меньше значения корневого узла.
- Часть дерева, расположенная правее корневого узла, содержит вершины, значения которых больше значения корневого узла.
- Любая часть дерева также является двоичным деревом поиска.

Таким образом, двоичное дерево поиска представляет собой двоичное дерево, в котором все вершины *упорядочены* так, что значения всех левых потомков всегда меньше значения их родительского узла, а значения всех правых потомков всегда больше значения их родительского узла. Поэтому в таком дереве операции поиска конкретного узла и их упорядоченного обхода выполняются эффективно (по логарифмическому и линейному законам соответственно). Пример двоичного дерева поиска приведен на рис. 6.7.

Самобалансирующиеся двоичные деревья поиска

Под *глубиной* (depth) узла дерева будем понимать количество его родительских узлов, отсчитываемых от корневого узла. Узлы, расположенные внизу дерева, не имеют дочерних узлов и называются *листьями*. Под *высотой* (height) дерева будем понимать глубину его самого нижнего узла. Под *сбалансированным двоичным деревом поиска* (balanced binary search tree) будем понимать двоичное дерево поиска, в котором глубина всех его листьев отличается не более чем на единицу (рис. 6.8). *Самобалансирующееся двоичное дерево поиска* (self-balancing binary search tree) — это двоичное дерево поиска, которое в силу своего нормального функционирования стремится почти всегда оставаться сбалансированным.

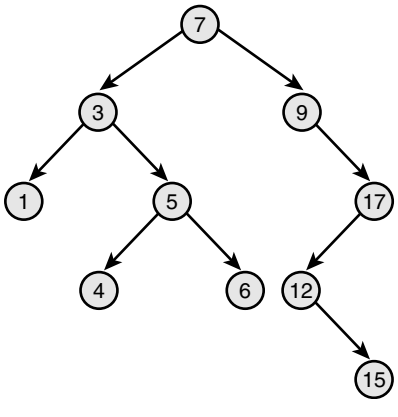


Рис. 6.7. Двоичное дерево поиска

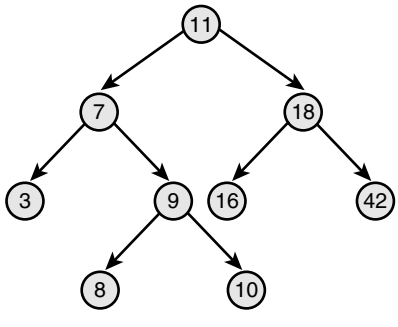


Рис. 6.8. Сбалансированное двоичное дерево поиска

Красно-черные деревья

Красно-черное дерево является подвидом самобалансирующегося двоичного дерева поиска. Основные структуры данных в Linux, относящиеся к двоичным деревьям, являются красно-черным деревом. Такие деревья имеют специальный атрибут цвета, который может быть либо *красным*, либо *черным*. Красно-черные деревья остаются почти сбалансированными за счет того, что всегда соблюдаются шесть условий.

1. Все узлы являются либо красными, либо черными.
2. Листья являются только черными.
3. Листья не содержат данных.
4. Все узлы, не являющиеся листьями, содержат двух потомков.
5. Если узел красный, то оба его потомка черные.

6. Путь, проложенный от произвольного узла к одному из его листьев, проходит через одинаковое количество черных узлов и является кратчайшим путем к любому другому листу.

Все перечисленные выше шесть условий гарантируют, что глубина самого "глубокого" листа не будет отличаться от глубины самого "мелкого" листа более чем в два раза. А это значит, что дерево будет почти всегда сбалансированным. Доказать данный факт можно на удивление просто. Согласно пятому условию, красный узел не может быть дочерним или родительским узлом другого красного узла. Согласно шестому условию, все пути, проложенные от некоторого узла через дерево до его листьев, будут проходить через одинаковое число черных узлов. Самый длинный путь через дерево будет проходить через чередующиеся черные и красные узлы. Таким образом, кратчайший путь, который должен проходить через одинаковое количество черных узлов, проходит только через черные узлы. По этой причине самый длинный путь, проложенный от корня дерева до листа, не может более чем в два раза превышать самый короткий путь от корня до любого другого листа.

Если в результате выполнения операции вставки узла в дерево или удаления его из дерева будут соблюдаться все перечисленные выше шесть условий, дерево будет оставаться почти сбалансированным. Вам может показаться странным, что при выполнении операций вставки и удаления требуется соблюдать именно *эти* шесть условий. Почему бы не реализовать операции так, чтобы они соблюдали другие, более простые условия и при этом в результате бы получалось сбалансированное дерево? Как оказалось, эти условия относительно легко можно соблюсти, хотя и не так просто реализовать. А это означает, что при выполнении операций вставки и удаления можно гарантировать, что дерево будет всегда оставаться почти сбалансированным, причем для этого не нужно будет прикладывать никаких особых усилий.

Описание того, *как* при выполнении операций вставки и удаления удается соблюсти все шесть условий, выходит за рамки данной книги. Несмотря на то что условия просты, реализация операций достаточно сложная. Если вы хотите во всем глубоко разобраться, обратитесь к любому хорошему учебнику по структурам данных и алгоритмам.

Реализация в Linux

Реализация красно-черных деревьев в Linux называется *rbtrees*. Код находится в файле `lib/rbtree.c`, а файл `<linux/rbtree.h>` содержит сигнатуры функций и определения структур данных. Если не учитывать оптимизационные моменты, то красно-черные деревья, реализованные в Linux, напоминают "классические" красно-черные деревья, описанные в предыдущем разделе. Они всегда остаются сбалансированными, поэтому время выполнения операции вставки зависит от числа узлов в дереве по логарифмическому закону.

Корневой узел красно-черного дерева в Linux описывается структурой `rb_root`. Чтобы создать новое дерево, нужно выделить память под структуру `rb_root` и проинициализировать ее специальным значением `RB_ROOT`, как показано ниже.

```
struct rb_root root = RB_ROOT;
```

Отдельные узлы в красно-черном дереве описываются структурой `rb_node`. Чтобы перейти от текущего узла `rb_node` к его левому или правому потомку, нужно воспользоваться соответствующими указателями, которым присвоены имена `rb_left` и `rb_right`.

В реализации красно-черных деревьев в Linux не предусмотрены функции для поиска и вставки узлов в дерево. Подразумевается, что эти операции должны определяться в клиентском коде. Все дело в том, что в языке C не так-то просто реализовать обобщенное программирование. Поэтому разработчики ядра посчитали, что самым эффективным решением будет ручная реализация операций поиска и вставки в клиентском коде с использованием стандартных вспомогательных функций и собственных операторов сравнения.

Лучше всего показать, как выполняются операции поиска и вставки в красно-черных деревьях, на реальном примере. Сначала рассмотрим поиск. В приведенной ниже функции реализован поиск фрагмента файла, представленного в виде пары "индексный узел–смещение", в страничной кеш-памяти системы Linux. Для каждого индексного узла существует собственное красно-черное дерево, в узлах которого содержатся смещения страницы в файле. Таким образом, в приведенной ниже функции выполняется поиск индексного узла в красно-черном дереве, которому соответствует заданное смещение в файле.

```
struct page * rb_search_page_cache(struct inode *inode,
                                   unsigned long offset)
{
   struct rb_node *n = inode->i_rb_page_cache.rb_node;

   while (n) {
      struct page *page = rb_entry(n, struct page, rb_page_cache);

      if (offset < page->offset)
          n = n->rb_left;
      else if (offset > page->offset)
          n = n->rb_right;
      else
          return page;
   }
   return NULL;
}
```

В данном примере в цикле выполняется обход красно-черного дерева. При этом в зависимости от значения смещения выбирается дочерний элемент либо из левой, либо из правой ветви дерева. Поскольку элементы красно-черного дерева упорядочены, их функция сравнения реализована в виде условного оператора `if-else`. Поиск завершается, как только будет найден узел с требуемым смещением. При этом функция возвращает указатель на связанную с ним структуру `page`. Если в цикле будет достигнут последний элемент красно-черного дерева, а нужное смещение найдено не будет, значит, требуемый элемент в дереве отсутствует и функция возвращает значение `NULL`.

Операция вставки в дерево немного сложнее, поскольку в ней задействуется логика как поиска, так и вставки. Приведенная ниже функция достаточно сложная, правда, если вам нужно будет реализовать собственную процедуру вставки, она послужит хорошим примером.

```
struct page * rb_insert_page_cache(struct inode *inode,
                                   unsigned long offset,
                                   struct rb_node *node)
{
   struct rb_node **p = &inode->i_rb_page_cache.rb_node;
   struct rb_node *parent = NULL;
   struct page *page;

   while (*p) {
       parent = *p;
       page = rb_entry(parent, struct page, rb_page_cache);
```

```
        if (offset < page->offset)
            p = &(*p)->rb_left;
        else if (offset > page->offset)
            p = &(*p)->rb_right;
        else
            return page;
    }
    rb_link_node(node, parent, p);
    rb_insert_color(node, &inode->i_rb_page_cache);

    return NULL;
}
```

Как и в примере с функцией поиска, здесь в цикле также выполняется обход дерева и в зависимости от значения смещения выбирается направление следующего перехода. Однако, в отличие от предыдущей функции, в нашей функции *не требуется* находить нужное смещение. Мы должны достигнуть последнего листового элемента дерева, который как раз и будет точкой вставки нового узла с новым смещением. После того как точка вставки будет найдена, вызывается функция `rb_link_node()` для вставки нового узла после текущего. Затем вызывается функция `rb_insert_color()`, которая выполняет сложную задачу по перебалансировке дерева. Функция `rb_insert_page_cache()` возвращает значение `NULL`, если страница была добавлена в кеш-память, либо указатель на существующую структуру `page`, если требуемая страница уже была в кеш-памяти.

Какие структуры данных следует использовать, если...

Выше мы уже ознакомились с четырьмя основными структурами данных системы Linux: связанными списками, очередями, таблицами отображения и красно-черными деревьями. В этом разделе будет приведено несколько советов, которые помогут вам выбрать правильную структуру данных при написании своего кода.

Если основным методом доступа к структуре данных является последовательный перебор ее элементов, используйте связанный список. Интуитивно понятно, что не существует структур данных, обеспечивающих лучший, чем линейный, коэффициент сложности, при обходе каждого элемента. Поэтому вам следует отдать предпочтение самой простой структуре данных для выполнения такой простейшей работы. Связанным спискам также следует отдать предпочтение, если вопросы быстродействия не ставятся на первый план, а также если вам нужно хранить сравнительно небольшое число элементов списка или если нужно взаимодействовать с другим кодом ядра, в котором используются связанные списки.

Если в вашем коде реализован шаблон “производитель и потребитель” (producer and consumer), используйте очередь, особенно если вам нужен (или вы можете этого добиться) буфер фиксированного размера. Очереди позволяют очень просто и эффективно добавлять и удалять элементы, а также реализовать стратегию обслуживания типа FIFO (т.е. “первым пришел — первым обслужен”). Подобная стратегия обслуживания как раз и требуется в большинстве случаев применения шаблона “производитель и потребитель”. С другой стороны, если количество сохраняемых в очереди элементов заранее неизвестно и может быть очень большим, лучше все-таки использовать связанные списки, поскольку они позволяют динамически добавить произвольное число элементов в список.

Если вам нужно сопоставить идентификатор UID указателю на какой-нибудь объект, используйте таблицы отображения. Они позволяют выполнить такое сопоставление просто и эффективно, кроме того, автоматически поддерживают и распределяют UID за вас. Однако следует заметить, что интерфейс таблиц отображения, принятый в Linux, предназначен исключительно для сопоставления идентификаторов UID и указателей на объекты, поэтому для других целей его применять нежелательно. Если вам нужно обрабатывать дескрипторы, полученные от пользовательской программы, также используйте таблицы отображения.

Если вам нужно сохранить большой массив данных и эффективно выполнять в нем поиск нужного элемента, используйте красно-черные деревья. При этом время поиска будет зависеть от количества элементов в дереве по логарифмическому закону, а время упорядоченного обхода элементов — по линейному закону. Несмотря на то что красно-черные деревья достаточно сложны в реализации по сравнению с другими структурами данных, они занимают не так уж и много памяти. Если же вам не нужно выполнять большое количество критичных по времени операций поиска, красно-черные деревья, пожалуй, не самый лучший выбор. В подобных случаях используйте связанные списки.

А что если ни одна из перечисленных выше структур данных вам не подходит? В ядре Linux реализованы и другие редко используемые структуры данных, одна из которых вам наверняка подойдет. Например, *дерево остатков* (radix trees), которое является вариантом *нагруженного*, или *префиксного дерева* (trie), а также *битовые массивы* (bitmaps). И только после того как будут рассмотрены *все* варианты структур данных ядра и ни один из них вам не подойдет, можете “изобретать свой велосипед”. Одной из универсальных структур данных, которая часто реализуется в виде самостоятельных исходных файлов, является хеш-таблица. Поскольку ее код реализации не намного больше, чем у остальных структур данных, а хеш-функция весьма специфична для конкретных случаев использования, не имеет особого смысла предоставлять в ядре ее реализацию на таком языке, как C, не поддерживающем обобщенное программирование.

Алгоритмическая сложность

Очень часто в информатике и связанных с нею дисциплинах нужно как-то количественно выразить сложность, или, другими словами, масштабируемость, алгоритмов. Для этого существуют различные методы. Одной из часто используемых методик является изучение асимптотической характеристики алгоритма. Речь идет об оценке эффективности алгоритма в случае, когда количество обрабатываемых им исходных данных очень велико и стремится к бесконечности. Асимптотическая характеристика показывает, насколько хорошо происходит масштабирование алгоритма по мере увеличения количества исходных данных. Анализ масштабируемости алгоритма (т.е. насколько хорошо он справляется с увеличением потока исходных данных) позволяет нам оценить скорость его работы и лучше понять его характеристику.

Что такое алгоритм?

Алгоритм можно определить как набор инструкций, примененных, вероятно, к одному или нескольким исходным данным, позволяющих получить заданный результат, или выходные данные. Например, последовательность действий, которая позволяет подсчитать количество людей, находящихся в комнате, есть алгоритм. При этом люди являются исходными данными, а их количество — результатом работы алгоритма, или выходными

данными. В ядре Linux примерами алгоритмов могут служить процессы вытеснения страниц и планирования задач. С математической точки зрения алгоритм подобен функции (по крайней мере, вы всегда можете смоделировать алгоритм в виде функции). Например, если обозначить алгоритм подсчета людей в комнате как `f`, а количество людей как `x`, функцию можно записать так:

```
y = f(x),
```

где `y` — время, необходимое для подсчета `x` человек.

Понятие большого "О"

Одна из самых полезных оценок асимптотической характеристики алгоритма — его *верхняя оценка* (upper bound), которая является функцией, чье значение после некоторой начальной точки всегда больше, чем значение исследуемой функции. При этом говорят, что верхняя оценка растет так же быстро, как и исследуемая функция. Для обозначения этого роста было введено специальное понятие — большое "О". Используя его, можно записать: `f(x) ∈ O(g(x))`. Это означает, что функция `f(x)` принадлежит множеству `O` функций `g(x)`. Формальное математическое определение этого выглядит так:

Если `f(x) ∈ O(g(x))`, то

$\exists c, x'$, такие, что $f(x) \le c \cdot g(x), \forall x > x'$.

На понятном всем языке это означает, что, как только исходное значение `x` становится больше, чем некоторое начальное значение `x'`, время вычисления `f(x)` всегда меньше или равно времени вычисления `g(x)`, умноженному на некоторую произвольную постоянную.

По сути, для выполнения оценки эффективности алгоритма нужно подобрать такую функцию, асимптотическая характеристика которой была бы хуже, чем у исследуемого алгоритма. Тогда на основе этой функции вы сможете посмотреть, как будет вести себя алгоритм при больших массивах исходных данных, и сделать верхнюю оценку его асимптотической характеристики.

Понятие большой "тета"

Когда мы рассуждаем о большом "О", то на самом деле чаще всего имеем в виду асимптотическую характеристику алгоритма, которую Дональд Кнут обозначил как большая "тета". Формально большая "О" означает верхнюю оценку. Например, число 7 является верхней оценкой числа 6, так же, как и числа 9, 12 и 65. Поэтому когда большинство из нас говорят об оценке роста функции, то имеют в виду как раз наименьшую верхнюю оценку, либо функцию, с помощью которой можно промоделировать как верхнюю, так и нижнюю оценку[3]. Профессор Кнут, "отец-основатель" области анализа алгоритмов, обозначил это через большую "тета" и дал следующее определение.

```
Если f(x) принадлежит множеству большой "тета" функций g(x),
то g(x) является как верхней, так и нижней границей функции f(x).
```

[3] Интересно, что нижняя оценка обозначается большой "омега". Она определяется так же, как и большое "О", за исключением того, что функция `g(x)` всегда меньше или равна `f(x)`. Большая "омега" менее полезна, чем большое "О", поскольку нахождение функции, меньшей, чем исследуемая функция, редко позволяет оценить асимптотическую характеристику алгоритма.

Тогда мы можем сказать, что `f(x)` — функция одного *порядка*, что и `g(x)`. Понятие большой "тета", или порядок алгоритма, является одной из самых основных математических характеристик, применяемых для анализа алгоритмов ядра Linux.

Следовательно, когда мы говорим о большом "О", то чаще всего имеем в виду наименьшее из возможных больших "О", т.е. большую "тета". Однако, если вам не нужно сдавать зачет профессору Кнуту, особо беспокоиться не о чем.

Временная сложность алгоритма

Снова рассмотрим пример из предыдущего раздела, связанный с подсчетом количества людей, находящихся в комнате. Предположим, мы можем считать со скоростью один человек в секунду. Следовательно, если в комнате находится 7 человек, то для их подсчета нам потребуется 7 секунд. В обобщенной форме это можно записать так: для подсчета всех `n` человек нам потребуется `n` секунд. Таким образом, можно сказать, что этот алгоритм принадлежит множеству `O(n)`. А что, если наша задача сводится к тому, чтобы выступить с речью перед всеми, кто находится в комнате? Поскольку выступление будет длиться одно и то же время, независимо от того, сколько людей (5 или 5000) находится в комнате, то эта задача относится к множеству `O(1)`. В табл. 6.1 приведены и другие часто используемые оценки сложности алгоритмов.

Таблица 6.1. Часто используемые оценки временной сложности алгоритмов

`O(g(x))`	Описание
`1`	Постоянная сложность (идеальный случай масштабируемости)
`log n`	Логарифмическая зависимость
`n`	Линейная зависимость
n^2	Квадратичная зависимость
n^3	Кубическая зависимость
2^n	Экспоненциальная зависимость
`n!`	Зависимость n-факториал

Как оценить временную сложность алгоритма, связанного со знакомством отдельного человека, находящегося в комнате, с каждым из остальных людей? С помощью какой из функций мы можем промоделировать этот алгоритм? Предположим, знакомство каждого человека занимает 30 секунд. За сколько времени можно познакомить друг с другом 10 человек? А что, если нам нужно познакомить друг с другом 100 человек? Понимание того, насколько эффективно алгоритм выполняет свою работу и насколько хорошо он масштабируется, является ключевым моментом в определении наилучшего алгоритма для выполнения конкретной работы.

Безусловно, вы должны избегать алгоритмов с временной сложностью `O(n!)` или `O(2n)`. Более того, обычно стараются заменить алгоритм с `O(n)` эквивалентным ему по функциональности алгоритмом с `O(log n)`. Тем не менее такое не всегда возможно, кроме того, не следует слепо принимать решения исключительно на основе большого "О". Давайте вспомним, что для данного `O(g(x))` существует постоянная `c`, на которую умножается `g(x)`. Поэтому возможны ситуации, когда алгоритмы с временной сложностью `O(1)` будут выполняться порядка трех часов. Очевидно, что три часа это всегда много,

независимо от того, насколько большим будет массив исходных данных, особенно если алгоритм с `O(n)` и несколькими исходными данными выполняется намного быстрее. Поэтому при сравнении эффективности алгоритмов не стоит забывать о размере массива исходных данных.

Следует всегда отдавать предпочтение менее сложным алгоритмам, однако не стоит забывать о снижении их эффективности при типичном изменении массива исходных данных. Не стоит слепо выполнять оптимизацию алгоритма, основываясь лишь на его уровне масштабируемости, — так вы никогда не добьетесь успеха!

Резюме

В этой главе мы рассмотрели довольно много обобщенных структур данных, которые используются разработчиками ядра Linux для реализации нужной функциональности, начиная с системного планировщика и заканчивая драйверами устройств. По мере того как мы будем продвигаться вглубь в изучении ядра Linux, материал этой главы вам будет полезен не раз. При написании собственного кода ядра всегда старайтесь использовать готовую инфраструктуру и не пытайтесь изобретать велосипед!

В этой главе мы также ознакомились с понятием алгоритмической сложности и средствами, с помощью которых ее можно оценить и выразить. Самым значимым из них является понятие большого “O”. На протяжении всей книги, а также в ядре Linux мы будем опираться на него при анализе того, насколько хорошо выбранные алгоритмы и компоненты ядра масштабируются в свете большого числа пользователей, процессов, процессоров, сетевых соединений и других постоянно растущих исходных данных.

7
Прерывания и их обработка

Одной из самых ответственных функций ядра любой операционной системы является управление аппаратными устройствами, которые подключены к вычислительной машине, — жесткими дисками, накопителями Blu-ray, клавиатурой, мышью, графическими 3D-процессорами и беспроводными устройствами. Частью этой работы является необходимость взаимодействия с отдельными устройствами машины. Поскольку процессоры вычислительной машины обычно работают во много раз быстрее, чем оборудование, с которым они должны взаимодействовать, то с точки зрения ядра получается неэффективным отправлять запросы и тратить время на ожидание ответа от существенно более медленного устройства. 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.

Как же может процессор работать с аппаратными устройствами и не замедлять при этом работу системы в целом? Одно из возможных решений этой проблемы — *периодический опрос* оборудования (polling). Ядро периодически может проверять состояние аппаратного устройства системы и соответствующим образом реагировать на него. Однако такой подход вносит дополнительные издержки, поскольку, 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 будет создание механизма, который позволит подать сигнал ядру о том, что нужно уделить внимание оборудованию. Такой механизм называется *прерыванием* (interrupt). В этой главе мы рассмотрим прерывания и способы их обработки в ядре, а также специальные функции, которые называются *обработчиками прерываний* (interrupt handlers).

Прерывания

Прерывания позволяют аппаратным устройствам послать сигнал процессору. Например, при нажатии клавиши на клавиатуре контроллер клавиатуры

(или другое устройство, которое управляет ею) генерирует специальный электрический сигнал процессору, чтобы привлечь внимание операционной системы и сообщить ей, что появились новые коды нажатых клавиш. Эти электрические сигналы и называются прерываниями. Получив сигнал прерывания, процессор сообщает операционной системе о том, что она может обработать новые данные. Аппаратные устройства генерируют сигналы 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 прерывание.

Физически прерывания вызываются специальными электрическими сигналами, которые генерируются устройствами и направляются на входные контакты микросхемы *контроллера прерываний* (interrupt controller). Это довольно простая микросхема, в задачи которой входит объединение сигналов прерывания, поступающих от нескольких устройств, и генерирование единственного сигнала прерывания для процессора. После получения сигнала прерывания от устройства контроллер прерываний посылает соответствующий сигнал процессору. Обнаружив этот сигнал на одном из контактов своей микросхемы, процессор прерывает выполнение текущей команды и переключается на обработку прерывания. После этого процессор извещает операционную систему о том, что произошло прерывание и операционная система может соответствующим образом обработать это прерывание.

Различным устройствам могут быть назначены отдельные сигналы прерывания, которые идентифицируются с помощью уникальных числовых значений, соответствующих каждому прерыванию. Таким образом, прерывание, поступившее от клавиатуры, можно легко отличить от прерывания, поступившего от жесткого диска. Это позволяет операционной системе различать прерывания и иметь информацию о том, какое аппаратное устройство вызвало данное прерывание. В свою очередь операционная система может обработать каждое прерывание с помощью соответствующего обработчика.

Сигналы прерывания, поступающие от устройств, часто называют *линиями запросов на прерывание* (interrupt request lines, IRQ). Каждой линии IRQ назначается соответствующий номер, который является восьмиразрядным двоичным числом. Например, в классическом ПК компании IBM нулевое IRQ соответствует сигналу таймера, а IRQ1 — прерыванию от клавиатуры. Однако не все номера прерываний жестко закреплены за соответствующими устройствами. Например, прерывания, связанные с устройствами шины PCI, как правило, назначаются динамически. На других аппаратных платформах, отличных от IBM PC, также предусмотрен аналогичный механизм для динамического назначения устройствам номеров прерываний. Основная идея состоит в том, что определенные номера прерываний назначены определенным устройствам и что у ядра есть вся эта информация. Чтобы привлечь внимание ядра, аппаратное устройство генерирует прерывание, сходное с тем, как если бы вы попросили кого-нибудь что-то сделать: *"Эй, мастер! Была нажата новая клавиша, и я жду, пока ты не прочитаешь ее код и не обработаешь поступившие данные!"*

Исключительные ситуации

В книгах, посвященных операционным системам, *исключительные ситуации* (exceptions) часто описываются вместе с прерываниями. В отличие от прерываний, они возникают синхронно с тактовым генератором процессора. И действительно, их часто называют *синхронными прерываниями* (synchronous interrupts). Исключительные ситуации генерируются самим процессором при выполнении машинных команд как реакция на ошибку в программе (например, деление на нуль) или как реакция на аварийную ситуацию, которая должна быть

обработана ядром (например, прерывание из-за отсутствия страницы (page fault)). Так как на большинстве аппаратных платформ исключительные ситуации обрабатываются аналогично обработке прерываний, 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 одной из предыдущих глав. Там было рассказано, как для аппаратной платформы x86 реализован механизм вызова системных функций на основе программных прерываний. При этом генерируется исключительная ситуация, которая заставляет операционную систему переключиться в режим ядра и в конечном итоге запустить нужный обработчик системной функции. Прерывания работают аналогичным образом, за исключением того, что они генерируются не программным, а аппаратным обеспечением.

Обработчики прерываний

Функция, которую запускает ядро в ответ на поступление сигнала определенного прерывания, называется *обработчиком прерывания* (interrupt handler), или *программой обслуживания прерывания* (*interrupt service routine*). Каждому устройству, которое может генерировать прерывания, в ядре соответствует свой обработчик прерывания. Например, одна функция обрабатывает прерывание от системного таймера, а другая — прерывания, сгенерированные клавиатурой. Обработчик прерывания для определенного 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 передавать информацию обработчику, 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 ниже в этой главе. Этот специальный контекст иногда называют *неделимым* (atomic), поскольку, как мы увидим чуть позже, код, выполняемый в нем, не может быть блокирован. В этой же книге мы будем называть его контекстом прерывания.

Поскольку прерывание может возникнуть в любой момент времени, соответственно, и обработчик прерывания может быть вызван когда угодно. Крайне важно, чтобы обработчик прерывания выполнялся очень быстро и возобновлял управление прерванного кода как можно скорее. 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 выполнить обработчик прерывания, — это отправить подтверждение устройству, что прерывание получено. Это как если бы вы попросили мастера обслужить прерывание, а он ответил: *"Я тебя слышу и скоро начну работу!"* Однако, как правило, в обработчике прерывания выполняется гораздо больше работы. В качестве примера рассмотрим обработчик прерывания от сетевой карты. Кроме отправки подтверждения о получении прерывания сетевому устройству, обработчик прерывания должен скопировать сетевые пакеты из памяти устройства в память компьютера, обработать их и отправить соответствующему стеку протоколов или соот-

ветствующей программе. Очевидно, что для этого требуется выполнить много работы, особенно если речь идет о современных сетевых устройствах Ethernet со скоростью передачи данных 10 Гбит/с.

Верхняя и нижняя половины

Очевидно, что два выдвинутых выше требования о том, что обработчик прерывания должен выполняться как можно быстрее *и* при этом делать много работы, являются противоречивыми. В связи с этим процесс обработки прерываний разбивается на две части, или половины. Обработчик прерывания относится к *верхней половине* (top half). Он запускается сразу после получения прерывания и выполняет только ту работу, которая критична к задержкам по времени, такую, как отправка подтверждения о получении прерывания или сброс аппаратного устройства. Работа, которую можно выполнить позже, откладывается до запуска *нижней* (или основной) *половины* (bottom half). Обработка нижней половины начинается позже, в более удобное время, когда все прерывания разрешены. В Linux предусмотрены различные механизмы реализации нижних половин. Все они рассмотрены в главе 8, “Нижняя половина обработчика и отложенные действия”.

Рассмотрим пример разделения обработчика прерывания на верхнюю и нижнюю половины на основе старой доброй сетевой платы. Когда сетевая плата получает прибывшие из сети пакеты, она должна уведомить ядро о том, что доступны новые данные. Причем это необходимо сделать как можно скорее, чтобы получить оптимальную пропускную способность и минимальное время задержки при передаче информации по сети. Поэтому немедленно генерируется прерывание: “*Эй, ядро! Есть свежие пакеты!*” В ответ ядро запускает зарегистрированный обработчик прерывания для сетевого адаптера.

Обработчик прерывания запускается, аппаратному обеспечению направляется подтверждение о получении прерывания, пакеты копируются в основную память, и после этого сетевой адаптер снова готов к получению новых пакетов. Эта часть работы является важной, критичной ко времени выполнения и специфичной для каждого типа оборудования. Как правило, ядро должно очень быстро скопировать сетевой пакет в основную память компьютера, поскольку буфер данных сетевой платы имеет небольшой фиксированный размер (по сравнению с размером основной памяти компьютера). Любая задержка с копированием пакета может привести к ситуации переполнения буфера, когда следующие входящие пакеты просто затрут содержимое буфера, в результате чего старый пакет будет потерян. После того как сетевые данные будут успешно скопированы в основную память компьютера, процесс обработки прерывания считается выполненным, и система возобновляет выполнение прерванной программы. Остальная часть обработки сетевого пакета выполняется позже — в нижней половине обработчика прерывания. В этой главе мы рассмотрим верхнюю половину обработчика прерываний, а в следующей — нижнюю.

Регистрация обработчика прерывания

Обработчики прерываний являются частью кода драйверов устройств, которые управляют определенным типом аппаратного обеспечения. С каждым устройством связан такой драйвер, и если устройство использует прерывания (а большинство использует), то драйвер должен выполнить регистрацию обработчика прерывания.

Драйвер может зарегистрировать обработчик прерывания, поступающего от заданной линии IRQ, с помощью функции `request_irq()`, которая определена в файле `<linux/interrupt.h>`.

```
/* request_irq: занимает указанную линию IRQ */
int request_irq(unsigned int irq,
                irq_handler_t handler,
                unsigned long flags,
                const char *name,
                void *dev)
```

В первом параметре, `irq`, указывается занимаемый номер прерывания. Для некоторых типов устройств, таких, как, например, стандартные устройства персонального компьютера, таймер и клавиатура, это значение, как правило, жестко закреплено. Для большинства других устройств это значение определяется программно путем *зондирования* (probed) или другим динамическим способом.

Во втором параметре, `handler`, передается указатель на функцию обработчика прерывания, которая обслуживает данное прерывание. Эта функция вызывается операционной системой в момент получения сигнала прерывания. Следует обратить внимание на специфический прототип функции-обработчика.

```
typedef irqreturn_t (*irq_handler_t)(int, void *);
```

Обратите внимание на специфический прототип этой функции. Ей передаются два параметра, а она возвращает значение типа `irqreturn_t`. Эту функцию мы рассмотрим в данной главе чуть позже.

Флаги обработчика прерываний

Третий параметр, `flags`, может быть равным нулю или содержать битовую маску флагов, описанных в файле `<linux/interrupt.h>`. Самые важные из них перечислены ниже.

- `IRQF_DISABLED`. Если данный флаг установлен, то во время выполнения обработчика прерываний все прерывания будут запрещены. Если флаг не установлен, то при выполнении обработчика прерываний все другие прерывания, кроме собственного, разрешены. В большинстве обработчиков прерываний данный флаг не устанавливается, поскольку запрещать все прерывания — это плохая идея. Данный флаг используется для критичных по времени обработчиков прерываний, которые должны выполняться очень быстро. Флаг `IRQF_DISABLED` заменил собой флаг `SA_INTERRUPT`, который в предыдущих версиях ядра Linux использовался для различия “быстрых” и “медленных” прерываний.
- `IRQF_SAMPLE_RANDOM`. Этот флаг указывает, что прерывания, сгенерированные данным устройством, должны вносить вклад в пул энтропии ядра. Последний обеспечивает генерацию истинно случайных чисел на основе различных случайных событий. Если этот флаг указан, то моменты времени, когда приходят прерывания, заносятся в пул в виде 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.

- IRQF_TIMER. Этот флаг указывает, что данный обработчик прерывания обслуживает системный таймер.
- IRQF_SHARED. Этот флаг указывает, что данный номер IRQ может совместно использоваться несколькими обработчиками прерываний. Его нужно использовать при регистрации обработчика прерываний, если последний использует одну и ту же линию IRQ с другими обработчиками прерываний. Если флаг сброшен, то для одной линии IRQ может существовать только один обработчик прерывания. Более подробно совместно используемые обработчики прерываний будут рассмотрены в следующем разделе.

Остальные параметры функции обработки прерывания

В четвертом параметре, name, указывается ASCII-строка, описывающая устройство, связанное с данным обработчиком прерывания. Например, для обработчика прерывания клавиатуры персонального компьютера это значение равно "keyboard". Текстовые имена устройств используются для взаимодействия с пользователями с помощью интерфейсов /proc/irq и /proc/interrupts, которые будут рассмотрены ниже в этой главе.

Пятый параметр, dev, используется в обработчиках прерываний, обслуживающих общие линии IRQ. При освобождении обработчика (этот процесс будет рассмотрен ниже) в параметре dev указывается уникальный идентификатор, который позволяет удалить только требуемый обработчик из заданной линии IRQ. Без этого параметра ядро не смогло бы определить, какой *именно* обработчик прерывания следует удалить с данной линии IRQ. Если линия IRQ используется монопольно, то в качестве параметра dev указывается значение NULL. Если линия IRQ используется совместно несколькими обработчиками прерываний, то вы должны указать для каждого из них уникальный идентификатор. Если ваше устройство не подключено к шине ISA, то, скорее всего, оно поддерживает совместно используемые линии IRQ. Этот параметр также передается обработчику прерывания при каждом его вызове. Чаще всего в качестве такого уникального идентификатора используется указатель на структуру, описывающую устройство. Дело в том, что значение этого указателя уникально для каждого устройства, к тому же в обработчике прерывания такой указатель может всегда пригодиться.

В случае успешного выполнения функция request_irq() возвращает нулевое значение. Ненулевое значение свидетельствует о возникшей ошибке. В последнем случае обработчик прерывания не регистрируется. Чаще всего встречается ошибка -EBUSY, указывающая на то, что выбранная линия IRQ уже используется в другом обработчике прерывания, и либо вы, либо автор другого обработчика при регистрации забыли указать флаг IRQF_SHARED.

Обратите внимание на то, что функция request_irq() может переходить в состояние ожидания (sleep). По этой причине ее нельзя вызывать из контекста прерывания или в других ситуациях, когда код не допускает блокировок. Распространенной ошибкой является вызов функции request_irq() в ситуациях, когда нельзя безопасно перейти в состояние ожидания. Это происходит отчасти из-за того, что действительно сразу непонятно, почему функция request_irq() должна чего-то ожидать. Дело в том, что при регистрации обработчика происходит добавление информации о линии IRQ в каталоге /proc/irq. Для создания новых элементов в файловой системе procfs используется функция proc_mkdir(). Эта функция вызывает функцию proc_create() для создания новых элементов файловой системы procfs, которая в свою очередь вызывает функ-

цию kmalloc() для выделения под них памяти. Как будет показано в главе 12, "Управление памятью", функция kmalloc() может переходить в состояние ожидания. Вот так вот!

Пример обработчика прерывания

Для захвата линии IRQ и регистрации обработчика прерываний в драйвере устройства используется функция request_irq(), как показано ниже.

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

В этом примере в параметре irqn указывается запрашиваемый номер IRQ. Параметр my_interrupt определяет обработчик этой линии IRQ, которая может совместно использоваться, поскольку указан флаг IRQF_SHARED. Имя устройства — "my_device", а в параметре my_dev передается значение идентификатора устройства dev. В случае ошибки выводится сообщение о ней и выполняется возврат в вызывающую программу. Если функция регистрации возвращает нулевое значение, то обработчик прерывания успешно инсталлирован. С этого момента обработчик прерывания будет вызываться в ответ на приходящие прерывания. Важно произвести инициализацию оборудования и регистрацию обработчика прерывания в правильной последовательности, чтобы исключить ситуацию, когда обработчик прерывания будет вызываться до момента инициализации устройства.

Освобождение обработчика прерывания

При выгрузке драйвера устройства из памяти компьютера необходимо отменить регистрацию обработчика прерывания и по возможности заблокировать линию IRQ. Для этого вызывается приведенная ниже функция.

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

Если указанная линия IRQ используется монопольно, то эта функция удаляет обработчик прерывания и блокирует линию IRQ. Если линия IRQ используется совместно с другими обработчиками прерываний, то удаляется обработчик, соответствующий параметру dev. Линия IRQ блокируется только тогда, когда будет удален последний обработчик прерывания. Теперь вы должны понимать, почему так важно передавать уникальное значение параметра dev. При совместном использовании линии IRQ уникальный идентификатор требуется для того, чтобы отличать друг от друга различные обработчики, связанные с одним и тем же номером IRQ. Это позволяет в функции free_irq() удалять нужный обработчик. В любом случае (совместного или монопольного использования линии IRQ), если параметр dev не равен значению NULL, то он должен соответствовать тому обработчику, который удаляется. Вызов функции free_irq() должен выполняться из контекста процесса.

Функции для регистрации и удаления обработчиков прерываний перечислены в табл. 7.1.

Таблица 7.1. Функции управления регистрацией прерываний

Функция	Описание
request_irq()	Зарегистрировать заданный обработчик прерывания для заданной линии IRQ
free_irq()	Удалить указанный обработчик прерывания. Если с линией IRQ больше не связан ни один обработчик, то она блокируется

Написание обработчика прерывания

Приведенное ниже описание является типичным для обработчика прерывания.

```
static irqreturn_t intr_handler(int irq, void *dev)
```

Обратите внимание на то, что оно соответствует прототипу второго аргумента `handler`, который передается функции `request_irq()`. В первом параметре, `irq`, задается числовое значение номера линии запроса на прерывание, которую будет обслуживать обработчик. Несмотря на то что это значение передается обработчику прерываний, оно используется очень редко, в основном в диагностических сообщениях, выводимых в системный журнал. В версиях ядра Linux, предшествовавших версии 2.0, параметра `dev` не было. Поэтому значение параметра `irq` использовалось для того, чтобы можно было различать устройства, обслуживаемые одним драйвером и, следовательно, одним обработчиком прерываний. В качестве примера можно привести компьютер, оснащенный несколькими контроллерами жесткого диска одного типа.

Второй параметр, `dev`, — это уникальный идентификатор устройства, совпадающий со значением параметра `dev`, который был передан функции `request_irq()` при регистрации обработчика прерываний. Если значение данного параметра будет уникально (а это необходимо для работы с общими номерами IRQ), то его можно использовать как идентификатор, позволяющий отличать друг от друга различные устройства, которые потенциально могут обслуживаться одним обработчиком прерывания. В связи с этим в параметре `dev` передается указатель на структуру устройства, которая используется в обработчике прерывания. Поскольку такие структуры создаются для каждого устройства в системе и являются уникальными (кроме того, они необходимы при обработке прерывания), то обычно их адрес и передается обработчику в качестве параметра `dev`.

Функция обработчика прерывания может возвращать одно из двух значений специального типа: `irqreturn_t`: `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, в самом деле вызвало прерывание. Кроме того, можно использовать макрос `IRQ_RETVAL(val)`. Если значение параметра `val` не равно нулю, то макрос возвращает значение `IRQ_HANDLED`, иначе возвращается значение, равное `IRQ_NONE`. Эти специальные значения позволяют проинформировать ядро о том, генерирует ли устройство паразитные (т.е. необрабатываемые) прерывания. Если все обработчики прерывания, которые обслуживают данную линию IRQ, 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 возможности. До серии ядер 2.6 возвращаемое значение обработчика прерывания описывалось как `void`. В коде новых драйверов можно просто переопределить тип `irqreturn_t` в `void`, заменить различные возвращаемые значения пустыми операциями, и драйверы могут работать с ядрами серии 2.4 без дальнейшей модификации. Обработчик прерывания, как правило, помечается как `static`, поскольку он никогда не вызывается непосредственно из других файлов кода.

Выполняемые обработчиком прерывания функции зависят только от устройства и тех причин, по которым это устройство генерирует прерывания. В самом простом случае обра-

ботчик прерывания должен отправить устройству подтверждение о том, что прерывание получено. Для более сложных устройств в обработчике прерывания необходимо дополнительно отправить и принять данные, а также выполнить другую более сложную работу. Как уже упоминалось, сложная работа должна по возможности выполняться в нижней половине обработчика прерывания, которая будет рассмотрена в следующей главе.

> **Реентерабельность и обработчики прерываний**
>
> Обработчики прерываний в операционной системе Linux не обязательно должны быть реентерабельными (повторно входимыми). Когда выполняется некоторый обработчик прерывания, соответствующая линия запроса на прерывание маскируется на всех процессорах, что предотвращает возможность приема другого запроса на прерывание с этой линии IRQ. Обычно все остальные прерывания в этот момент разрешены, поэтому другие прерывания могут обслуживаться, тогда как текущая линия IRQ всегда замаскирована. Следовательно, никакой обработчик прерывания никогда не может быть вызван параллельно самому себе для обработки вложенных запросов на прерывание. Данный факт позволяет значительно упростить написание обработчиков прерываний.

Обработчики общих запросов на прерывание

Обработчики прерываний данного типа регистрируются и выполняются практически так же, как обработчики монопольных линий IRQ. Ниже описаны их основные отличия.

- При вызове функции `request_irq()` нужно установить значение флага `IRQF_SHARED`, который передается в параметре `flags`.
- Значение параметра `dev` этой же функции должно быть уникальным для каждого зарегистрированного обработчика прерываний. Для этого достаточно передать указатель на любую структуру, описывающую данное устройство. Обычно в качестве значения параметра `dev` выбирается указатель на структуру `device`, поскольку она уникальна для каждого устройства и, кроме того, может понадобиться в обработчике прерывания. Для обработчиков общих запросов на прерывание значение параметра `dev` *не может* быть равно `NULL`!
- В обработчике прерывания нужно определить, 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 IRQ является источником прерывания.

Все драйверы устройств, рассчитанные на работу с общей линией IRQ, должны удовлетворять перечисленным выше требованиям. Если хотя бы одно из устройств не делает это корректно, то все остальные устройства также не смогут совместно использовать общую линию IRQ. Если при вызове функции `request_irq()` указан флаг `IRQF_SHARED`, то обработчик прерывания может быть зарегистрирован только в случае, если данная линия IRQ свободна либо если при регистрации всех остальных обработчиков прерывания также был указан флаг `IRQF_SHARED`. Несмотря на это, при регистрации обработчиков общих запросов на прерывание можно произвольным образом устанавливать значение флага `IRQF_DISABLED`.

Когда ядро получает запрос на прерывание, оно последовательно вызывает все обработчики, зарегистрированные для данной линии IRQ. 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 текущее 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 поддерживает данную возможность.

Пример настоящего обработчика прерывания

Рассмотрим настоящий обработчик прерывания, который используется в драйвере *часов реального времени* (real-time clock, RTC), находящегося в файле `drivers/char/rtc.c`. Устройство RTC есть во многих вычислительных системах, включая персональные компьютеры (PC). Это отдельное от системного таймера устройство используется для установки системных часов, для подачи *сигналов таймера* (alarm) или для реализации генераторов *периодических сигналов* (periodic timer). В большинстве аппаратных платформ установка системных часов обычно выполняется путем записи значений в специальный регистр ввода-вывода или в несколько регистров, определяемых диапазоном номеров портов. Подача сигналов таймера или генератора периодических сигналов в ядро обычно реализуется через прерывания. В этом случае прерывание аналогично будильнику обычных часов. Получение прерывания свидетельствует о том, что будильник зазвенел.

При загрузке драйвера устройства RTC вызывается функция `rtc_init()` для инициализации драйвера. Одна из ее задач — регистрация обработчика прерывания, как показано ниже.

```
/* Регистрация обработчика rtc_interrupt на rtc_irq */
if (request_irq(rtc_irq, rtc_interrupt, IRQF_SHARED, "rtc",
                (void *)&rtc_port)) {
   printk(KERN_ERR "rtc: cannot register IRQ %d\n", rtc_irq);
   return -EIO;
}
```

В данном примере значение номера IRQ хранится в переменной `rtc_irq`. Ее значение зависит от конкретной аппаратной платформы. Для платформы PC устройство RTC подключено к IRQ 8. Во втором параметре указывается обработчик прерывания `rtc_interrupt`. Благодаря установке флага `IRQF_SHARED` этот обработчик по мере необходимости может совместно с другими обработчиками использовать общую линию IRQ. Значение четвертого параметра определяет имя устройства — `"rtc"`. Поскольку устройство RTC может использовать общую линию IRQ, в пятом параметре передается уникальное для устройства значение параметра `dev`.

И наконец, собственно сам обработчик прерывания:

```
static irqreturn_t rtc_interrupt(int irq, void *dev)
{
   /*
    * Прерывание может произойти от таймера, при завершении обновления
    * или быть периодическим прерыванием.
    * Состояние (причина) прерывания хранится в самом
    * младшем байте, а общее количество прерываний, полученных
    * с момента последнего чтения, — в оставшейся 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)
       mod_timer(&rtc_irq_timer, jiffies + HZ/rtc_freq + 2*HZ/100);

    spin_unlock(&rtc_lock);

   /*
    * Теперь выполним оставшуюся часть работы
    */
    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 спин-блокировок: первая группа вызовов функций гарантирует, что к переменной `rtc_irq_data` не будет конкурентных обращений со стороны других процессоров на SMP-машине. Вторая группа вызовов функций защищает в аналогичной ситуации поля структуры `rtc_callback`. Различные типы блокировок будут описаны в главе 10, “Средства синхронизации ядра”.

Переменная `rtc_irq_data` имеет тип `unsigned long`. В ней хранится информация об устройстве RTC, которая обновляется при поступлении каждого прерывания и отражает состояние прерывания.

Далее, если запущен генератор периодических сигналов, обновляется значение системного таймера с помощью функции `mod_timer()`. Таймеры будут описаны в главе 11, “Таймеры и управление временем”.

В последней части кода, находящейся после комментария `"Теперь выполним оставшуюся часть работы"`, запускается функция обратного вызова, если таковая установлена. Драйвер RTC позволяет устанавливать функцию обратного вызова, которая может быть зарегистрирована из вне. В результате она будет запускаться при каждом прерывании, приходящем от устройства RTC.

В конце функция обработки прерывания возвращает значение `IRQ_HANDLED`, чтобы указать, что прерывание от данного устройства корректно обработано. Поскольку данный обработчик прерывания не поддерживает общих запросов на прерывание и в устройстве RTC не существует механизма, позволяющего обнаружить паразитные прерывания, этот обработчик всегда возвращает значение `IRQ_HANDLED`.

Контекст прерывания

При выполнении обработчика прерывания ядро находится в *контексте прерывания*. Напомним, что контекст процесса — это режим, в котором работает ядро, выполняя работу от имени процесса, например выполнение вызова системной функции или потока ядра. В контексте процесса макрос `current` возвращает указатель на связанную с ним задачу. Более того, поскольку при вызове функций ядра не происходит переключение контекста процесса, система может приостановить выполнение текущей задачи и переключиться на выполнение другой.

В отличие от контекста процесса контекст прерывания не связан ни с одним процессом. Поэтому макрос `current` не имеет особого смысла, хотя он и возвращает указатель на процесс, выполнение которого было прервано. Поскольку в контексте прерывания нет сопровождающего его процесса, этот контекст нельзя перевести в состояние ожидания. Дело в том, что тогда нарушилась бы работа системного планировщика: нельзя перепланировать контекст, в котором нет задачи. 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, а это, в свою очередь, ограничивает набор функций, которые можно использовать в обработчиках прерываний.

Контекст прерывания является критичным ко времени выполнения, поскольку на время обработки прерывания выполнение некоторого программного кода прекращается. Код же самого обработчика прерывания должен быть простой и быстрый. Использование в нем циклов проверки *состояния чего-либо* (busy loop) крайне нежелательно, хотя и возможно. 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 прерываний должны быть как можно более быстрыми и простыми. Максимально возможную часть работы необходимо изъять из обработчика прерывания и переложить на его нижнюю половину, которая выполняется в более подходящее время.

Во время конфигурации системы можно определить размер стека обработчика прерывания. Исторически так сложилось, что обработчик прерывания не имел собственного стека. Вместо этого он должен был использовать стек ядра прерванного процесса[1]. Стандартный размер стека ядра составляет две страницы памяти, что обычно соответствует 8 Кбайт для 32-разрядных аппаратных платформ и 16 Кбайт — для 64-разрядных. Поскольку подразумевается, что обработчики прерываний используют общий стек, они должны очень экономно расходовать память в этом стеке. Конечно, размер стека ядра будет всегда ограничен, поэтому при разработке любого кода ядра следует всегда иметь это в виду.

В ранних версиях ядер серии 2.6 была введена возможность уменьшить размер стека ядра от двух до одной страницы памяти, что равно 4 Кбайт для 32-разрядных аппаратных платформ. Это позволило резко уменьшить затраты памяти, потому что ранее каждому процессу требовались две страницы памяти ядра, которая не могла быть вытеснена на диск. Чтобы можно было работать со стеком уменьшенного размера, каждому обработчику прерывания выделили свой стек, отдельный для каждого процессора, размером

[1] В системе всегда выполняется какой-нибудь процесс. Если активных задач нет, то система переключается на выполнение *холостой задачи* (idle task).

в одну страницу памяти. Этот стек назвали *стеком прерывания*. Хотя общий размер стека прерывания и равен половине первоначально размера совместно используемого стека, тем не менее в результате выходит, что суммарный размер стека получается бо́льшим, потому что теперь 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. Поэтому всегда нужно использовать минимально возможное количество памяти в стеке.

Реализация системы обработки прерываний

Вы, наверное, уже догадываетесь, что реализация системы обработки прерываний в операционной системе Linux зависит от используемой аппаратной платформы: типа процессора и контроллера прерываний, особенностей аппаратной платформы и конструкции самой вычислительной машины.

На рис. 7.1 показана схема прохождения запроса прерывания, сгенерированного аппаратным устройством, по ядру системы.

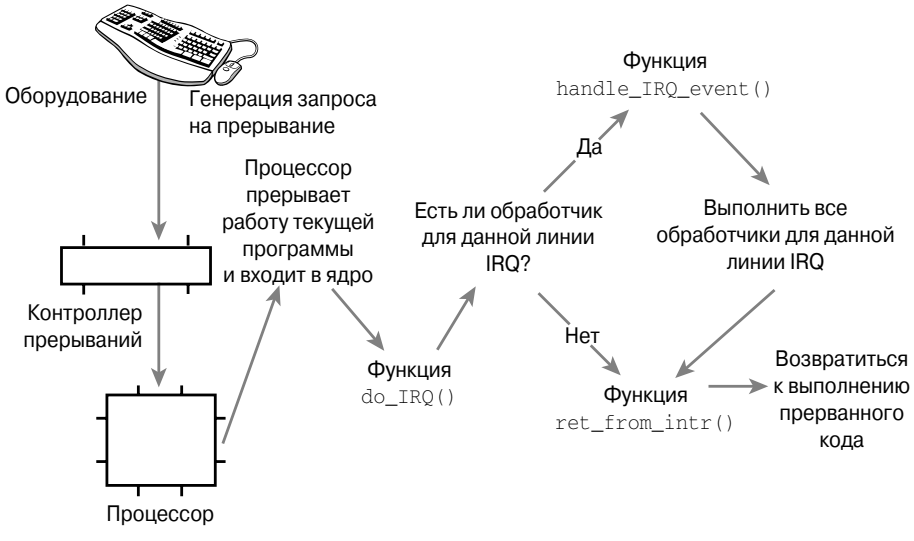


Рис. 7.1. Схема прохождения запроса на прерывание от аппаратного обеспечения до ядра системы

Устройство инициирует прерывание путем отправки электрического сигнала контроллеру прерываний по аппаратной шине. Если соответствующая линия запроса на прерывание (IRQ) не запрещена (линия может быть в данный момент времени замаскирована), то контроллер прерываний отправляет сигнал прерывания процессору. На большинстве аппаратных платформ это осуществляется путем подачи сигнала на специальный вход микросхемы центрального процессора. Если прерывания в процессоре разрешены (иногда может случиться, что они запрещены), то процессор завершает выполнение текущей машинной команды, запрещает прием новых прерываний, осуществляет переход на специальный предопределенный адрес в памяти и начинает выполнять программный код, который находится по этому адресу. Этот предопределенный адрес памяти был заранее сконфигурирован ядром и является *точкой входа* в обработчики прерываний.

Вход в систему обработки прерываний в ядре начинается со строго определенной точки, так же как и при вызове системных функций вход в ядро выполняется через преопределенный обработчик исключительных ситуаций. Для каждой линии IRQ в памяти машины предусмотрена своя уникальная точка входа, куда и переходит процессор, после чего он начинает выполнять расположенный там код. Именно таким образом ядро узнает о номере IRQ произошедшего прерывания. В точке входа сначала сохраняется в стеке значение номера прерывания и значения всех регистров процессора, которые относились к прерванной задаче. После этого ядро вызывает функцию `do_IRQ()`. Далее, начиная с этого момента, почти весь код обработчика прерываний написан на языке программирования C, хотя он все же и остается зависимым от аппаратной платформы.

Функция `do_IRQ()` определена следующим образом:

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

Поскольку соглашение о вызовах функций в языке C предусматривает размещение их аргументов в вершине стека, первоначальные значения всех регистров процессора, которые были сохранены ассемблерной программой в точке входа, передаются в функцию `do_IRQ()` через структуру `pt_regs`. Так как в этой структуре также сохраняется значение номера прерывания, функция `do_IRQ()` может его легко извлечь. После вычисления значения номера IRQ функция `do_IRQ()` отправляет уведомление о получении прерывания и запрещает генерирование прерываний по данной линии. Для обычных машин платформы PC эти действия выполняются с помощью функции `mask_and_ack_8295A()`.

Далее в функции `do_IRQ()` выполняется проверка, что для данной линии IRQ зарегистрирован корректный обработчик прерывания, что этот обработчик разрешен и не выполняется в данный момент. Если все эти условия выполняются, то вызывается функция `handle_IRQ_event()`, определенная в файле `kernel/irq/handler.c`, которая запускает установленные для данной линии IRQ обработчики прерывания.

```
/**
  * handle_IRQ_event - обработчик цепочки действий прерывания для данного
irq
  * @irq: номер прерывания
  * @action: цепочка действий прерывания для данного irq
  *
  * Обрабатывает цепочку действий для данного irq-события
  */
irqreturn_t handle_IRQ_event(unsigned int irq, struct irqaction *action)
{
   irqreturn_t ret, retval = IRQ_NONE;
   unsigned int status = 0;

   if (!(action->flags & IRQF_DISABLED))
      local_irq_enable_in_hardirq();

   do {
      trace_irq_handler_entry(irq, action);
      ret = action->handler(irq, action->dev_id);
      trace_irq_handler_exit(irq, action, ret);

      switch (ret) {
         case IRQ_WAKE_THREAD:
         /*
          * Установим признак того, что прерывание обработано,
          * чтобы не запускалась проверка паразитных прерываний
```

```
            */
              ret = IRQ_HANDLED;

            /*
             * Перехватим драйверы, которые вернули WAKE_THREAD, но
             * при этом не инсталлировали функцию thread
             */
              if (unlikely(!action->thread_fn)) {
                 warn_no_thread(irq, action);
                 break;
               }

            /*
             * Активизируем поток обработчика для данного действия.
             * В случае если поток аварийно завершился и был удален из
             * системы, просто сделаем вид, что прерывание обработано.
             * В обработчике аппаратного irq, который был вызван до нас,
             * прерывания от этого устройства были запрещены, поэтому
             * irq-шторма не будет.
             */
              if (likely(!test_bit(IRQTF_DIED,
                           &action->thread_flags))) {
                 set_bit(IRQTF_RUNTHREAD, &action->thread_flags);
                 wake_up_process(action->thread);
              }

            /* После выхода из цикла добавим элемент случайности */
            case IRQ_HANDLED:
               status |= action->flags;
               break;

            default:
               break;
         }

         retval |= ret;
         action = action->next;
      } while (action);

      if (status & IRQF_SAMPLE_RANDOM)
         add_interrupt_randomness(irq);

      local_irq_disable();

      return retval;
}
```

Поскольку процессор запретил прерывания, сначала они разрешаются, если при регистрации обработчика прерываний не был указан флаг `IRQF_DISABLED`. Напомним, что флаг `SA_INTERRUPT` указывает, что обработчик должен выполняться при всех запрещенных прерываниях. Далее в цикле вызываются все потенциальные обработчики прерываний. Если эта линия IRQ используется монопольно, то цикл заканчивается после первой итерации. В противном случае будут вызваны все обработчики. После этого вызывается функция `add_interrupt_randomness()`, если при регистрации обработчика был указан флаг `IRQF_SAMPLE_RANDOM`. Данная функция использует временные характеристики прерывания, чтобы сгенерировать значение энтропии для генератора случайных чисел. В конце обработчика прерывания снова запрещаются (для работы функции `do_IRQ()` требуется, чтобы прерывания были запрещены) и на этом работа

функции завершается. После возврата в функцию do_IRQ() выполняется очистка стека и возврат к первоначальной точке входа, откуда осуществляется переход к функции ret_from_intr().

Функция ret_from_intr() и код входа в прерывание написаны на языке ассемблера. В этой функции проверяется, есть ли ожидающий запрос на перепланирование процессов. В главе 4, "Системный планировщик и диспетчеризация процессов", мы говорили о том, что для этого должен быть установлен 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 код ядра не может быть безопасно вытеснен. После возврата из функции schedule() или если нет никакой ожидающей работы, восстанавливаются исходные значения регистров процессора и ядро продолжает свою работу там, где оно было прервано.

Для платформы x86 подпрограммы, написанные на языке ассемблера, находятся в файле arch/x86/kernel/entry_64.S (entry_32.S для 32-разрядного режима x86), а соответствующие функции на языке C — в файле arch/x86/kernel/irq.c. Для других поддерживаемых аппаратных платформ имеются аналогичные файлы.

Интерфейс /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 в файловой системе procfs приводит к вызовам функций ядра, которые имитируют чтение или запись обычных файлов. Характерный пример — 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-hcd, eth0
5:    0         XT-PIC   EMU10K1
12:   85077     XT-PIC   uhci-hcd
15:   24571     XT-PIC   aic7xxx
NMI:  0
LOC:  3602236
ERR:  0
```

В первом столбце указывается номер линии IRQ. В нашей системе присутствуют линии запросов на прерывания с номерами 0–2, 4, 5, 12 и 15. Линии IRQ, для которых не инсталлирован обработчик, не отображаются. Во втором столбце указывается общее количество полученных прерываний по данной линии IRQ. На многопроцессорных платформах здесь будет отображено несколько столбцов (по числу процессоров), в которых указывается общее количество прерываний для каждого процессора. Как видите, в сис-

теме произошло 3 602 371 прерывание от таймера[2] и ни одного прерывания от звуковой платы EMU10K1 (это означает, что плата попросту не использовалась с момента перезагрузки системы). В третьем столбце указывается контроллер прерываний, обрабатывающий данное прерывание. Значение XT-PIC соответствует стандартному программируемому контроллеру прерываний компьютера IBM PC. Для систем, оснащенных устройством I/O APIC, для большинства прерываний в качестве контроллера прерываний будет указано значение IO-APIC-level или IO-APIC-edge. И наконец, в последнем столбце отображается имя устройства, связанного с прерыванием. Это имя передается в виде параметра devname функции request_irq(), как говорилось выше. Если какая-то из линий IRQ совместно используется несколькими устройствами (как IRQ4 в нашем примере), то в последнем столбце будут перечислены имена всех зарегистрированных для данной линии IRQ устройств.

Для тех, кому интересно, код файловой системы procfs расположен в основном в каталоге fs/proc. Функция, которая обеспечивает работу интерфейса /proc/interrupts, называется show_interrupts() и является зависимой от аппаратной платформы, что, в общем-то, и не удивительно.

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 запроса на прерывание для всей машины. Все эти функции зависят от аппаратной платформы и находятся в файлах <asm/system.h> и <asm/irq.h>. Полный список интерфейсов приведен ниже, в табл. 7.2.

Причины, по которым иногда нужно управлять системой обработки прерываний, в основном связаны с обеспечением синхронизации. Путем запрещения прерываний можно гарантировать, что обработчик прерывания не вытеснит текущий выполняемый код. Более того, запрещение прерываний также запрещает и вытеснение кода ядра. Однако ни запрещение генерации прерываний, ни запрещение вытеснения кода ядра не дает никакой защиты от конкурентного обращения со стороны других процессоров. Поскольку операционная система Linux поддерживает многопроцессорные системы, в большинстве случаев код ядра должен захватить некоторую блокировку, чтобы предотвратить одновременный доступ другого процессора к совместно используемым данным. Эти блокировки обычно захватываются в комбинации с запрещением прерываний на текущем процессоре. Блокировка обеспечивает защиту от доступа со стороны другого процессора, а запрещение прерываний — от возможного одновременного доступа из обработчика прерывания. В главе 9, “Общие сведения о синхронизации кода ядра”, и главе 10, “Средства синхронизации ядра”, обсуждаются различные аспекты проблем синхронизации и их решение. Тем не менее изучение интерфейсов ядра для управления прерываниями очень важно.

[2] В качестве упражнения после прочтения главы 11, “Таймеры и управление временем”, попробуйте подсчитать, как долго система находится в работоспособном состоянии (в секундах), зная количество прерываний, прошедших с момента перезагрузки системы и частоту колебаний таймера в Гц.

Запрещение и разрешение прерываний

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

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

Эти функции обычно реализованы в виде одной команды на языке ассемблера (что, конечно же, зависит от аппаратной платформы). В самом деле, для платформы x86 функция `local_irq_disable()` представляет собой всего одну машинную команду `cli`, а функция `local_irq_enable()` — команду `sti`. Ассемблерные команды `cli` (Clear interrupt flag) и `sti` (Set interrupt flag) соответственно сбрасывают и устанавливают флажок разрешения прерывания в регистре флагов процессора. Другими словами, они запрещают или разрешают генерацию прерываний на вызвавшем их процессоре.

Функцию `local_irq_disable()` довольно опасно использовать в случае, если *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 восстановить состояние системы обработки прерываний 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, представьте себе, что приведенный выше фрагмент кода является частью большой функции. Предположим также, что 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 проще сохранять состояние системы обработки прерываний перед запрещением прерываний. Тогда, вместо вызова функции разрешения прерываний, вам нужно просто восстановить первоначальное состояние системы обработки прерываний, как показано ниже

```
unsigned long flags;
local_irq_save(flags); /* Прерывания запрещены */
/* ... */
local_irq_restore(flags); /* Восстановлено первоначальное состояние
                           * системы обработки прерываний */
```

Следует отметить, что эти функции, по крайней мере частично, реализованы в виде макросов. Поэтому передача параметра `flags` (а он должен быть определен как `unsigned long`) выглядит как передача по значению. Этот параметр содержит зависящие от аппаратной платформы данные, определяющие состояние системы прерываний. Поскольку, по крайней мере для одной поддерживаемой аппаратной платформы (SPARC), в этой переменной хранится информация о стеке, параметр `flags` *нельзя* передавать в другие функции (т.е. он должен оставаться в одном и том же стековом фрейме). По этой причине вызовы функций сохранения и восстановления состояния системы прерываний должны выполняться в теле одной функции.

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

Больше нет глобальной функции `cli()`

В прежних версиях ядра была функция, с помощью которой можно было запретить прерывания на *всех* процессорах системы. Более того, если какой-либо процессор вызывал эту функцию, то для продолжения работы он должен был ждать, пока прерывания не будут разрешены. Эта функция называлась `cli()`, а соответствующая ей разрешающая функция — `sti()`. Их имена характерны для платформы x86, хотя функции были доступны для всех аппаратных платформ. Эти функции были изъяты во время разработки ядер серии 2.5, и, следовательно, теперь во всех операциях по синхронизации обработки прерываний должна использоваться комбинация функций по управлению локальными прерываниями и функций работы со спин-блокировками (они рассматриваются в главе 9, "Общие сведения о синхронизации кода ядра"). Следовательно, если раньше для того, чтобы получить монопольный доступ к совместно используемым данным, достаточно было в коде лишь глобально запретить прерывания, то теперь нужно 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()` гарантировала, что ни один из обработчиков прерываний (в том числе и другие экземпляры текущего обработчика) не будет выполняться. 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 вынуждает разработчиков драйверов использовать реальные блокировки. 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 заданной линии IRQ

В предыдущем разделе были рассмотрены функции, которые позволяют запретить генерирование всех прерываний на определенном процессоре. В некоторых случаях может понадобиться запретить прерывание, поступающее от *определенной* линии IRQ для *всей* системы. Это называется *маскированием* линии запроса на прерывание. Например, может потребоваться запретить прерывание, поступающее от некоторого устройства, перед изменением его состояния. Для этой цели в операционной системе Linux предусмотрены 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 IRQ в контроллере прерываний. В результате запрещается генерация прерывания от данной линии для *всех* процессоров 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, для вызывающей программы гарантируется не только то, что прерывания с данной линии IRQ не будут генерироваться, но и то, что не будет выполняться ни один обработчик данного прерывания. Функция `disable_irq_nosync()` не ждет, пока текущий обработчик прерывания завершит свою работу, а возвращается сразу.

Функция synchronize_irq() ожидает, пока указанный обработчик прерывания завершит свою работу, если он, конечно, был запущен.

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

Все эти три функции могут быть вызваны как из контекста прерывания, так и из контекста процесса, поскольку они не переходят в состояние ожидания. Однако при их вызове из контекста прерывания следует быть осмотрительным! Например, не стоит размаскировать линию прерывания во время выполнения обработчика прерывания (вспомним, что линия запроса на прерывание обработчика, который в данный момент выполняется, является замаскированной).

В высшей степени неразумно запрещать линию IRQ, которая совместно используется в нескольких обработчиках прерываний. Это вызовет запрет генерации прерываний от *всех* устройств, использующих эту линию. Поэтому в драйверах новых устройств не рекомендуется использовать описываемые в этом разделе интерфейсы[3]. Так как устройства PCI должны согласно спецификации поддерживать общие линии IRQ, в их драйверах эти интерфейсы не должны использоваться вообще. Поэтому функция disable_irq() и связанные с ней обычно используются в драйверах 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 irqs_disabled(), определенный в файле <asm/system.h>, возвращает ненулевое значение, если обработка прерываний на локальном процессоре запрещена. В противном случае возвращается нуль.

Для проверки текущего контекста ядра используются два приведенных ниже макроса, определенных в файле <linux/hardirq.h>.

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

Самый полезный из них — первый. Он возвращает ненулевое значение, если ядро в данный момент выполняет любой тип обработки прерываний: либо обработчик прерывания, либо его нижнюю половину. Макрос in_irq() возвращает ненулевое значение, только если ядро выполняет исключительно обработчик прерывания.

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

Чаще всего нужно проверить, не выполняется ли код в контексте процесса. Это означает, что нужно убедиться в том, что код выполняется *не* в контексте прерывания. Такая задача возникает перед выполнением в коде действий, разрешенных только в контексте процесса, например замораживание задачи. Если макрос `in_interrupt()` возвращает нуль, ядро находится в контексте процесса.

Несмотря на то что выбранные имена функций и макросов часто сбивают с толку и мало что говорят о выполняемых ими действиях, все же в табл. 7.2 приведено описание методов управления прерываниями.

Таблица 7.2. Методы управления прерываниями

Метод	Описание
`local_irq_disable()`	Запретить генерацию прерываний на локальном процессоре
`local_irq_enable()`	Разрешить генерацию прерываний на локальном процессоре
`local_irq_save()`	Сохранить текущее состояние системы обработки прерываний на локальном процессоре и запретить прерывания
`local_irq_restore()`	Восстановить указанное состояние системы прерываний на локальном процессоре
`disable_irq()`	Замаскировать указанную линию IRQ с гарантией, что после возврата из этой функции не будет выполняться ни один обработчик прерывания данной линии
`disable_irq_nosync()`	Замаскировать указанную линию IRQ
`enable_irq()`	Размаскировать указанную линию IRQ
`irqs_disabled()`	Возвратить ненулевое значение, если запрещена генерация прерываний на локальном процессоре, в противном случае возвращается нуль
`in_interrupt()`	Возвратить ненулевое значение, если код выполняется в контексте прерывания, и нуль — если в контексте процесса
`in_irq()`	Возвратить ненулевое значение, если код выполняется в обработчике прерывания, и нуль — в противном случае

Резюме

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

В большинстве современных устройств прерывания используются в качестве механизма взаимодействия устройства с операционной системой. Драйвер устройства, который управляет некоторым оборудованием, должен зарегистрировать обработчик прерывания, чтобы отвечать на эти прерывания и обрабатывать их. В обработчике прерывания устройству отправляется подтверждение о получении прерывания, выполняется инициализация оборудования, копирование данных из памяти устройства в память системы и, наоборот, обработка аппаратных запросов и отправка ответов на них.

В ядре предусмотрен ряд интерфейсов для регистрации и удаления обработчиков прерываний, запрещения прерываний, маскирования линий IRQ и проверки состояния системы прерываний. В табл. 7.2 приведен обзор некоторых из этих функций.

Поскольку прерывания прерывают выполнение другого кода (кода процессов, кода ядра и даже кода других обработчиков прерываний), их обработчики должны выполняться максимально быстро. Тем не менее в обработчике прерывания часто приходится выполнять много работы. Для достижения компромисса между большим количеством работы и необходимостью быстрого выполнения обработка прерывания делится на две половины. Собственно обработчик прерывания, рассмотренный в этой главе, относится к верхней половине. В следующей главе мы рассмотрим нижнюю половину процесса обработки прерывания.

8

Нижняя половина обработчика и отложенные действия

В предыдущей главе были рассмотрены обработчики прерываний — механизм ядра, предназначенный для управления аппаратными прерываниями. Разумеется, обработчики прерываний очень важны и являются необходимой частью ядра любой операционной системы. Но в связи с некоторыми ограничениями они представляют собой лишь первую половину процесса обработки любого прерывания. Ниже приведен список этих ограничений.

- Обработчики прерываний запускаются асинхронно и поэтому потенциально могут прерывать выполнение другого важного кода, в том числе и другие обработчики прерываний. Поэтому, чтобы сильно не замедлять работу прерванного кода, обработчики прерываний должны выполняться как можно быстрее.
- В лучшем случае обработчики прерываний выполняются при замаскированной обрабатываемой линии IRQ, а в худшем случае (когда установлен флаг `IRQF_DISABLED`) — при запрещении всех прерываний на текущем процессоре. Поскольку при запрещенных прерываниях аппаратное устройство не может взаимодействовать с операционной системой, обработчики прерываний опять же 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 их нельзя заблокировать. Данный факт ограничивает выполняемые ими действия.

Теперь вы уже должны понимать, что обработчики прерываний являются только частью полного решения проблемы обработки аппаратных прерываний. Безусловно, в любой операционной системе должен присутствовать быстрый, асинхронный и простой механизм для немедленного реагирования на

запросы оборудования и выполнения критичной ко времени работы. И обработчики прерываний прекрасно справляются с этой работой. Однако другая, менее критичная ко времени выполнения работа должна быть отложена до того момента, когда прерывания будут разрешены.

Поэтому процесс обработки прерываний делится на две части, или *половины*. Первая часть обработчика прерывания, *верхняя половина* (*top half*), выполняется ядром асинхронно и немедленно в ответ на аппаратное прерывание, как было показано в предыдущей главе. В этой главе мы рассмотрим вторую часть процесса обработки прерываний — его *нижнюю половину* (*bottom half*).

Нижняя половина

В задачу нижней половины процесса обработки прерывания входит выполнение всей связанной с прерываниями работы, которую не смог сделать обработчик прерывания. В идеальной ситуации — это практически вся работа, так как необходимо, чтобы обработчик прерывания выполнил по возможности ее меньшую часть (т.е. выполнился максимально быстро). Вынеся максимально возможную часть работы в нижнюю половину, мы тем самым разгружаем обработчик прерывания и способствуем тому, чтобы он побыстрее завершил свою работу и вернул управление прерванной задаче.

Тем не менее обработчик прерывания все же должен выполнить *некоторые* действия. Например, почти всегда обработчик прерывания должен отправить устройству уведомление о том, что прерывание получено. Ему также может понадобиться скопировать некоторые данные из внутренней памяти аппаратного устройства в память компьютера. Так как эта работа критична ко времени выполнения, имеет смысл выполнить ее в самом обработчике прерывания.

Практически все остальные действия будет правильным выполнить в нижней половине обработчика. Например, если в верхней половине были скопированы данные из аппаратного устройства в память компьютера, то в нижней половине обработчика, конечно, имеет смысл обработать эти данные. К сожалению, не существует четких 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*. Не забывайте, что обработчики прерываний выполняются асинхронно при запрещенной, по крайней мере текущей, линии запроса на прерывание. Поэтому минимизация времени выполнения обработчика прерывания так важна. Хотя нет строгих правил по поводу того, как делить работу между верхней и нижней половинами обработчика прерываний, все же можно привести несколько полезных советов.

- Если работа критична по времени, то ее необходимо выполнять в обработчике прерывания.
- 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.
- Всю остальную работу следует выполнять в нижней половине обработчика.

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

Когда используется нижняя половина обработчика

Ключевым моментом при обработке прерывания является понимание того, зачем и в какой именно момент нужно откладывать работу на потом. Необходимо минимизировать количество работы, которая выполняется в обработчике прерывания, потому что это происходит при замаскированной текущей линии запроса на прерывание для всех процессоров. Более того, обработчики прерываний, при регистрации которых был указан флаг `IRQF_DISABLED`, выполняются при *всех* замаскированных линиях IRQ на локальном процессоре (плюс текущая линия запроса на прерывание замаскирована глобально). Чтобы уменьшить время реакции и увеличить производительность системы, необходимо минимизировать время, в течение которого прерывания запрещены. 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. Обработка входящего трафика, поступающего от сетевого устройства, не должна мешать ядру вовремя считывать коды нажатых клавиш на клавиатуре. Поэтому напрашивается очевидное решение — часть работы отложить на потом.

Но когда должно наступить это "потом"? Важно понять, что *позже* часто означает просто *не сейчас*. Основной момент в нижней половине обработчика прерывания заключается в том, что его выполнение не откладывается до *определенного* момента времени в будущем, а откладывается до "лучших времен", когда система будет не так загружена и все прерывания снова будут разрешены. Часто нижняя половина обработчика прерывания запускается сразу после возврата из обработчика прерывания. Основной момент здесь состоит в том, что он должен запускаться, когда будут разрешены все прерывания.

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

Многообразие нижних половин

В отличие от верхних половин обработчиков, которые могут быть реализованы только непосредственно в самих обработчиках прерываний, для реализации нижних половин обработчиков прерываний существует несколько механизмов. Они представляют собой различные интерфейсы и подсистемы, которые позволяют пользователю реализовать нижнюю половину обработчика прерывания. В предыдущей главе мы рассмотрели единственно возможный способ реализации верхней половины обработчика прерывания, а в этой главе рассмотрим несколько механизмов реализации его нижних половин. На самом деле за всю историю развития операционной системы Linux было придумано много вариантов

реализации нижних половин обработчиков прерываний. Иногда сбивает с толку то, что все эти варианты имеют очень схожие или очень неудачные названия. Для того чтобы придумывать названия вариантам нижних половин обработчиков прерываний, необходимы программисты с особым складом ума.

В этой главе мы рассмотрим принципы работы и способы реализации нижних половин обработчиков прерываний, которые существуют в ядрах операционной системы Linux серии 2.6. Также будет показано, как их можно использовать при написании собственного кода ядра. Старые и давно изъятые из употребления способы реализации нижних половин обработчиков прерывания также представляют собой историческую ценность, поэтому мы не забудем упомянуть о них там, где это будет уместно.

Первоначальный вариант нижней половины обработчика прерываний

С самого начала в операционной системе Linux существовал только единственный способ реализации нижних половин обработчиков прерывания, который так и назывался — *нижние половины* (bottom half). Это название было логичным, 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 собой статический список из 32 нижних половин обработчиков прерываний для всей системы. Путем установки соответствующего бита в 32-разрядном целом числе в верхней половине обработчика прерывания отмечалось, какую из нижних половин обработчика прерываний нужно запустить. 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 и представлял собой узкое место в плане производительности.

Очереди задач

Позже разработчики ядра придумали механизм *очередей задач* (*task queue*), который можно было одновременно использовать и как средство для выполнения отложенной обработки, и как замену механизма BH. В ядре определялось семейство очередей. Каждая очередь содержала связанный список функций, которые нужно было вызвать для выполнения соответствующих действий. Эти функции запускались в определенные моменты времени, в зависимости от того, в какой очереди они находились. Драйверы могли зарегистрировать собственные нижние половины обработчиков прерываний 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.

Отложенные прерывания и тасклеты

Во время разработки серии ядер 2.3 был предложен механизм *отложенных прерываний*[1] (*softirq*) и механизм *тасклетов* (*tasklet*). За исключением решения проблемы совместимости с существующими драйверами механизмы отложенных прерываний и таск-

[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*. — *Примеч. ред.*

летов были в состоянии полностью заменить интерфейс BH[2]. Отложенные прерывания — это набор статически определенных нижних половин обработчиков прерываний, которые могут одновременно выполняться на разных процессорах (даже два обработчика одного типа могут выполняться параллельно). Название "тасклет" придумано неудачно и часто сбивает с толку[3]. Тасклеты представляют собой гибкие, динамически создаваемые нижние половины обработчиков, которые являются надстройкой над механизмом отложенных прерываний. 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 нижних половин обработчиков прерываний достаточно использовать таскелеты. Обработчики отложенных прерываний становятся полезными, когда критична производительность, например для сетевой подсистемы. Использование механизма отложенных прерываний требует от программиста особой внимательности, 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 программисты называют все нижние половины обработчиков прерываний программными прерываниями, или отложенными прерываниями (software interrupt, или softirq). Другими словами, они называют и механизм отложенных прерываний, и нижние половины обработчиков в целом программными прерываниями. На таких людей лучше не обращать внимания, — они из той же категории, что и те, которые придумали названия "*BH*" и "тасклет".

Во время разработки ядер серии 2.5 интерфейс BH был в конце концов удален, потому что все его пользователи конвертировали свой код и перешли на другие интерфейсы нижних половин обработчиков. В дополнение к этому интерфейс *очередей задач* (task queue) был заменен на новый интерфейс очередей *отложенных действий* (work queue). Последний представляет собой простой и в то же время полезный механизм, позволяющий отложить некоторое действие на более поздний срок для выполнения в контексте процесса. Мы вернемся к нему чуть позже.

Таким образом, на сегодняшний день в ядре серии 2.6 предусмотрены три механизма создания нижних половин обработчиков: отложенные прерывания, тасклеты и очереди отложенных действий. Старые интерфейсы, такие как *BH* и очереди задач, канули в Лету.

Таймеры ядра

Существует еще один механизм выполнения отложенных действий — это *таймеры ядра* (kernel timers). В отличие от механизмов, рассмотренных выше в этой главе, таймеры ядра позволяют отсрочить работу на указанный интервал времени. Другими словами, несмотря на то, что средства, рассмотренные в предыдущем разделе этой главы, могут отсрочить выполнение работы *на неопределенный срок*, таймеры ядра предназначены для отсрочки выполнения работы *на заданный срок*.

---

[2] В связи с тем, что обработчики BH выполнялись глобально синхронно друг с другом, не так просто было их преобразовать для использования механизмов отложенных прерываний и тасклетов. Подразумевалось, что во время выполнения обработчика BH никакой другой обработчик BH не мог быть запущен. Однако в ядрах серии 2.5 это наконец-то удалось сделать.

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

Поэтому таймеры ядра, в отличие от обобщенных механизмов, описанных в данной главе, имеют другое назначение. Подробно таймеры будут описаны в главе 11, "Таймеры и управление временем".

Путаница с терминологией

Существует некоторая путаница с нижними половинами обработчиков прерываний, но на самом деле — это только проблема выбора названий. Давайте снова вернемся к этому вопросу.

Под термином *нижняя половина* (bottom half) подразумевается общий термин из области операционных систем, который обозначает отложенную часть процесса обработки прерывания. Выбор этого термина связан с тем, что он представляет вторую, или нижнюю, половину процесса обработки прерывания. В операционной системе Linux данный термин обозначает то же самое. Все механизмы ядра, предназначенные для отложенной обработки, называются нижними половинами. Некоторые программисты также называют нижние половины программными прерываниями или "softirq", что еще больше запутывает дело.

Термином *нижняя половина* также называют самый первый механизм выполнения отложенных действий в операционной системе Linux. Его еще называют "BH", поэтому далее так и будем называть именно этот механизм, а термин *нижняя половина* (bottom half) будем употреблять в контексте общего описания. Механизм BH был исключен из употребления и полностью изъят в ядрах серии 2.5.

В настоящий момент существуют три механизма выполнения отложенной работы: *отложенные прерывания* (softirq), тасклеты и очереди отложенных действий. Тасклеты используют механизм отложенных прерываний, а очереди отложенных действий имеют полностью отличную реализацию. История развития нижних половин обработчиков прерывания показана в табл. 8.1.

Таблица 8.1. Состояние нижних половин обработчиков

Механизм	Состояние
BH	Изъят в серии 2.5
Очереди задач	Изъяты в серии 2.5
Отложенные прерывания	Доступны начиная с серии 2.3
Тасклеты	Доступны начиная с серии 2.3
Очереди отложенных действий	Доступны начиная с серии 2.3

Разобравшись с этой путаницей в терминологии, рассмотрим каждый механизм в отдельности.

Отложенные прерывания

Обсуждение существующих методов построения нижних половин обработчиков прерываний начнем с механизма *отложенных прерываний* (softirq). Сам по себе этот механизм напрямую используется редко, поскольку он положен в основу более общего механизма создания нижних половин обработчиков прерываний — тасклетов. Исходя из этого, мы должны сначала рассмотреть механизм отложенных прерываний. Его код находится в файле `kernel/softirq.c` дерева исходных кодов ядра.

Реализация механизма отложенных прерываний

Отложенные прерывания определяются статически во время компиляции. В отличие от тасклетов, нельзя динамически зарегистрировать или удалить отложенное прерывание. Отложенные прерывания описываются с помощью структуры типа `softirq_action`, которая определена в файле `<linux/interrupt.h>`, как показано ниже.

```
struct softirq_action {
   void (*action)(struct softirq_action *);
};
```

Массив, состоящий из 32 экземпляров этой структуры, определен в файле `kernel/softirq.c`.

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

Каждое зарегистрированное отложенное прерывание соответствует одному элементу этого массива. Поэтому всего можно зарегистрировать `NR_SOFTIRQS` отложенных прерываний. Поскольку максимальное количество отложенных прерываний определяется статически во время компиляции, его нельзя динамически изменить во время выполнения программы. В ядре принудительно устанавливается лимит на регистрацию 32 отложенных прерываний, хотя в текущей версии ядра используется всего девять[4].

Обработчик отложенных прерываний

Прототип обработчика отложенного прерывания, `action`, выглядит следующим образом:

```
void softirq_handler(struct softirq_action *)
```

При выполнении в ядре этого обработчика отложенных прерываний запускается функция `action`, которой в качестве единственного аргумента передается указатель на соответствующую структуру типа `softirq_action`. Например, если переменная `my_softirq` содержит указатель на элемент массива `softirq_vec`, то ядро вызовет функцию-обработчик соответствующего отложенного прерывания в следующем виде:

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

Вас может несколько удивить тот факт, что ядро передает в обработчик указатель на всю структуру. Этот прием позволяет в будущем вводить дополнительные поля в структуру без необходимости внесения изменений в существующие обработчики.

Обработчик одного отложенного прерывания никогда не может вытеснить другой такой же обработчик. В действительности только одно событие может вытеснить обработчик отложенного прерывания — это аппаратное прерывание. Однако на другом процессоре одновременно с обработчиком отложенного прерывания может выполняться другой (и даже этот же) обработчик отложенного прерывания.

Запуск отложенных прерываний

Перед тем как зарегистрированное отложенное прерывание будет запущено, оно должно быть промаркировано. Этот процесс называется *генерацией отложенного прерывания* (*rise softirq*). Как правило, маркировка отложенных прерываний для их запуска

[4] В большинстве драйверов устройств для нижних половин обработчиков прерываний используется механизм тасклетов, которые построены на основе механизма отложенных прерываний, как будет показано ниже.

выполняется перед возвратом из обработчика прерываний. Проверка ожидающих выполнения обработчиков отложенных прерываний и их запуск осуществляется в перечисленных ниже случаях.

- При возврате из обработчика аппаратного прерывания.
- В контексте потока ядра `ksoftirqd`.
- В любом коде ядра, в котором явно проверяются и запускаются ожидающие обработчики отложенных прерываний, как, например, это делается в сетевой подсистеме.

Независимо от метода вызова отложенного прерывания его выполнение осуществляется в функции `__do_softirq()`, которая вызывается из функции `do_softirq()`. Эта функция очень проста. Если в системе есть ожидающие выполнения отложенные прерывания, то функция `__do_softirq()` в цикле проверяет их все и вызывает ожидающие обработчики. Рассмотрим упрощенный вариант самой важной части функции `__do_softirq()`.

```
u32 pending;

pending = local_softirq_pending();
if (pending) {
    struct softirq_action *h;

   /* Сбросим битовую маску ожидающих запросов */
   set_softirq_pending(0);

   h = softirq_vec;
   do {
      if (pending & 1)
         h->action(h);
      h++;
      pending >>= 1;
   } while (pending);
}
```

Этот фрагмент кода является “сердцем” обработчика отложенных прерываний. В нем проверяются и запускаются все ожидающие отложенные прерывания. В частности, выполняются перечисленные ниже действия.

1. Локальной переменной `pending` присваивается значение, возвращаемое макросом `local_softirq_pending()`. Это значение — 32-разрядная битовая маска ожидающих на выполнение отложенных прерываний. Если установлен бит с номером `n`, то отложенное прерывание с этим номером ожидает запуска.
2. Когда значение битовой маски отложенных прерываний сохранено, оригинальная битовая маска очищается[5].
3. Переменной `h` присваивается указатель на первый элемент массива `softirq_vec`.

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

4. Если первый бит маски, которая хранится в переменной `pending`, установлен, то вызывается функция `h->action(h)`.
5. Указатель `h` увеличивается на единицу, и теперь он указывает на второй элемент массива `softirq_vec`.
6. Выполняется 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.
7. Переменная `h` теперь указывает на второй элемент массива, а в битовой маске второй бит стал первым. Теперь необходимо повторить все ранее выполненные шаги.
8. Описанные выше действия последовательно повторяются 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 вполне достаточно. Она гарантирует, что указатель `h` всегда будет указывать на корректную запись в массиве `softirq_vec`. Поскольку битовая маска `pending` состоит из 32 битов, цикл не может выполниться больше 32 раз.

Использование отложенных прерываний

Отложенные прерывания используются для запуска самых важных и критичных ко времени выполнения нижних половин обработчиков прерываний в системе. В настоящий момент только в двух подсистемах (сетевой и блочных устройств) механизм отложенных прерываний используются напрямую. Кроме того, на его основе построены таймеры ядра и тасклеты. Перед тем как добавить в систему новое отложенное прерывание, спросите себя, почему для этой цели нельзя использовать тасклет. Тасклеты создаются динамически, кроме того, их легче использовать в связи с более простыми требованиями к блокировкам, а их производительность остается очень высокой. Тем не менее для задач, критичных ко времени выполнения, которые способны сами обеспечивать эффективные блокировки, использование механизма отложенных прерываний будет правильным решением.

Назначение индексов

Отложенные прерывания объявляются на этапе компиляции с помощью соответствующего перечисления (`enum`), определенного в файле `<linux/interrupt.h>`. Указанный в перечислении индекс (он начинается с нуля) используется в ядре как значение относительного приоритета отложенных прерываний. При этом отложенные прерывания с меньшим номером выполняются раньше отложенных прерываний с бо́льшим номером.

Для создания отложенного прерывания нужно добавить новую запись в указанный выше перечень (`enum`). При этом новая строчка помещается не в конец списка, как это делается в других местах, а в то его место, которое соответствует приоритету отложенного прерывания. Исходя из принятых соглашений, строчка `HI_SOFTIRQ` всегда указывается первой, а `RCU_SOFTIRQ` — последней. Новая запись, скорее всего, должна быть где-то между записями `BLOCK_SOFTIRQ` и `TASKLET_SOFTIRQ`. В табл. 8.2 приведены все типы существующих отложенных прерываний.

Таблица 8.2. Список отложенных прерываний

Отложенное прерывание	Приоритет	Описание
HI_SOFTIRQ	0	Высокоприоритетные тасклеты
TIMER_SOFTIRQ	1	Таймеры
NET_TX_SOFTIRQ	2	Отправка сетевых пакетов
NET_RX_SOFTIRQ	3	Прием сетевых пакетов
BLOCK_SOFTIRQ	4	Блочные устройства
TASKLET_SOFTIRQ	5	Тасклеты с обычным приоритетом
SCHED_SOFTIRQ	6	Планировщик
HRTIMER_SOFTIRQ	7	Высокоточные таймеры
RCU_SOFTIRQ	8	RCU-блокировка

Регистрация обработчика

Следующий этап состоит в регистрации обработчика отложенного прерывания во время выполнения программы с помощью функции `open_softirq()`. Ей передаются два параметра: индекс отложенного прерывания и функция-обработчик. Например, код регистрации обработчика отложенного прерывания для сетевой подсистемы находится в файле `net/core/dev.c` и выполняется примерно так, как показано ниже.

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

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. Тем не менее на другом процессоре обработчики отложенных прерываний могут выполняться. Если вдруг отложенное прерывание будет сгенерировано 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, а также глобальные данные, которые используются только в самом обработчике, должны быть корректным образом заблокированы (этот вопрос обсуждается в следующих двух главах). Этот момент очень важен, и именно из-за него предпочтительнее использовать тасклеты. Простое решение запретить параллельное выполнение обработчиков отложенных прерываний нельзя назвать идеальным. Если обработчик отложенного прерывания захватит блокировку, которая предотвратит его запуск параллельно самому себе, то не остается почти никакого смысла использовать механизм отложенных прерываний. Следовательно, чтобы избежать явных и ненужных блокировок и обеспечить отличную масштабируемость, в большинстве обработчиков отложенных прерываний используются уникальные для каждого процессора (и следовательно, не требующие блокировок) данные или какие-нибудь другие ухищрения.

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

Генерация отложенных прерываний

После добавления обработчика в перечень `enum` и его регистрации с помощью функции `open_softirq()` он готов к запуску. Чтобы промаркировать обработчик, в результате чего он будет запущен при следующем вызове функции `do_softirq()`, используется функция `raise_softirq()`. Например, эта функция вызывается в сетевой подсистеме так, как показано ниже.

```
raise_softirq(NET_TX_SOFTIRQ);
```

В результате будет сгенерировано отложенное прерывание с индексом `NET_TX_SOFTIRQ`. Его обработчик `net_tx_action()` будет запущен при следующем выполнении программных прерываний ядром. Эта функция запрещает аппаратные прерывания перед тем, как сгенерировать отложенное прерывание, а затем восстанавливает их в первоначальном состоянии. 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 уже запрещены!
 */
raise_softirq_irqoff(NET_TX_SOFTIRQ);
```

Чаще всего отложенные прерывания генерируются из обработчиков аппаратных прерываний. В этом случае в обработчике аппаратного прерывания выполняется вся основная работа, связанная с аппаратным обеспечением, генерируется отложенное прерывание, после чего его работа завершается. После завершения обработки аппаратных прерываний в ядре вызывается функция `do_softirq()`. Далее запускается обработчик отложенного прерывания, в котором будет продолжена работа, отложенная в обработчике аппаратного прерывания. В данном случае раскрывается истинный смысл названий “верхняя половина” и “нижняя половина” обработчика прерываний.

Тасклеты

Тасклеты — это механизм построения нижних половин обработчиков прерываний на основе механизма отложенных прерываний. Как уже отмечалось выше, они не имеют ничего общего с *задачами* (task). По своей природе и принципу работы они очень похожи на отложенные прерывания, но имеют более простой интерфейс и упрощенные правила блокировок.

Для разработчиков драйверов устройств существует простое правило выбора что использовать — отложенные прерывания или тасклеты. В большинстве случаев необходимо использовать тасклеты. Как было показано в предыдущем разделе, примеры использования отложенных прерываний можно посчитать на пальцах одной руки. Отложенные прерывания применяются там, где нужна очень высокая частота запуска обработчиков прерываний и интенсивно используется многопоточная обработка. С другой стороны, тасклеты гораздо легче использовать, и в большинстве случаев они работают очень хорошо.

Реализация тасклетов

Поскольку тасклеты реализованы на основе отложенных прерываний, они *также* являются отложенными прерываниями. Как уже отмечалось выше, тасклеты представлены двумя типами отложенных прерываний: `HI_SOFTIRQ` и `TASKLET_SOFTIRQ`. Единст-

венная разница между ними в том, что тасклеты типа HI_SOFTIRQ запускаются всегда раньше тасклетов типа TASKLET_SOFTIRQ.

Структуры тасклетов

Тасклеты представляются в виде структуры tasklet_struct. Каждый экземпляр структуры представляет собой уникальный тасклет. Эта структура определена в 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 */
};
```

В поле func указывается функция-обработчик тасклета. Данное поле эквивалентно полю action для структуры, представляющей отложенное прерывание. Единственный аргумент, который передается функции-обработчику тасклета, указывается в поле data.

В поле state может находиться одно из следующих значений: нуль, TASKLET_STATE_SCHED или TASKLET_STATE_RUN. Значение TASKLET_STATE_SCHED указывает на то, что тасклет запланирован на выполнение, а TASKLET_STATE_RUN — что тасклет выполняется. В качестве оптимизации следует отметить, что значение TASKLET_STATE_RUN используется только на многопроцессорной машине, поскольку на однопроцессорной машине и так известно, выполняется тасклет или нет (т.е. выполняется в настоящий момент код тасклета или нет).

В поле count указывается счетчик ссылок на тасклет. Если это значение не равно нулю, то тасклет отключен и не может быть запущен. Если оно равно нулю, то тасклет активизирован и может запуститься в случае, когда он промаркирован 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: tasklet_vec (для обычных тасклетов) и tasklet_hi_vec (для высокоприоритетных тасклетов). Каждая из этих структур представляет собой связанный список структур tasklet_struct. Каждый экземпляр структуры tasklet_struct представляет собой отдельный тасклет.

Тасклеты планируются на выполнение с помощью функций tasklet_schedule() и tasklet_hi_schedule(), которым передается единственный аргумент — указатель на структуру tasklet_struct. В каждой их этих функций проверяется, что тасклет не был запланирован на выполнение, после чего в зависимости от обстоятельств вызывается функция __tasklet_schedule() или __tasklet_hi_schedule(). Эти функции очень похожи. Разница состоит только в том, что в одной используется отложенное прерывание с номером TASKLET_SOFTIRQ, а в другой — с номером HI_SOFTIRQ. Процесс

[6] Это еще один пример плохой терминологии. Почему отложенные прерывания (softirq) генерируются (rise), а тасклеты (tasklet) планируются (schedule)? Кто знает? Оба термина означают, что нижние половины обработчиков прерываний промаркированы как ожидающие запуска и в скором времени будут выполнены.

написания и использования тасклетов описан в следующем разделе. А сейчас рассмотрим последовательность действий, выполняемых в функции `tasklet_schedule()`.

1. Проверяется, что значение поля `state` не равно `TASKLET_STATE_SCHED`. В противном случае тасклет уже запланирован на выполнение, и функция может немедленно возвратить управление.
2. Вызывается функция `__tasklet_schedule()`.
3. Сохраняется состояние системы прерываний и запрещаются прерывания на локальном процессоре. Это гарантирует, что ничто на данном процессоре не будет мешать выполнению кода тасклета в процессе манипуляции тасклетами в функции `tasklet_schedule()`.
4. Тасклет добавляется в очередь на выполнение. Это происходит путем его помещения в начало связанного списка `tasklet_vec` или `tasklet_hi_vec`, которые уникальны для каждого процессора в системе.
5. Генерируется отложенное прерывание с номером `TASKLET_SOFTIRQ` или `HI_SOFTIRQ`. В результате наш тасклет будет запущен при следующем вызове функции `do_softirq()`.
6. Восстанавливается первоначальное состояние системы прерываний и управление возвращается в вызывающую программу.

Как уже отмечалось в предыдущем разделе, функция `do_softirq()` запускается при первом же удобном случае. Поскольку большинство тасклетов маркируются как готовые к выполнению в обработчиках прерываний, то, скорее всего, функция `do_softirq()` будет запущена сразу же после возврата управления из последнего обработчика прерывания. Так как отложенные прерывания с номерами `TASKLET_SOFTIRQ` или `HI_SOFTIRQ` к этому моменту уже сгенерированы, то в функции `do_softirq()` будут вызваны соответствующие обработчики. Эти обработчики, а также функции `tasklet_action()` и `tasklet_hi_action()` являются основой механизма обработки тасклетов. Рассмотрим, что в них происходит.

1. Запрещаются прерывания для текущего процессора и выбирается соответствующий список `tasklet_vec` или `tasklet_hi_vec`. Однако при этом нет необходимости сохранять текущее состояние системы прерываний, поскольку наш код всегда будет выполняться в обработчике отложенных прерываний, а в нем прерывания всегда разрешены.
2. Очищается список тасклетов для текущего процессора путем присвоения ему значения `NULL`.
3. Разрешаются прерывания для текущего процессора. Опять же нет необходимости восстанавливать предыдущее состояние системы прерываний, поскольку мы и так знаем, что первоначально прерывания всегда разрешены.
4. Организовывается цикл по всем тасклетам в выбранном списке.
5. На многопроцессорной машине нужно проверить, не выполняется ли текущий тасклет на другом процессоре, т.е. проверить, не установлен ли флаг `TASKLET_STATE_RUN`. Если тасклет уже запущен, то его необходимо пропустить и перейти к следующему тасклету в списке (вспомним, что только один тасклет данного типа может выполняться в любой момент времени).

6. Если тасклет еще не запущен, то нужно установить флаг TASKLET_STATE_RUN, чтобы другой процессор не запустил его.
7. Значение поля count проверяется на равенство нулю, чтобы убедиться, что тасклет не отключен. Если тасклет отключен (поле count не равно нулю), то нужно перейти к следующему тасклету, находящемуся в очереди на выполнение.
8. Теперь нам точно известно, что тасклет не выполняется и не будет запущен на другом процессоре, поскольку мы только что его промаркировали как выполняющийся, и что значение его поля count равно нулю. Можно смело запускать обработчик тасклета.
9. После запуска тасклета следует сбросить флаг TASKLET_STATE_RUN в его поле state.
10. Описанный выше процесс нужно повторить для всех тасклетов, ожидающих в очереди на выполнение.

Реализация тасклетов проста, но в то же время очень остроумна. Как было показано, 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 с помощью специальных функций, в которых запускаются все запланированные на выполнение тасклеты. В этих функциях предприняты меры, которые позволяют гарантировать, что только один тасклет данного типа будет выполняться в любой момент времени (но тасклеты разных типов могут выполняться одновременно). Вся эта сложность спрятана за простым и понятным интерфейсом.

Использование тасклетов

В большинстве случаев тасклеты — самый предпочтительный механизм, с помощью которого следует реализовывать нижние половины обработчиков прерываний для обычных аппаратных устройств. Тасклеты можно создавать динамически, их просто использовать, и они сравнительно быстро работают. Хотя их название способно сбить с толку любого, вам наверняка понравится остроумная идея, положенная в основу их работы.

Объявление тасклетов

Тасклеты можно создавать статически и динамически. Все зависит от того, как необходимо (или желательно) обращаться в программах к тасклету: прямо или через указатель. Для статического создания тасклета (и соответственно обеспечения прямого доступа к нему) используется один из двух макросов, определенных в файле <linux/interrupt.h>, как показано ниже.

```
DECLARE_TASKLET(name, func, data)
DECLARE_TASKLET_DISABLED(name, func, data);
```

Оба макроса статически создают экземпляр структуры tasklet_struct с указанным в параметре name именем. Как только тасклет будет запланирован на выполнение, вызывается функция func, которой передается аргумент data. Различие между этими макросами состоит только в начальном значении счетчика ссылок на тасклет (поле count). Первый макрос создает тасклет, у которого значение поля count равно нулю, и, соответственно, этот тасклет разрешен для выполнения. Второй макрос создает тасклет

и устанавливает для него значение поля count, равное единице, и, соответственно, этот тасклет будет отключен. Ниже приведен пример.

```
DECLARE_TASKLET(my_tasklet, my_tasklet_handler, dev);
```

Эта строка эквивалентна приведенному ниже объявлению.

```
struct tasklet_struct my_tasklet = { NULL, 0, ATOMIC_INIT(0),
                                     my_tasklet_handler, dev };
```

В данном примере создается тасклет my_tasklet, который разрешен для выполнения, а в качестве его обработчика выступает функция tasklet_handler. При вызове этой функции ей передается значение параметра dev.

Для инициализации тасклета, на который указывает заданный указатель struct tasklet_struct* t — косвенная ссылка на динамически созданную ранее структуру, — используется приведенная ниже функция.

```
tasklet_init(t, tasklet_handler, dev); /* динамически, а не статически */
```

Написание собственной функции-обработчика тасклета

Функция-обработчик тасклета должна соответствовать корректному прототипу.

```
void tasklet_handler(unsigned long data)
```

Так же как и в случае отложенных прерываний, тасклет не может переходить в состояние ожидания. Это означает, что в тасклетах нельзя использовать семафоры или другие функции, которые могут вызвать состояние его блокировки. Кроме того, тасклеты выполняются при всех разрешенных прерываниях, поэтому необходимо принять соответствующие меры предосторожности (например, запретить прерывания и захватить блокировку), если в тасклете используются общие с обработчиком прерывания данные. В отличие от отложенных прерываний один и тот же тасклет не может выполняться параллельно самому себе, хотя два разных тасклета могут выполняться на разных процессорах параллельно. Поэтому, если в тасклете используются общие с обработчиком прерывания или другим тасклетом данные, необходимо выполнять соответствующие блокировки (подробнее об этом — в главе 9, “Общие сведения о синхронизации кода ядра”, и главе 10, “Средства синхронизации ядра”).

Планирование тасклета на выполнение

Для планирования тасклета на выполнение нужно вызвать функцию tasklet_schedule(), которой в качестве аргумента передается указатель на соответствующий экземпляр структуры tasklet_struct.

```
tasklet_schedule(&my_tasklet); /* маркировка my_tasklet
                                  как ожидающего выполнения */
```

После планирования тасклета на выполнение он будет запущен всего один раз в некоторый момент времени в ближайшем будущем. Если до запуска тасклет будет запланирован на выполнение еще раз, то он также запустится всего один раз. Если тасклет уже выполняется, скажем, на другом процессоре, то он будет запланирован еще раз и запустится снова. Учитывая вопросы оптимизации, тасклет будет всегда запускаться на том же процессоре, который запланировал его на выполнение. Это позволяет надеяться на лучшее использование кеш-памяти процессора.

Для отключения заданного тасклета используется функция tasklet_disable(). Если тасклет в данный момент времени выполняется, то эта функция не возвратит

управление, пока тасклет не закончит свою работу. Существует и альтернативная функция `tasklet_disable_nosync()`, которая отключает указанный тасклет, но при этом не ожидает завершения его выполнения и сразу же возвращает управление в вызвавшую ее программу. Как правило, эту функцию нужно использовать осмотрительно, поскольку в данном случае нельзя гарантировать, что тасклет уже закончил свое выполнение. Для активизации тасклета используется функция `tasklet_enable()`. Эту функцию также нужно вызвать, чтобы активизировать тасклет, созданный с помощью макроса `DECLARE_TASKLET_DISABLED()`. Ниже приведен пример использования описанных выше функций.

```
tasklet_disable(&my_tasklet); /* тасклет теперь отключен */

/* Здесь мы можем делать все, что угодно, зная,
    что тасклет не может быть запущен... */

tasklet_enable(&my_tasklet); /* Теперь тасклет активизирован */
```

Для удаления тасклета из очереди ожидающих на выполнение программ используется функция `tasklet_kill()`. Ей передается единственный аргумент — указатель на структуру `tasklet_struct`. Удаление запланированного на выполнение тасклета из очереди пригодится в случае, когда мы имеем дело с тасклетами, которые сами себя планируют на выполнение. Эта функция сначала ожидает, пока тасклет не завершит свою работу, а затем удаляет его из очереди. Однако это не исключает возможности, когда этот же тасклет может быть запланирован на выполнение из другого кода. Поскольку данная функция может переходить в состояние ожидания, ее нельзя вызывать из контекста прерывания.

Демон ksoftirqd

Обработка *отложенных прерываний* (softirq) и соответственно тасклетов может осуществляться с помощью набора потоков ядра (по одному потоку на каждый процессор). Эти потоки помогают обрабатывать отложенные прерывания, когда система перегружена их большим количеством. Поскольку тасклеты реализованы через механизм отложенных прерываний, описанный ниже материал применим в равной степени и к тасклетам, и к отложенным прерываниям. Поэтому для краткости мы в основном будем рассматривать отложенные прерывания.

Как уже упоминалось, ядро обрабатывает отложенные прерывания в нескольких местах, но чаще всего это происходит после возврата из обработчика прерывания. Отложенные прерывания могут генерироваться с очень большой частотой (как, например, в случае интенсивного сетевого трафика). Более того, функции-обработчики отложенных прерываний могут самостоятельно возобновлять свое выполнение (реактивизировать себя). Иными словами, во время выполнения функции-обработчики отложенных прерываний могут сгенерировать для себя отложенное прерывание, чтобы запуститься снова (на самом деле сетевая подсистема именно так и делает!). Частая генерация отложенных прерываний в сочетании с возможностью активизации самих себя может привести к тому, что пользовательским программам будет выделяться недостаточное количество процессорного времени. В свою очередь, несвоевременная обработка отложенных прерываний также не допустима. Возникает дилемма, которая требует решения, но ни одно из двух очевидных решений не является подходящим. Поэтому сначала рассмотрим оба этих очевидных решения.

Первое решение состоит в незамедлительной обработке всех отложенных прерываний по мере их возникновения, а также обработке 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, все вновь активизированные отложенные прерывания также будут немедленно обработаны. Проблема возникает в системах, работающих при большой загрузке и в которых возникает большое количество отложенных прерываний, которые к тому же постоянно сами себя активизируют. В таком случае ядро может постоянно обслуживать отложенные прерывания без возможности выполнять что-либо еще. При этом пользовательские программы будут попросту игнорироваться, поскольку в системе выполняются только лишь обработчики прерываний и отложенных прерываний. В результате пользователи системы начинают нервничать. Подобный подход может использоваться в слабо загруженных системах. Если же система испытывает хотя бы умеренную нагрузку, вызванную обработкой прерываний, то такое решение неприемлемо. Пользовательские приложения не должны продолжительное время испытывать недостаток процессорного времени.

Второе решение состоит в том, что вновь активизируемые отложенные прерывания *не должны* сразу же обрабатываться. При этом после возврата из очередного обработчика прерывания ядро должно просмотреть список всех ожидающих на выполнение отложенных прерываний и запустить их как обычно. Если какое-то из отложенных прерываний активизирует само себя, то его обработчик не будет запущен до того момента, пока ядро *в следующий раз* снова не приступит к их обработке. Однако такое скорее всего произойдет только при возникновении следующего аппаратного прерывания. А это может означать, что новое (или вновь активизированное) отложенное прерывание не будет выполнено в течение длительного промежутка времени. Такое решение плохо тем, что на не загруженной системе лучше всего 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 пользовательских приложений, создает нехватку ресурсов для отложенных прерываний, и к тому же он не эффективен для систем, работающих при малых нагрузках.

При проектировании механизма отложенных прерываний разработчики ядра осознавали, что необходим какой-нибудь компромисс. Решение, которое в конечном счете было реализовано в ядре, состояло в том, что вновь активизированные отложенные прерывания *не должны* обрабатываться немедленно. Вместо этого при существенном увеличении количества отложенных прерываний ядро возобновляет выполнение семейства потоков ядра, которые помогают справиться с нагрузкой. Данные потоки работают с самым минимально возможным приоритетом (значение параметра `nice` равно 19). Это гарантирует, что они не будут выполняться вместо чего-то более важного. При большой частоте генерации отложенных прерываний такой подход предотвращает монопольное использование процессора для их обработки. Также он гарантирует, что даже в случае большого количества отложенных прерываний все они в конечном итоге будут выполнены. И наконец, такое решение гарантирует, что в случае незагруженной системы отложенные прерывания также будут обрабатываются достаточно быстро (потому что соответствующие потоки ядра будут планироваться на выполнение немедленно).

Для каждого процессора существует свой поток ядра. Каждому потоку присвоено 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 запущен свой поток ядра, гарантирует, что если в системе есть свободный

процессор, то он всегда будет в состоянии обработать отложенные прерывания. После запуска потоков ядра в них выполняется замкнутый цикл, похожий на приведенный ниже.

```
for (;;) {
   if (!softirq_pending(cpu))
      schedule();

   set_current_state(TASK_RUNNING);

   while (softirq_pending(cpu)) {
      do_softirq();
      if (need_resched())
         schedule();
   }
   set_current_state(TASK_INTERRUPTIBLE);
}
```

Если в системе есть отложенные прерывания, ожидающие обработки (а это определяет вызов функции `softirq_pending()`), то поток ядра `ksoftirqd` вызывает функцию `do_softirq()`, которая обрабатывает эти прерывания. Происходит это периодически, чтобы можно было обработать также вновь активизированные отложенные прерывания. После каждой итерации по мере необходимости вызывается функция `schedule()`, чтобы могли выполняться более важные процессы. После того как вся обработка будет выполнена, поток ядра устанавливает свое состояние в `TASK_INTERRUPTIBLE` и вызывает системный планировщик для выбора нового процесса, готового к выполнению.

Поток ядра, обрабатывающий отложенные прерывания, вновь возвращается в состояние готовности к выполнению, когда функция `do_softirq()` определяет, что отложенное прерывание повторно активизировало само себя.

Старый механизм BH

К счастью, старый интерфейс BH 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 и представляет собой лишь историческую ценность.

Интерфейс BH очень древний, и это видно невооруженным глазом. Каждый обработчик BH должен быть определен статически, и количество этих обработчиков не должно превышать 32. Поскольку все обработчики BH определялись 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 ограничение максимального количества нижних половин обработчиков, равного 32, стали основной преградой для его использования.

Все обработчики BH выполнялись строго последовательно — никакие два обработчика BH, даже разных типов, не могли выполняться параллельно. Такой подход позволял обеспечить простую синхронизацию, но все же для получения высокой производительности при многопроцессорной обработке он был не очень хорош. Производительность на больших многопроцессорных машинах была ниже среднего. Драйверы, в которых использовался интерфейс BH, очень плохо масштабировались на несколько процессоров. В частности, от этого сильно 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 на

основе тасклетов. Максимальное количество нижних половин обработчиков, равное 32, определялось в виде констант в заголовочном файле `<linux/interrupt.h>`. Для того чтобы промаркировать обработчик BH как ожидающий запуска, нужно было вызвать функцию `mark_bh()`, которой в качестве параметра передавался номер обработчика BH. В ядрах серии 2.4 при этом планировался на выполнение тасклет BH, который запускался с помощью функции `bh_action()`. До появления серии ядер 2.4 механизм BH существовал самостоятельно. При его реализации не использовались никакие низкоуровневые механизмы нижних половин обработчиков прерываний, что во многом напоминало современную реализацию механизма отложенных прерываний.

В связи с описанными недостатками механизма BH разработчики ядра предложили заменить его *очередями задач* (task queue), которые, однако, так и не смогли справиться с поставленной целью, хотя и завоевали расположение очень многих новых пользователей. При разработке серии ядер 2.3 были предложены механизмы *отложенных прерываний* (softirq) и механизм *тасклетов* (tasklet), что и положило конец механизму BH. Механизм BH при этом был реализован на основе механизма тасклетов. К сожалению, достаточно сложно переносить код нижних половин обработчиков прерываний, в которых использовался интерфейс BH, на новые механизмы 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, в которых использовался механизм BH) наконец-то перевели на использование отложенных прерываний. И в завершение разработчики ядра наконец-то совсем убрали интерфейс BH. Скатертью дорога, 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 в потоке ядра. Данный тип нижних половин обработчиков всегда выполняется в контексте процесса и, как следствие, пользуется всеми преимуществами контекста процесса. Главное из них состоит в том, что этим процессом управляет системный планировщик, поэтому выполняющийся код может переходить в состояние ожидания.

Как правило, не составляет большого труда принять решение о том, что выбрать — очереди отложенных действий или отложенные прерывания/тасклеты. Если в отложенном коде необходимо переходить в состояние ожидания, то следует использовать очереди действий. Если же этого не требуется, то воспользуйтесь тасклетами или отложенными прерываниями. Обычно альтернатива использованию очередей отложенных действий состоит в создании новых потоков ядра. Поскольку при введении новых потоков ядра разработчики обычно хмурят брови (а у некоторых народов это означает смертельную обиду), настоятельно рекомендуется использовать очереди отложенных действий. Их действительно *очень* просто использовать.

---

[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 будет выполняться один и тот же код в одно и то же время разными тасклетами? Однако, после того как конвертация выполнена, это окупается повышением производительности.

Если для нижних половин обработчиков прерываний необходимо использовать нечто, что планируется на выполнение системным планировщиком, то воспользуйтесь очередями отложенных действий. Это единственный механизм, который всегда выполняется в контексте процесса и позволяет переходить нижним половинам обработчиков прерываний в состояние ожидания. Следовательно, им можно воспользоваться в ситуациях, когда необходимо выделить много памяти, захватить семафор или выполнить блочные операции ввода-вывода. Если для выполнения отложенных операций нет необходимости использовать поток ядра, то стоит подумать об использовании тасклетов.

Реализация очередей отложенных действий

В самом общем случае подсистема очередей отложенных действий — это интерфейс для создания потоков ядра, выполняющих действия, которые кем-то были поставлены 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, в подсистеме очередей отложенных действий предусмотрен специальный стандартный рабочий поток, выполняющий нужную работу. Поэтому в самом общем случае очереди отложенных действий — это простой интерфейс пользователя для откладывания работы, которая будет стандартным потоком ядра.

Стандартные рабочие потоки называются `events/n`, где `n` — номер процессора. Для каждого процессора запускается один такой поток. Например, в однопроцессорной системе будет запущен только один поток `events/0`. В двухпроцессорной системе добавляется еще один поток — `events/1`. Стандартные рабочие потоки выполняют отложенные действия, которые были инициированы из разных мест. Многие драйверы в ядре перекладывают обработку своих нижних половин на стандартные рабочие потоки. Если для драйвера или подсистемы нет строгой необходимости в создании собственного потока ядра, то предпочтительнее использовать стандартные рабочие потоки.

Тем не менее ничто не запрещает коду ядра создавать собственные рабочие потоки. Это может понадобиться, если в рабочем потоке выполняется большое количество вычислительных операций. Для операций, интенсивно использующих центральный процессор или критичных ко времени выполнения, также имеет смысл создавать отдельные рабочие потоки. В результате будет уменьшена нагрузка на стандартные рабочие потоки и устранена проблема нехватки ресурсов для выполнения остальных отложенных действий.

Структуры данных для представления рабочих потоков

Рабочие потоки представляются с помощью структуры `workqueue_struct`, показанной ниже.

```
/*
 * 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];
       struct list_head list;
       const char *name;
       int singlethread;
       int freezeable;
       int rt;
};
```

Эта структура определена в файле kernel/workqueue.c. В ней содержится массив структур cpu_workqueue_struct, каждый элемент которого соответствует одному процессору в системе. Поскольку рабочие потоки создаются для каждого процессора в системе, для каждого рабочего потока, работающего на каждом процессоре машины, существует такая структура. Структура cpu_workqueue_struct является основной и также определена в файле kernel/workqueue.c, как показано ниже.

```
struct cpu_workqueue_struct {
   spinlock_t lock;              /* Блокировка, защищающая
                                  *  данную структуру       */
   struct list_head worklist;    /* Список действий */
   wait_queue_head_t more_work;
   struct work_struct *current_struct;
   struct workqueue_struct *wq; /* Связанная структура workqueue_struct */
   task_t *thread;               /* Связанный поток */
};
```

Обратите внимание на то, что каждый *mun* рабочего потока имеет одну, связанную с ним структуру workqueue_struct. В этой структуре предусмотрено по одному экземпляру структуры cpu_workqueue_struct для каждого рабочего потока и, следовательно, для каждого процессора в системе, так как на каждом процессоре может существовать только один рабочий поток каждого типа.

Структуры для представления отложенных действий

Все рабочие потоки реализованы как обычные потоки ядра, которые выполняют функцию worker_thread(). После начальной инициализации эта функция входит в бесконечный цикл и переходит в состояние ожидания. Когда какая-либо работа ставится в очередь, поток активизируется и выполняет ее. Когда в очереди не остается работы, которую нужно выполнить, поток снова переходит в состояние ожидания.

Каждое действие представлено с помощью структуры work_struct, определенной в файле <linux/workqueue.h>. Эта структура показана ниже.

```
struct work_struct {
   atomic_long_t data;
   struct list_head entry;
   work_func_t func;
};
```

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 стандартными рабочими потоками. При активизации рабочий поток начинает выполнять все действия, которые находятся в его списке. После завершения работы он удаляет соответствующие структуры work_struct из связанного списка. Когда список становится пустым, поток переходит в состояние ожидания.

Рассмотрим упрощенную основную часть функции worker_thread().

```
for (;;) {
   prepare_to_wait(&cwq->more_work, &wait, TASK_INTERRUPTIBLE);
   if (list_empty(&cwq->worklist))
      schedule();
   finish_wait(&cwq->more_work, &wait);
   run_workqueue(cwq);
}
```

В этой функции в бесконечном цикле выполняются перечисленные ниже действия.

1. Поток изменяет свое состояние на состояние ожидания (флаг состояния задачи устанавливается в значение `TASK_INTERRUPTIBLE`) и добавляет себя в очередь ожидания.
2. Если связанный список действий пуст, то поток вызывает функцию `schedule()` и переходит в состояние ожидания.
3. Если список не пустой, то поток не переходит в состояние ожидания. Он устанавливает свое состояние в значение `TASK_RUNNING` и удаляет себя из очереди ожидания.
4. Если список не пустой, то вызывается функция `run_workqueue()` для выполнения отложенных действий.

Вся работа по выполнению отложенных действий выполняется в функции `run_workqueue()`, как показано ниже.

```
while (!list_empty(&cwq->worklist)) {
   struct work_struct *work;
   work_func_t f;
   void *data;

   work = list_entry(cwq->worklist.next, struct work_struct, entry);
   f = work->func;
   list_del_init(cwq->worklist.next);
   work_clear_pending(work);
   f(work);
}
```

В бесконечном цикле эта функция обрабатывает каждый элемент связанного списка отложенных действий и запускает функцию, указатель которой находится в поле `func` структуры `workqueue_struct`. При этом выполняются перечисленные ниже действия.

1. Если список не пустой, выбирается следующий элемент списка.
2. Из поля `func` выбирается указатель на функцию, которую необходимо вызвать, а из поля `data` — аргумент этой функции.
3. Текущий элемент удаляется из списка и сбрасывается бит ожидания в его структуре.
4. Вызывается сама функция.
5. Перечисленные выше действия повторяются.

Основные моменты реализации очереди отложенных действий

Взаимосвязи между различными рассмотренными в этом разделе структурами данных достаточно запутанные. Чтобы прояснить ситуацию, на рис. 8.1 показана схема, которая поможет вам во всем разобраться.

На самом верхнем уровне находятся рабочие потоки, которых может быть несколько типов. Для каждого процессора может существовать только один рабочий поток заданного типа. По мере необходимости в различных подсистемах ядра могут создаваться новые рабочие потоки. По умолчанию создаются только стандартные рабочие потоки с именами `events/n`. Каждый рабочий поток представляется в виде структуры `cpu_workqueue_struct`. Структура `workqueue_struct` представляет все рабочие потоки одного типа.

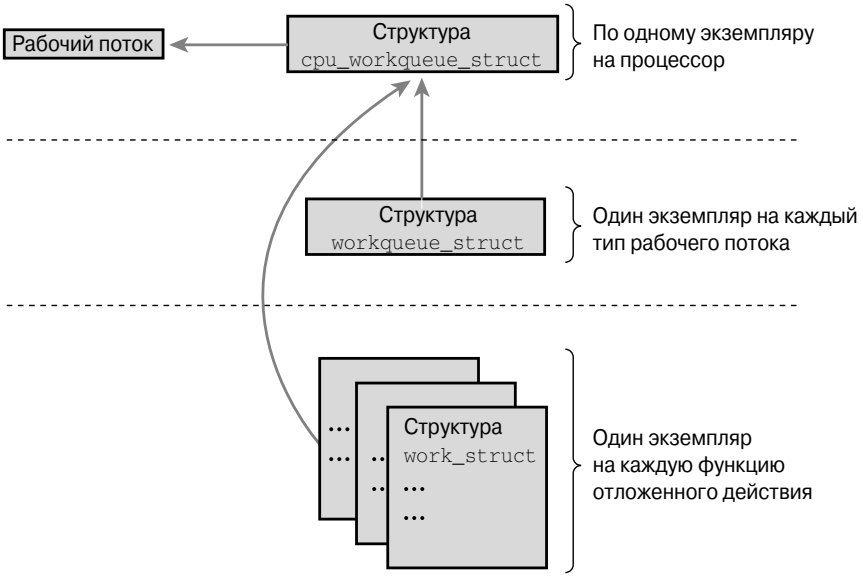


Рис. 8.1. Взаимосвязи между отложенными действиями, очередями действий и рабочими потоками

В качестве примера предположим, что, кроме обычных типов рабочих потоков `events/n`, был создан еще один тип рабочего потока — `falcon`. Также предположим, что в нашем распоряжении имеется четырехпроцессорный компьютер. Тогда в системе будут существовать четыре потока типа `events` (и, следовательно, четыре структуры типа `cpu_workqueue_struct`) и четыре потока типа `falcon` (и, соответственно, еще четыре дополнительных структуры типа `cpu_workqueue_struct`). Для потоков типа `events` определяется свой экземпляр структуры `workqueue_struct`, а для потоков типа `falcon` — свой.

А теперь зайдем с другой стороны: на самом нижнем уровне находятся отложенные действия. Написанный вами драйвер создает отложенное действие, которое должно выполниться позже. Это действие представляется в системе структурой `work_struct`. Среди прочего в этой структуре находится указатель на функцию, которая должна обработать отложенное действие. Отложенное действие отправляется на выполнение *определенному* рабочему потоку — в данном случае заданному потоку типа `falcon`. После этого рабочий поток выводится из состояния ожидания и выполняет отложенную работу.

В большинстве драйверов используются существующие стандартные рабочие потоки, которые называются `events`. Они просты в реализации, и их легко использовать. Однако в некоторых более серьезных ситуациях необходимо создавать новые специальные рабочие потоки. Например, в драйвере файловой системы XFS создаются два новых типа рабочих потоков.

Использование очередей отложенных действий

Очереди отложенных действий использовать довольно легко. Сначала мы рассмотрим, как используются стандартные рабочие потоки `events`, а затем опишем создание новых типов рабочих потоков.

Создание отложенного действия

Первый этап состоит в создании самого действия, которое должно быть отложено. Для создания статической структуры на этапе компиляции используется макрос DECLARE_WORK, как показано ниже.

```
DECLARE_WORK(name, void (*func)(void *), void *data);
```

В результате будет создана структура типа work_struct с именем name, функцией-обработчиком func и аргументом функции-обработчика data.

Во время выполнения программы отложенное действие создается путем передачи указателя на структуру с использованием макроса INIT_WORK, как показано ниже.

```
INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
```

В результате отложенное действие будет проинициализировано динамически с помощью структуры, на которую указывает указатель work, функции-обработчика func и аргумента data.

Обработчик отложенного действия

Функция-обработчик отложенного действия имеет следующий прототип:

```
void work_handler(void *data)
```

Эта функция будет выполняться в рабочем потоке и, следовательно, в контексте процесса. При этом по умолчанию все прерывания разрешены и не захватываются никакие блокировки. По мере необходимости функция может переходить в состояние ожидания. Обратите внимание: хотя функция-обработчик и выполняется в контексте процесса, из рабочего потока нельзя обращаться к содержимому памяти пространства пользователя. Все дело в том, что адресное пространство пользователя не отображается в адресное пространство потока ядра. Ядро может обращаться к пользовательской памяти только тогда, когда оно выполняется от имени пользовательского процесса, как, например, в случае вызова системной функции. Только в этом случае адресное пространство пользователя отображается в память ядра.

Блокировки между очередями отложенных действий и другими частями ядра осуществляются так же, как и в случае любого другого кода, работающего в контексте процесса. Это позволяет существенно упростить процесс написания обработчиков отложенных действий. Подробнее блокировки будут описаны в следующих двух главах.

Планирование отложенных действий на выполнение

После создания отложенного действия системный планировщик должен поставить его в очередь на выполнение. Для того чтобы оно выполнилось в стандартном рабочем потоке events, нужно вызвать функцию.

```
schedule_work(&work);
```

Данное отложенное действие будет запланировано на выполнение немедленно и выполнено, как только рабочий поток events, запущенный на текущем процессоре, выйдет из состояния ожидания.

Иногда требуется выполнить отложенное действие не немедленно, а с некоторой задержкой. Для этого следует запланировать это действие в заданный момент времени в будущем, как показано ниже.

```
schedule_delayed_work(&work, delay);
```

В этом случае действие, представленное структурой work_struct, с адресом &work, не будет выполнено, пока не пройдет заданное в параметре delay количество импульсов таймера. Об использовании импульсов таймера для измерения интервалов времени речь пойдет в главе 11, “Таймеры и управление временем”.

Ожидание завершения выполнения отложенного действия

Отложенные действия, поставленные в очередь, выполняются, когда рабочий поток в следующий раз будет выведен из состояния ожидания. Иногда нужно гарантировать, чтобы перед началом некоторой работы заданный пакет отложенных действий завершил свое выполнение. Очевидно, например, что перед выгрузкой из памяти загружаемых модулей ядра все установленные ими отложенные действия должны завершиться. В других местах ядра также может понадобиться гарантировать, что не будет конфликтов из-за доступа к системным ресурсам, возникающих из-за ожидающих выполнения отложенных действий.

Для выполнения описанных выше действий предусмотрена функция flush_scheduled_work, ожидающая, пока заданная очередь отложенных действий не будет очищена, как показано ниже.

```
void flush_scheduled_work(void);
```

Данная функция перед своим возвратом ожидает, пока все отложенные действия в очереди не будут выполнены. При этом данная функция переводит вызывающий процесс в состояние ожидания. Поэтому ее можно вызывать только из контекста процесса.

Эта функция не аннулирует никаких отложенных действий, запланированных на выполнение с задержкой. Другими словами, любые отложенные действия, запланированные на выполнение с помощью функции schedule_delayed_work() и интервал времени задержки которых еще не истек, не аннулируются с помощью вызова функции flush_scheduled_work(). Для отмены отложенных действий с задержками следует использовать приведенную ниже функцию.

```
int cancel_delayed_work(struct work_struct *work);
```

Эта функция отменяет отложенное действие, которое связано с данной структурой work_struct, если оно запланировано на выполнение.

Создание новых очередей отложенных действий

Если для выполнения поставленных целей недостаточно стандартной очереди отложенных действий, то можно создать новую очередь и соответствующие рабочие потоки. Поскольку при этом создается по одному рабочему потоку на каждый процессор, новые очереди отложенных действий нужно создавать только в случае, если необходима большая производительность за счет выделенного набора потоков.

Новая очередь отложенных действий и связанные с ней рабочие потоки создаются с помощью простого вызова функции create_workqueue, как показано ниже.

```
struct workqueue_struct *create_workqueue(const char *name);
```

В параметре name задаются имена рабочих потоков. Например, стандартная очередь с рабочими потоками events создается с помощью приведенного ниже кода.

```
struct workqueue_struct *keventd_wq;
keventd_wq = create_workqueue("events");
```

В результате будут созданы *все* рабочие потоки (по одному на каждый процессор в системе) и выполнены подготовительные операции для выполнения ими своей работы.

Создание отложенных действий выполняется одинаково, независимо от типа очереди. После этого для их планирования на выполнение используются приведенные ниже функции, аналогичные `schedule_work()` и `schedule_delayed_work()`. Они отличаются тем, что работают с указанной, а не стандартной очередью отложенных действий.

```
int queue_work(struct workqueue_struct *wq, struct work_struct *work)

int queue_delayed_work(struct workqueue_struct *wq,
struct work_struct *work,
unsigned long delay)
```

И наконец, ожидание завершения отложенных в заданной очереди действий выполняется с помощью функции `flush_workqueue`, как показано ниже.

```
flush_workqueue(struct workqueue_struct *wq)
```

Эта функция аналогична функции `flush_scheduled_work()`, как описывалось выше, за исключением того, что она ожидает, пока заданная очередь не станет пустой.

Старый механизм очередей задач

Так же как и в случае интерфейса BH, который дал начало интерфейсам отложенных прерываний (softirq) и тасклетов (tasklet), интерфейс очередей отложенных действий возник благодаря недостаткам интерфейса очередей задач (task queue). Интерфейс очередей задач (который в ядре называют просто *tq*), так же как и тасклеты, не имеет ничего общего с задачами (task), в смысле с процессами[8]. Все подсистемы, которые использовали механизм очередей задач, 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 задач. Все, что осталось от интерфейса очередей задач, перешло в интерфейс очередей отложенных действий. Совершим небольшой экскурс в историю и рассмотрим очереди задач, которыми пользовались в течение достаточно долгого времени.

Интерфейс очередей задач позволял определять набор очередей. Очередям были присвоены имена, такие как `scheduler queue` (очередь планировщика), `immediate queue` (очередь немедленного запуска) и `timer queue` (очередь таймера). Задачи из каждой очереди запускались на выполнение в определенных местах ядра. Поток ядра `keventd` выполнял работу, связанную с очередью планировщика. Эта очередь была предшественником интерфейса очередей отложенных действий. Очередь таймера запускалась при поступлении каждого импульса от системного таймера. Очередь немедленного запуска обрабатывалась в нескольких местах ядра, чтобы гарантировать "немедленное" выполнение действий (*какая халтура!*). Кроме перечисленных в ядре, также существовали и другие очереди. Кроме того, можно было динамически создавать новые очереди.

На первый взгляд все сказанное выше могло показаться очень полезным, но на практике интерфейс очередей задач не обладал четкой структуризацией и был довольно спонтанным. По сути, все очереди представляли собой произвольные абстракции и были разбросаны по всему ядру, словно кто-то подбросил их в воздух, и они упали в ядро, как

[8] Названия для механизмов обработчиков нижних половин, очевидно, выбираются из соображений конспирации, чтобы сбивать с толку молодых и неопытных разработчиков ядра. А если говорить серьезно, то это еще один пример неудачного выбора терминологии.

на землю. Единственной ценной очередью оказалась очередь планировщика, которая позволяла выполнять отложенные действия в контексте процесса.

Еще одним преимуществом механизма очередей задач была простота интерфейса. Несмотря на большое количество очередей и разнообразие правил, по которым они обрабатывались, интерфейс был максимально прост. Все остальное, что касается очередей задач, необходимо было убрать.

Со временем различные очереди задач были заменены другими механизмами обработки нижних половин — главным образом тасклетами. Осталось только то, что касалось очереди планировщика. В конечном итоге код демона `keventd` был обобщен в отличный механизм очередей отложенных действий, который мы имеем сегодня, а очереди задач были полностью удалены из ядра.

Какие механизмы обработчиков нижних половин следует использовать

Решение о том, какой из механизмов обработчиков нижних половин следует использовать, является очень 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 механизма отложенных прерываний, и поэтому они очень похожи. Механизм очередей отложенных действий полностью отличается от них и построен на основе потоков 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. Дело в том, что несколько экземпляров отложенного прерывания одного и того же типа могут выполняться параллельно на разных процессорах. Если рассматриваемый код уже очень хорошо распараллелен для многопоточного выполнения, как, например, в сетевой подсистеме, которая использует данные, связанные с процессорами, то отложенные прерывания — это хороший выбор. Они предоставляют самый быстрый механизм обработки для критичных ко времени или частоте выполнения задач.

Тасклеты имеет смысл использовать для кода, который не очень хорошо распараллелен для многопоточной обработки. Они имеют более простой интерфейс, и поскольку тасклеты одного типа не могут выполняться параллельно, то их легче реализовать. 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 столкнуться 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 — очереди отложенных 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 отложенных действий сильнее нагружают систему, поскольку в них используются потоки ядра и, соответственно, переключение контекста. Нельзя сказать, что они так уж и не эффективны, но в свете тех нескольких тысяч прерываний в секунду, которые может сгенерировать сетевая подсистема,

использование других механизмов может иметь больший смысл. Хотя в большинстве случаев очереди отложенных действий бывает вполне достаточно.

В плане простоты использования пальма первенства принадлежит очереди отложенных действий. Использование стандартной очереди `events` — это просто детские игры. Далее идут тасклеты, которые тоже имеют простой интерфейс. Последними в списке стоят отложенные прерывания, которые должны быть определены статически и поэтому требуют особого внимания при их реализации.

В табл. 8.3 приведено сравнение различных механизмов обработчиков нижних половин.

Таблица 8.3. Сравнение механизмов обработчиков нижних половин

Механизм	Контекст выполнения	Последовательное выполнение
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 говорить коротко, то разработчики обычных драйверов имеют всего два варианта выбора. Сначала следует решить, необходимо ли использовать возможности планировщика для выполнения отложенных действий, т.е. необходимо ли переходить в состояние ожидания по какой-либо причине? Если ответ на этот вопрос положительный, то единственный вариант выбора — очереди отложенных действий. В противном случае предпочтительнее использовать тасклеты. И только если важна масштабируемость, то стоит обратиться к механизму отложенных прерываний.

Блокировки между нижними половинами обработчиков

До сих пор мы вовсе не рассматривали вопросы, связанные с блокировками. Эти темы настолько занимательны, сложны и обширны, что им целиком посвящены следующие две главы. Тем не менее важно понимать, что решающим моментом при разработке нижней половины обработчика является защита общих данных доступа от конкурентных изменений, даже на однопроцессорной машине. Следует помнить, что нижняя половина обработчика прерывания потенциально может выполняться в любой момент времени. Если концепция блокировок вам еще не знакома, прочитайте следующие две главы, а затем вернитесь к данному разделу.

Одно из преимуществ использования тасклетов состоит в том, что они всегда выполняются последовательно по отношению к самим себе: один и тот же тасклет никогда не выполняется параллельно самому себе даже на двух разных процессорах. Это означает, что нет необходимости заботиться о проблемах, связанных с параллельным выполнением тасклетов одного типа. Однако при параллельном выполнении тасклетов разных типов (в случае, если они совместно используют одни и те же данные) следует использовать блокировки.

Так как отложенные прерывания не обеспечивают строгой последовательности выполнения (даже два обработчика одного и того же отложенного прерывания могут выполняться параллельно), все совместно используемые данные требуют соответствующих блокировок.

Если некоторые данные могут совместно использоваться в контексте процесса и в нижней половине обработчика прерываний, то перед обращением к ним нужно запретить обработку нижних половин и захватить соответствующую блокировку. Это позволяет гарантировать защиту совместно используемых данных как на локальном процессоре, так и на разных процессорах SMP-системы, а также предотвратить взаимоблокировку.

Если имеются данные, которые могут совместно использоваться и в контексте прерывания, и в нижней половине обработчика, то перед обращением к ним нужно запретить прерывания и захватить соответствующую блокировку. Эти две операции также позволяют предотвратить взаимоблокировку и обеспечить защиту совместно используемых данных как на локальном процессоре, так и на разных процессорах SMP-системы.

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, поскольку очереди действий всегда выполняются в контексте процесса.

В главе 9, "Общие сведения о синхронизации кода ядра", будут рассмотрены вопросы, относящиеся к проблеме параллельной обработки данных, а в главе 10, "Средства синхронизации ядра", приведены основные сведения о выполнении блокировок в ядре. Прочитав эти две главы, вы узнаете, как можно защитить данные, которые используются в нижних половинах обработчиков прерываний.

Запрещение обработки нижних половин

Обычно только одного запрещения обработки нижних половин недостаточно. Чаще всего для полной защиты совместно используемых данных нужно захватить соответствующую блокировку *и* запретить обработку нижних половин. Методы, которые позволяют это сделать, обычно используются при разработке драйверов, и описаны в главе 10, "Средства синхронизации ядра". Однако при разработке самого кода ядра иногда нужно просто запретить обработку нижних половин.

Для запрещения запуска всех типов обработчиков нижних половин (всех отложенных прерываний и, соответственно, тасклетов), вызывается функция `local_bh_disable()`. Для разрешения обработки нижних половин используется функция `local_bh_enable()`. Само собой разумеется, что у этих функций "неправильные" названия. Никто не потрудился переименовать эти функции, когда интерфейс BH уступил место интерфейсу отложенных прерываний. Эти функции описаны в табл. 8.4.

Таблица 8.4. Функции управления обработкой нижних половин

Функция	Описание
`void local_bh_disable()`	Запрещает обработку всех отложенных прерываний (softirq) и тасклетов (tasklet) на локальном процессоре
`void local_bh_enable()`	Разрешает 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 запуск нижних половин обработчиков прерываний. Например, при первом вызове функции `local_bh_disable()` запрещается выполнение отложенных прерываний на текущем процессоре. Если эта функция вызывается еще три раза, то выполнение отложенных прерываний на локальном процессоре 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 preempt_count, который поддерживается для каждой задачи (интересно, что этот же счетчик используется и для вытеснения процессов в режиме ядра)[9]. Когда значение этого счетчика достигает нуля, обработка нижних половин разрешается. Поскольку при вызове функции local_bh_enable() обработка нижних половин была запрещена, эта функция также проверяет наличие ожидающих запуска нижних половин обработчиков прерываний и выполняет их.

Реализация описанных выше функций уникальна для каждой поддерживаемой аппаратной платформы и обычно осуществляется с помощью сложных макросов, описанных в файле <asm/softirq.h>. Для любопытных ниже приведены соответствующие реализации на языке программирования C.

```
/*
 * Запретим обработку нижних половин, увеличив на единицу
 * значение счетчика preempt_count
 */
void local_bh_disable(void)
{
   struct thread_info *t = current_thread_info();

   t->preempt_count += SOFTIRQ_OFFSET;
}

/*
 * Уменьшим на единицу значение счетчика preempt_count.
 * Обработка нижних половин 'автоматически' разрешается,
 * когда значение этого счетчика достигнет нуля.
 *
 * По необходимости запустим все ожидающие выполнения нижние
 * половины обработчиков прерываний.
 */
void local_bh_enable(void)
{
   struct thread_info *t = current_thread_info();

   t->preempt_count -= SOFTIRQ_OFFSET;

   /*
    * Проверим, не достиг ли нуля счетчик preempt_count и нет ли в очереди
    * ожидающих выполнения нижних половин обработчиков прерываний
    */
    if (unlikely(!t->preempt_count &&
softirq_pending(smp_processor_id())))
       do_softirq();
}
```

Приведенные выше функции не запрещают обработку очередей отложенных действий. Поскольку очереди отложенных действий запускаются в контексте процесса, нет

[9] На самом деле этот счетчик используется как системой обработки прерываний, так и системой обработки нижних половин. Наличие одного счетчика для задачи позволяет реализовать в операционной системе Linux атомарность задач. Как показала практика, такой подход очень полезен для нахождения ошибок, связанных с тем, что задание переходит в состояние ожидания в то время, когда выполняет атомарные операции (sleeping-while-atomic bug).

никаких проблем с их асинхронным выполнением, а следовательно, нет необходимости их запрещать. Из-за того что отложенные прерывания и тасклеты могут "возникать" асинхронно (например, при возвращении из обработчика аппаратного прерывания), то в коде ядра может потребоваться запрещать их. В случае использования очередей отложенных действий защита совместно используемых данных осуществляется так же, как и в любой программе, работающей в контексте процесса. Подробнее об этом речь пойдет в главе 9, "Общие сведения о синхронизации кода ядра", и главе 10, "Средства синхронизации ядра".

Резюме

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 работают и реализованы эти механизмы. Также обсуждались основные моменты, связанные с использованием этих механизмов в собственном программном коде, и отмечалось, что их названия выбраны неудачно. Для того чтобы восстановить историческую справедливость, мы также рассмотрели те механизмы нижних половин обработчиков прерываний, которые существовали в предыдущих версиях ядра Linux: BH и очередь задач.

Очень часто в настоящей главе поднимались вопросы синхронизации и параллельного выполнения кода, поскольку они имеют прямое отношение к нижним половинам обработчиков. В главу специально был включен раздел, посвященный запрещению обработки нижних половин, в котором шла речь о защите совместно используемых данных от конкурентного доступа. Теперь пришло время погрузиться в эти темы с головой. В главе 9, "Общие сведения о синхронизации кода ядра", будут рассмотрены особенности синхронизации и параллельного выполнения кода в ядре. Этот материал позволит вам изучить основы и понять основные проблемы, которые часто приходится устранять. В главе 10, "Средства синхронизации ядра", будут рассмотрены интерфейсы, позволяющие осуществлять синхронизацию в ядре и решать указанные проблемы. Вооружившись материалов следующих двух глав, вы сможете покорить мир!

9

Общие сведения о синхронизации кода ядра

В приложениях, использующих *общую память* (shared memory), разработчики должны позаботиться о том, чтобы совместно используемые ресурсы были защищены от конкурентного доступа. В этом смысле ядро Linux не является исключением. Совместно используемые ресурсы требуют защиты от конкурентного доступа в связи с тем, что несколько одновременно запущенных потоков команд[1] могут одновременно изменять одни и те же данные. В результате один поток может затереть в памяти изменения, внесенные другим потоком, или обратиться к данным в тот момент, когда они находятся в несогласованном (т.е. противоречивом) состоянии. Конкурентный доступ к совместно используемым данным часто является причиной нестабильной работы системы, которую, как показывает опыт, впоследствии очень сложно обнаружить и исправить. В связи с этим важно при разработке кода *сразу* сделать все правильно.

Выполнить корректную защиту совместно используемых ресурсов не всегда просто. Много лет назад, когда операционная система Linux еще не поддерживала симметричную многопроцессорную обработку, предотвратить конкурентный доступ к данным было просто. Поскольку в системе существовал только один процессор, конкурентный доступ к данным мог возникнуть только при получении аппаратного прерывания или при явном вызове планировщика задач, в результате которого управление передавалось другой задаче. Да, раньше жизнь разработчиков ядра была проще!

Теперь это безмятежное время закончилось. Поддержка симметричной многопроцессорной обработки была введена в ядрах серии 2.0, и с тех пор она постоянно совершенствуется. Поддержка мультипроцессорности предполагает, что код ядра может одновременно выполняться на нескольких

[1] Под термином *поток команд* (thread), или просто поток, подразумевается любой выполняющийся в системе фрагмент кода. В качестве примера можно привести задачу в ядре, обработчик прерывания и его нижнюю половину или просто рабочий поток в ядре. Для краткости в этой главе будем называть все эти типы потоков команд одним термином — *поток*. Имейте в виду, что под этим термином подразумевается *любой* фрагмент выполняющегося кода.

процессорах. Следовательно, если не предпринимать никаких действий, фрагменты кода ядра, выполняющиеся на двух разных процессорах, могут обратиться к совместно используемым данным в один и тот же момент времени. Начиная с серии ядер 2.6 стала поддерживаться приоритетная многозадачность в ядре операционной системы Linux, т.е. оно стало вытесняемым. А это означает, что, если опять же не предпринимать никаких действий, системный планировщик может в произвольный момент времени прервать выполнение одного фрагмента кода ядра и передать управление другой задаче. На сегодняшний день существует довольно много сценариев возникновения конкурентного доступа к данным в ядре, и во всех из них требуется выполнить корректную защиту данных.

В этой главе мы кратко рассмотрим проблемы, связанные с параллельным выполнением кода и синхронизацией задач, возникающие в любой операционной системе. В следующей главе детально рассмотрены механизмы и интерфейсы, реализованные в ядре операционной системы Linux, предназначенные для решения проблем синхронизации и предотвращения конфликтов из-за доступа к системным ресурсам.

Критические участки и конфликты из-за доступа к системным ресурсам

Фрагменты кода, в которых выполняется доступ к совместно используемым данным и их изменение, называются *критическими участками* (critical sections). Как правило, нельзя безопасно обращаться к одному и тому же ресурсу одновременно из нескольких потоков команд. Для предотвращения конкурентного доступа к ресурсам из критических участков программист должен обеспечить *неделимое* выполнение кода. Другими словами, выполнение критического участка кода не должно прерываться, как если бы весь этот участок был одной неделимой машинной командой. Если возможна ситуация, когда критический участок кода может одновременно выполняться несколькими потоками команд, то это явная ошибка в программе. Однако если такое вдруг случается, то между потоками возникает так называемый *конфликт* из-за доступа к одному и тому же ресурсу (*race condition*). При этом потоки как бы соревнуются за то, кто первым получит к нему доступ. Обратите внимание: такую ситуацию довольно редко можно выявить при анализе кода программы. Кроме того, ее также тяжело выявить в процессе отладки, поскольку состояние конфликта не так-то легко воспроизвести. Процесс предотвращения небезопасного доступа к ресурсам и устранения конфликтов доступа к ним называется *синхронизацией (synchronization*).

Зачем вообще нужно что-то защищать?

Чтобы лучше понять, зачем нужна синхронизация, рассмотрим примеры из повседневной жизни, в которых часто возникают конфликты из-за доступа к ресурсам. В качестве первого примера рассмотрим ситуацию из реальной жизни — банкомат, который еще называют ATM (Automated Teller Machine), кеш-машиной (cash machine), банковским автоматом (cashpoint) или (за пределами США) ABM (Automatic Banking Machine).

Чаще всего банкомат используется для снятия денег с персонального банковского счета физического лица. Человек подходит к банкомату, вставляет пластиковую карточку, вводит PIN-код, проходит аутентификацию, выбирает пункт меню Выдача 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 необходимо вычесть снимаемую сумму из общего количества доступных денег. Код, который выполняет эту операцию, может выглядеть следующим образом:

```
/* Общее количество денег на счету */
int total = get_total_from_account();

/* Запрошенная сумма */
int withdrawal = get_withdrawal_amount();

/* Проверим, есть ли у пользователя деньги на счету */
if (total < withdrawal) {
    error("На вашем счету нет требуемой суммы!")
    return -1;
}

/* Все в порядке, у пользователя достаточно денег.
 * Теперь нужно вычесть снимаемую сумму из общей суммы денег на счету */
total -= withdrawal;
update_total_funds(total);

/* Выдадим деньги пользователю */
spit_out_money(withdrawal);
```

А теперь представим, что одновременно со счета этого же пользователя снимается еще одна сумма денег. Не имеет значения, *каким* образом это делается. Например, жена пользователя может снять деньги с другого банкомата, или кто-то еще выполняет электронный платеж, или переводит деньги со счета на счет, или банк снимает нужную сумму со счета в качестве платы за что-то (как это обычно любят делать банки). Любой из описанных случаев подходит для данного примера.

В обеих системах, где снимаются деньги со счета, выполняется код, аналогичный приведенному выше. Сначала проверяется, что снятие денег возможно, после этого вычисляется остаток денег на счету и, наконец, деньги снимаются физически. А теперь рассмотрим данный пример в конкретных числах. Допустим, 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 видеть не хотят). Предположим также, что у пользователя на счету есть сумма, равная $105. Очевидно, что одна из двух транзакций не может завершиться успешно без получения минуса на счету.

Вы можете предположить, что 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 выполняться снятие денег через банкомат, но оно завершится неудачно, так как $95 — это меньше, чем $100.

Без конфликтов из-за доступа к ресурсам жизнь была бы намного интереснее! Предположим, две описанные выше транзакции начинаются практически в один и тот же момент времени. Проверки состояния счета в обеих транзакциях пройдут успешно, ведь на счету достаточно денег: $105 — это больше $100 и больше $10. После этого запускается процесс снятия денег из банкомата, в результате которого из $105 вычитается $100 и в остатке на счету получается $5. В это же самое время в процессе снятия платы за вход в банк из $105 вычитается $10, и в остатке на счету получается уже $95. Далее в процессе снятия денег обновляется состояние счета клиента — на его счету окажется сумма $5. В конце концов завершается транзакция снятия платы за вход в банк, в процессе которой также обновляется состояние счета клиента, на котором окажется уже $95. Получаем деньги из воздуха!

Очевидно, что финансовые учреждения должны гарантировать, чтобы такая ситуация никогда не произошла. Необходимо блокировать счет во время выполнения некоторых операций, чтобы гарантировать неделимость транзакций по отношению к другим транзакциям. Такие транзакции должны либо полностью выполняться, не прерываясь, либо не запускаться вовсе.

Общая переменная

А теперь рассмотрим более специфичный пример, связанный с компьютерами. Предположим, у нас есть общий ресурс — глобальная переменная целого типа и простой критический участок, в котором значение этой переменной увеличивается на единицу.

```
i++;
```

Это выражение можно перевести в команды процессора вычислительной машины следующим образом:

```
Загрузить текущее значение переменной i из памяти в регистр.
Прибавить единицу к значению, которое находится в регистре.
Записать новое значение переменной i обратно в память.
```

Теперь предположим, что этот критический участок одновременно выполняется двумя потоками команд и что начальное значение переменной `i` равно `7`. Тогда желаемая последовательность действий показана в табл. 9.1, где каждой строке соответствует одна единица времени.

Таблица 9.1. Желаемая последовательность действий при выполнении критического участка

Поток 1	Поток 2
Загрузить **i** (7)	–
Увеличить **i** на 1 (7 -> 8)	–
Записать **i** (8)	–
–	Загрузить **i** (8)
–	Увеличить **i** на 1 (8 -> 9)
–	Записать **i** (9)

Как и следовало ожидать, если к числу `7` два раза прибавить по единице, в результате получим число `9`. Однако описанный выше сценарий может выполняться иначе, как показано в табл. 9.2.

Таблица 9.2. Возможная последовательность действий при выполнении критического участка

Поток 1	Поток 2
Загрузить **i** (7)	Загрузить **i** (7)
Увеличить **i** на 1 (7 -> 8)	–
–	Увеличить **i** на 1 (7 -> 8)
Записать **i** (8)	–
–	Записать **i** (8)

Если значение переменной `i` прочитано в обоих потоках *до* ее увеличения, то в обоих потоках будет увеличено на единицу и сохранено в память одно и то же значение. В результате значение переменной `i` будет равно `8`, тогда как по идее оно должно быть равно `9`. Мы привели пример одного из самых простых критических участков. К счастью, существует простое решение этой проблемы — необходимо сделать так, чтобы описанные выше три операции процессора выполнялись как одна непрерываемая команда. В большинстве процессоров для этой цели предусмотрена специальная машинная команда инкремента, которая, не прерываясь, читает значение переменной из памяти в регистр, увеличивает его на единицу и сохраняет обратно в память. При использовании этой *неделимой команды* возможны только два варианта развития событий, которые показаны в табл. 9.3.

Таблица 9.3. Последовательность действий при использовании неделимой команды инкремента

Поток 1	Поток 2
`Увеличить i на 1 и записать в память (7 -> 8)`	–
–	`Увеличить i на 1 и записать в память (8 -> 9)`
либо так	
–	`Увеличить i на 1 и записать в память (7 -> 8)`
`Увеличить i на 1 и записать в память (8 -> 9)`	–

Результаты работы двух неделимых команд никогда не могут перекрывать друг друга. Это гарантируется на физическом уровне при разработке процессора. Использование подобной машинной команды полностью решает нашу проблему. В ядре операционной системы предусмотрен ряд интерфейсов, которые позволяют реализовать подобные неделимые операции. Они и будут рассмотрены в следующей главе.

Блокировки

А теперь рассмотрим более сложный пример конфликта из-за доступа к ресурсу, который нельзя так просто решить, как это было описано в предыдущем разделе. Предположим, у нас есть очередь запросов, которые нужно обработать. Для примера будем считать, что очередь реализована в виде связанного списка, каждый элемент которого соответствует одному запросу. Для работы с очередью существуют две функции. Одна из них добавляет новый запрос в конец очереди, а другая — извлекает запрос из начала очереди и делает с ним что-то полезное. Эти функции могут вызываться из различных частей ядра, поэтому запросы будут постоянно добавляться, удаляться и обрабатываться. Очевидно, что обработку запроса, находящегося в очереди, нельзя выполнить с помощью всего одной машинной команды. Поэтому если один из потоков попытается извлечь очередной запрос из начала очереди, а другой в это время будет находиться в самом разгаре его обработки, то это означает, что для первого потока очередь будет находиться в неопределенном и несогласованном состоянии. Уже должно быть понятно, что при конкурентном доступе

к элементам очереди ее данные могут быть запорчены. Часто случается, что если совместно используемый ресурс представляет собой сложную структуру данных, то в результате возникновения конфликта из-за доступа к нему данные этой структуры разрушаются.

На первый взгляд может показаться, что описанная выше ситуация не имеет своего логичного и очевидного решения. Как можно предотвратить чтение очереди на одном процессоре в тот момент, когда другой процессор ее обновляет? Вполне логично выглядит реализация на аппаратном уровне простых неделимых арифметических или логических машинных команд. Однако было бы полным абсурдом реализовывать на аппаратном уровне команды для поддержки критических участков кода неопределенного размера, как в приведенном выше примере. В данном случае нам нужен какой-то механизм, с помощью которого можно было бы гарантировать, что в произвольный момент времени только один поток команд может обрабатывать конкретную структуру данных. Этот механизм должен предотвращать доступ к ресурсу на то время, пока другой поток выполняет команды, расположенные в критическом участке.

И такой механизм был создан, он называется *блокировкой* (*lock*). Его работа во многом напоминает дверной замок (lock). Представьте себе, что комната, которая находится за дверью, и есть наш критический участок. В комнате в произвольный момент времени может находиться только один поток команд. Как только поток заходит в комнату, он запирает за собой дверь. Когда поток заканчивает обработку совместно используемых данных, он отпирает дверь и выходит из комнаты. Если другой поток подойдет к двери, когда она заперта, то он должен будет подождать, пока поток, который находится в комнате, не отопрет дверь и не выйдет. Таким образом, поток захватывает блокировку (запирает замок), которая позволяет защитить данные.

В приведенном выше примере для защиты очереди запросов может использоваться всего одна блокировка. Если какому-то потоку потребуется добавить новый запрос в очередь, то вначале он должен захватить соответствующую блокировку. После этого он может безопасно поместить запрос в очередь и затем освободить блокировку. Если потоку понадобится извлечь запрос из очереди, он также должен сначала захватить блокировку. После этого он может проанализировать данные запроса, удалить его из очереди и только после этого освободить блокировку. То есть перед любым обращением к очереди нужно сначала захватить соответствующую блокировку. Поскольку в произвольный момент времени блокировку может захватить только один поток, то только он и может манипулировать элементами очереди в это время. Если какому-либо потоку понадобится обновить данные в очереди в то время, когда это делает другой поток, то первый поток должен будет подождать, пока второй поток не освободит блокировку. Таким образом, блокировка позволяет предотвратить конкурентный доступ к очереди и защитить ее от конфликтов из-за доступа к ресурсу.

Если в каком-то фрагменте кода требуется получить доступ к элементам очереди, то вначале нужно захватить соответствующую блокировку. Если при этом аналогичные действия нужно сделать в другом потоке команд, то эта блокировка предотвратит конкуренцию, как показано в табл. 9.4.

Таблица 9.4. Последовательность действий при использовании блокировок

Поток 1	Поток 2
`Захват блокировки`	`Захват блокировки`
`Успешно: блокировка захвачена`	`Отказ: ожидаем освобождения...`

Окончание табл. 9.4

Поток 1	Поток 2
`Доступ к очереди`	`Ожидаем...`
`Освобождение блокировки`	`Ожидаем...`
	`Успешно: блокировка захвачена`
	`Доступ к очереди`
	`Освобождение блокировки`

Обратите внимание: захват блокировки — дело *не обязательное* и *строго добровольное*. Блокировки — это чисто программные конструкции, которыми *должны* пользоваться программисты, желающие написать корректно работающий код. Ничто и никто не может помешать вам написать программный код, в котором обрабатываются элементы вымышленной очереди без всякой блокировки. Однако подобная практика, скорее всего, приведет к возникновению конфликтов из-за доступа к ресурсам и, как следствие, к разрушению данных.

Блокировки бывают различных "форм" и "размеров". В операционной системе Linux реализовано несколько разных механизмов блокировок. Основное отличие между ними состоит в реакции системы на ситуацию, когда требуемая блокировка недоступна, поскольку ранее была захвачена другим процессом. При захвате блокировок некоторых типов задачи должны ожидать их освобождения, постоянно выполняя проверку в *замкнутом цикле* (busy wait[2]), в то время как другие типы блокировок переводят задачу в состояние ожидания до освобождения блокировки. В следующей главе рассказывается о том, как ведут себя различные типы блокировок в операционной системе Linux, и об интерфейсах взаимодействия с этими блокировками.

Вдумчивый читатель в этом месте должен воскликнуть: "Блокировки не решают проблемы, они просто сужают набор всех возможных критических участков до кода, в котором захватываются и освобождаются блокировки. Тем не менее в них потенциально может возникать конфликт из-за доступа к ресурсам, хотя и с меньшей вероятностью!" К счастью, блокировки реализованы на основе неделимых операций, которые гарантируют, что конфликт из-за доступа к ресурсам никогда не возникнет. С помощью всего одной машинной команды можно проверить, захвачен ли ключ, и, если нет, захватить его. Конкретная реализация этого процесса зависит от используемой аппаратной платформы. Однако практически во всех процессорах существует неделимая машинная команда типа *test-and-set* (проверить и установить), которая позволяет проверить значение целочисленной переменной и присвоить ей указанное число, если ее значение равно нулю. Нулевое значение соответствует отсутствию блокировки. На популярной аппаратной платформе x86 блокировки реализованы с помощью аналогичной машинной команды, которая называется `CMPXCHG` (Compare and Exchange — сравнение с обменом).

Причины возникновения параллелизма

При работе в пространстве пользователя необходимость синхронизации возникает из-за того, что выполнение программы может быть принудительно прервано по воле планировщика. Поскольку в любой момент времени планировщик может вытеснить текущий

[2] Иными словами, "вращаются" (spin) в замкнутом цикле, ожидая освобождения блокировки.

процесс, выполняющийся на данном процессоре, и заменить его другим процессом, не исключено, что процесс может быть вытеснен по независящим от него причинам во время выполнения критического участка кода. 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 или записывать данные в общий файл), то может возникнуть конфликт из-за доступа к ресурсам. Аналогичная проблема может возникнуть при выполнении нескольких однопоточных программ, обращающихся к общему файлу, или даже в рамках одной программы, использующей сигналы, так как сигналы могут приходить асинхронно. Такой тип параллелизма, когда два события происходят не одновременно, а как бы накладываются друг на друга, вроде они происходят в один момент времени, называется *псевдопараллелизмом* (pseudo-concurrency).

На машине с симметричной многопроцессорной обработкой два процесса могут действительно заходить в критические участки кода в один и тот же момент времени. Такой вид параллелизма 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.

Причины возникновения параллелизма в ядре перечислены ниже.

- **Прерывания.** Могут возникать асинхронно, практически в любой момент времени, прерывая код, который выполняется в данный момент.
- **Отложенные прерывания и тасклеты.** Ядро может запустить обработчики отложенных прерываний и тасклетов практически в любой момент времени и прерывать код, который выполняется в данный момент времени.
- **Мультипрограммный режим работы ядра.** Так как ядро является вытесняемым, одна задача, которая работает в режиме ядра, может вытеснить другую задачу, тоже работающую в пространстве ядра.
- **Переход в состояние ожидания и синхронизация с пространством пользователя.** Задача, работающая в пространстве ядра, может переходить в состояние ожидания, что вызывает активизацию планировщика и выполнение нового процесса.
- **Симметричная многопроцессорная обработка.** Один и тот же код ядра может выполняться одновременно на нескольких процессорах.

Разработчики ядра должны осознать все эти причины и заранее предусмотреть в своем коде возможные случаи параллелизма. Если прерывание возникает во время выполнения кода, который работает с некоторым ресурсом, и обработчик прерывания также обращается к этому же ресурсу, то это является грубой ошибкой. Точно так же, если код ядра вытесняется в момент обращения к совместно используемому ресурсу, то это тоже является грубейшей ошибкой. Также ошибкой является случай, когда код ядра переводится в состояние ожидания во время выполнения критического участка кода. И наконец, два процессора никогда не должны одновременно обращаться к совместно используемым данным. Когда ясно, какие данные требуют защиты, уже нетрудно применить соответствующие блокировки, чтобы обеспечить стабильную работу системы. Гораздо сложнее идентифицировать возможные условия возникновения таких ситуаций и определить, что для предотвращения конкуренции необходима та или иная форма защиты.

А теперь подробнее рассмотрим перечисленные выше моменты, потому что они очень важны. Применить блокировки в коде для защиты совместно используемых данных

совсем не сложно, особенно если сделать это на самых ранних этапах разработки. Сложность как раз и состоит в том, чтобы найти эти самые совместно используемые данные и самые критические участки кода. Вот почему требование аккуратного использования блокировок с самого начала разработки кода — а не когда-нибудь потом — имеет первостепенную важность. Впоследствии будет очень сложно отследить, что необходимо блокировать, и правильно внести изменения в существующий код. Результаты подобной практики обычно не очень хорошие. Из всего этого вы должны сделать один важный вывод: нужно *всегда* аккуратно учитывать необходимость применения блокировок с самого начала процесса разработки кода.

Код, который можно безопасно выполнять параллельно с обработчиком прерывания, называют *устойчивым к прерываниям* (interrupt-safe). Код, который можно безопасно параллельно выполнять на нескольких процессорах симметричной многопроцессорной машины, называют *устойчивым к симметричной обработке* (SMP-safe). Код, который можно безопасно параллельно выполнять в вытесняемой среде ядра, называется *устойчивым к мультипрограммированию* (preempt-safe)[3]. Реальные механизмы, используемые для синхронизации и защиты от конфликтов за доступ к ресурсам во всех перечисленных выше случаях, будут рассмотрены в следующей главе.

Что нужно защищать?

Жизненно важно определить, какие данные требуют защиты. Поскольку практически все данные, к которым выполняется конкурентный доступ, нужно тем или иным способом защищать, чаще всего легче определить, какие данные *не требуют* защиты, и работать дальше, отталкиваясь от этого. Очевидно, что в пределах одного потока все локальные данные не требуют защиты, поскольку доступ к ним может выполняться только со стороны этого потока. Например, локальные переменные, которые автоматически выделяются в стеке при входе в процедуру, а также динамически создаваемые структуры данных (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 одна задача, не требуют применения блокировок, поскольку 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`**
>
> Поскольку ядро операционной системы Linux может быть сконфигурировано на этапе компиляции, имеет смысл "подогнать" параметры ядра под данный тип машины. Важной функцией ядра является поддержка симметричной многопроцессорной обработки (SMP), которая включается с помощью параметра конфигурации ядра `CONFIG_SMP`. Например, на однопроцессорной (uniprocessor, UP) машине не требуется выполнять многие типы блокировок, поскольку процессор всего один. Поэтому если не установить параметр конфигурации `CONFIG_SMP`, то код, в котором нет необходимости, не компилируется в исполняемый образ ядра. Например, подобная настройка позволяет на однопроцессорной машине отказаться от

[3] Далее будет показано, что за некоторыми исключениями код, устойчивый к симметричной обработке, будет также устойчив и к мультипрограммированию.

накладных расходов, связанных со *спин-блокировками* (spin locks). Аналогичный прием используется для параметра конфигурации ядра `CONFIG_PREEMPT`, который указывает, будет ли в ядре поддерживаться мультипрограммный режим работы. Подобный подход является примером отличного проектного решения, поскольку при этом появляется возможность поддерживать только один отлаженный базовый код ядра, а различные механизмы блокировок задействовать по мере необходимости. Различные комбинации параметров `CONFIG_SMP` и `CONFIG_PREEMPT` на разных аппаратных платформах позволяют компилировать в ядро различные механизмы блокировок.

При написании кода необходимо обеспечить соответствующие варианты защиты для самого сложного случая — работы ядра в мультипрограммном режиме на многопроцессорной машине.

При написании кода ядра следует всегда учитывать перечисленные ниже моменты.

- Являются ли данные глобальными? Может ли к ним обращаться, кроме текущего, и другой поток команд?
- Будут ли данные совместно использоваться как в контекста процесса, так и в контексте прерывания? А может быть, они совместно используются в двух обработчиках 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) при выполнении какой-либо операции? Если да, то в каком состоянии он оставит все совместно используемые данные?
- Что мешает освободить память, в которой находятся данные?
- Что произойдет, если эта же функция будет вызвана на другом процессоре?
- Отталкиваясь от заданной точки, как можно обеспечить безопасное выполнение кода в конкурентной среде?

Если говорить коротко, то почти все глобальные и разделяемые данные в ядре требуют применения тех или иных методов синхронизации, которые будут рассмотрены в следующей главе.

Взаимоблокировки

Под *взаимоблокировкой* (deadlock) будем понимать ситуацию, при которой для продолжения работы одному или нескольким потокам требуется захватить один или несколько ресурсов, но при этом все ресурсы оказались занятыми. Все потоки переходят в состояние ожидания, пока какой-нибудь из них не освободит ресурсы, требуемые для работы другого потока. Однако это принципиально невозможно, поскольку все потоки находятся в состоянии ожидания. Таким образом, ни один из потоков не может продолжить свою работу, и возникает так называемая ситуация взаимоблокировки.

Хорошая аналогия — это перекресток, на котором стоят четыре машины, подъехавшие с четырех разных сторон. Если водители всех машин решат применить правило “правой руки” (т.е. каждый из них будет уступать дорогу тому, кто находится справа), то ни одна из машин не сдвинется с места, и мы получим тупиковую ситуацию, или взаимоблокировку дорожного движения.

Самый простой пример взаимоблокировки — это *самоблокировка*[4] (self-deadlock). Если поток команд попытается захватить блокировку, которую он уже захватил ранее, то ему необходимо дождаться, пока блокировка не будет освобождена. Однако поток никогда не сможет освободить блокировку, потому что он ожидает на ее захват, что и приводит к тупиковой ситуации, как показано ниже.

```
Захватить блокировку.
Захватить блокировку еще раз.
Ждать, пока блокировка не будет освобождена.
....
```

По аналогии предположим, что существует `n` потоков и `n` блокировок. Если каждый поток удерживает блокировку, освобождения которой ожидает другой поток, то все потоки будут заблокированы до тех пор, пока не освободятся те блокировки, освобождение которых они ожидают. Чаще всего в качестве примера приводят два потока и две блокировки и называют это *тупиковой ситуацией* (deadly embrace), или *взаимоблокировкой типа ABBA* (ABBA deadlock), как показано в табл. 9.5.

Таблица 9.5. Взаимоблокировка двух потоков

Поток 1	Поток 2
`Захват блокировки A`	`Захват блокировки B`
`Попытка захвата блокировки B`	`Попытка захвата блокировки A`
`Ожидание блокировки B`	`Ожидание блокировки A`

В данном случае оба потока будут ожидать друг друга, поскольку ни один из них, находясь в состоянии ожидания, не сможет освободить первоначально захваченную блокировку.

Важно не допускать появление взаимоблокировок. Несмотря на то что сложно проверить готовый код на наличие в нем взаимоблокировок, тем не менее *можно* написать код, в котором состояние взаимоблокировки никогда не возникнет. Ниже приведено несколько простых правил.

- Определите порядок захвата блокировок. Вложенные блокировки *всегда* должны захватываться в одном и том же порядке. Это предотвратит возникновение взаимоблокировки между несколькими потоками. Порядок захвата блокировок необходимо документировать, чтобы другие разработчики также могли его соблюдать.
- Не допускайте длительных ожиданий при доступе к ресурсу. Задайте себе простой вопрос: “Всегда ли этот код сможет завершить свою работу?” Если не выполнится какое-либо условие, то не будет ли что-то ожидать его вечно?
- Не захватывайте одну и ту же блокировку дважды.
- Стремитесь к простоте. Сложная система блокировок часто приводит к возникновению тупиковых ситуаций.

[4] В некоторых ядрах операционных систем такой тип тупиковой ситуации предотвращается с помощью рекурсивных блокировок, которые позволяют одному потоку 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.

Первый пункт — самый важный и наименее сложный для выполнения. Если одновременно нужно захватить две или более блокировки, то они *всегда* должны захватываться в строго определенном порядке. Предположим, у нас есть три блокировки: `cat`, `dog` и `fox`, которые используются для защиты структур данных с аналогичными именами. А теперь представьте себе, что у нас есть функция, которая должна работать с этими тремя структурами данных одновременно, — например, она должна распределить данные между структурами. В любом случае, для того чтобы гарантировать целостность данных, эти структуры нужно защитить с помощью блокировок. Так, если в одной из функций эти блокировки захватываются в следующем порядке: `cat`, `dog` и затем `fox`, то в *любой* другой функции эти же блокировки (или только некоторые из них) должны захватываться в том же порядке. Например, если в некоторой функции сначала будет захвачена блокировка `fox`, а затем — `dog`, то это может привести к возникновению взаимоблокировки (а значит, ошибки в работе), потому что блокировка `dog` всегда должна захватываться перед блокировкой `fox`. В табл. 9.6 приведен пример того, как это может произойти.

Таблица 9.6. Пример возникновения взаимоблокировки между двумя потоками

Поток 1	Поток 2
`Захват блокировки cat`	`Захват блокировки fox`
`Захват блокировки dog`	`Попытка захвата блокировки dog`
`Попытка захвата блокировки fox`	`Ожидание блокировки dog`
`Ожидание блокировки fox`	—

Первый поток ожидает освобождения блокировки `fox`, которую удерживает второй поток. При этом второй поток ожидает освобождения блокировки `dog`, которую удерживает первый поток. Ни один из потоков никогда не освободит свои блокировки, и, соответственно, оба потока будут ждать вечно — возникает тупиковая ситуация. Если оба потока будут всегда захватывать блокировки в одном и том же порядке, то подобной тупиковой ситуации возникнуть не может.

Если несколько процедур, в которых захватываются блокировки, вложены одна в другую, то должен быть принят определенный порядок их захвата. Хорошая практика — перед захватом блокировки поместите комментарий с описанием порядка ее захвата, как показано ниже.

```
/*
 * cat_lock - блокирует доступ к структуре данных cat.
 * Всегда должна захватываться перед блокировкой dog!
 */
```

Следует отметить, что порядок *освобождения* блокировок не приводит к появлению взаимоблокировок. Поэтому обычно блокировки освобождаются в обратном порядке по отношению к их захвату.

Очень важно не допускать возможности появления взаимоблокировок. В ядре Linux предусмотрено несколько простых отладочных средств, которые позволяют обнаружить взаимоблокировки при выполнении кода ядра. Эти средства будут рассмотрены в следующей главе.

Конфликт при блокировке и масштабируемость

Термин *конфликт при блокировке* (lock contention) используется для описания блокировки, которая в данный момент захвачена и освобождения которой ожидают другие потоки. Освобождения блокировки с *высоким уровнем конфликтов* (highly contended) всегда ожидает много потоков. Высокий уровень конфликтов может возникнуть, если некоторая блокировка захватывается слишком часто, удерживается продолжительное время или происходит и то и другое. Поскольку основная задача блокировки состоит в упорядочивании доступа к ресурсу, то нет нечего удивительного в том, что применение блокировок снижает производительность системы в целом. Блокировка с высоким уровнем конфликтов может стать узким местом в системе и существенно снизить ее производительность. Конечно, блокировки необходимы для того, чтобы предотвратить разрушение 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 сильно система может быть расширена. В случае операционных систем, когда говорят о масштабируемости, подразумевают большее количество процессов, большее количество процессоров или больший объем оперативной памяти. О маштабируемости можно говорить по отношению практически к любому компоненту компьютера, который можно охарактеризовать количественным параметром. В идеальном случае 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 позволила поразительным образом увеличить производительность системы в целом. Это произошло с того момента, когда в ядро серии 2.0 была введена поддержка многопроцессорной обработки. В те дни, когда поддержка многопроцессорности в операционной системе Linux только-только появилась, в режиме ядра в любой момент времени могла выполняться всего одна задача. В ядрах серии 2.2 это ограничение было снято, поскольку были придуманы мелко структурированные механизмы блокировок. В серии 2.4 и выше блокировки стали еще более мелкоструктурными. В современных ядрах серии 2.6 блокировки имеют очень мелкую структурированность, при которой получается очень хорошая масштабируемость.

Под структурированностью блокировки понимают размер или объемы данных, которые защищаются этой блокировкой. Применение очень грубой блокировки позволяет защитить большие объемы данных, например все структуры данных одной подсистемы. С другой стороны, блокировка на уровне очень мелких структурных единиц (fine grained), используется для защиты маленького объема данных, например одного поля большой структуры. В реальных ситуациях большинство блокировок находится между двумя этими крайностями. Они используются и не для защиты подсистемы в целом, и не для защиты одного поля или даже отдельного экземпляра структуры. Скорее всего, блокировки будут защищать отдельные структурные единицы или списки структур данных. Большинство блокировок начинали использовать на уровне крупных структурных единиц (cource graine), а потом их стали разделять на более мелкие структурные уровни, как только конфликты при захвате этих блокировок выходили на первый план.

Один из примеров перевода блокировок на более мелкий структурный уровень был приведен в главе 4, “Системный планировщик и диспетчеризация процессов”, где рассматривались блокировки очередей выполнения системного планировщика (runqueue).

В ядрах серии 2.4 и более ранних системный планировщик имел всего одну очередь выполнения (напомним, что в очереди выполнения хранится список готовых к выполнению процессов). В ранних версиях ядра серии 2.6 был предложен планировщик типа O(1), поддерживающий очереди выполнения для каждого процессора. Каждая из очередей защищалась своей уникальной блокировкой. Таким образом, система блокировок эволюционировала от одной глобальной блокировки для всей очереди до нескольких отдельных блокировок для каждой очереди, обрабатываемых отдельным процессором. Эта оптимизация была очень существенной, поскольку на машинах с большим количеством процессоров блокировка всей очереди выполнения имела очень высокий уровень конфликтов. В результате планирование всех процессов выполнялось строго последовательно. Иными словами, в любой момент времени код планировщика мог выполняться только на одном процессоре системы, а остальные процессоры при этом ожидали. В поздних версиях ядра серии 2.6 появился планировщик типа *CFS*, который улучшил масштабируемость.

В общем, за счет такого улучшения масштабируемости удалось кардинально повысить производительность операционной системы Linux на больших и более мощных системах. Однако чрезмерное увлечение "ростом" масштабируемости может привести к снижению производительности на небольших много- и однопроцессорных машинах. Дело в том, что для небольших машин не требуются такие мелкоструктурные блокировки, которые влекут за собой бо́льшую сложность системы и только увеличивают накладные расходы. В качестве примера рассмотрим связанный список. В первоначальной схеме блокировки предусматривается одна блокировка на весь список. Со временем эта единственная блокировка может стать узким местом на большой многопроцессорной машине, на которой очень часто обращаются к связанному списку. Для решения проблемы одна блокировка может быть разбита на несколько блокировок на уровне элементов списка. В результате перед каждым обращением к любому элементу этого списка необходимо захватить его уникальную блокировку. Теперь конфликт при захвате блокировки будет возникать только в случае, когда к одному элементу списка одновременно будут обращаться несколько процессоров. Однако что делать, если эти меры не помогли и в системе все равно остается высокий уровень конфликтов при блокировках? Стоит ли в таком случае использовать блокировку для каждого поля элемента списка? Или для каждого бита каждого элемента? Ответ на эти вопросы — *нет!* Несмотря на то что такие мелко структурированные блокировки позволяют добиться великолепного уровня масштабируемости на больших многопроцессорных машинах, как они будут работать на обычных двухпроцессорных машинах? Если на двухпроцессорной машине нет существенных конфликтов при работе с блокировками, то все эти меры, связанные с дополнительными блокировками, будут напрасными.

Тем не менее масштабируемость — это очень важный фактор. Необходимо с самого начала разрабатывать схему блокировок с учетом обеспечения хорошей масштабируемости. Блокировки на уровне крупных структурных единиц могут стать узким местом даже на машинах с небольшим количеством процессоров. Между крупно- и мелкоструктурными блокировками существует очень тонкая грань. Слишком крупноструктурные блокировки приводят к большому уровню конфликтов и снижают масштабируемость, а слишком мелкоструктурные — к напрасным накладным расходам, если уровень конфликтов при захвате блокировок остается не очень высоким. Оба варианта приводят к снижению производительности системы. *Необходимо начинать с простого и переходить к сложному только по мере необходимости. Простота — залог успеха!*

Резюме

Разработку кода, устойчивого к симметричной обработке, нельзя откладывать на потом. Правильная синхронизация, система блокировок без возникновения тупиковых ситуаций, масштабируемость и ясность кода — все эти факторы следует учитывать при разработке с самого начала и до самого конца. При написании кода ядра, будь то новая системная функция или модификация драйвера устройства, необходимо прежде всего позаботиться об обеспечении защиты данных от конкурентного доступа.

Обеспечение достаточной защиты для любого возможного случая, как-то использование симметричной многопроцессорной обработки, мультипрограммный режим работы ядра и тому подобное, приведет к тому, что все данные будут надежно защищены на любой машине и при любой конфигурации. В следующей главе речь пойдет о том, как это осуществить.

После изучения основ и теории синхронизации, параллельной обработки и блокировок пора перейти к рассмотрению конкретных средств, предусмотренных в ядре Linux для устранения конфликтов при доступе к ресурсам и взаимоблокировок в коде.

10
Средства синхронизации ядра

В предыдущей главе рассматривались причины возникновения конфликтов из-за доступа к ресурсам и способы их разрешения. К счастью, в ядре Linux реализовано большое семейство средств синхронизации, которые позволяют разработчикам писать эффективный и свободный от конфликтов код. В этой главе обсуждаются эти средства, интерфейсы к ним, а также особенности их работы и использования.

Неделимые операции

Начнем обсуждение средств синхронизации с рассмотрения неделимых операций, потому что именно они составляют основу, на которой построены эти средства. *Неделимые операции* (atomic operations) представляют собой набор машинных команд, которые выполняются как одна *цельная* машинная команда, т.е. не прерываясь. В этом смысле они чем-то напоминают атом, который вначале тоже считался неделимой частицей. Поэтому по аналогии такие операции часто называют атомарными. Например, в предыдущей главе была рассмотрена неделимая операция инкремента, которая позволяла прочитать из памяти значение переменной, увеличить ее на единицу и снова сохранить в памяти за один неделимый и непрерываемый шаг. Напомним простой пример из предыдущей главы с возникновением конфликтов из-за доступа к общему ресурсу, который возникал при увеличении значения переменной на единицу (табл. 10.1).

Таблица 10.1. Пример возникновения конфликта из-за доступа к общему ресурсу

Поток 1	Поток 2
`Загрузить i (7)`	`Загрузить i (7)`
`Увеличить i на 1 (7 -> 8)`	–
–	`Увеличить i на 1 (7 -> 8)`
`Записать i (8)`	–
–	`Записать i (8)`

При использовании неделимых операций подобный конфликт из-за доступа к общей переменной не может возникнуть в принципе. События будут происходить по одному из двух сценариев, описанных в табл. 10.2.

Таблица 10.2. Последовательность действий при использовании неделимой операции инкремента

Поток 1	Поток 2
Получить значение **i**, увеличить его на 1 и записать в память (7 -> 8)	–
–	Получить значение **i**, увеличить его на 1 и записать в память (8 -> 9)
либо так	
–	Получить значение **i**, увеличить его на 1 и записать в память(7 -> 8)
Получить значение **i**, увеличить его на 1 и записать в память(8 -> 9)	–

В результате всегда получится правильное значение — девять. Две неделимые операции никогда не могут одновременно обратиться и изменить значение одной и той же переменной. Они всегда выполняются строго последовательно, поэтому для операции неделимого инкремента никогда не может возникнуть конфликт из-за доступа к ресурсам.

В ядре предусмотрены два набора интерфейсов для выполнения неделимых операций. Один из них предназначен для работы с целыми числами, а другой — для операций с отдельными битами. Эти интерфейсы реализованы для всех аппаратных платформ, на которых может работать операционная система Linux. На большинстве аппаратных платформ предусмотрен специальный набор машинных команд, выполняющих простые неделимые арифметические операции. Если на какой-либо аппаратной платформе такой набор команд не предусмотрен, то на время доступа к переменной в памяти процессор блокирует шину данных. Это позволяет предотвратить одновременный доступ к памяти со стороны других процессоров.

Неделимые целочисленные операции

Средства выполнения неделимых операций с целыми числами работают со специальным типом данных `atomic_t`. Вместо того чтобы использовать функции, которые работают непосредственно с типом данных `int` языка C, по ряду причин используется специальный тип данных. Во-первых, в функции, выполняющие неделимые операции, можно передать только аргументы типа `atomic_t`. Это гарантирует, что неделимые операции будут выполняться только с данными этого специального типа. Более того, это также гарантирует, что данные типа `atomic_t` нельзя будет передать в другие функции, которые не выполняют неделимых операций. В самом деле, что хорошего будет в таких неделимых операциях, если нарушится единообразие в используемых данных? Во-вторых, использование типа `atomic_t` позволяет гарантировать, что компилятор для повышения эффективности (в данном случае это совсем не нужно!) не будет оптимизировать операции обращения к таким переменным. Важно, чтобы при выполнении неделимых операций использовалось правильное значение адреса переменной в памяти, а не адреса ее временных копий. В-третьих, использование типа `atomic_t` позволяет не

учитывать различия в реализации для конкретной аппаратной платформы. Тип atomic_t определен в файле <linux/types.h>, как показано ниже.

```
typedef struct {
   volatile int counter;
} atomic_t;
```

Несмотря на то что значение целого типа на всех аппаратных платформах, поддерживаемых Linux, является 32-разрядным, разработчики почему-то однажды решили, что значение типа atomic_t должно быть всего лишь 24-разрядным. Это было связано с аппаратной платформой SPARC, в которой реализация неделимых операций была несколько странной — значение флага блокировки содержалось в младших 8 битах 32-разрядного числа типа int, как показано на рис. 10.1. Этот флаг использовался для защиты данных при одновременном доступе к значениям типа atomic_t, поскольку на платформе SPARC не было предусмотрено специальных неделимых машинных команд. Следовательно, на машинах SPARC можно было использовать только 24 бита. Хотя код, который рассчитан на использование полного 32-битового диапазона значений, будет работать 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 реализовать тип atomic_t, который бы позволял хранить полноценное 32-разрядное целое число. Поэтому указанного ограничения больше не существует.

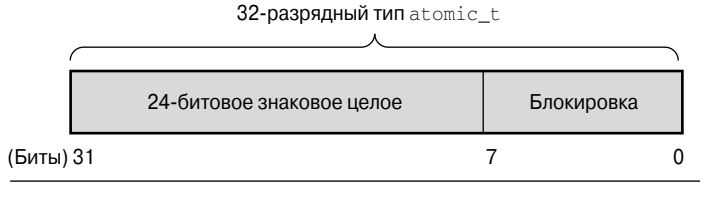


Рис. 10.1. Структура 32-битового типа atomic_t для аппаратной платформы SPARC в старой реализации

Определения, необходимые для использования неделимых целочисленных операций, находятся в файле <asm/atomic.h>. Для некоторых аппаратных платформ существуют дополнительные уникальные средства, но для всех аппаратных платформ существует минимальный набор операций, которые используются в ядре повсюду. При написании кода ядра необходимо убедиться в том, что соответствующие операции доступны и правильно реализованы для всех аппаратных платформ.

Объявление переменных типа atomic_t выполняется как обычно. По мере необходимости можно установить начальное значение этой переменной, как показано ниже.

```
atomic_t v;                    /* Объявление переменной v */
atomic_t u = ATOMIC_INIT(0); /* Объявление переменной u и присваивание
                                ей нулевого начального значения */
```

Все операции достаточно просты и понятны:

```
atomic_set(&v, 4); /* v = 4          (неделимая операция) */
atomic_add(2, &v); /* v = v + 2 = 6 (неделимая операция) */
atomic_inc(&v);    /* v = v + 1 = 7 (неделимая операция) */
```

Для преобразования типа atomic_t в тип int нужно использовать функцию atomic_read(), как показано ниже.

```
printk("%d\n", atomic_read(&v)); /* will print "7" */
```

Чаще всего неделимые целочисленные операции используются при реализации счетчиков. Защищать один счетчик с помощью сложной системы блокировок — это глупо, поэтому разработчики используют системные функции `atomic_inc()` и `atomic_dec()`, которые к тому же значительно быстрее.

Неделимые целочисленные операции используются также для выполнения ряда операций с проверкой. Чаще всего используется неделимая операция декремента с проверкой, как показано ниже.

```
int atomic_dec_and_test(atomic_t *v)
```

Эта функция уменьшает на единицу значение заданной переменной типа `atomic_t`. Если результат выполнения операции равен нулю, то возвращается значение `true`, иначе возвращается `false`. Полный список всех неделимых операций с целыми числами (т.е. тех, которые доступны для всех аппаратных платформ) приведен в табл. 10.3. Определения всех операций, которые реализованы для конкретной аппаратной платформы, приведены в файле `<asm/atomic.h>`.

Таблица 10.3. Полный список всех неделимых целочисленных операций

Операция	Описание
`ATOMIC_INIT(int i)`	Инициализация переменной типа `atomic_t` значением `i` во время объявления
`int atomic_read(atomic_t *v)`	Неделимое считывание значения целочисленной переменной `v`
`void atomic_set(atomic_t *v, int i)`	Неделимое присвоение переменной `v` значения `i`
`void atomic_add(int i, atomic_t *v)`	Неделимое прибавление значения `i` к переменной `v`
`void atomic_sub(int i, atomic_t *v)`	Неделимое вычитание значения `i` из переменной `v`
`void atomic_inc(atomic_t *v)`	Неделимое увеличение на единицу значения переменной `v`
`void atomic_dec(atomic_t *v)`	Неделимое уменьшение на единицу значения переменной `v`
`int atomic_sub_and_test(int i, atomic_t *v)`	Неделимое вычитание значения `i` из переменной `v`; возвращается `true`, если результат равен нулю, и `false` в противном случае
`int atomic_add_negative (int i, atomic_t *v)`	Неделимое прибавление значения `i` к переменной `v`; возвращается `true`, если результат меньше нуля, иначе возвратить `false`
`int atomic_add_return (int i, atomic_t *v)`	Неделимое прибавление значения `i` к переменной `v` и возврат результата
`int atomic_sub_return (int i, atomic_t *v)`	Неделимое вычитание значения `i` из переменной `v` и возврат результата
`int atomic_inc_return (atomic_t *v)`	Неделимое увеличение на единицу значения переменной `v` и возврат результата
`int atomic_dec_return (atomic_t *v)`	Неделимое уменьшение на единицу значения переменной `v` и возврат результата

Окончание табл. 10.3

Операция	Описание
`int atomic_dec_and_test (atomic_t *v)`	Неделимое вычитание единицы из переменной `v`; возвращается `true`, если результат равен нулю, иначе возвращается `false`
`int atomic_inc_and_test (atomic_t *v)`	Неделимое прибавление единицы к переменной `v`; возвращается `true`, если результат равен нулю, иначе возвращается `false`

Обычно неделимые операции реализуются в виде встраиваемых функций языка C со вставками на языке ассемблера. В случае если какая-либо из функций неделима по своей природе, то чаще всего она определяется в виде макроса. Например, для большинства распространенных аппаратных платформ считывание одного машинного слова данных — это неделимая операция. Другими словами, слово не может быть прочитано в середине цикла его записи. Поэтому операция чтения всегда возвращает корректное состояние слова либо до, либо после завершения операции записи, но никак не посередине. Следовательно, функция `atomic_read()` обычно реализуется в виде макроса, возвращающего целочисленное значение переменной типа `atomic_t`, как показано ниже.

```
/**
  * atomic_read - неделимое чтение значения переменной
  * @v: указатель на переменную типа atomic_t
  *
  * Неделимое чтение значения переменной @v.
  */
static inline int atomic_read(const atomic_t *v)
{
return v->counter;
}
```

Порядок выполнения неделимых команд

В приведенном выше описании неделимой операции чтения мы вскользь затронули тему порядка выполнения команд, которая требует дальнейшего пояснения. Как уже было сказано, операция чтения одного машинного слова всегда выполняется неделимо. Она никогда не перекрывается с операцией записи того же самого слова. Поэтому при чтении всегда возвращается корректное состояние этого слова, которое было либо перед началом операции записи, либо после нее, но никогда в процессе записи. Например, если целочисленное значение вначале было равно 42, а потом стало 365, то операция чтения всегда вернет значение 42 или 365, но никогда смесь двух значений. Это и называется *неделимостью*.

Иногда в процессе написания кода разработчику нужно усилить требования к выполнению команд, например, чтобы операция чтения всегда выполнялась *перед* ожидающей операцией записи. Этот тип требований относится *не* к неделимости команд, а к *порядку* их выполнения. Неделимость гарантирует, что команды выполняются не прерываясь и что они либо выполняются полностью, либо не выполняются совсем. Порядок выполнения же гарантирует, что две или более команды, если они выполняются в разных потоках или даже на разных процессорах, всегда будут выполнены в нужном порядке.

В этом разделе мы рассмотрели только неделимые операции, при выполнении которых гарантируется их непрерывность. Для гарантирования порядка выполнения используются *барьеры операций* (barrier operations), которые будут рассмотрены далее в текущей главе.

Как правило, при разработке кода, там, где это возможно, предпочтительнее использовать неделимые операции, а не сложные механизмы блокировок. Для большинства аппаратных платформ использование одной или двух неделимых операций вызывает меньшие накладные расходы и приводит к более эффективному использованию кешпамяти процессора, чем в случае более сложных методов синхронизации. Поэтому, если код критичен ко времени выполнения, всегда лучше протестировать несколько вариантов.

64-разрядные неделимые операции

По мере роста популярности 64-разрядных аппаратных платформ нет ничего удивительного в том, что разработчики ядра Linux на основе 32-разрядного типа `atomic_t` сделали его 64-разрядный вариант `atomic64_t`. С целью переносимости размер типа `atomic_t` не может варьироваться в зависимости от используемой аппаратной платформы, поэтому даже на 64-разрядных платформах тип `atomic_t` всегда остается 32-разрядным. Вместо него и был введен тип `atomic64_t`, который позволяет выполнять 64-разрядные неделимые операции, идентичные их 32-разрядным аналогам. Правила их использования остаются такими же, за исключением размеров операндов — вместо 32-разрядных теперь используются 64-разрядные. В 64-разрядном варианте реализованы практически все 32-разрядные неделимые функции, только вместо префикса `atomic` они имеют префикс `atomic64`. Их полный список приведен в табл. 10.4. На некоторых аппаратных платформах реализовано большее количество функций, но они не являются переносимыми. Так же как и `atomic_t`, тип `atomic64_t` является простой оболочкой для целочисленного типа, в данном случае `long`.

```
typedef struct {
   volatile long counter;
} atomic64_t;
```

Таблица 10.4. Список 64-разрядных неделимых целочисленных операций

Операция	Описание
`ATOMIC64_INIT(long i)`	Инициализация переменной типа `atomic64_t` значением `i` во время объявления
`long atomic64_read(atomic64_t *v)`	Неделимое считывание значения целочисленной переменной `v`
`void atomic64_set(atomic64_t *v, long i)`	Неделимое присвоение переменной `v` значения `i`
`void atomic64_add(long i, atomic64_t *v)`	Неделимое прибавление значения `i` к переменной `v`
`void atomic64_sub(long i, atomic64_t *v)`	Неделимое вычитание значения `i` из переменной `v`
`void atomic64_inc(atomic64_t *v)`	Неделимое увеличение на единицу значения переменной `v`
`void atomic64_dec(atomic64_t *v)`	Неделимое уменьшение на единицу значения переменной `v`
`long atomic64_sub_and_test(long i, atomic64_t *v)`	Неделимое вычитание значения `i` из переменной `v`; возвращается `true`, если результат равен нулю, и `false` в противном случае

Окончание табл. 10.4

Операция	Описание
`long atomic64_add_negative (long i, atomic64_t *v)`	Неделимое прибавление значения `i` к переменной `v`; возвращается `true`, если результат меньше нуля, иначе возвратить `false`
`long atomic64_add_return (long i, atomic64_t *v)`	Неделимое прибавление значения `i` к переменной `v` и возврат результата
`long atomic64_sub_return (long i, atomic64_t *v)`	Неделимое вычитание значения `i` из переменной `v` и возврат результата
`long atomic64_inc_return (atomic64_t *v)`	Неделимое увеличение на единицу значения переменной `v` и возврат результата
`long atomic64_dec_return (atomic64_t *v)`	Неделимое уменьшение на единицу значения переменной `v` и возврат результата
`long atomic64_dec_and_test (atomic64_t *v)`	Неделимое вычитание единицы из переменной `v`; возвращается `true`, если результат равен нулю, иначе возвращается `false`
`long atomic64_inc_and_test (atomic64_t *v)`	Неделимое прибавление единицы к переменной `v`; возвращается `true`, если результат равен нулю, иначе возвращается `false`

На всех 64-разрядных аппаратных платформах предусмотрены тип `atomic64_t` и семейство функций для работы с ним. Однако на большинстве 32-разрядных платформ, за исключением x86-32, тип `atomic64_t` не поддерживается. Для переносимости ядра Linux на все поддерживаемые аппаратные платформы разработчики должны использовать 32-разрядный тип `atomic_t`. Новый 64-разрядный тип предназначен для использования в коде, который зависит от конкретной платформы и которому нужны эти 64 разряда.

Неделимые битовые операции

Кроме неделимых целочисленных операций, в ядре также предусмотрено семейство функций, позволяющих работать на уровне отдельных битов. Нет ничего удивительного в том, что они зависят от используемой аппаратной платформы и определены в файле `<asm/bitops.h>`.

Тем не менее может вызвать удивление то, что в функциях, реализующих битовые операции, используются обычные адреса памяти. Аргументами функций являются указатель и номер бита. При этом бит 0 является самым младшим по указанному адресу. На 32-разрядных машинах бит номер 31 является самым старшим в текущем машинном слове, а бит номер 32 — самым младшим в следующем слове. На значение номера бита, который передается в функцию, не накладывается каких-либо ограничений. Однако в большинстве случаев приходится работать в пределах одного машинного слова. Поэтому на 32-разрядных машинах номера битов будут находиться в диапазоне от 0 до 31, а на 64-разрядных — от 0 до 63.

Поскольку в функциях используются обычные указатели, 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); /* Устанавливается бит 0 (неделимая операция) */
set_bit(1, &word); /* Устанавливается бит 1 (неделимая операция) */

printk("%ul\n", word); /* Выводится число "3" */

clear_bit(1, &word); /* Сбрасывается бит 1 (неделимая операция) */
change_bit(0, &word); /* Значение бита 0 изменяется на противоположное.
                         Теперь он сброшен (неделимая операция) */


/* Устанавливаем бит 0 и возвращаем его предыдущее значение (нуль).
   Неделимая операция. */
if (test_and_set_bit(0, &word)) {
   /* Условие никогда не выполнится ... */
}
/* Следующий оператор вполне допустим.
   Обычные операторы языка С можно использовать вперемешку
   с неделимыми */
word = 7;
```

Список стандартных неделимых битовых операций приведен в табл. 10.5.

Таблица 10.5. Стандартные неделимые битовые операции

Операция	Описание
`void set_bit(int nr, void *addr)`	Неделимая операция установки `nr`-го бита в области памяти, которая начинается с адреса `addr`
`void clear_bit(int nr, void *addr)`	Неделимая операция сброса `nr`-го бита в области памяти, которая начинается с адреса `addr`
`void change_bit(int nr, void *addr)`	Неделимая операция инвертирования `nr`-го бита в области памяти, которая начинается с адреса `addr`
`int test_and_set_bit (int nr, void *addr)`	Неделимая операция установки `nr` -го бита в области памяти, которая начинается с адреса `addr`; возвращается предыдущее значение этого бита
`int test_and_clear_bit(int nr, void *addr)`	Неделимая операция очистки `nr`-го бита в области памяти, которая начинается с адреса `addr`; возвращается предыдущее значение этого бита
`int test_and_change_bit(int nr, void *addr)`	Неделимая операция инвертирования `nr`-го бита в области памяти, которая начинается с адреса `addr`; возвращается предыдущее значение этого бита
`int test_bit(int nr, void *addr)`	Неделимая операция возврата значения `nr`-го бита в области памяти, которая начинается с адреса `addr`

Для удобства работы также предусмотрены обычные (а не неделимые) версии всех битовых операций. Они аналогичны описанным выше неделимым функциям, но при этом не гарантируется непрерывность выполнения операции. Имена этих функций начинаются с двух символов подчеркивания. Например, обычная форма функции `test_bit()` будет называться `__test_bit()`. Если нет необходимости в том, чтобы операции были неделимыми, например, когда данные уже защищены с помощью блокировки, обычные версии функций могут выполняться быстрее.

В ядре также предусмотрены функции, позволяющие отыскать номер первого установленного или сброшенного 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)
```

Зачем нужны обычные варианты битовых функций?

На первый взгляд может показаться, что битовые операции всегда выполняются неделимо и в обычном варианте в них нет никакого смысла. Действительно, поскольку в операции задействован всего один бит, как можно нарушить его целостность? Если одна из операций с ним завершается успешно, то что еще нужно? Бесспорно, порядок выполнения операций важен, но здесь же идет речь об их неделимости. В конце концов, если значение бита равно тому, которое устанавливается хотя бы одной из операций, то все замечательно, не так ли?

Давайте вспомним, что же такое неделимость? Неделимость означает, что операция или завершается полностью, не прерываясь, или не выполняется вообще. Следовательно, если выполняются две неделимые битовые операции, то предполагается, что они обе должны успешно выполниться. После завершения обеих операций, значение бита должно быть таким, каким бы оно было после выполнения второй операции. Более того, в некоторый момент времени перед выполнением второй операции значение бита должно быть таким, каким бы оно было после выполнения первой операции. Вообще говоря, требование неделимости означает, что все промежуточные состояния переменной должны быть корректно учтены.

Например, предположим, что нам нужно выполнить две неделимые битовые операции — сначала установить бит, а затем его сбросить. Без требования неделимости этот бит будет в конце концов сброшен, но он может быть так и не будет *никогда* установлен. Дело в том, что операция установки бита может начаться одновременно с операцией сброса и поэтому не выполниться совсем. Операция сброса бита завершится успешно, и бит будет обнулен, как и предполагалось. Однако при выполнении неделимых операций бит будет в самом деле установлен. При этом может существовать некоторый довольно небольшой промежуток времени, когда операция чтения подтвердит это, после чего выполнится операция сброса и значение бита станет равным нулю.

Иногда требуется именно такое поведение системы, особенно когда важен порядок выполнения команд или нужно работать на аппаратном уровне с регистрами какого-либо физического устройства.

Обеим функциям в качестве первого параметра передается указатель на область памяти, где нужно провести поиск, а в качестве второго параметра — общее количество битов, среди которых будет проводиться поиск. Эти функции возвращают номер первого установленного или не установленного бита соответственно. Если нужно провести поиск всего в одном машинном слове, используйте функции `__ffs()` и `ffz()`, которым в качестве единственного параметра передается адрес слова, где будет выполняться поиск.

В отличие от неделимых операций с целыми числами, при написании кода обычно нет возможности выбора, использовать или не использовать рассмотренные выше битовые операции. Часто они являются единственными переносимыми средствами, которые позволяют установить или сбросить определенный бит в памяти. Вопрос лишь в том, какие разновидности этих операций использовать — неделимые или обычные. Если при выполнении кода не могут возникнуть конфликты из-за доступа к ресурсам, то следует использовать обычные битовые операции, поскольку на некоторых аппаратных платформах они могут выполняться быстрее их неделимых аналогов.

Спин-блокировки

Было бы просто замечательно, если бы все критические участки кода сводились к таким простым действиям, как инкремент или декремент переменной, однако в жизни все куда сложнее. В реальной жизни критические участки кода могут включать в себя множество

вызовов функций. Например, очень часто данные необходимо извлечь из одной структуры, затем отформатировать, провести синтаксический анализ и добавить результат в другую структуру. Весь этот набор операций должен выполняться неделимо. Никакой другой код не должен иметь возможности читать или записывать данные ни в одну из структур до того, как данные этих структур будут полностью обновлены. Очевидно, что в таком случае простые неделимые операции не смогут обеспечить необходимую защиту данных, поэтому следует использовать более сложный метод защиты — *блокировки* (lock).

Чаще всего в ядре Linux используются так называемые *спин-блокировки* (*spin lock*). Это такой тип блокировки, которая может быть захвачена не более чем в одном потоке команд. Если в каком-то из потоков команд будет предпринята попытка захвата спин-блокировки, которая удерживается другим потоком, возникает состояние *конфликта*. При этом первый поток входит в цикл и начинает в нем постоянно проверять, не освободилась ли уже требуемая блокировка. Поскольку код при этом постоянно находится в цикле и как бы вращается (от англ. *spin* — вращаться), отсюда и возникло название этого типа блокировки. Если блокировка свободна, поток может сразу же ее захватить и продолжить свое выполнение. Циклическая проверка предотвращает ситуацию, когда в критическом участке кода одновременно может находиться более одного потока команд. Одна и та же спин-блокировка может захватываться в разных местах кода ядра. В результате всегда будет гарантирована защита и синхронизация при доступе, например, к какой-нибудь структуре данных.

А сейчас вернемся к аналогии с дверью и ключом, описанной в предыдущей главе. Так вот, спин-блокировка напоминает человека, стоящего перед дверью и ожидающего, пока из комнаты выйдет другой человек и передаст ему ключ. Если вы подошли к двери и в комнате никого нет, то вы можете свободно взять ключ, открыть дверь и войти в комнату. Если же в комнате кто-то есть, вы должны будете обождать за дверью, периодически проверяя, не освободилась ли комната. Как только комната освободится, вы сможете взять ключ, открыть дверь и войти внутрь. Благодаря тому что ключ (читай спин-блокировка) может взять только один человек (читай поток команд), только один человек может находиться в один и тот же момент в комнате (читай критического участка кода).

Тот факт, что при возникновении конфликта из-за спин-блокировки потоки переходят в замкнутый цикл, ожидая ее освобождения, и, соответственно, понапрасну расходуют процессорное время, является важным, поскольку такое поведение характерно для спин-блокировок. Очевидно, что неразумно удерживать спин-блокировку в течение длительного промежутка времени. По своей сути спин-блокировка — это простая и быстрая блокировка, которая должна захватываться на короткое время одним потоком. Существует и альтернативный вариант реакции системы на возникновение конфликта при блокировке. Если один из потоков попытается захватить блокировку, которая удерживается другим потоком, то первый поток переводится в состояние ожидания и возвращается к выполнению, когда блокировка освобождается. В этом случае процессор может начать выполнение другого кода. Такая реакция системы влечет за собой дополнительные издержки, основная из которых — два переключения контекста, требуемые при переходе на другую задачу, и возвращения к выполнению текущей задачи при освобождении блокировки. Очевидно, что для реализации всего этого требуется довольно большой объем кода, который никак нельзя сравнить с теми строчками кода, с помощью которых реализуются спин-блокировки. Поэтому разумно использовать спин-блокировку, если время ее удержания меньше длительности двух переключений контекста. Так как у большинства из нас есть более интересные занятия, чем измерение времени переключения контекста,

необходимо стараться удерживать блокировки по возможности в течение максимально короткого периода времени[1]. Ниже в этой главе мы рассмотрим *семафоры* (*semaphore*) — механизм блокировок, который позволяет переводить потоки, ожидающие освобождения блокировки, в состояние ожидания. В отличие от спин-блокировок они не расходуют понапрасну время центрального процессора при выполнении проверок.

Функции для спин-блокировки

Спин-блокировки являются зависимыми от аппаратной платформы и реализованы на языке ассемблера. Зависимый от аппаратной платформы код определен в заголовочном файле `<asm/spinlock.h>`. Сами интерфейсы использования спин-блокировок определены в файле `<linux/spinlock.h>`. Ниже приведен простой пример использования спин-блокировки.

```
DEFINE_SPINLOCK(mr_lock);
spin_lock(&mr_lock);

/* Критический участок кода ... */

spin_unlock(&mr_lock);
```

В любой момент времени блокировка может удерживаться не более чем одним потоком команд. Следовательно, только один поток может зайти в критический участок кода в данный момент времени. При этом на многопроцессорных машинах обеспечивается нужная защита ресурсов от конкурентного использования. На однопроцессорной машине блокировки не компилируются в исполняемый код, и, соответственно, их просто не существует. При этом блокировки просто играют роль маркеров, запрещающих или разрешающих вытеснение кода ядра (мультипрограммирование в режиме ядра). Если мультипрограммный режим ядра отключен, то блокировки совсем не компилируются.

> **Внимание: спин-блокировки не рекурсивны!**
>
> В отличие от реализаций в других операционных системах и библиотеках поддержки потоковых вычислений, спин-блокировки в операционной системе Linux не рекурсивны. Это означает, что если поток попытается захватить блокировку, которую он уже удерживает, то этот поток войдет в бесконечный цикл, 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 ситуация самоблокировки. Поэтому будьте внимательны!

Спин-блокировки могут использоваться в обработчиках прерываний, а семафоры — нет, поскольку они переводят процесс в состояние ожидания. Если блокировка используется в обработчике прерывания, то перед тем, как захватить ее, необходимо запретить все локальные прерывания (т.е. запросы на прерывания на текущем процессоре). Иначе может возникнуть такая ситуация, когда обработчик прерывания прервет выполнение кода ядра, удерживающего данную блокировку, и снова попытается захватить эту же блокировку. При этом обработчик прерывания начнет проверять в цикле, не освободилась ли нужная ему блокировка. С другой стороны, код ядра, который удерживает блокировку, не будет выполняться до тех пор, пока обработчик прерывания не закончит выполнение. Этот пример взаимоблокировки (двойного захвата блокировки) уже обсуждал-

[1] Сейчас это требование становится особенно важным, поскольку в ядре поддерживается режим мультипрограммирования. Время, в течение которого удерживаются блокировки, эквивалентно времени задержки системного планировщика.

ся в предыдущей главе. Следует заметить, что прерывания необходимо запрещать только на *текущем* процессоре. Если прерывание возникнет на другом процессоре (по отношению к коду ядра, захватившего блокировку) и его обработчик будет ожидать освобождения блокировки, то это не приведет к взаимоблокировке, так как первый процессор в конечном итоге освободит захваченную им блокировку.

Для удобства в ядре предусмотрен специальный интерфейс, который позволяет сразу запретить прерывания и захватить блокировку. Пример его использования приведен ниже.

```
DEFINE_SPINLOCK(mr_lock);
unsigned long flags;

spin_lock_irqsave(&mr_lock, flags);
/* Критический участок кода... */

spin_unlock_irqrestore(&mr_lock, flags);
```

Процедура `spin_lock_irqsave()` сохраняет текущее состояние системы прерываний, запрещает локальные прерывания и захватывает указанную блокировку. Соответственно функция `spin_unlock_irqrestore()` освобождает указанную блокировку и восстанавливает предыдущее состояние системы прерываний. Таким образом, если прерывания были уже запрещены, после выполнения приведенного выше кода они по ошибке не будут разрешены. Обратите внимание на то, что переменная `flags` передается по значению. Дело в том, что указанные процедуры частично реализованы в виде макросов.

Поскольку на однопроцессорной машине средства блокировок не компилируются в ядро, описанная выше функция `spin_lock_irqsave()` только лишь запрещает прерывания, чтобы предотвратить доступ обработчика прерывания к совместно используемым данным. Функции захвата и освобождения блокировки также запрещают и разрешают мультипрограммный режим ядра соответственно.

Что нужно блокировать?

Важно, чтобы каждая блокировка была четко связана с блокируемым объектом. Еще важнее защищать *данные*, а не *код*. Несмотря на то что во всех примерах этой главы речь идет о важности защиты критических участков кода, на самом деле на этих участках выполняется обращение к важным данным, которые и защищаются, а не к коду.

Существует важное правило: блокировки, которые попросту охватывают участки кода, затрудняют его чтение и понимание и являются возможным источником конфликтов. Поэтому защищайте данные, а не код!

Всегда связывайте общие данные с соответствующей блокировкой. Например, структура данных `foo` может быть защищена с помощью блокировки `foo_lock`. Перед доступом к общим данным сначала убедитесь, что это можно сделать безопасно. Чаще всего это означает, что перед манипуляцией с данными нужно захватить соответствующую блокировку и освободить ее после завершения работы с данными.

Если точно известно, что перед захватом блокировки 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()`, как показано ниже.

```
DEFINE_SPINLOCK(mr_lock);

spin_lock_irq(&mr_lock);
/* Критический участок кода... */
```

```
spin_unlock_irq(&mr_lock);
```

По мере увеличения размеров и сложности ядра Linux не легко гарантировать, что в нужном нам участке кода прерывания будут всегда разрешены. Поэтому не рекомендуется использовать функцию `spin_lock_irq()`. Если стоит вопрос об использовании этой функции, лучше сразу убедиться в том, что прерывания разрешены. Иначе можно столкнуться с непредсказуемым поведением системы, особенно в тех местах, где прерывания должны быть запрещены, а они почему-то оказались разрешены.

> **Отладка спин-блокировок**
>
> Параметр конфигурации ядра `CONFIG_DEBUG_SPINLOCK` подключает несколько отладочных проверок в коде спин-блокировок. Например, если указать этот параметр, то в коде будут выполняться проверки на использование неинициализированных спин-блокировок, а при освобождении блокировки будет проверяться, была ли захвачена блокировка. При тестировании кода всегда следует включать отладку спин-блокировок. Чтобы включить дополнительные отладочные средства для блокировок, задайте параметр конфигурации ядра `CONFIG_DEBUG_LOCK_ALLOC`.

Другие средства работы со спин-блокировками

Функция `spin_lock_init()` используется для инициализации динамически созданных спин-блокировок в виде переменной типа `spinlock_t`, к которой нет прямого доступа, а есть только указатель на нее.

Функция `spin_trylock()` пытается захватить указанную спин-блокировку. Если блокировка уже кем-то захвачена, то вместо циклической проверки и ожидания на ее освобождение эта функция немедленно возвращает нулевое значение. Если блокировка была захвачена успешно, то функция возвращает ненулевое значение. Аналогично функция `spin_is_locked()` возвращает ненулевое значение, если блокировка в данный момент захвачена, в противном случае возвращается нуль. Эта функция ни при каких обстоятельствах не захватывает блокировку[2].

Полный список функций для работы со спин-блокировками приведен в табл. 10.6.

Таблица 10.6. Функции для работы со спин-блокировками

Функция	Описание
`spin_lock()`	Захватывает указанную блокировку
`spin_lock_irq()`	Запрещает прерывания на локальном процессоре и захватывает указанную блокировку
`spin_lock_irqsave()`	Сохраняет текущее состояние системы прерываний, запрещает прерывания на локальном процессоре и захватывает указанную блокировку
`spin_unlock()`	Освобождает указанную блокировку

[2] При использовании описываемых функций код часто становится весьма запутанным. В коде нет никакой необходимости часто проверять состояние спин-блокировки. Дело в том, что перед вызовом кода нужно сначала захватить блокировку либо вызывать код, только если блокировка уже захвачена. Однако в некоторых ситуациях описываемые функции логично использовать, поэтому и были предусмотрены эти интерфейсы.

Окончание табл. 10.6

Функция	Описание
`spin_unlock_irq()`	Освобождает указанную блокировку и разрешает прерывания на локальном процессоре
`spin_unlock_irqrestore()`	Освобождает указанную блокировку и восстанавливает состояние системы прерываний на локальном процессоре в первоначальное значение
`spin_lock_init()`	Инициализирует динамический объект типа `spinlock_t` в заданной области памяти
`spin_trylock()`	Выполняется попытка захвата указанной блокировки; если блокировка успешно захвачена, возвращается ненулевое значение
`spin_is_locked()`	Возвращается ненулевое значение, если указанная блокировка в данный момент захвачена, и нулевое значение — в противном случае

Спин-блокировки и нижние половины обработчиков прерываний

Как уже отмечалось в главе 8, "Нижняя половина обработчика и отложенные действия", при использовании блокировок в нижних половинах обработчиков прерываний необходимо принимать некоторые меры предосторожности. Функция `spin_lock_bh()` позволяет захватить указанную блокировку и запретить обработку нижних половин. Функция `spin_unlock_bh()` выполняет обратные действия.

Нижняя половина обработчика прерываний может вытеснять код, выполняемый в контексте процесса. Поэтому, если некоторые данные используются и в нижней половине обработчика, и в контексте процесса, их нужно защитить в контексте процесса, запретив обработку нижних половин и захватив блокировку. Точно так же, поскольку обработчик прерывания может вытеснить свою нижнюю половину, пред обращением к общим данным необходимо захватить блокировку и запретить прерывания.

Вспомним, что два тасклета одного типа не могут выполняться параллельно. Поэтому нет необходимости защищать данные, которые используются только в тасклетах одного типа. Если же данные используются в тасклетах разных типов, то необходимо использовать обычную спин-блокировку перед тем, как обращаться к таким данным в нижней половине обработчика. В этом случае нет необходимости запрещать обработку нижних половин, так как один тасклет никогда не вытесняет другой тасклет, выполняющийся на том же процессоре.

В случае отложенных прерываний, независимо от того, отложенные ли это прерывания одного типа или разных, данные, совместно используемые в обработчиках 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 по чтению-записи

Иногда в соответствии с целью использования блокировок их можно разделить на два типа — *блокировки по чтению* (reader lock) и *блокировки по записи* (writer lock). В качестве

примера рассмотрим произвольный список, который может обновляться и в котором одновременно может выполняться поиск. При обновлении списка (т.е. когда в него осуществляется запись) важно, чтобы никакой другой поток кода не мог параллельно осуществлять запись *или* чтение данных из этого списка. При записи данных требуется монопольный доступ к списку. С другой стороны, если в списке выполняется поиск (чтение информации), важно только, чтобы никто другой не выполнял запись в этот список. Параллельный доступ к списку со стороны нескольких потоков возможен при условии, что ни один из потоков не будет записывать данные в список. Алгоритм работы со списком задач в системе, описанный в главе 3, "Управление процессами", как раз соответствует приведенному только что описанию. Поэтому нет ничего удивительного в том, что список задач в системе защищен с помощью спин-блокировки по чтению-записи.

Если работа со структурой данных может быть четко разделена на этапы чтения/записи или соответствует шаблону "производитель и потребитель", то имеет смысл использовать механизмы блокировок с аналогичной семантикой. Для таких ситуаций в операционной системе Linux предусмотрены спин-блокировки по чтению-записи. Они обеспечивают раздельные блокировки для операций чтения и записи. Блокировку по чтению могут одновременно захватывать несколько потоков, в которых выполняется считывание данных. В отличие от этого, блокировку по записи может захватить только один поток. При этом не допускаются параллельные операции по считыванию данных. Блокировку по чтению-записи иногда называют *разделяемой/монопольной* или *совместной/монопольной*, поскольку она доступна в разделяемом режиме для считывания данных и в монопольном режиме для записи данных.

Описываемый тип блокировки используется точно так же, как и спин-блокировка. Блокировка по чтению-записи инициализируется с помощью приведенного ниже фрагмента кода.

```
DEFINE_RWLOCK(mr_rwlock);
```

После этого для чтения данных используется следующий фрагмент кода:

```
read_lock(&mr_rwlock);

/* Критический участок кода для чтения данных... */

read_unlock(&mr_rwlock);
```

При записи блокировка выполняется так, как показано ниже.

```
write_lock(&mr_rwlock);

/* Критический участок кода для чтения/записи данных... */

write_unlock(&mr_lock);
```

Обычно считывание и запись информации осуществляются в разных ветках кода, как это показано в данном примере.

Обратите внимание на то, что блокировку, захваченную для чтения, нельзя "повышать" до блокировки, захваченной для записи. В качестве примера рассмотрим приведенный ниже фрагмент кода.

```
read_lock (&mr_rwlock);
write_lock(&mr_rwlock);
```

Выполнение приведенных выше двух функций вызовет взаимоблокировку. Дело в том, что для блокировки по записи требуется, чтобы все потоки, включая текущий, освободили блокировку по чтению. Поэтому, если в каком-либо месте кода вам нужно будет выпол-

нить запись, сразу же захватывайте блокировку по записи. Если же вы не можете четко разделить ветки кода, выполняющие чтение и запись, то это значит, что блокировку по чтению-записи использовать нельзя. В таком случае оптимальным будет использование обычных спин-блокировок.

Одну и ту же блокировку по чтению можно безопасно захватывать в нескольких потоках. Более того, один поток также может безопасно захватывать повторно одну и ту же блокировку по чтению. Это позволяет выполнить полезную и всеобщую оптимизацию. Если в обработчиках прерываний осуществляется только чтение данных и не выполняется их 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 захвате блокировки для записи данных нужно все равно запрещать прерывания, например использовать функцию `write_lock_irqsave()`. Дело в том, что в обработчике прерывания может возникнуть взаимоблокировка в связи с ожиданием захвата блокировки по чтению при захваченной блокировке по записи. Полный список функций для работы с блокировками по чтению-записи приведен в табл. 10.7.

Таблица 10.7. Список функций для работы со спин-блокировками по чтению-записи

Функция	Описание
`read_lock()`	Захватывает указанную блокировку по чтению
`read_lock_irq()`	Запрещает прерывания на локальном процессоре и захватывает указанную блокировку по чтению
`read_lock_irqsave()`	Сохраняет состояние системы прерываний на текущем процессоре, запрещает прерывания на локальном процессоре и захватывает указанную блокировку по чтению
`read_unlock()`	Освобождает указанную блокировку, захваченную по чтению
`read_unlock_irq()`	Освобождает указанную блокировку, захваченную по чтению, и разрешает прерывания на локальном процессоре
`read_unlock_irqrestore()`	Освобождает указанную блокировку, захваченную по чтению, и восстанавливает предыдущее состояние системы прерываний
`write_lock()`	Захватывает указанную блокировку по записи
`write_lock_irq()`	Запрещает прерывания на локальном процессоре и захватывает указанную блокировку по записи
`write_lock_irqsave()`	Сохраняет состояние системы прерываний на текущем процессоре, запрещает прерывания на локальном процессоре и захватывает указанную блокировку по записи
`write_unlock()`	Освобождает указанную блокировку, захваченную по записи
`write_unlock_irq()`	Освобождает указанную блокировку, захваченную по записи, и разрешает прерывания на локальном процессоре
`write_unlock_irqrestore()`	Освобождает указанную блокировку, захваченную по записи, и восстанавливает предыдущее состояние системы прерываний
`write_trylock()`	Пытается захватить заданную блокировку по записи и в случае неудачи возвращает нулевое значение
`rw_lock_init()`	Инициализирует объект типа `rwlock_t` в заданной области памяти

Еще один факт, который необходимо принимать во внимание при работе с блокировками по чтению-записи в операционной системе Linux, — это то, что блокировка по чтению всегда имеет большее преимущество по сравнению с блокировкой по записи. Так, если некоторый поток удерживает блокировку по чтению, а другой поток ожидает ее освобождения, чтобы получить монопольный доступ к ресурсу по записи, то все другие потоки, пытающиеся захватить блокировку по чтению, будут без проблем добиваться своей цели. Поток записи, который периодически проверяет на освобождение блокировки, не сможет ее захватить, пока все потоки чтения не освободят эту блокировку. Поэтому большое количество потоков чтения будет приводить к "подвисанию" ожидающих потоков записи. Это важное обстоятельство всегда нужно помнить при разработке схемы блокировок, поскольку иногда такое поведение системы желательно, а иногда нет.

Спин-блокировки являются очень простым и быстрым средством 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 в подобных случаях следует использовать семафоры.

Семафоры

В операционной системе Linux семафоры (semaphore) — это блокировки, которые переводят процессы в состояние ожидания. При попытке захвата семафора, который уже захвачен другой задачей, текущая задача помещается в очередь ожидания (wait queue) и замораживается. После этого процессор переходит к выполнению другой задачи. Как только требуемый семафор освобождается, одна из задач[3], находящаяся в очереди ожидания, активизируется, после чего она может захватить требуемый семафор.

Давайте снова вернемся к аналогии с дверью и ключом. Подойдя к двери, человек может взять ключ, открыть дверь и войти в комнату. Однако если в этот момент кто-либо другой также подойдет к двери, то ключ уже не будет доступен. Вот здесь и начинаются основные отличия в поведении системы по сравнению со спин-блокировками. Вместо того чтобы ожидать под дверью, периодически проверяя, не доступен ли ключ, человек просто записывает на клочке бумаги, висящей на двери, свое имя и номер и ложится спать. Тот, кто находится внутри комнаты, должен при выходе взглянуть на этот список. Если он не пустой, выходящий человек должен выбрать первое имя из списка, разбудить того, чье имя там указано, и позволить ему войти в комнату. Таким образом, ключ (читай семафор) позволяет только одному человеку (читай потоку) находиться в комнате (читай в критическом участке) в один момент времени. 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 бо́льшими накладными расходами. Жизнь — это всегда компромисс!

Из-за того что при захвате семафора задачи переводятся в состояние ожидания, следует несколько интересных выводов, приведенных ниже.

[3] Как будет показано ниже, несколько процессов могут по мере необходимости одновременно удерживать один семафор.

- Поскольку задача, которая хочет захватить блокировку, переводится в состояние ожидания до момента освобождения блокировки, семафоры хорошо подходят для реализации блокировок, которые могут удерживаться в течение длительного периода времени.
- С другой стороны, не рекомендуется использовать семафоры для реализации блокировок, которые удерживаются в течение короткого периода времени. Дело в том, что задержки, связанные с переводом процессов в состояние ожидания, обслуживания очереди ожидания и последующей активизации процесса, могут легко превысить время, в течение которого удерживается сама блокировка.
- Так как в случае возникновения конфликта при захвате блокировки поток будет переведен в состояние ожидания, семафоры можно захватывать только в контексте процесса. Дело в том, что в контексте прерывания потоки не обрабатываются планировщиком.
- При удержании семафора процесс может переходить в состояние ожидания, хотя обычно так не делают. Это не приведет к тупиковой ситуации, когда другой процесс попытается захватить ту же блокировку (просто он переводится в состояние ожидания и не мешает выполняться первому процессу).
- При захвате семафора нельзя удерживать спин-блокировку, поскольку процесс может переходить в состояние ожидания до освобождения семафора, а при удержании спин-блокировки в состояние ожидания переходить нельзя.

В перечисленных выше пунктах отражены особенности использования семафоров по сравнению со спин-блокировками. В большинстве случаев выбор решения, какой тип блокировки следует использовать, очень прост. Если код должен переходить в состояние ожидания, что очень часто возникает при выполнении синхронизации с пространством пользователя, то семафоры — единственное решение. Использовать семафоры всегда проще, даже если в них нет особой необходимости, поскольку они предоставляют гибкость, связанную с переходом процессов в состояние ожидания. Если же возникает необходимость выбора между семафорами и спин-блокировками, то решение должно основываться на времени удержания блокировки. В идеале все блокировки нужно удерживать в течение минимально возможного промежутка времени. Однако при использовании семафоров вполне допустимы более длительные периоды удержания блокировок. Кроме того, применение семафоров не отменяет вытеснение процессов ядра, следовательно, код, который удерживает семафор, может быть вытеснен. Это означает, что семафоры не оказывают вредного влияния на планировщик и не увеличивают задержки при планировании.

Счетные и бинарные семафоры

Наконец, существует еще одно очень полезное свойство семафоров: один и тот же семафор может одновременно удерживаться в нескольких потоках. В отличие от этого спин-блокировка в любой момент времени может удерживаться только в одном потоке. При объявлении семафора можно указать максимальное количество потоков, которым разрешено одновременно удерживать семафор. Это значение называется *счетчиком использования* (usage count), или просто *счетчиком* (count).

Чаще всего, как и для спин-блокировок, один и тот же семафор может одновременно удерживаться только в одном потоке. При этом счетчик его использования равен единице,

и такой семафор называют *бинарным семафором* (binary semaphore), потому что он может либо удерживаться только в одном потоке, либо не удерживаться вовсе нигде. Также такой тип семафора называют *мьютексом* (*mutex*, или mutual exclusion), потому что он гарантирует взаимоисключающий доступ к ресурсу.

Кроме того, счетчику при инициализации может быть присвоено значение, большее единицы. В этом случае семафор называется *счетным семафором* (counting semaphore), или *семафором со счетчиком*. Такой семафор одновременно могут удерживать столько потоков, сколько указано в счетчике, и не более того. Семафоры со счетчиком не используются для предоставления монопольного (взаимоисключающего) доступа к ресурсу, так как они позволяют нескольким потокам команд одновременно находиться в критическом участке кода. Поэтому они используются для установки ограничений на доступ к ресурсу в определенном коде. В ядре они практически не используются, так как в большинстве случаев требуется предоставить монопольный доступ к ресурсу, т.е. применить семафор со счетчиком, равным единице.

В качестве обобщенного механизма блокировок семафоры были формализованы Эдсгером Вибе Дейкстрой[4] (Edsger Wybe Dijkstra) в 1968 году. Семафор поддерживает две неделимые операции, `P()` и `V()`, название которых происходит от голландских слов *Proberen* (дословно — 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 получаемое в результате значение счетчика больше или равно нулю, то блокировка захватывается успешно и задача может войти в критический участок. Если в результате получается отрицательное значение счетчика, то задача помещается в очередь ожидания, и процессор переходит к выполнению каких-либо других действий. Об использовании этой функции говорят в форме глагола — для захвата семафора его нужно *опустить* (*down*).

Операция `up()` используется для того, чтобы освободить семафор после завершения выполнения критического участка кода. Ее часто называют *поднятием* (*upping*) семафора. Операция `up()` используется для инкремента значения счетчика. Если очередь ожидания семафора не пуста, то одна из задач 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>`. Объекты-семафоры представляются в виде структуры `semaphore`. Статическое определение семафоров выполняется так, как показано ниже. В приведенном фрагменте кода `name` — это имя переменной типа семафор, а `count` — счетчик семафора.

```
struct semaphore name;
sema_init(&name, count);
```

[4] Доктор Дейкстра (1930–2002) — один из самых талантливых ученых за всю, конечно, не очень долгую историю существования информатики как науки. Его многочисленные труды включают работы по 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 операторов `GOTO` в ядре Linux.

Для создания мьютексов, которые используются чаще всего, предусмотрена сокращенная форма записи, как показано ниже. В приведенном фрагменте кода `name` — это имя переменной бинарного семафора.

```
static DECLARE_MUTEX(name);
```

Однако обычно семафоры создаются динамически, чаще всего как часть большой структуры данных. В таком случае для инициализации семафора, создаваемого динамически и на который есть только косвенная ссылка в виде указателя, необходимо использовать функцию `sema_init()`, как показано ниже. В приведенном фрагменте кода `sem` — это указатель на структуру типа "семафор", а `count` — счетчик его использования.

```
sema_init(sem, count);
```

По аналогии для инициализации динамически создаваемого мьютекса можно использовать приведенный ниже код.

```
init_MUTEX(sem);
```

Мне так и не удалось узнать, почему слово "mutex" в названии функции `init_MUTEX()` набрано прописными буквами и почему слово "init" стоит на первом месте, тогда как в названии функции `sema_init()` оно идет вторым. Полагаю, что после прочтения главы 8, "Нижняя половина обработчика и отложенные действия", вы перестанете удивляться подобным нестыковкам.

Использование семафоров

Функция `down_interruptible()` выполняет попытку захватить указанный семафор. Если эта попытка не удалась, то вызывающий ее процесс переводится в состояние ожидания с флагом `TASK_INTERRUPTIBLE`. Вспомните материал главы 3. Там мы говорили, что процесс, находящийся в таком состоянии, может быть возвращен к выполнению при поступлении сигнала и что в этом нет ничего плохого. Если сигнал приходит в тот момент, когда процесс ожидает освобождения семафора, то он возвращается к выполнению, а функция `down_interruptible()` возвращает значение `-EINTR`. Альтернативой рассмотренной функции выступает функция `down()`, которая переводит процесс в состояние ожидания с флагом `TASK_UNINTERRUPTIBLE`. В большинстве случаев это нежелательно, так как процесс, который ожидает освобождения семафора, не будет отвечать на сигналы. Поэтому функция `down_interruptible()` используется значительно более широко (и это правильно!), чем функция `down()`. Конечно, имена этих функций далеки от идеала.

Функция `down_trylock()` используется для захвата указанного семафора без блокировки. Если семафор уже захвачен, то функция немедленно возвращает ненулевое значение. В противном случае выполняется захват семафора и возвращается нулевое значение.

Для освобождения захваченного семафора вызовите функцию `up()`. Рассмотрим приведенный ниже пример.

```
/* Определим семафор с именем mr_sem и первоначальным значением
   счетчика, равным 1 */
static DECLARE_MUTEX(mr_sem);

/* Попытаемся захватить семафор... */
if (down_interruptible(&mr_sem)) {
   /* Если получен сигнал, семафор захвачен не был... */
}
```

```
/* Критический участок ... */

/* Освободим захваченный семафор*/
up(&mr_sem);
```

Полный список функций для работы с семафорами приведен в табл. 10.8.

Таблица 10.8. Список функций для 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 *)`	Пытается захватить семафор. Если данный семафор уже кем-то захвачен, то процесс переводится в прерываемое состояние ожидания
`down(struct semaphore *)`	Пытается захватить семафор. Если данный семафор уже кем-то захвачен, то процесс переводится в непрерываемое состояние ожидания
`down_trylock(struct semaphore *)`	Выполняется попытка захвата семафора. Если семафор уже захвачен, то функция немедленно возвращает ненулевое значение. В противном случае выполняется захват семафора и возвращается нулевое значение
`up(struct semaphore *)`	Освобождение указанного семафора и возврат к выполнению ожидающего его задания, если такое есть

Семафоры для чтения–записи

Семафоры, так же как и спин-блокировки, могут быть предназначены для защиты операций чтения-записи. Ситуации, в которых предпочтительнее использовать семафоры для чтения-записи, такие же, как и в случае использования спин-блокировок по чтению-записи.

Семафоры для чтения-записи представляются в виде структуры `rw_semaphore`, которая определена в файле `<linux/rwsem.h>`. Статически определенный семафор для чтения-записи создается с помощью приведенного ниже кода, в котором `name` — это определяемое имя нового семафора.

```
static DECLARE_RWSEM(name);
```

Динамически создаваемые семафоры для чтения-записи инициализируются с помощью приведенной ниже функции.

```
init_rwsem(struct rw_semaphore *sem)
```

Все семафоры для чтения-записи являются мьютексами, т.е. их счетчик использования равен единице. Несмотря на это, они предоставляют монопольный доступ только для записывающих, а не считывающих данные потоков. Блокировку по чтению может одновременно удерживать любое количество потоков, при условии, что при этом нет ни одного потока записи. И наоборот, только один поток может удерживать блокировку по записи, если при этом нет ни одного потока, захватившего эту же блокировку по чтению.

Для всех семафоров чтения-записи используется непрерываемое состояние ожидания, поэтому существует только одна версия функции down(). Пример приведен ниже.

```
static DECLARE_RWSEM(mr_rwsem);

/* Попытаемся захватить семафор для чтения... */
down_read(&mr_rwsem);

/* Критический участок кода (только для чтения) ... */

/* Освободим семафор */
up_read(&mr_rwsem);

/* ... */
/* Попытаемся захватить семафор для записи... */
down_write(&mr_rwsem);

/* Критический участок кода (для чтения и записи)... */

/* Освободим семафор */
up_write(&mr_sem);
```

По аналогии с обычными семафорами для семафоров чтения-записи предусмотрены реализации функций down_read_trylock() и down_write_trylock(). Каждой из них передается один параметр — указатель на семафор чтения-записи. Обе функции возвращают ненулевое значение, если семафор был успешно ими захвачен, и нуль, если семафор уже кем-то захвачен. Будьте внимательны! По непонятным причинам поведение этих функций противоположно поведению аналогичных функций для обычных семафоров.

Семафоры для чтения-записи имеют уникальную функцию, аналога которой нет для спин-блокировок по чтению-записи. Это функция downgrade_writer(), которая автоматически превращает блокировку, захваченную для записи, в блокировку, захваченную для чтения.

Семафоры для чтения-записи, так же как и спин-блокировки аналогичного типа, должны использоваться, только если есть четкое разделение участков кода, которые выполняют чтение и которые осуществляют запись. Использование механизмов блокировок по чтению-записи приводит к дополнительным накладным расходам. Поэтому их стоит использовать, только если код можно четко разделить на участки чтения и записи.

Мьютексы

До недавнего времени семафоры были единственной блокировкой в ядре, которая переводила задачу в состояние ожидания. В большинстве случаев при создании семафора программисты присваивали единичное значение его счетчику и использовали его для предоставления монопольного доступа к некоторому ресурсу. По сути, семафор был аналогичен спин-блокировке с возможностью перевода процесса в состояние ожидания. К сожалению, семафоры являются слишком общим понятием и не могут налагать много полезных ограничений. В результате их можно с успехом использовать для предоставления монопольного доступа к ресурсу в совершенно туманных ситуациях наподобие сложных взаимодействий ядра и пространства пользователя. Однако реализовать с их помощью простую блокировку не так-то просто, а отсутствие жестких правил делает невозможным любой вид автоматической отладки или наложение ограничений.

В поисках простого средства блокировки, которое могло бы переводить задачу в состояние ожидания, разработчики ядра решили использовать *мьютексы* (mutex). Поскольку вы уже, должно быть, привыкли, что названия в Linux частенько сбивают с толку, давайте

немного проясним ситуацию. Под термином "мьютекс" подразумевается обобщенное название любых видов блокировок, которые могут переводить задачу в состояние ожидания, предназначенных для предоставления монопольного (взаимоисключающего) доступа к ресурсу. Типичный пример — семафор, счетчик которого равен единице. В современных версиях ядра Linux понятие "мьютекс" теперь также означает особый тип блокировки с возможностью перевода процесса в состояние ожидание, с помощью которого реализуется монопольный доступ к некоторому ресурсу. Так что в Linux мьютекс — это просто мьютекс, и ничего больше!

Мьютекс представляется в виде структуры `mutex`. Он ведет себя так же, как и семафор, счетчик использования которого равен единице, но имеет более простой интерфейс, более высокую эффективность и налагает дополнительные ограничения при использовании. Для статического определения мьютекса воспользуйтесь приведенным ниже кодом.

```
DEFINE_MUTEX(name);
```

Динамическая инициализация мьютекса выполняется так, как показано ниже.

```
mutex_init(&mutex);
```

Захват и освобождение мьютекса выполняется очень просто:

```
mutex_lock(&mutex);
/* Критический участок... */
mutex_unlock(&mutex);
```

Да, все на самом деле так и есть! Мьютексы проще, чем семафоры, и в них не нужно возиться со счетчиком использования. Основные функции для работы с мьютексами приведены в табл. 10.9.

Таблица 10.9. Основные функции для работы с мьютексами

Функция	Описание
`mutex_lock(struct mutex *)`	Захватывает указанный мьютекс. Если блокировка недоступна, процесс переводится в состояние ожидания
`mutex_unlock(struct mutex *)`	Освобождает указанный мьютекс
`mutex_trylock(struct mutex *)`	Пытается захватить указанный мьютекс. Если мьютекс успешно захвачен, возвращается значение 1, в противном случае возвращается нуль
`mutex_is_locked(struct mutex *)`	Возвращает единицу, если указанный мьютекс уже захвачен, и нуль — в противном случае

Простота и эффективность мьютексов вытекает из тех дополнительных ограничений, которые были наложены на их использование по сравнению с тем, что требовалось для семафоров. В отличие от семафоров, которые реализуют самую общую модель поведения согласно первоначальному замыслу Дейкстры, сфера использования мьютексов гораздо *уже* и более строго определена, как показано ниже.

- В один и тот же момент времени мьютекс может удерживать только одна задача. Другими словами, счетчик использования мьютекса всегда равен единице.
- Тот процесс, который захватил мьютекс, должен обязательно его освободить. Другими словами, нельзя захватить мьютекс в одном контексте, а затем освободить его в другом. Это означает, что мьютекс не пригоден для решения

сложных задач синхронизации между ядром и пространством пользователя. Тем не менее в большинстве случаев они аккуратно захватываются и освобождаются в одном и том же контексте.

- Повторные захваты и освобождения мьютексов не разрешаются. Это означает, что нельзя повторно захватить тот же самый мьютекс и нельзя освободить уже освобожденный мьютекс.
- Процесс не может завершить свою работу до тех пор, пока он не освободит мьютекс.
- Мьютекс нельзя захватить в обработчике прерываний или в его нижней половине, даже с помощью функции `mutex_trylock()`.
- Для работы с мьютексами может использоваться только официальное API. Они должны инициализироваться только с помощью описанных в данном разделе функций. Не допускается их копирование, ручная инициализация или повторная инициализация.

Вероятно, одной из самых полезных особенностей новой структуры `mutex` является возможность для ядра программно контролировать нарушение описанных выше ограничений и выводить соответствующие предупреждения при работе в специальном режиме отладки. Если при конфигурации ядра задан параметр `CONFIG_DEBUG_MUTEXES`, то в ядро включается специальный проверочный код, гарантирующий, что приведенные выше (а также другие) ограничения не будут никогда нарушаться. Это позволяет программистам просто и единообразно пользоваться мьютексами.

Сравнение семафоров и мьютексов

Мьютексы и семафоры очень похожи друг на друга. Поэтому сам факт, что они оба реализованы в ядре, часто сбивает с толку. К счастью, существует простое правило, когда и в каком случае их следует использовать. Всегда используйте мьютексы, а не семафоры, если перечисленные выше ограничения для мьютексов вам подходят. При написании нового кода семафоры используются только в очень специфических (часто низкоуровневых) модулях. Поэтому всегда начинайте с использования мьютексов и переходите к семафорам только в том случае, если столкнулись с ограничениями, которые никак нельзя обойти.

Сравнение спин-блокировок и мьютексов

Понимание того, когда следует использовать спин-блокировки, а когда мьютексы (или семафоры), очень важно для написания оптимального кода. Однако во многих случаях сделать выбор очень просто. В контексте прерывания могут использоваться только спин-блокировки, и только мьютекс может удерживаться процессом, который находится в состоянии ожидания. В табл. 10.10 приведен набор требований, облегчающих процесс выбора типа блокировки.

Таблица 10.10. Выбор между спин-блокировкой и мьютексом

Требование	Рекомендуемый тип блокировки
Блокировка с малыми дополнительными издержками	Спин-блокировки более предпочтительны
Малое время удержания блокировки	Спин-блокировки более предпочтительны

Окончание табл. 10.10

Требование	Рекомендуемый тип блокировки
Длительное время удержания блокировки	Мьютексы более предпочтительны
Использование блокировки в контексте прерывания	Необходима спин-блокировка
Возможен переход в состояние ожидания при захваченной блокировке	Необходимо использовать мьютекс

Условные переменные

Условные переменные (completion variable) — простое средство синхронизации между двумя задачами в ядре, когда необходимо, чтобы одна задача отправила сигнал другой о том, что произошло некоторое событие. С помощью условной переменной одна задача сообщает системе, что она переходит в состояние ожидания до тех пор, пока другая задача не выполнит некоторую работу. Когда вся нужная работа будет выполнена, другая задача с помощью условной переменной уведомляет об этом систему и та возвращает к выполнению все ожидающие задачи. Если вам кажется, что все описанное выше похоже на работу семафора, то именно так оно и есть — идея та же. По сути, условные переменные обеспечивают простое решение проблемы, для которой в других ситуациях использовались бы семафоры. Например, в системной функции `vfork()` условная переменная используется для возврата к выполнению родительского процесса, когда порожденный им процесс запускается на выполнение или завершает свою работу.

Условные переменные представляются с помощью структуры `completion`, которая определена в файле `<linux/completion.h>`. Статически условная переменная может быть создана с помощью следующего макроса:

```
DECLARE_COMPLETION(mr_comp);
```

Для инициализации динамически созданной условной переменной используется функция `init_completion()`.

Чтобы перевести задачу в состояние ожидания до завершения определенного события, с которым связана условная переменная, нужно вызвать функцию `wait_for_completion()`. После того как событие произошло, с помощью вызова функции `complete()` можно просигнализировать об этом всем ожидающим задачам и возвратить их к выполнению. Функции для работы с условными переменными приведены в табл. 10.11.

Таблица. 10.11. Функции для работы с условными переменными

Функция	Описание
`init_completion(struct completion *)`	Инициализирует указанную динамически созданную условную переменную
`wait_for_completion(struct completion *)`	Ожидает сигнала, связанного с указанной условной переменной
`complete(struct completion *)`	Отправляет сигнал всем ожидающим задачам и возвращает их к выполнению

Примеры использования условных переменных приведены в файлах `kernel/sched.c` и `kernel/fork.c`. Чаще всего используются условные переменные, создаваемые динамически, как часть структуры данных. Если в коде ядра нужно подождать завер-

шения инициализации структуры данных, то следует вызвать функцию `wait_for_completion()`. Когда инициализация закончена, ожидающие задания возвращаются к выполнению с помощью вызова функции `complete()`.

BKL: большая блокировка ядра

Добро пожаловать к "рыжему пасынку" ядра. Большая блокировка ядра (Big Kernel Lock, или BKL) — это глобальная спин-блокировка, которая была создана специально для того, чтобы облегчить переход от первоначальной реализации SMP в операционной системе Linux к мелкоструктурным блокировкам. Блокировка BKL обладает рядом интересных свойств, перечисленных ниже.

- Во время удержания BKL можно переходить в состояние ожидания. Блокировка автоматически освобождается, когда задача переходит в состояние ожидания, и снова захватывается, когда задача планируется на выполнение. Конечно, это не означает, что *всегда 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. В отличие от спин-блокировок нельзя захватывать блокировку BKL в контексте прерывания.
- Использовать блокировку BKL в новом коде запрещено. С выходом каждой новой версии ядра в драйверах устройств и в других подсистемах ядра блокировка BKL используется все меньше и меньше.

Рассмотренные свойства дали возможность упростить переход от ядер серии 2.0 к серии 2.2. Когда в ядро 2.0 была введена поддержка SMP, в любой момент времени в ядре могла выполняться только одна задача. Разумеется, сейчас ядро распараллелено очень хорошо, ведь был 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 приводит к ухудшению масштабируемости[5].

Использовать блокировку BKL не рекомендуется. По сути, в новом коде она никогда не должна использоваться. Тем не менее эта блокировка все еще достаточно интенсивно используется в некоторых частях ядра. Поэтому важно иметь представление о большой блокировке ядра и ее интерфейсах. Блокировка BKL ведет себя как обычная спин-блокировка, за исключением тех особенностей, которые были рассмотрены выше. Для захвата блокировки BKL используется функция `lock_kernel()`, а для освобождения — функция `unlock_kernel()`. Блокировка BKL может захватываться в каждом потоке ядра несколько раз подряд, но после этого нужно вызвать функцию `unlock_kernel()` соответствующее число раз, чтобы освободить блокировку BKL. Данная блокировка освобождается только после последнего вызова функции. Функция `kernel_locked()` воз-

[5] Хотя, может быть, она и не такая страшная, какой ее иногда пытаются представить, все же некоторые люди считают ее "воплощением дьявола" в ядре.

вращает ненулевое значение, если блокировка в данный момент захвачена, в противном случае возвращается нуль.

Интерфейсы блокировки BKL определены в файле <linux/smp_lock.h>. Рассмотрим простой пример использования этой блокировки.

```
lock_kernel();

/*
 * Критический участок, который синхронизирован со всеми пользователями
 * блокировки BKL...
 * Обратите внимание, что в этом месте можно безопасно переходить в состояние
 * ожидания. При этом блокировка будет автоматически освобождена. После
 * возврата задачи к выполнению блокировка будет автоматически захвачена.
 * Это гарантирует, что не возникнет состояния взаимоблокировки,
 * но все-таки лучше не переходить в состояние ожидания,
 * если необходимо обеспечить защиту данных!
 */

unlock_kernel();
```

После захвата блокировки BKL запрещается мультипрограммный режим работы ядра. При работе ядра на однопроцессорной машине код BKL на самом деле не выполняет никаких физических блокировок. Полный список функций для работы с BKL приведен в табл. 10.12.

Таблица 10.12. Функции для работы с большой блокировкой ядра

Функция	Описание
`lock_kernel()`	Захватывает блокировку BKL
`unlock_kernel()`	Освобождает блокировку BKL
`kernel_locked()`	Возвращает ненулевое значение, если блокировка захвачена, и нуль — в противном случае (на однопроцессорных машинах всегда возвращается ненулевое значение)

Одна из основных проблем, связанных с большой блокировкой ядра, — определение того, что защищается с помощью данной блокировки. Часто блокировка BKL визуально ассоциируется с кодом (например, она "синхронизирует вызовы функции `foo()`"), а не с данными ("защищает структуру `foo`"). Это приводит к тому, что заменить BKL обычными спин-блокировками бывает сложно, потому что нелегко определить, что же все-таки необходимо блокировать. На самом деле подобная замена еще более сложная, так как необходимо учитывать все взаимосвязи между всеми участками кода, в которых эта блокировка используется.

Последовательные блокировки

Последовательная блокировка (sequential lock, или сокращенно seq lock) — это новый тип блокировки, который появился в ядрах серии 2.6. Она обеспечивает очень простой механизм для чтения и записи совместно используемых данных. Работа таких блокировок основана на счетчике последовательности событий. Перед записью нужных данных захватывается блокировка, и значение счетчика увеличивается на единицу. Перед чтением и после чтения данных выполняется считывание значения счетчика. Если два полученных значения одинаковы, то во время чтения данных новый цикл записи не был начат. Более того, если оба эти значения четные, то к моменту начала чтения цикл записи

был уже закончен. Дело в том, что при захвате блокировки для записи значение счетчика становится нечетным, а перед освобождением — снова четным, так как начальное значение счетчика равно нулю.

Для определения последовательной блокировки используется приведенный ниже код.

```
seqlock_t mr_seq_lock = DEFINE_SEQLOCK(mr_seq_lock);
```

Участок кода, в котором выполняется запись данных, может выглядеть следующим образом:

```
write_seqlock(&mr_seq_lock);

/* Получена блокировка для записи... */

write_sequnlock(&mr_seq_lock);
```

Этот фрагмент кода напоминает тот, в котором используется обычная спин-блокировка. Необычное решение появляется в ветке кода для чтения данных, который несколько отличается от ранее рассмотренных.

```
unsigned long seq;

do {
     seq = read_seqbegin(&mr_seq_lock);
     /* Здесь происходит чтение данных ... */
} while (read_seqretry(&mr_seq_lock, seq));
```

Последовательные блокировки позволяют добиться упрощенной и масштабируемой блокировки для очень быстрого доступа к данным в случае, когда применяется много потоков чтения и мало потоков записи. Однако при использовании этого типа блокировок потоки записи получают более высокий приоритет перед потоками чтения. Запрос на блокировку для записи всегда будет удовлетворен в случае, если нет других потоков записи. В отличие от спин-блокировок по чтению записи и семафоров потоки чтения никак не влияют на захват блокировки для записи. Более того, в потоках, ожидающих разрешения на запись данных, будет постоянно повторяться цикл чтения (как в показанном выше примере) до тех пор, пока не останется ни одного потока, удерживающего блокировку для записи.

Использование последовательной блокировки оправдано в случаях, когда удовлетворяются почти все перечисленные ниже условия.

- Чтение общих данных выполняется множеством потоков.
- Существует один или несколько потоков для записи общих данных.
- Хотя потоков записи всего несколько, необходимо отдать им приоритет и сделать так, чтобы многочисленные потоки чтения не смогли помешать записи общих данных.
- Структура защищаемых данных должна быть достаточно простой, либо это должно быть обычное целое число, операцию доступа к которому по каким-либо причинам нельзя сделать неделимой.

“Классикой” применения последовательных блокировок считается процесс чтения и обновления глобальной переменной системы Linux `jiffies`, в которой хранится время работы системы (см. главу 11, “Таймеры и управление временем”). Она представляет собой 64-разрядный счетчик импульсов системного таймера, прошедших с момента последней перезагрузки. На тех компьютерах, где отсутствуют средства неделимого чтения

64-разрядной переменной `jiffies_64`, реализована функция `get_jiffies_64()`, в которой используются последовательные блокировки, как показано ниже.

```
u64 get_jiffies_64(void)
{
    unsigned long seq;
    u64 ret;

    do {
           seq = read_seqbegin(&xtime_lock);
           ret = jiffies_64;
    } while (read_seqretry(&xtime_lock, seq));
    return ret;
}
```

Обновление переменной `jiffies_64` происходит в обработчике прерывания от системного таймера и требует применения варианта последовательной блокировки для записи.

```
write_seqlock(&xtime_lock);
jiffies_64 += 1;
write_sequnlock(&xtime_lock);
```

Для более глубокого ознакомления с подсистемой учета времени и переменной `jiffies` обратитесь к главе 11, "Таймеры и управление временем", а также проанализируйте файлы `kernel/timer.c` и `kernel/time/tick-common.c` дерева исходных кодов ядра.

Отключение мультипрограммного режима работы ядра

Поскольку ядро системы Linux является вытесняемым, выполнение некоторого процесса в ядре может быть приостановлено в произвольный момент времени и управление передано более высокоприоритетному процессу. Это означает, что новый процесс может начать свое выполнение в том же критическом участке кода, в котором выполнялся вытесненный процесс. Чтобы предотвратить такое нежелательное явление, в коде, отвечающем за вытеснение задач, используются спин-блокировки в тех местах, где выполнение должно осуществляться в однопрограммном режиме. Если такая спин-блокировка захвачена, ядро работает в режиме без вытеснения задач. Поскольку и при многопроцессорной обработке, и при мультипрограммировании приходится решать одни и те же проблемы (кроме того, код ядра является устойчивым к многопроцессорной обработке), такое простое дополнение позволяет также сделать ядро устойчивым к вытеснению.

Будем надеяться, что так оно и есть на самом деле. В реальности возникает ряд ситуаций, при которых нет необходимости использовать спин-блокировки, но нужно просто запретить режим вытеснения ядра. Чаще всего такая ситуация возникает из-за доступа к данным, привязанным к определенному процессору (per-processor data). Если каждый процессор обращается только к своим уникальным данным, то защищать их с помощью блокировок нет никакой необходимости. Дело в том, что в подобных случаях к этим данным может обратиться только один процессор. Однако если никакая из блокировок не удерживается, ядро является вытесняемым. Поэтому появляется возможность доступа к тем же самым переменным для вновь запланированного задания, как показано в следующем примере:

```
1: Задача A работает с процессорной переменной foo, которая не защищена
   блокировкой.
2: Задача A вытеснена.
3: Задача B запланирована на выполнение.
4: Задача B работает с той же процессорной переменной foo.
5: Задача B завершена.
6: Задача A вновь запланирована на выполнение.
7: Задача A продолжает работать с процессорной переменной foo.
```

Как видите, даже на однопроцессорной машине к некоторой переменной может псевдопараллельно обращаться несколько процессов. В обычной ситуации для такой переменной требуется спин-блокировка (для ее защиты при истинном параллелизме на многопроцессорной машине). Однако если эта переменная связана только с одним процессором, то блокировка для нее может и не требоваться.

Для устранения указанной проблемы режим вытеснения ядра можно запретить с помощью функции `preempt_disable()`. Вызов этой функции может быть вложенным, т.е. функцию можно вызывать много раз подряд. Для каждого такого вызова требуется соответствующий вызов функции `preempt_enable()`. Режим вытеснения ядра будет разрешен только после последнего вызова функции `preempt_enable()`, как показано в следующем примере:

```
preempt_disable();
/* Вытеснение запрещено... */
preempt_enable();
```

В специальном счетчике вытеснения хранится число захваченных блокировок плюс количество вызовов функции `preempt_disable()`. Если значение этого счетчика равно нулю, то режим вытеснения ядра разрешен. Если значение счетчика больше или равно единице, то режим вытеснения ядра запрещен. Значение этого счетчика трудно переоценить! Он невероятно полезен для отладки неделимых операций совместно с переходами в состояние ожидания. Значение этого счетчика возвращает функция `preempt_count()`. Описание функций для управления режимом вытеснения ядра приведено в табл. 10.13.

Таблица 10.13. Функции для управления режимом вытеснения ядра

Функция	Описание
`preempt_disable()`	Запрещает режим вытеснения кода ядра путем увеличения на единицу значения счетчика вытеснения
`preempt_enable()`	Уменьшает на единицу значение счетчика вытеснения; если его значение равно нулю, разрешается режим вытеснения ядра, выполняется проверка и повторное планирование ожидающих процессов
`preempt_enable_no_resched()`	Аналогична `preempt_enable()`, за исключением того, что проверка на повторное планирование не выполняется
`preempt_count()`	Возвращает значение счетчика вытеснения

Для корректного доступа к данным процессора можно вначале определить номер процессора с помощью функции `get_cpu()`. Этот номер, по-видимому, используется в качестве индекса для доступа к данным, связанным с определенным процессором. Пе-

ред возвратом текущего номера процессора эта функция запрещает режим вытеснения ядра, как показано ниже.

```
int cpu;

/* Запретим режим вытеснения ядра и присвоим переменной "cpu"
   номер текущего процессора */
cpu = get_cpu();

/* Работа с данными процессора... */

/* Восстановим режим вытеснения ядра. Значение переменной "cpu"
   при этом может измениться, поэтому ее значение больше некорректно */
put_cpu();
```

Порядок выполнения операций и барьеры

При обеспечении синхронной работы нескольких процессоров или аппаратных устройств иногда нужно, чтобы операции чтения (загрузки) и записи (сохранения) в память выполнялись именно в том порядке, в каком это было указано в коде программы. При работе с аппаратными устройствами часто необходимо, чтобы заданная операция чтения была выполнена перед другими операциями чтения или записи. Кроме того, на симметричной многопроцессорной системе иногда требуется, чтобы операции записи выполнялись строго в том порядке, в каком они были указаны в исходном программном коде. Как правило, это нужно для того, чтобы последующие операции чтения могли извлечь данные в том же порядке, в котором они были записаны. Описанные выше проблемы усложняются еще и тем, что и компилятор, и процессор могут изменять порядок выполнения команд чтения и записи[6] для повышения производительности работы программы. К счастью, в тех процессорах, в которых может изменяться порядок выполнения команд чтения и записи, предусмотрены специальные машинные команды, позволяющие строго определить этот порядок. Также существует возможность указать компилятору, что не следует изменять порядок выполнения операций вокруг некоторой точки программы. Эти указания называются *барьерами* (*barriers*).

По сути, при выполнении приведенного ниже кода на некоторых процессорах значение переменной `b` может быть сохранено в памяти *раньше*, чем значение переменной `a`.

```
a = 1;
b = 2;
```

Дело в том, что и компилятор, и процессор не “видят” явной связи между переменными `a` и `b`. Поэтому компилятор может выполнить такую перестановку статически еще на этапе компиляции. При этом в результирующем объектном коде значение переменной `b` будет изменяться перед значением переменной `a`. Тем не менее процессор во время цикла выборки и предварительного планирования может динамически изменять порядок выполнения отдельных, не связанных на первый взгляд друг с другом команд по только ему известному алгоритму. В большинстве случаев такая перестановка команд будет оптимальной, так как между переменными `a` и `b` нет никакой взаимосвязи. Тем не менее иногда программисту все-таки виднее.

[6] В процессорах семейства Intel x86 порядок выполнения команд записи никогда не изменяется, т.е. они выполняются всегда в том порядке, в каком это было указано в коде программы. Тем не менее в других процессорах дела могут обстоять совсем иначе.

Хотя в предыдущем примере процессор и может изменить порядок выполнения команд, ни процессор, ни компилятор никогда не будет менять порядок выполнения следующего кода, поскольку между значениями переменных a и b существует очевидная взаимосвязь.

```
a = 1;
b = a;
```

Однако ни компилятор, ни процессор не имеет никакой информации о коде, который выполняется в других контекстах. Иногда важно, чтобы результаты записи в память могли быть считаны в нужном нам порядке в другом коде, который выполняется за пределами нашей досягаемости, или даже на другой машине, расположенной в противоположной точке мира. Такая ситуация часто происходит при работе с аппаратными устройствами, а также возникает на многопроцессорных машинах.

Функция rmb() позволяет установить в памяти барьер для чтения (read memory barrier). Она гарантирует, что при изменении порядка выполнения команд чтения из памяти они не будут "перепрыгивать" через ее вызов. Другими словами, команды чтения из памяти, выполняемые перед вызовом функции rmb(), не будут переставлены местами с теми командами, которые выполняются после вызова этой функции. И наоборот, те команды чтения из памяти, которые выполняются после вызова этой функции, не будут переставлены местами с командами, выполняемыми до вызова функции rmb().

Функция wmb() позволяет установить в памяти барьер по записи (write barrier). Она работает так же, как и функция rmb(), но не с операциями чтения, а с операциями записи. При этом гарантируется, что операции записи, которые находятся по разные стороны барьера, никогда не будут переставлены местами друг с другом.

Функция mb() позволяет создать барьер в памяти как для чтения, так и для записи. Никакие команды чтения и записи, которые указаны по разные стороны вызова функции mb(), не будут переставлены местами друг с другом. Эта функция была реализована, поскольку существует аналогичная машинная команда (часто та же, что используется в функции rmb()), которая позволяет установить барьер на чтение и запись.

Вариант функции rmb() — read_barrier_depends() — позволяет создать барьер в памяти для чтения, но *только для тех команд, от которых зависят следующие за ними команды 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 в памяти барьер для чтения, так же как и функция rmb(), но этот барьер будет установлен только для некоторых команд 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 холостая команда noop (нет операции).

Рассмотрим пример использования функций mb() и rmb(). Предположим, первоначальное значение переменной a равно 1, а переменной b — 2 (табл. 10.14).

Без использования барьеров в памяти на некоторых процессорах возможна ситуация, когда переменной c будет присвоено *новое* значение переменной b, а переменной d — старое значение переменной a (скажем, 1). Например, c может равняться 4 (то, что нам нужно!), а d — 1 (этого мы никак не ожидали!). Использование функции mb() позволяет гарантировать, что значения переменных a и b будут записаны в память в указанном

порядке, а функция rmb() гарантирует, что чтение переменных c и d будет выполнено в указанном порядке.

Таблица 10.14. Пример использования барьеров в памяти

Поток 1	Поток 2
a = 3;	—
mb();	—
b = 4;	c = b;
—	rmb();
—	d = a;

Такое изменение порядка выполнения операций в современных процессорах может возникнуть из-за оптимизации использования конвейера на фазах предварительного планирования и выполнения команд. В предыдущем примере это может привести к тому, что машинные команды, присваивающие значения переменным b и a, будут выполнены не в том порядке. Функции rmb() и wmb() соответствуют машинным командам, которые заставляют процессор выполнить все незаконченные операции чтения и записи перед тем, как продолжить работу.

Теперь рассмотрим похожий пример, в котором вместо функции rmb() используется функция read_barrier_depends(). В этом примере первоначально переменной a присвоено значение 1, переменной b — 2, а переменная p содержит адрес переменной b (табл. 10.15).

Таблица 10.15. Обновленный пример использования барьеров в памяти

Поток 1	Поток 2
a = 3;	—
mb();	—
p = &a;	pp = p;
—	read_barrier_depends();
—	b = *pp;

Опять-таки без установки барьеров в памяти возможна ситуация, когда переменной b будет присвоено значение, на которое указывает переменная pp, до того, как переменной pp будет присвоено значение переменной p. Однако функция read_barrier_depends() обеспечивает достаточный барьер в памяти, поскольку выборка значения, на которое указывает переменная pp, зависит от выборки значения переменной p. Здесь также будет достаточно использовать функцию rmb(), но так как операции чтения взаимозависимы, лучше использовать потенциально более быструю функцию read_barrier_depends(). Обратите внимание на то, что в любом случае требуется использовать функцию mb(), 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 определяют обычные барьеры в памяти, а для ядра, рассчитанного на однопроцессорную машину, — только

барьер для 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 чтения 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 чтения в случаях, когда это может повлиять на правильность выполнения кода, написанного на языке C, или на существующие зависимости между данными. Однако у компилятора нет информации о событиях, которые могут произойти вне текущего контекста. Например, компилятор не имеет информации о прерываниях, в контексте которых может выполняться считывание данных, записывающихся в данный момент. Поэтому нужно сделать так, чтобы операции записи выполнялись перед операциями чтения. Описанные выше барьеры в памяти работают и как барьеры для компилятора, но последний значительно проще, чем барьер в памяти, и практически не влияет на производительность. В самом деле, использовать на практике барьер для компилятора необязательно, поскольку он просто препятствует *возможному* изменению порядка выполнения операций компилятором.

В табл. 10.16 приведен полный список функций установки барьеров в памяти и для компилятора, доступных для разных аппаратных платформ, поддерживаемых ядром Linux.

Таблица 10.16. Средства установки барьеров для компилятора и в памяти

Функция	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 взаимозависимых операций
`wmb()`	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()`
`smp_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 оптимизацию компилятора для команд чтения и записи данных при переходе через барьер

Следует заметить, что эффекты установки барьеров могут быть разными для разных аппаратных платформ. Например, если процессор не изменяет порядка выполнения опе-

раций записи (как в случае микропроцессоров семейства Intel x86), функция `wmb()` не выполняет никаких действий. Используйте соответствующий барьер в памяти для самой плохой ситуации (т.е. для процессора с самым плохим порядком выполнения), и ваш код будет скомпилирован оптимально для вашей аппаратной платформы.

Резюме

В этой главе рассказывалось о том, как применять на практике понятия, описанные в предыдущей главе, чтобы лучше разобраться с функциями ядра, которые помогают осуществлять синхронное и параллельное выполнение операций. Вначале были рассмотрены самые простые методы, обеспечивающие синхронизацию выполнения, — неделимые операции. Далее были описаны спин-блокировки — часто используемые типы блокировок в ядре. Они построены на основе периодической проверки в цикле условия освобождения блокировки и позволяют гарантировать, что доступ к ресурсу получит только один поток выполнения. После этого были рассмотрены семафоры — блокировки, которые переводят вызывающий процесс в состояние ожидания, а также их более специализированные и чаще используемые типы — мьютексы. Следом за мьютексами были описаны более специфические (и менее используемые) типы блокировок, такие как условные переменные и последовательные блокировки. Мы в шутливом тоне описали большую блокировку ядра BKL, рассмотрели методы запрещения режима вытеснения кода ядра и коснулись барьеров. Спектр информации был очень большой.

Вооружившись арсеналом средств синхронизации, описанных в данной главе, вы сможете писать код ядра, свободный от конфликтов из-за использования ресурсов, обеспечивающий необходимую синхронизацию с помощью самого подходящего для этого инструментария и корректно работающий на машинах с несколькими процессорами.

11
Таймеры и управление временем

Отслеживание хода времени — одна из важных функций ядра. В ядре существует довольно большое количество функций, которые запускаются по сигналам времени (time-driven) и отличаются от тех, которые запускаются по событиям[1] (event-driven). Часть из этих функций запускаются периодически, как, например, функции балансировки очередей выполнения планировщика или обновления содержимого экрана. Такие функции вызываются по жесткому графику, например 100 раз в секунду. Другие функции, такие как обработка отложенных дисковых операций ввода-вывода, планируются ядром на выполнение в некоторый относительный момент времени в будущем. Например, ядро может запланировать работу на выполнение в момент времени, который наступит через 500 миллисекунд относительно текущего момента времени. И наконец, ядро должно уметь вычислять время непрерывной работы системы (uptime), а также текущую дату и время.

Следует обратить внимание на разницу между относительным и абсолютным временем. Планирование выполнения некоторой работы на будущее, скажем, через 5 секунд, 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.

Кроме того, имеются отличия в реализации обработчиков событий, возникающих периодически, и тех, которые планируются на выполнение ядром в некоторый фиксированный момент времени в будущем. События, возникающие периодически (скажем, каждые 10 мс), генерируются *системным таймером*. Последний представляет собой программируемое аппаратное устройство, которое генерирует аппаратные прерывания с фиксированной частотой. В обработчике этого прерывания, которое называется *прерыванием*

[1] Если быть точными, то функции, запускаемые по сигналам времени, на самом деле запускаются по событиям, соответствующим ходу времени. В этой главе будут рассмотрены в основном события, связанные со временем, так как они встречаются очень часто и являются важными для ядра.

от таймера (*timer interrupt*), обновляется значение системного времени и выполняются периодические действия. Системный таймер и его прерывания являются основной движущей силой, отвечающий за работу операционной системы Linux, поэтому в текущей главе им уделяется основное внимание.

Кроме того, в главе будут рассмотрены *динамические таймеры* (*dynamic timers*) — средства, позволяющие планировать события, которые выполняются один раз, после истечения некоторого интервала времени. Например, такой таймер используется в драйвере накопителя на гибких магнитных дисках для остановки двигателя привода, если тот не используется в течение некоторого периода времени. Подобные таймеры создаются и аннулируются в ядре динамически, по мере необходимости. В этой главе мы рассмотрим реализацию динамических таймеров, а также их интерфейс, который используется в программном коде.

Основная идея учета времени в ядре

Очевидно, что концепция *времени* с точки зрения компьютера является несколько неопределенной. В действительности, для того чтобы получать информацию о текущей дате и управлять системным временем, ядро должно взаимодействовать с системным аппаратным обеспечением. В это аппаратное обеспечение помимо прочего входит системный таймер, который используется ядром для измерения прошедшего времени. “Сердцем” системного таймера является кварцевый резонатор, обеспечивающий очень точное значение периода вырабатываемых импульсов. Такой же резонатор используется в цифровых электронных часах или тактовом генераторе процессора. Системный таймер вырабатывает эти импульсы с заранее *заданной частотой* (tick rate), называемой *частотой импульсов* (tick rate). Как только истекает период очередного импульса, процессору посылается прерывание от таймера, в результате чего вызывается специальный обработчик прерывания.

Поскольку ядру известна частота импульсов системного таймера, оно может точно вычислить интервал времени, прошедший между двумя прерываниями от таймера. Этот интервал времени между двумя импульсами системного таймера называется *периодом следования* (tick) импульсов, измеряется в долях секунды и равен обратному значению частоты системного таймера. Собственно, вам уже должно быть понятно, как в ядре отслеживается абсолютное значение времени, а также относительный интервал времени, прошедший с момента последней перезагрузки системы. Абсолютное значение времени соответствует текущему значению времени суток и очень важно для работы пользовательских приложений. Ядро в состоянии отследить это время просто потому, что оно обрабатывает прерывания от таймера. В ядре предусмотрено семейство системных функций, позволяющих пользовательским приложениям получать информацию о дате и времени дня. Время непрерывной работы системы (system uptime) является относительным и отсчитывается от момента последней перезагрузки системы. Оно используется как в системных, так и в пользовательских приложениях. Во многих программах требуется знать интервал времени, *прошедший* между двумя событиями. Проще всего его измерить как разность между двумя значениями времени непрерывной работы системы, считанными в нужные моменты.

Прерывание от таймера очень важно для управления работой всей операционной системы. Существует большое количество функций ядра, которые запускаются и завершают свою работу в соответствии с ходом времени. Кроме того, ряд функций запускается периодически по сигналу прерывания от таймера. Они перечислены ниже.

- Обновление значения времени непрерывной работы системы (uptime).
- Обновление абсолютного значения времени (time of day).
- Для SMP-систем выполняется проверка сбалансированности очередей выполнения планировщика. В случае выявления разбалансировки выполняется их повторная балансировка, как было рассказано в главе 4, “Системный планировщик и диспетчеризация процессов”.
- Запуск обработчиков всех динамических таймеров, для которых истек период времени.
- Обновление статистики использования процессорного времени и других ресурсов.

Часть из перечисленных выше действий выполняется 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. Другими словами, эти функции выполняются с частотой, которая кратна частоте системного таймера. Сам обработчик прерываний от системного таймера будет рассмотрен ниже, в разделе “Обработчик прерываний от таймера”.

Частота импульсов таймера: директива `HZ`

Частота импульсов системного таймера (tick rate) программируется при загрузке системы на основании значения статически определенной директивы препроцессора `HZ`. Значение параметра `HZ` зависит от используемой аппаратной платформы. Более того, 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 в герцах равна значению параметра `HZ`, а период таймера равен `1/HZ` секунды. Например, стандартное значение параметра `HZ` для платформы x86 равно 100. Следовательно, частота системного таймера для машин, оснащенных процессором семейства i386, равна 100 Гц. Иначе говоря, каждую секунду происходит 100 прерываний от системного таймера, т.е. каждое прерывание происходит раз в одну сотую доли секунды, или каждые 10 мс. Используются и другие значение параметра `HZ`, такие как 250 и 1000, соответствующие периоду 4 и 1 мс. В табл. 11.1 приведен полный список всех поддерживаемых аппаратных платформ и определенных для них значений частоты системного таймера.

Таблица 11.1. Значение частоты системного таймера

Аппаратная платформа	Частота, Гц	Аппаратная платформа	Частота, Гц
Alpha	1024	Mips	100
Arm	100	mn10300	100
avr32	100	parisc	100
Blackfin	100	powerpc	100
Cris	100	Score	100
h8300	100	s390	100
ia64	1024	sh	100

Окончание табл. 11.1

Аппаратная платформа	Частота, Гц	Аппаратная платформа	Частота, Гц
m32r	100	sparc	100
m68k	100	Um	100
m68knommu	50, 100 или 1000	x86	100
Microblaze	100		

При написании кода ядра нельзя считать, что параметр `HZ` имеет заранее определенное фиксированное значение. В наши дни это уже не такая часто встречающаяся ошибка, так как поддерживается много различных аппаратных платформ с разными частотами системного таймера. Однако раньше аппаратная платформа Alpha была единственной, для которой частота системного таймера отличалась от 100 Гц, и не редко 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.

Значение частоты системного таймера достаточно важно. Как вы увидите чуть ниже, в обработчике прерывания от системного таймера выполняется достаточно большой объем работы. Очевидно, что идея отслеживания времени в ядре основана на периодичности колебаний системного таймера. Поэтому выбор правильного значения периода колебаний сродни установке хороших дружеских отношений — все дело в компромиссе!

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. И хотя значение частоты системного таймера в современных версиях ядра Linux снова равно 100 Гц, в них был введен специальный параметр конфигурации, позволяющий пользователям самим задавать значение параметра `HZ`. Поскольку работа системы во многом завязана на прерываниях от таймера, изменение значения его частоты должно оказывать сильное влияние на саму систему. Разумеется, как в случае больших, так и малых значений параметра `HZ` есть свои положительные и отрицательные стороны.

Увеличение частоты системного таймера означает, что обработчик прерываний от таймера будет запускаться более часто. Следовательно, вся работа, которую он делает, также будет выполняться чаще. Это позволяет получить перечисленные ниже преимущества.

- Прерывание от таймера имеет б*о*льшую разрешающую способность по времени, и, следовательно, все события, которые выполняются во времени, также имеют б*о*льшую разрешающую способность.
- Увеличивается точность выполнения событий во времени.

Разрешающая способность увеличивается во столько же раз, во сколько раз возрастает частота импульсов системного таймера. Например, точность работы таймеров при частоте импульсов 100 Гц равна 10 мс. Это означает, что все периодические события могут возникать только синхронно с очередным прерыванием от таймера, которое генери-

руется раз в 10 мс, при этом бо́льшая *точность*[2] не гарантируется. Тем не менее при частоте, равной 1000 Гц, разрешающая способность составляет уже 1 мс, т.е. в десять раз выше. Хотя ядро позволяет создавать таймеры с временны́м разрешением, равным 1 мс, при частоте системного таймера 100 Гц нет возможности гарантированно получить временной интервал короче 10 мс.

Точность измерения времени также возрастает аналогичным образом. Предположим, таймеры ядра запускаются в случайные моменты времени. Тогда в среднем таймеры будут срабатывать с точностью до половины периода прерывания системного таймера. Дело в том, что период времени таймера может исчерпаться в любой момент, а обработчик таймера может запуститься только в момент возникновения прерывания. Например, при частоте 100 Гц события в среднем будут возникать с точностью до +/– 5 мс от желаемого момента времени. Поэтому ошибка измерения времени в среднем составит 5 мс. При частоте 1000 Гц ошибка измерения в среднем уменьшается до 0,5 мс, т.е. точность возрастает в десять раз!

Преимущества больших значений параметра `HZ`

Как уже отмечалось, при увеличении частоты системного таймера возрастает разрешающая способность системы по времени и увеличивается точность измерения интервалов времени. Это позволяет получить перечисленные ниже преимущества.

- Таймеры ядра запускаются более часто, в результате чего обеспечивается лучшая точность измерения времени. (Это позволяет получить много разных преимуществ, часть из которых описана ниже.)
- Системные функции, такие как `poll()` и `select()`, которые позволяют при желании указать в качестве параметра время ожидания (timeout), выполняются с большей точностью.
- Результаты измерений, такие как время использования ресурсов или время непрерывной работы системы, фиксируются с большей точностью.
- Вытеснение процессов выполняется более корректно.

Некоторые из самых заметных улучшений производительности системы связаны с увеличением точности измерения периодов времени ожидания при выполнении системных функций `poll()` и `select()`. Это улучшение может быть достаточно существенным. Если в прикладной программе достаточно интенсивно используются вызовы этих системных функций, то при ожидании очередного поступления прерывания от таймера может пройти достаточно много времени. При этом может оказаться, что период ожидания уже давно истек. Не забывайте, что средняя ошибка измерения времени (т.е. потенциально зря потраченное время) равна половине длительности периода прерывания от таймера.

Еще одно преимущество высокой частоты системного таймера связано с более корректным вытеснением процессов. В результате планирование процессов выполняется быстрее, поскольку уменьшаются задержки на запуск системного планировщика. Вспомним из материала главы 4, “Системный планировщик и диспетчеризация процессов”,

[2] 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.

что в обработчике прерывания от таймера уменьшается значение кванта времени выполняющегося процесса. Как только оно достигает нуля, в ядре устанавливается флаг `need_resched` и при первой же удобной возможности запускается системный планировщик. А теперь рассмотрим ситуацию, когда у выполняемого в данный момент времени процесса значение кванта времени равно 2 мс. Это значит, что через 2 мс системный планировщик *должен* будет во что бы то ни стало прервать выполнение текущего процесса и передать управление новому процессу. К сожалению, это событие не может произойти раньше, чем будет сгенерировано следующее прерывание от системного таймера, а это не всегда будет ровно через 2 мс. В самом худшем случае следующее прерывание от таймера может возникнуть аж через `1/HZ` секунд! В случае, когда параметр `HZ=100`, процессу может быть выделено порядка 10 лишних миллисекунд процессорного времени. 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 задержек, связанных с запуском планировщика. Если в планируемой на выполнение задаче выполняются какие-нибудь критичные ко времени действия, как, например, заполнение буфера аудиоустройства, то подобные задержки не допустимы. Увеличение частоты системного таймера до 1000 Гц уменьшает задержку запуска планировщика в худшем случае до 1 мс, а в среднем — до 0,5 мс.

Недостатки больших значений параметра `HZ`

Очевидно, что должна существовать и обратная сторона увеличения частоты системного таймера, иначе она была бы с самого начала равна 1000 Гц (или больше). На самом деле существует одна большая проблема. Повышение частоты системного таймера вызывает рост накладных расходов, поскольку в единицу времени будет происходить большее количество прерываний, которые должны быть обработаны процессором. Чем выше частота, тем больше времени процессор должен тратить на выполнение кода обработчика прерывания от таймера. Это приводит не только к тому, что другим задачам отводится меньше процессорного времени, но и к периодической очистке процессорного кеша, падению его производительности и увеличению потребляемой мощности. Проблема, связанная с накладными расходами, вызывает споры. Очевидно, что увеличение значения с `HZ=100` до `HZ=1000` в десять раз увеличивает накладные расходы. Но от какого реального значения накладных расходов следует отталкиваться? Если “ничего” умножить на 10, то получится тоже “ничего”. Однако все сошлись на том, что, по крайней мере для современных систем, значение параметра `HZ=1000` не приводит к недопустимым накладным расходы. Тем не менее для ядер серии 2.6 существует возможность скомпилировать ядро с другим значением параметра `HZ`[3].

Возможна ли операционная система без периодических прерываний от таймера

У вас может возникнуть вопрос, всегда ли для функционирования операционной системы нужно задействовать постоянные прерывания от таймера? Использование таймера считалось нормой при проектировании операционных систем на протяжении последних сорока лет. При этом практически во всех операционных системах общего назначения он применялся именно так, как было описано выше в этой главе. И несмотря на все это, в ядре Linux предусмотрен

[3] В связи с ограничениями, накладываемыми аппаратной платформой и протоколом NTP, значение переменной HZ не может быть произвольным. Для платформы x86 значения 100, 500 и 1000 работают хорошо.

специальный параметр конфигурации, который обеспечивает работоспособность системы без периодических прерываний от таймера (tickless operation). Если при сборке ядра был указан параметр конфигурации `CONFIG_HZ`, то система будет динамически управлять прерываниями от таймера в соответствии с активными в настоящий момент таймерами. Вместо того чтобы генерировать прерывания от таймера периодически, скажем, каждую 1 мс, в обработчике прерывания этот интервал изменяется динамически по мере необходимости. Например, если очередной таймер должен сработать только через 3 мс, следующее прерывание от таймера возникнет именно через 3 мс. А если после этого в течение 50 мс ни один таймер не должен сработать, то ядро запланирует следующее прерывание от таймера только через 50 мс.

Уменьшение накладных расходов, связанных с обслуживанием периодических прерываний от таймера, — это, конечно, хорошо. Однако главной целью здесь было уменьшение потребляемой мощности, особенно в тех системах, которые практически простаивают. При использовании стандартного подхода с периодическими прерываниями от таймера ядро операционной системы вынуждено их обслуживать даже в моменты простоя. При использовании подхода с динамическим изменением интервала прерывания от таймера в моменты простоя система не будет обрабатывать совершенно ненужные прерывания от таймера, что уменьшит потребляемую мощность. Не имеет значения, каков интервал простоя, 200 мс или 200 с, по прошествии некоторого времени экономия потребляемой мощности будет довольно внушительной.

Переменная `jiffies`

В глобальной переменной `jiffies` содержится количество импульсов системного таймера, которые были получены со времени загрузки системы. Во время начальной загрузки ядро обнуляет значение этого параметра; при каждом прерывании системного таймера он будет увеличиваться на единицу. Поскольку каждую секунду возникает `HZ` прерываний от системного таймера, за секунду значение переменной `jiffies` увеличивается на `HZ`. Таким образом, время непрерывной работы системы (uptime) равно `jiffies/HZ` секунд. Однако на практике все немного сложнее. Ядро присваивает переменной `jiffies` специальное начальное значение, которое вызывает ее частые переполнения, а затем обрабатывает возникшую исключительную ситуацию. Когда запрашивается реальное значение переменной `jiffies`, это "смещение" вначале вычитается.

Этимология слова `jiffy`

Происхождение слова *jiffy* (миг, мгновение) точно неизвестно. Считается, что фразы типа *"in a jiffy"* (*в одно мгновение*) появились в Англии в XVIII 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 мс). В физике это слово иногда используется для указания интервала времени, за который свет проходит 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, вызвавшими прерывания, которые были успешно обработаны. В электротехнике *jiffy* — период переменного тока. В США *jiffy* — это 1/60 с.

В приложении к операционным системам, в частности к Unix, *jiffy* — это интервал времени между двумя соседними успешно обработанными прерываниями от системного таймера. Исторически это значение равно 10 мс. Как уже было показано, интервал времени jiffy в операционной системе Linux может иметь разные значения.

Переменная jiffies определяется в файле <linux/jiffies.h> так, как показано ниже.

```
extern unsigned long volatile jiffies;
```

Определение этой переменной достаточно специфичное, и оно будет рассмотрено более подробно в следующем разделе. А сейчас рассмотрим пример кода ядра. Пересчет интервала времени, выраженного в секундах, в значение переменной jiffies можно выполнить следующим образом:

```
(секунды * HZ)
```

Отсюда следует, что преобразование из значения переменной jiffies в интервал времени, выраженный в секундах, можно выполнить так:

```
(jiffies / HZ)
```

Первый вариант преобразования времени из секунд в отсчеты таймера встречается чаще. Например, часто необходимо установить значение некоторого момента времени в будущем.

```
unsigned long time_stamp = jiffies;          /* Текущее время */
unsigned long next_tick = jiffies + 1;       /* Через один импульс таймера
*/
unsigned long later = jiffies + 5*HZ;        /* Через пять секунд */
unsigned long fraction = jiffies + HZ / 10; /* Через 1/10 с */
```

Преобразование интервалов времени, выраженных в импульсах системного таймера, в секунды обычно используется при взаимодействии с пространством пользователя, так как в самом ядре редко используется абсолютное время.

Обратите внимание на то, что переменная jiffies имеет тип unsigned long, и использовать какой-либо другой тип для сохранения ее значения — некорректно.

Внутреннее представление переменной jiffies

Исторически переменная jiffies всегда представлялась с помощью типа unsigned long и, следовательно, имеет длину 32 бита для 32-разрядных аппаратных платформ и 64 бита для 64-разрядных. В случае 32-разрядного значения переменной jiffies и частоты системного таймера 100 Гц переполнение этой переменной будет происходить примерно каждые 497 дней. Однако при увеличении значения параметра HZ до 1000 период переполнения уменьшается всего до 47,9 дня! В случае 64-разрядного значения переменной jiffies ее переполнение 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 переменной jiffies тип unsigned long. Для решения проблемы пришлось немного подумать и сделать небольшой трюк при компоновке ядра.

Как уже говорилось, переменная jiffies имеет тип unsigned long и определяется так, как показано ниже.

```
extern unsigned long volatile jiffies;
```

Вторая переменная определяется в файле <linux/jiffies.h> в следующем виде:

```
extern u64 jiffies_64;
```

В сценарии компоновщика ld(1), который используется для сборки главного образа ядра (для аппаратной платформы x86 он находится в файле arch/x86/kernel/

vmlinux.lds.S), указана специальная директива, указывающая 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. В результате в коде не нужно ничего менять и можно обращаться к 32-разрядной переменной jiffies, как и раньше. Так как в большинстве случаев переменная jiffies используется для измерения промежутков времени, для большей части кода существенными являются только младшие 32 бита. В коде, который используется для управления ходом времени, задействуются все 64 бита, что предотвращает возможность переполнения 64-разрядного значения счетчика. На рис. 11.1 показана структура переменных jiffies и jiffies_64.

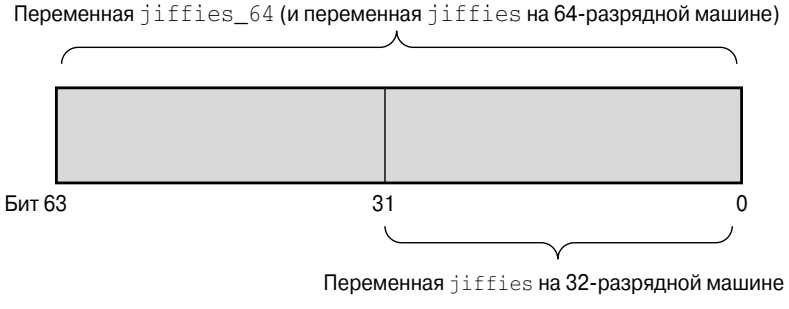


Рис. 11.1. Структура переменных jiffies и jiffies_64

Таким образом, если в коде требуется получить значение переменной jiffies, при этом просто выбирается значение младших 32 битов переменной jiffies_64. Для получения полного 64-разрядного значения переменной jiffies_64 используется специальная функция get_jiffies_64()[4]. Однако такая необходимость возникает редко, поскольку в большей части кода младшие 32 разряда считываются непосредственно из переменной jiffies.

На 64-разрядных аппаратных платформах переменные jiffies_64 и jiffies просто совпадают. При этом в коде можно либо непосредственно считывать значение переменной jiffies, либо использовать функцию get_jiffies_64(); оба действия позволяют получить одинаковый эффект.

Переполнение переменной jiffies

Переменная jiffies, так же как и любое целое число языка программирования C, после достижения своего 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 4 294 967 295 импульсов таймера. При этом, если значение счетчика равно данному значению и счетчик увеличивается на 1, значение счетчика становится равным нулю.

[4] Эта функция необходима, так как на 32-разрядных аппаратных платформах нельзя выполнить неделимую операцию загрузки двух машинных слов для получения 64-разрядного значения. Перед чтением значения эта функция блокирует доступ к переменной jiffies с помощью блокировки xtime_lock.

Рассмотрим пример, в котором может быть переполнение.

```
unsigned long timeout = jiffies + HZ/2; /* Тайм-аут в полсекунды */

/* Выполнение некоторой работы ... */
/* Проверим, не слишком ли много времени это заняло */

if (timeout > jiffies) {
        /* Мы не превысили лимит времени, отлично... */
} else {
        /* Мы превысили лимит времени, ошибка... */
}
```

Вначале этого фрагмента кода вычисляется значение лимита времени до наступления некоторого события в будущем. В нашем примере он равен полсекунды относительно текущего момента. Затем в коде продолжает выполняться некоторая работа, например, запись данных в аппаратное устройство и ожидание ответа. После выполнения, если весь процесс превысил лимит установленного времени, в коде соответствующим образом обрабатывается ошибка.

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 вычисления значения переменной `timeout`[5]? Тогда первое условие будет всегда выполняться и мы не сможем определить, был ли на самом деле тайм-аут или нет. Дело в том, что после переполнения значение переменной `jiffies` будет всегда меньше, чем значение переменной `timeout`, хотя логически оно должно быть больше. При тайм-ауте значение переменной `jiffies` должно быть огромным числом, всегда бо́льшим значения переменной `timeout`. Так как эта переменная переполнилась, 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. Ой!

К счастью, в ядре предусмотрены четыре макроса для сравнения двух значений счетчика импульсов таймера с учетом возможного переполнения. Все они определены в файле `<linux/jiffies.h>`. Ниже приведены упрощенные версии этих макросов.

```
#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 истинное значение, если оба параметра равны друг другу.

Версия кода из предыдущего примера, в которой учтены ошибки, связанные с переполнением, будет выглядеть следующим образом:

---

[5] Здесь предполагается, что при вычислении значения `timeout` переполнения еще не было, но оно оказалось достаточно близко к переполнению.

```
unsigned long timeout = jiffies + HZ/2; /* Тайм-аут в полсекунды */

/* ... */

if (time_before(jiffies, timeout)) {
        /* Мы не превысили лимит времени, отлично... */

} else {
        /* Мы превысили лимит времени, ошибка... */
}
```

Если вы заинтересовались, 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.

Пользовательские программы и параметр `HZ`

В ранних версиях ядер, предшествовавших серии 2.6, изменение значения параметра `HZ` приводило к различным 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, в пользовательских приложениях считалось, что параметр `HZ` имеет предопределенное значение. Следовательно, при изменении значения параметра `HZ` изменялись значения, которые экспортировались в пространство пользователя, в одинаковое число раз. Информация о том, во сколько раз изменились значения, в пространство пользователя не передавалась! В результате полученное от ядра значение времени работы системы могло интерпретироваться как 20 часов, хотя на самом деле оно равнялось только двум часам.

Чтобы исправить такое положение вещей, в коде ядра нужно было преобразовывать все значения переменной `jiffies`, которые экспортировались в пространство пользователя. Такое преобразование реализуется путем определения константы `USER_HZ`, равной значению параметра `HZ`, которое *должно* передаваться в пространстве пользователя. Поскольку для аппаратной платформы x86 значение параметра `HZ` исторически равно 100, значение константы `USER_HZ` также равно `100`.

Для преобразования значения счетчика импульсов системного таймера, выраженного в единицах `HZ`, в значение счетчика импульсов, выраженное в единицах `USER_HZ`, используется функция `jiffies_to_clock_t()`, определенная в файле `kernel/time.c`. Используемое в этой функции выражение зависит от того, кратны ли значения параметров `USER_HZ` и `HZ` один другому и не является ли значение `USER_HZ` меньшим или равным значению `HZ`. Если оба условия соблюдаются, то для большинства используемых систем выражение для преобразования значений таймера очень простое:

```
return x / (HZ / USER_HZ);
```

Если значения не кратны, то используется более сложный алгоритм.

Кроме того, для преобразования 64-разрядных значений переменной `jiffies` из `HZ` в `USER_HZ` используется функция `jiffies_64_to_clock_t()`.

Данные 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;
unsigned long total_time;

start = jiffies;

/* Выполнение работы ... */

total_time = jiffies - start;
printk("Работа заняла %lu импульсов таймера\n", jif-
fies_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 функция выполнит нужное преобразование, и все будут счастливы. Конечно, этот пример несколько нелогичный: больше смысла имело бы вывести значение времени не в импульсах системного таймера, а в секундах, как показано ниже.

```
printk("Работа заняла %lu секунд\n", total_time / HZ);
```

Аппаратные часы и таймеры

Для упрощения отсчета времени на всех поддерживаемых компьютерных платформах предусмотрены два аппаратных устройства — системный таймер, который был рассмотрен выше, и часы реального времени. Реализация и принцип работы этих устройств могут быть различными для машин разного типа, но общее их назначение и структура почти всегда одинаковы.

Часы реального времени

Часы реального времени (real-time clock, RTC) представляют собой энергонезависимое устройство, в котором хранится системное время. Устройство RTC продолжает отслеживать ход времени, даже когда компьютер обесточен, благодаря небольшой батарейке, которая обычно находится на системной плате. Для аппаратной платформы PC устройство RTC и энергонезависимая КМОП-память (CMOS) находятся в одной микросхеме. При этом используется общая батарейка, обеспечивающая и работу устройства RTC, и сохранение установок BIOS.

При выполнении начальной загрузки 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 повторно, 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 самую важную роль в процессе отслеживания хода времени в ядре и чаще всего используется для этой цели. Независимо от используемой аппаратной платформы идея, лежащая в основе работы системного таймера, всегда одна и та же — обеспечение механизма управления прерываниями, возникающими периодически с постоянной частотой. На некоторых аппаратных платформах это реализуется с помощью электронного генератора, выдающего колебания с программируемой частотой. На других аппаратных платформах используется декрементный счетчик (decrementer). В него

записывается 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 в качестве основного системного таймера служит устройство, называемое программируемым интервальным таймером (Programmable Interval Timer, PIT). Таймер PIT существует на всех машинах платформы PC и используется для управления прерываниями еще со времен операционной системы DOS. Во время начальной загрузки в коде ядра программируется таймер PIT (он подключен к нулевой линии прерывания IRQ0), чтобы периодически генерировать прерывание с частотой `HZ`. Хотя этот таймер представляет собой очень простое устройство с ограниченными возможностями, тем не менее оно хорошо выполняет свою работу. На платформе x86 существуют и другие источники времени, такие как таймер APIC (Advanced Programmable Interrupt Controller — расширенный программируемый контроллер прерываний) и счетчик временных меток (Time Stamp Counter, TSC).

Обработчик прерываний от таймера

Теперь, когда мы разобрались, что такое `HZ` и `jiffies`, а также какова роль системного таймера, рассмотрим реализацию обработчика прерываний от системного таймера. Обработчик прерываний от таймера состоит из двух частей: зависимой от аппаратной платформы и независимой от используемого оборудования.

Подпрограмма, зависящая от аппаратной платформы, регистрируется в качестве обработчика прерываний от системного таймера и запускается, когда срабатывает системный таймер. Разумеется, выполняемые ею действия зависят от аппаратной платформы, но в большинстве обработчиков выполняются перечисленные ниже действия.

- Захватывается блокировка `xtime_lock`, которая закрывает доступ к переменной `jiffies_64` и значению текущего времени — переменной `xtime` — со стороны других процессов.
- Считывается или сбрасывается состояние системного таймера, если это необходимо.
- Периодически записывается новое значение абсолютного времени в CMOS-память часов реального времени.
- Вызывается аппаратно-независимая функция таймера `tick_periodic()`.

В аппаратно-независимой функции `tick_periodic()` выполняется значительно больше действий.

- Значение переменной `jiffies_64` увеличивается на единицу (это безопасная операция даже для 32-разрядных аппаратных платформ, так как блокировка `xtime_lock` была захвачена раньше).
- Для выполняющегося в данный момент процесса обновляется статистка использования системных ресурсов, таких как процессорное время, затраченное на выполнение кода ядра и кода пользовательского приложения.
- Выполняются обработчики динамических таймеров, для которых истек период времени ожидания (это будет рассмотрено в следующем разделе).

- Вызывается функция `scheduler_tick()`, которая была описана в главе 4, “Системный планировщик и диспетчеризация процессов”.
- Обновляется значение абсолютного времени, которое хранится в переменной `xtime`.
- Вычисляется значение печально известного показателя средней загруженности системы (load average).

Сама по себе описываемая функция очень проста, так как большинство рассмотренных действий выполняется в других функциях.

```
static void tick_periodic(int cpu)
{
   if (tick_do_timer_cpu == cpu) {
       write_seqlock(&xtime_lock);

       /* Определим следующее событие таймера */
       tick_next_period = ktime_add(tick_next_period, tick_period);

       do_timer(1);
       write_sequnlock(&xtime_lock);
   }

   update_process_times(user_mode(get_irq_regs()));
   profile_tick(CPU_PROFILING);
}
```

Б*о*льшая часть важной работы выполняется в функциях `do_timer()` и `update_process_times()`. В первой функции значение переменной `jiffies_64` увеличивается на единицу.

```
void do_timer(unsigned long ticks)
{
   jiffies_64 += ticks;
   update_wall_time();
   calc_global_load();
}
```

В функции `update_wall_time()`, как следует из ее названия, выполняется обновление абсолютного значения времени в соответствии со значением текущего счетчика импульсов системного таймера. В функции `calc_global_load()` обновляются показатели средней загруженности системы.

После возврата из функции `do_timer()` запускается функция `update_process_times()`, в которой обновляются различные статистические данные в связи с окончанием очередного периода таймера. С помощью параметра `user_tick` функции сообщается, в каком режиме (пользовательском или ядра) произошло прерывание.

```
void update_process_times(int user_tick)
{
    struct task_struct *p = current;
    int cpu = smp_processor_id();

   /* На заметку: данный контекст прерывания от таймера должен
      быть также учтен. */
    account_process_tick(p, user_tick);
    run_local_timers();
    rcu_check_callbacks(cpu, user_tick);
    printk_tick();
    scheduler_tick();
    run_posix_cpu_timers(p);
}
```

Как вы уже видели в функции `tick_periodic()`, значение параметра `user_tick` определяется путем анализа системных регистров, как показано ниже.

```
update_process_times(user_mode(get_irq_regs()));
```

Реальное обновление статистики использования процессорного времени выполняется в функции `account_process_tick()`.

```
void account_process_tick(struct task_struct *p, int user_tick)
{
   cputime_t one_jiffy_scaled = cputime_to_scaled(cputime_one_jiffy);
   struct rq *rq = this_rq();

   if (user_tick)
       account_user_time(p, cputime_one_jiffy, one_jiffy_scaled);
   else if ((p != rq->idle) || (irq_count() != HARDIRQ_OFFSET))
       account_system_time(p, HARDIRQ_OFFSET, cputime_one_jiffy,
                           one_jiffy_scaled);
   else
       account_idle_time(cputime_one_jiffy);
}
```

Нетрудно заметить, что при таком подходе предполагается, что за период следования импульса системного таймера процесс *все время* выполнялся *в том же* режиме, в котором он был в момент прихода прерывания. На самом же деле в течение этого короткого промежутка времени процесс мог несколько раз переходить из пользовательского режима в режим ядра, и наоборот. Более того, это мог быть и не единственный процесс, который выполнялся в течение данного времени! К сожалению, без применения более сложной системы учета такой классический для систем Unix способ является лучшим из всех тех, которые реализованы в ядре. Кроме того, данный факт является одной из причин для увеличения частоты системного таймера.

Далее функция `run_local_timers()` помечает отложенные прерывания как готовые к выполнению (см. главу 8, “Нижняя половина обработчика и отложенные действия”), для запуска обработчиков всех таймеров, для которых закончился период времени ожидания. Таймеры будут рассмотрены ниже, в разделе “Таймеры”.

Наконец, функция `scheduler_tick()` уменьшает значение кванта времени для текущего выполняющегося процесса и по необходимости устанавливает флаг `need_resched`. Для SMP-машин в этой функции также, если нужно, выполняется балансировка очередей выполнения. Все это обсуждалось в главе 4, “Системный планировщик и диспетчеризация процессов”.

Функция `tick_periodic()` возвращается в аппаратно-зависимый обработчик прерывания, в котором выполняются все необходимые завершающие операции, освобождается блокировка `xtime_lock` и в конце концов управление возвращается прерванной программе.

Все описанное выше происходит каждые `1/HZ` секунд, т.е. 100 или 1000 раз в секунду на машине типа PC!

Абсолютное время

Текущее значение абсолютного времени (time of day, wall time — время дня) представляется с помощью структуры `timespec` и определяется в файле `kernel/time/timekeeping.c` следующим образом:

```
struct timespec xtime;
```

Структура данных timespec определена в файле <linux/time.h> в следующем виде:

```
struct timespec {
    __kernel_time_t tv_sec; /* в секундах */
    long tv_nsec;           /* в наносекундах */
};
```

В переменной xtime.tv_sec содержится количество секунд, которые прошли с 0 часов 1 января 1970 года. Время задано по всеобщему скоординированному времени (UTC, Universal Coordinated Time). Указанная дата называется началом *эпохи* (epoch) Unix. В большинстве Unix-подобных операционных систем отсчет времени ведется от момента начала эпохи Unix. В переменной xtime.v_nsec хранится количество наносекунд, которые прошли в последней секунде.

При чтении или записи переменной xtime требуется захватить блокировку xtime_lock. Это не обычная спин-блокировка, а последовательная блокировка, которая рассматривалась в главе 10, "Средства синхронизации ядра".

Для обновления значения переменной xtime необходимо захватить последовательную блокировку по записи, как показано ниже.

```
write_seqlock(&xtime_lock);

/* Обновление переменной xtime ... */

write_sequnlock(&xtime_lock);
```

При считывании значения переменной xtime используются функции read_seqbegin() и read_seqretry(), как показано ниже.

```
unsigned long seq;

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 они не были перезаписаны. Если во время выполнения цикла происходит прерывание от таймера и значение переменной xtime обновляется, возвращаемый номер последовательности будет неправильным и цикл повторится снова.

С точки зрения пользовательских приложений главной функцией для получения значения абсолютного времени является gettimeofday(), которая реализована как функция sys_gettimeofday() в файле kernel/time.c:

```
asmlinkage long sys_gettimeofday(struct  timeval *tv,
                                 struct timezone *tz)
{
    if (likely(tv)) {
        struct timeval ktv;
```

```
        do_gettimeofday(&ktv);
        if (copy_to_user(tv, &ktv, sizeof(ktv)))
              return -EFAULT;
    }

    if (unlikely(tz)) {
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
              return -EFAULT;
    }
    return 0;
}
```

Если из пользовательской программы передано ненулевое значение параметра `tv`, то вызывается аппаратно-зависимая функция `do_gettimeofday()`. В этой функции в основном выполняется цикл считывания значения переменной `xtime`, который был рассмотрен выше. Кроме того, если параметр `tz` не равен нулю, пользователю возвращается информация о часовом поясе (значение переменной `sys_tz`), соответствующем значению абсолютного времени операционной системы. Если при копировании в пространство пользователя значения абсолютного времени или часового пояса возникли ошибки, то функция возвращает значение `-EFAULT`. Если все прошло без ошибок, возвращается нулевое значение.

В ядре также реализована функция `time()`[6], однако системная функция `gettimeofday()` полностью перекрывает ее возможности. В библиотеке функций языка C также предусмотрены и другие функции, связанные с абсолютным временем, такие как `ftime()` и `ctime()`.

Системная функция `settimeofday()` позволяет задать значение абсолютного времени. Для ее выполнения процессу должно быть назначено право использования `CAP_SYS_TIME`.

Если не считать обновления переменной `xtime`, то значение абсолютного времени не так часто используется в ядре, как в пользовательских программах. Однако существует одно важное исключение — в коде управления файловыми системами значения абсолютного времени записываются в виде временных меток в индексные дескрипторы при выполнении различных файловых операций (чтения, записи и т.п.).

Таймеры

Таймеры (timers), или, как их еще иногда называют, *динамические таймеры*, или *таймеры ядра*, необходимы для управления ходом времени в ядре. В коде ядра часто необходимо отложить выполнение некоторых функций на более позднее время. В предыдущих главах были рассмотрены механизмы реализации нижних половин обработчиков прерываний, которые являются прекрасным примером отложенных действий. Здесь намеренно выбрано не очень четкое понятие "*позднее время*". Основной смысл нижних половин обработчиков прерываний — не задерживать выполнение некоторой работы на заданный промежуток времени, а *не выполнять работу прямо сейчас*. В связи с этим необходимо средство, позволяющее отложить выполнение работы на некоторый интервал времени. Если этот интервал времени не очень маленький, но и не очень большой, то для решения проблемы используются таймеры ядра.

[6] На некоторых аппаратных платформах функция `sys_time()` не реализована, а вместо этого она эмулируется библиотекой функций языка C на основании функции `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 сам таймер. Указанная вами функция будет выполнена, когда закончится интервал времени ожидания таймера. Таймеры *не являются* циклическими. Когда заканчивается интервал времени ожидания, таймер аннулируется. Это одна из причин, почему таймеры называют *динамическими*[7]. Таймеры постоянно создаются и аннулируются, на количество создаваемых таймеров не накладывается никаких ограничений. Таймеры используются практически во всех частях ядра.

Использование таймеров

Таймеры представляются в ядре с помощью структуры `timer_list`, которая определена в файле `<linux/timer.h>` следующим образом:

```
struct timer_list {
   struct list_head entry; /* Элемент связанного списка таймеров */
   unsigned long expires;  /* Время окончания срока ожидания
                              в импульсах системного таймера (jiffies) */
   void (*function)(unsigned long); /* Функция-обработчик таймера */
   unsigned long data;              /* Единственный аргумент,
                                       передающийся обработчику */
   struct tvec_t_base_s *base;      /* Внутренние данные таймера,
                                       не трогать! */
};
```

К счастью, использование таймеров не требует глубокого понимания назначения полей этой структуры. Однако крайне не рекомендуется использовать поля этой структуры не по назначению, чтобы сохранить совместимость с возможными будущими изменениями кода. Для упрощения работы с таймерами в ядре предусмотрено специальное семейство интерфейсов. Все необходимые определения находятся в файле `<linux/timer.h>`. Бо́льшая часть кода реализации находится в файле `kernel/timer.c`.

Перед использованием таймера его нужно объявить, как показано ниже.

```
struct timer_list my_timer;
```

Далее должны быть проинициализированы поля структуры, которые предназначены для внутреннего использования. Это делается с помощью специальной вспомогательной функции. Ее нужно вызвать *перед* вызовом *любых* других функций, которые работают с таймером.

```
init_timer(&my_timer);
```

После этого необходимо заполнить все остальные поля структуры, например, следующим образом:

```
my_timer.expires = jiffies + delay; /* Время срабатывания таймера, выраженное
                                       в импульсах системного таймера */
my_timer.data = 0;                  /* В функцию-обработчик будет передан
                                       параметр, равный нулю */
my_timer.function = my_function;    /* Функция, которая будет выполнена после
                                       истечения интервала времени таймера */
```

[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.

В поле my_timer.expires указывается время срабатывания таймера, выраженное в импульсах системного таймера. Как только текущее значение переменной jiffies становится бо́льшим 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 приведенному ниже прототипу.

```
void my_timer_function(unsigned long data);
```

Благодаря параметру data можно зарегистрировать несколько таймеров с одним и тем же обработчиком и различать таймеры по значению этого параметра. Если в аргументе нет необходимости, то можно просто указать нулевое (или любое другое) значение.

Последняя операция — это запуск таймера.

```
add_timer(&my_timer);
```

И, вуаля, таймер запустился и работает! Следует обратить внимание на важность значения, занесенного в поле expired. Ядро запускает функцию-обработчик, когда текущее значение счетчика импульсов системного таймера *больше*, чем указанное значение времени срабатывания таймера, *или равно* ему. Хотя в ядре и гарантируется, что функция-обработчик таймера не запустится раньше, чем истечет время ожидания таймера, тем не менее возможны задержки с запуском этой функции. Как правило, функции-обработчики таймеров запускаются в момент времени, близкий к моменту времени срабатывания таймера, однако их запуск может быть также отложен и до прихода 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 запущен. Для этого в ядре предусмотрена функция mod_timer().

```
mod_timer(&my_timer, jiffies + new_delay); /* Новое время срабатывания */
```

Функцию mod_timer() можно также использовать в случае, если таймер уже проинициализирован, но еще не запущен. Если таймер не активен, то эта функция mod_timer() запускает его. Функция возвращает значение 0, если таймер был незапущенный, и значение 1, если таймер был уже запущен. В любом случае перед возвратом из функции mod_timer() таймер будет запущен и настроен на новое время срабатывания.

Если нужно остановить таймер до момента его срабатывания, используется функция del_timer().

```
del_timer(&my_timer);
```

Эту функцию можно использовать как с запущенными, так и с незапущенными таймерами. Если таймер уже остановлен, то функция возвращает значение 0, в противном случае возвращается значение 1. Обратите внимание на то, что нет необходимости вызывать эту функцию для таймеров, интервал ожидания которых уже истек, так как они останавливаются автоматически.

При удалении таймеров потенциально может возникнуть конфликтная ситуация из-за доступа к ресурсу. При возврате из функции del_timer() гарантируется только, что указанный таймер остановлен, т.е. его функция-обработчик не будет запущена в будущем. Тем не менее на многопроцессорной машине функция-обработчик таймера в этот момент уже может выполняться на другом процессоре. Для того чтобы остановить таймер и подождать, пока завершится его функция-обработчик, которая потенциально может уже выполняться, используется функция del_timer_sync():

```
del_timer_sync(&my_timer);
```

В отличие от функции `del_timer()`, функция `del_timer_sync()` не может вызываться из контекста прерывания.

Конфликты из-за доступа к ресурсам при использовании таймеров

Поскольку функции-обработчики таймеров запускаются асинхронно по отношению к выполняемому в данный момент коду, потенциально могут возникнуть несколько типов конфликтов из-за доступа к ресурсам. Во-первых, ни при каких условиях нельзя использовать приведенный ниже код как замену функции `mod_timer()`, поскольку на многопроцессорных машинах он работает неустойчиво.

```
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 функция-обработчик таймера в данный момент не выполняется. Собственно по этой причине в приведенном выше фрагменте кода функция `del_timer()` вызвана первой! Представьте себе, что произойдет, если после удаления таймера в коде будет освобождена память или ресурсы, которые используются в функции-обработчике таймера, будут задействованы для других целей. Поэтому синхронизированная версия этой функции более предпочтительна.

Наконец, необходимо гарантировать защиту всех совместно используемых данных, к которым обращается функция-обработчик таймера. Эта функция запускается ядром асинхронно по отношению к другому коду. Совместно используемые данные должны быть защищены так, как рассматривалось в главе 9, “Общие сведения о синхронизации кода ядра”, и главе 10, “Средства синхронизации ядра”.

Реализация таймеров

В ядре функции-обработчики таймеров запускаются в виде отложенного прерывания (softirqs) после завершения обработки прерывания от таймера. Все это происходит в контексте нижней половины обработчика прерывания. В обработчике прерывания от таймера вызывается функция `update_process_times()`, которая в свою очередь вызывает функцию `run_local_timers()`, имеющую такой вид:

```
void run_local_timers(void)
{
    hrtimer_run_queues();
    raise_softirq(TIMER_SOFTIRQ); /* Активизировать отложенное
                                     прерывание от таймера */
    softlockup_tick();
}
```

Отложенное прерывание с номером `TIMER_SOFTIRQ` обрабатывается функцией `run_timer_softirq()`. В этой функции на локальном процессоре запускаются обработчики всех таймеров, для которых истек период времени ожидания (если такие есть).

Структуры, представляющие таймеры, хранятся в виде связанного списка. Однако было бы неразумным просматривать весь список в ядре в поисках таймеров, для которых истекло время ожидания, или поддерживать этот список в отсортированном состоянии

на основании времени срабатывания таймеров. В последнем случае вставка и удаление таймеров заняли бы много времени. Поэтому в ядре все таймеры разбиваются на пять 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 запуск функций-обработчиков таймеров, поиск таймеров, у которых истек период ожидания, не займет много времени. Следовательно, код управления таймерами должен быть очень эффективен.

Задержка выполнения

Часто в коде ядра (особенно драйверов) необходимо на некоторое время задерживать выполнение программы без использования таймеров или привлечения механизма нижних половин. Обычно это необходимо для того, чтобы дать аппаратному обеспечению время на завершение выполнения операции. Такой интервал времени, как правило, достаточно короткий. Например, в спецификации платы сетевого интерфейса может быть указано время переключения режима работы Ethernet-контроллера, равное 2 мкс. Другими словами, после установки желаемой скорости передачи данных драйвер должен подождать хотя бы 2 мкс перед тем, как продолжить работу.

В зависимости от величины задержки в ядре предусмотрено несколько способов решения этой проблемы. Все они имеют различные свойства. Некоторые решения на время задержки загружают процессор, не давая возможности выполнять другую, более полезную работу. Другие решения не загружают процессор, но и не гарантируют, что код возобновит выполнение точно в нужный момент времени[8].

Задержка с помощью цикла

Самым простым, хотя и не самым оптимальным решением для создания задержек в выполняемом коде, является использование *циклов ожидания* (busy waiting), или холостых циклов (busy loop). Такой метод можно использовать, если длительность времени задержки кратна периоду системного таймера или точность задержки не очень важна.

Идея проста — в программе нужно запустить холостой бесконечный цикл, пока не будет получено необходимое количество импульсов системного таймера, как показано в следующем примере:

```
unsigned long timeout = jiffies + 10; /* 10 импульсов таймера */

while (time_before(jiffies, timeout))
  ;
```

Этот цикл будет выполняться до тех пор, пока значение переменной `jiffies` не станет больше, чем значение переменной `timeout`, 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 мс. Аналогично можно поступить следующим образом:

[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.

```
unsigned long delay = jiffies + 2*HZ; /* две секунды */

while (time_before(jiffies, delay))
  ;
```

В этом случае цикл будет выполняться, пока не поступит `2*HZ` импульсов системного таймера, что всегда равно 2 секундам, независимо от частоты системного таймера.

Такой подход не очень хорош с точки зрения всей системы. Пока код ожидает, процессор загружен выполнением холостого цикла, не делая при этом никакой полезной работы! Поэтому к такому простому методу нужно прибегать по возможности реже. Мы его описали здесь лишь потому, что он является простым и понятным способом осуществления задержки. Однако вы можете с ним столкнуться, анализируя чей-нибудь не очень хороший код.

Существует значительно лучшее решение: во время ожидания нужно явно указать планировщику, чтобы тот перепланировал ваш процесс. Тогда процессор сможет выполнить другую полезную работу.

```
unsigned long delay = jiffies + 5*HZ;

while (time_before(jiffies, delay))
   cond_resched();
```

При вызове функции `cond_resched()` ваш процесс будет вытеснен и запущен другой процесс только в случае, если установлен флаг `need_resched`. Другими словами, данное решение позволяет запустить планировщик, но только в случае, когда есть более важное задание, которое нужно выполнить. Обратите внимание: 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 того, что указанные циклы будут корректно работать? Как правило, компилятор C загружает значение указанной переменной в регистр перед началом цикла и больше к памяти не обращается. Следовательно, в обычной ситуации нет никакой гарантии, что переменная `jiffies` будет считываться из памяти на каждой итерации цикла. Нам же необходимо совсем обратное! Значение переменной `jiffies` *должно* считываться на *каждой* итерации цикла, поскольку это значение увеличивается в другом месте — в обработчике прерывания от таймера. Именно поэтому данная переменная определена в файле `<linux/jiffies.h>` с атрибутом `volatile`. Ключевое слово `volatile` указывает компилятору, что значение переменной необходимо считывать в цикле из оперативной памяти и никогда не использовать копию, хранящуюся в регистре процессора. Это гарантирует, что указанный цикл будет работать так, как и ожидалось.

Короткие задержки

Иногда в коде ядра (опять же, как правило, в драйверах) необходимо создать задержки на очень короткие интервалы времени (короче, чем период системного таймера), причем интервал должен отслеживаться с достаточно высокой точностью. Довольно часто в программе необходимо синхронизировать работу с аппаратным обеспечением, для которого оговорено некоторое минимальное время, необходимое для завершения текущей операции. Как правило, это время бывает меньше одной миллисекунды. В случае таких

малых значений времени невозможно использовать задержки на основании переменной `jiffies`, как показано в предыдущем примере. При частоте, равной 100 Гц, значение периода системного таймера достаточно большое — 10 мс! И даже при частоте 1000 Гц период системного таймера равен всего одной миллисекунде. Ясно, что необходимо искать другое решение, обеспечивающее более короткие и точные величины задержек.

Для этой цели в ядре предусмотрены три функции, обеспечивающие задержки на уровне микро-, нано- и миллисекунд. Они определены в файлах `<linux/delay.h>` и `<asm/delay.h>`, и в этих функциях не используется переменная `jiffies`.

```
void udelay(unsigned long usecs)
void ndelay(unsigned long nsecs)
void mdelay(unsigned long msecs)
```

Первая функция позволяет задержать выполнение программы на указанное количество *микросекунд* с использованием холостого цикла. Последняя функция задерживает выполнение программы на указанное количество *миллисекунд*. Напомним, что одна секунда равна 1000 миллисекундам, что эквивалентно 1 000 000 *микросекунд*. Использовать эти функции очень просто.

```
udelay(150); /* Задержка на 150 мкс */
```

Функция `udelay()` выполнена на основе холостого цикла, количество итераций которого точно вычисляется на основании указанного периода времени. Функция `mdelay()` реализована на основе функции `udelay()`.Так как в ядре известно, сколько циклов может выполнить процессор в одну секунду (смотрите ниже врезку по поводу характеристики BogoMIPS), в функции `udelay()` это значение просто масштабируется 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 в функциях `udelay()` и `mdelay()`. Название этого параметра состоит из двух частей: *bogus* (фиктивный) и *MIPS* (million of instructions per second — миллион инструкций в секунду). Наверняка при загрузке системы вы уже сталкивались с сообщением, похожим на приведенное ниже (данное сообщение соответствует процессору Intel Xeon серии 7300 с частотой 2,4 ГГц).

```
Detected 2400.131 MHz processor.
Calibrating delay loop... 4799.56 BogoMIPS
```

Значение параметра BogoMIPS — это просто количество холостых циклов, которые процессор может выполнить за заданный период времени. По сути, эта характеристика показывает, насколько быстро процессор может ничего не делать! Это значение хранится в переменной `loops_per_jiffy`, и его можно прочитать из файла `/proc/cpuinfo`. Оно используется в функциях `udelay()` и `mdelay()` для определения количества итераций холостого цикла, которые необходимо выполнить, чтобы обеспечить необходимую задержку (причем достаточно точно!).

В ядре значение переменной `loops_per_jiffy` вычисляется с помощью функции `calibrate_delay()` при начальной загрузке системы, реализация которой описана в файле `init/main.c`.

Функция `udelay()` должна вызываться только для создания небольших задержек, поскольку при большом времени задержки на быстрой машине может возникнуть переполнение счетчика цикла. Общее правило: по возможности не используйте функцию

udelay() для задержек больше одной миллисекунды. Для более продолжительных задержек предназначена функция mdelay(). Так же как и другие методы задержки выполнения, основанные на холостых циклах, эти функции (особенно функция mdelay(), так как она дает длительные задержки) должны использоваться, только если это абсолютно необходимо. Не забывайте, что не стоит использовать циклы задержек, когда удерживается блокировка или запрещены прерывания, потому что это очень сильно влияет на производительность и время реакции системы. Однако если нужно обеспечить точное время задержки, то эти функции — наилучшее решение. Как правило, они используются для создания очень коротких задержек, измеряемых в микросекундах.

Функция schedule_timeout()

Существует более оптимальный метод создания задержек при выполнении кода — использование функции schedule_timeout(). Вызов этой функции переводит текущий процесс в состояние ожидания (sleep) как минимум 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 несложно.

```
/* Установим прерываемое состояние ожидания задачи */
set_current_state(TASK_INTERRUPTIBLE);

/* Переходим в приостановленное состояние на s секунд */
schedule_timeout(s * HZ);
```

Единственный параметр функции — это желаемое время ожидания, выраженное в количестве импульсов системного таймера. В этом примере задача переводится в прерываемое состояние ожидания на s секунд. Поскольку задача находится в состоянии TASK_INTERRUPTIBLE, она может быть возвращена к выполнению раньше времени при получении сигнала. Если в коде не нужно обрабатывать внешние сигналы, то можно использовать состояние TASK_UNINTERRUPTIBLE. Перед вызовом функции schedule_timeout() задача должна находиться в одном из этих двух состояний, иначе она не будет переведена в состояние ожидания.

Поскольку в функции schedule_timeout() вызывается планировщик, код, который ее вызывает, должен быть способен перейти в состояние ожидания. О том, что такое неделимые операции и как выполняется переход в состояние ожидания, мы говорили в главе 9, “Общие сведения о синхронизации кода ядра”, и главе 10, “Средства синхронизации ядра”. Если говорить коротко, то эту функцию нужно вызывать в контексте процесса и не удерживать при этом блокировку.

Реализация функции schedule_timeout()

Функция schedule_timeout() достаточно проста. В ней используются таймеры ядра. Рассмотрим ее 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;
}
```

В этой функции создается таймер со своеобразным именем `timer` и устанавливается время его срабатывания через `timeout` импульсов системного таймера. В качестве обработчика таймера устанавливается функция `process_timeout()`, которая вызывается, когда истекает период времени таймера. После этого таймер запускается и вызывается функция `schedule()`. Поскольку предполагается, что текущий процесс должен находиться в состоянии `TASK_INTERRUPTIBLE` или `TASK_UNINTERRUPTIBLE`, планировщик *не* будет продолжать его выполнение, а выберет для запуска другой процесс.

После срабатывания таймера вызывается функция `process_timeout()`, которая имеет следующий вид:

```
void process_timeout(unsigned long data)
{
    wake_up_process((task_t *) data);
}
```

В данной функции устанавливается состояние задачи `TASK_RUNNING`, и она помещается в очередь выполнения.

Как только задача будет снова запланирована на выполнение, управление возвратится в функцию `schedule_timeout()` (сразу после вызова функции `schedule()`). Если после получения сигнала задача возвращается к выполнению преждевременно, то таймер удаляется и функция возвращает значение интервала времени, которое она должна еще находиться в состоянии ожидания.

Код оператора `switch()` служит для обработки специальных случаев и не является основной частью функции. Проверка на значение `MAX_SCHEDULE_TIMEOUT` позволяет задаче находиться в состоянии ожидания неопределенное время. В этом случае таймер не запускается (поскольку нет ограничений на интервал времени ожидания) и сразу же

вызывается планировщик. Если вы применяете такую возможность, то должны предусмотреть какой-то способ вернуть задание в состояние выполнения!

Ожидание события в течение заданного интервала времени

В главе 4, "Системный планировщик и диспетчеризация процессов", рассматривалось, как код ядра, находящийся в контексте процесса, может поместить сам себя в очередь ожидания до наступления определенного события, а затем вызвать планировщик, который выберет новое задание для выполнения. Если в другом месте ядра произойдет указанное событие, то для всех ожидающих в очереди задач вызывается функция `wake_up()`. В результате эти задачи возвращаются к выполнению и могут продолжить свою работу.

Иногда требуется ожидать наступления некоторого события только в течение определенного интервала времени. Иначе говоря, задача должна быть выведена из состояния ожидания либо при наступлении нужного события, либо после исчерпания интервала времени, в зависимости от того, что наступит раньше. В таких случаях после перемещения себя в очередь ожидания вместо функции `schedule()` нужно вызвать функцию `schedule_timeout()`. Тогда задача будет возвращена к выполнению, когда произойдет желаемое событие или пройдет указанный интервал времени. В коде обязательно нужно проверить, *почему* именно процесс был возвращен к выполнению. Это может произойти либо после наступления нужного события, исчерпания интервала времени, либо в результате получения сигнала. После этого необходимо соответствующим образом продолжить выполнение.

Резюме

В этой главе были рассмотрены понятия, связанные с представлением времени в ядре, а также способы отслеживания абсолютного и относительного хода времени. Были показаны отличия абсолютного и относительного времени, а также периодических и разовых событий. Далее были рассмотрены прерывания от таймера, представление времени в количестве импульсов таймера, константа `HZ` и переменная `jiffies`.

Затем речь шла 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 реализации временных задержек.

При написании большей части кода ядра вам потребуются знания о представлении времени в ядре и его отслеживании. С очень большой вероятностью, особенно при разработке драйверов, вы столкнетесь с таймерами ядра. Материал этой главы принесет вам практическую пользу и не будет простой тратой времени.

12
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 легко и без проблем выделить память. Например, в режиме ядра не так-то просто обрабатывать ошибки, связанные с выделением памяти, кроме того, часто нельзя переходить в состояние ожидания. Из-за этих ограничений, а также из-за необходимости, чтобы процесс выделения памяти был быстрым, работа с памятью в режиме ядра становится более сложной, чем в режиме пользователя. Конечно, нельзя сказать, что выделение памяти в ядре — очень сложная процедура, просто это делается несколько по-другому.

В этой главе рассматриваются средства, предназначенные для выделения памяти внутри ядра. Перед изучением интерфейсов, предназначенных для выделения памяти, необходимо рассмотреть сам процесс управления памятью в ядре.

Страничная организация памяти

Основными единицами управления памятью в ядре являются страницы. Хотя наименьшими единицами памяти, которую может адресовать процессор, являются байт и машинное слово, модуль управления памятью (MMU, Memory Management Unit) — аппаратное устройство, которое отвечает за обращение к памяти и выполняет преобразование виртуальных адресов в физические, — обычно работает со страницами. Таким образом, модуль MMU управляет таблицами страниц на уровне страничной детализации (отсюда и название). С точки зрения организации виртуальной памяти страница является наименьшим элементом, с которым приходится иметь дело.

Как будет показано в главе 19, “Переносимость”, размеры страниц для каждой аппаратной платформы могут быть разными. Более того, на многих аппаратных платформах поддерживается несколько размеров страниц. В большинстве 32-разрядных аппаратных платформ размер страницы равен 4 Кбайт, а в большинстве 64-разрядных платформ — 8 Кбайт. Это означает, что на машине, оснащенной одним гигабайтом физической памяти, размер страницы которой равен 4 Кбайт, существует 262 144 отдельные страницы памяти.

В ядре *каждая* физическая страница памяти представляется в виде структуры page, которая определена в файле <linux/mm_types.h>. В приведенном ниже коде я несколько упростил это определение, исключив два сбивающих с толку *объединения* (union), которые никак не проясняют основные моменты.

```
struct page {
   unsigned long         flags;
   atomic_t              _count;
   atomic_t              _mapcount;
   unsigned long         private;
   struct address_space  *mapping;
   pgoff_t               index;
   struct list_head      lru;
   void                  *virtual;
};
```

Сначала рассмотрим самые важные поля этой структуры. В поле flags хранится информация о состоянии текущей страницы. В частности, эти флаги отображают, изменилось ли содержимое страницы с момента ее последнего вытеснения на диск и фиксирована ли страница в памяти или нет. Значение каждого флага представлено одним битом, поэтому всего может быть до 32 разных флагов. Значения флагов определены в файле <linux/page-flags.h>.

В поле _count хранится счетчик использования страницы, отражающий количество ссылок в системе на эту страницу. Как только значение этого счетчика становится отрицательным (-1), это означает, что данная страница больше нигде не используется и ее можно выделить при следующем запросе на память. Значение этого поля не должно проверяться в коде ядра напрямую. Вместо этого следует использовать функцию page_count(), которой передается единственный параметр — указатель на структуру page. Хотя в случае незанятой страницы памяти значение счетчика _count отрицательно и равно -1 (во внутреннем представлении), функция page_count() возвращает значение нуль для незанятой страницы памяти и положительное значение — для страницы, которая используется в данный момент. Страница может использоваться в страничном кеше (в таком случае поле mapping указывает на объект типа address_space, который связан с данной страницей памяти), в качестве приватных данных, на которые в таком случае указывает поле private, или отображаться в таблицу страниц процесса.

В поле virtual хранится виртуальный адрес страницы. Обычно он соответствует адресу данной страницы в виртуальной памяти ядра. Некоторая часть памяти (называемая областью верхней памяти (high memory) постоянно не отображается в адресное пространство ядра. В этом случае значение данного поля равно NULL, а страница по мере необходимости отображается динамически. Верхняя память будет рассмотрена в одном из следующих разделов.

Самый важный момент, который необходимо понять, — 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 на физической странице, продолжают существовать, то это не значит, что они будут всегда привязаны к одной и той же физической странице памяти и соответственно к одной и той же структуре page, например, в результате вытеснения страницы (swapping) или по другим причинам. В ядре эта структура данных используется для описания всего того, что содержится в *данный* момент на связанной с ней странице физической памяти. Структура page предназначена для описания именно фрагмента физической памяти, а не тех данных, которые в нем находятся.

Рассматриваемая структура данных используется в ядре для учета всех страниц физической памяти в системе и определения, какая из страниц свободна (т.е. соответствующая область физической памяти никому не выделена). И если конкретная страница занята, то ядро должно иметь информацию о ее владельце. Страница может быть выделена пользовательскому процессу, в ней могут находиться данные из динамически выделяемой памяти в пространстве ядра, статический код ядра, страничный кеш (page cache) и т.п.

Разработчики часто удивляются тому, что для каждой физической страницы в системе создается экземпляр данной структуры. При этом они думают: "Как много для этого теряется памяти!" Давайте посмотрим, насколько плохо (или хорошо) расходуется адресное пространство для хранения информации о страницах памяти. Будем считать, что размер структуры `page` равен 40 байт, а система оснащена 4 Гбайт физической памяти и разбита на страницы по 8 Кбайт. Тогда в системе должно существовать 524 288 страниц и соответственно структур `page`. Все вместе структуры `page` будут занимать 20 Мбайт памяти. Возможно, в абсолютных величинах это и много, но если посчитать все в процентах относительно доступного размера памяти, равного 4 Гбайт, то получается, что для учета всех страниц памяти в системе требуется заплатить не так уж и много.

Зоны

В связи с ограничениями, наложенными используемым аппаратным обеспечением, в ядре нельзя считать все страницы памяти равноценными. Часть страниц из-за значений их физических адресов памяти нельзя использовать для решения некоторых типов задач. По этой причине в ядре вся физическая память разделена на *зоны*. Они используются в ядре для группировки физических страниц памяти, обладающих одинаковыми свойствами. В частности, в операционной системе Linux должны учитываться перечисленные ниже особенности 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 32-разрядных аппаратных платформах адресуемый объем физической памяти может превышать адресуемый объем виртуальной памяти. Следовательно, часть памяти не может постоянно отображаться в адресное пространство ядра.

В связи с приведенными выше ограничениями в операционной системе Linux выделяют четыре основных зоны памяти.

- `ZONE_DMA`. Содержит страницы памяти, к которым может обращаться контроллер DMA.
- `ZONE_DMA32`. Как и в `ZONE_DMA`, в этой зоне находятся страницы, доступные для контроллера DMA. В отличие от первой зоны обращаться к этим страницам могут только 32-разрядные устройства. На некоторых аппаратных платформах эта зона занимает больший объем физической памяти.
- `ZONE_NORMAL`. Содержит страницы памяти, которые отображаются в адресные пространства обычным образом.
- `ZONE_HIGHMEM`. Соответствует "верхней памяти" компьютера и состоит из страниц, которые не могут постоянно отображаться в адресное пространство ядра.

Эти и две другие менее значимые зоны определяются в заголовочном файле <linux/mmzone.h>.

Разделение памяти на зоны и ее реальное использование зависят от аппаратной платформы. Например, для некоторых аппаратных платформ прямой доступ к памяти может без проблем выполняться по любому адресу. Для таких платформ зона ZONE_DMA является пустой, и для всех типов выделения памяти используется зона ZONE_NORMAL, независимо от ее использования. В качестве контрпримера можно привести платформу x86, устройства ISA[1] которой не могут выполнять операции прямого доступа к памяти в полном 32-разрядном пространстве адресов, поскольку они могут обращаться только к первым 16 Мбайт физической памяти. Следовательно, зона ZONE_DMA для платформы x86 содержит только страницы памяти с физическими адресами в диапазоне 0–16 Мбайт.

Аналогичным образом используется и зона ZONE_HIGHMEM. Только от конкретной аппаратной платформы зависит, какая часть физических адресов памяти может отображаться в адресное пространство ядра. Например, для платформы x86 к зоне ZONE_HIGHMEM относится вся физическая память, адреса которой лежат выше отметки 896 Мбайт. На остальных аппаратных платформах зона ZONE_HIGHMEM пуста, так как может непосредственно отображаться вся память. Память, которая содержится в зоне ZONE_HIGHMEM, называется *верхней памятью*[2] (*high memory*). Вся остальная память в системе называется *нижней памятью* (*low memory*).

В зоне ZONE_NORMAL обычно содержится все, что не попало в две предыдущие зоны памяти. Например, для аппаратной платформы x86 в нее попадает вся физическая память в диапазоне от 16 до 896 Мбайт. Для других, более удачных аппаратных платформ в зону ZONE_NORMAL попадает доступная физическая память. В табл. 12.1 приведен список зон для аппаратной платформы x86 и занимаемый ими диапазон адресов памяти.

Таблица 12.1. Зоны памяти для аппаратной платформы x86-32

Зона	Описание	Физическая память
ZONE_DMA	Страницы памяти, к которым имеет доступ контроллер прямого доступа в память	< 16 Мбайт
ZONE_NORMAL	Страницы памяти, адресуемые обычным образом	16–896 Мбайт
ZONE_HIGHMEM	Страницы, динамически отображаемые в память	> 896 Мбайт

Благодаря разбиению страниц памяти на зоны в операционной системе Linux создаются пулы памяти разного типа, из которых по мере необходимости происходит выделение памяти. Например, пул зоны ZONE_DMA дает возможность ядру удовлетворить запрос на выделение памяти, которая необходима для операций DMA. И если такая память понадобится, то ядро может просто выделить необходимое количество страниц из зоны ZONE_DMA. Следует обратить внимание на то, что зоны не связаны с каким-либо аппаратным обеспечением. Это просто логические группы, используемые в ядре и облегчающие учет страниц памяти.

[1] Некоторые ущербные PCI-устройства также могут выполнять прямой доступ к памяти только в 24-битовом адресном пространстве.

[2] Это не имеет ничего общего с верхней памятью операционной системы DOS, появление которой было связано с ограничениями, связанными с режимом адресации процессора 8086 и ее эмуляции в реальном режиме платформы x86.

Хотя в некоторых запросах на выделение памяти могут требоваться страницы из определенной зоны, чаще всего память выделяется из разных зон. Например, при выполнении операций с прямым доступом в память требуются страницы из зоны ZONE_DMA, тогда как при обычном выделении памяти страницы могут браться как из зоны ZONE_NORMAL, так и из зоны ZONE_DMA, но не из обеих сразу. Выделенный участок памяти не должен пересекать границы зон. Разумеется, при выполнении запросов на обычную память ядро будет стараться выделять страницы из зоны ZONE_NORMAL, чтобы сохранить страницы в зоне ZONE_DMA для случая, когда эта память действительно нужна. Если же наступает критический момент (скажем, становится мало физической памяти), то ядро может обратиться к любой доступной и подходящей зоне.

Перечисленные выше зоны определены далеко не на всех аппаратных платформах. Например, в 64-разрядных системах типа Intel x86-64 имеется полный доступ ко всему 64-разрядному адресному пространству физической памяти. Поэтому для них зона ZONE_HIGHMEM не определена (поскольку в ней нет особого смысла), а вся физическая память распределена между зонами ZONE_DMA и ZONE_NORMAL.

Каждая из зон представляется в виде структуры zone, которая описана в файле <linux/mmzone.h>.

```
struct zone {
     unsigned long          watermark[NR_WMARK];
     unsigned long          lowmem_reserve[MAX_NR_ZONES];
     struct per_cpu_pageset pageset[NR_CPUS];
     spinlock_t             lock;
     struct                 free_area free_area[MAX_ORDER]
     spinlock_t             lru_lock;
     struct zone_lru {
                      struct list_head list;
                      unsigned long    nr_saved_scan;
                    } lru[NR_LRU_LISTS];
     struct zone_reclaim_stat reclaim_stat;
     unsigned                 long pages_scanned;
     unsigned                 long flags;
     atomic_long_t            vm_stat[NR_VM_ZONE_STAT_ITEMS];
     int                      prev_priority;
     unsigned int             inactive_ratio;
     wait_queue_head_t        *wait_table;
     unsigned long            wait_table_hash_nr_entries;
     unsigned long            wait_table_bits;
     struct pglist_data       *zone_pgdat;
     unsigned                 long zone_start_pfn;
     unsigned long            spanned_pages;
     unsigned long            present_pages;
     const char               *name;
};
```

Эта структура довольно большая, но в системе существует всего три зоны и соответственно три такие структуры. Рассмотрим самые важные поля данной структуры.

В поле lock хранится спин-блокировка, которая защищает структуру от одновременного доступа со стороны других процессов. Обратите внимание: она защищает только структуру, а не все страницы, принадлежащие зоне. Для защиты отдельных страниц нет специальных блокировок, хотя в некоторых частях кода могут блокироваться данные, оказавшиеся на указанных страницах.

В массиве watermark хранятся значения минимального, нижнего и верхнего уровней для текущей зоны. Они используются в ядре для оценки параметров расхода памяти

в пределах зоны. При изменении уровней по отношению к свободной памяти изменяется и значение оценки.

Как можно догадаться, в поле name хранится указатель на строку символов, оканчивающуюся нулем, в которой содержится имя соответствующей зоны. Ядро инициализирует указанное поле при загрузке системы с помощью кода, который описан в файле mm/page_alloc.c. Трем зонам присваиваются имена "DMA", "Normal" и "HighMem".

Выделение страниц памяти

Теперь, имея некоторое понятие о том, как ядро управляет памятью с помощью страниц, зон и тому подобного, рассмотрим интерфейсы ядра, посредством которых можно выделить и освободить память в ядре.

В ядре предусмотрен один низкоуровневый механизм 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>. Прототип основной функции выделения памяти приведен ниже.

```
struct page * alloc_pages(gfp_t gfp_mask, unsigned int order)
```

Данная функция позволяет выделить 2^{order} (т.е. 1 << order) смежных страницы (один непрерывный участок) физической памяти и возвращает указатель на структуру page, которая соответствует первой выделенной странице памяти. В случае ошибки возвращается значение NULL. Тип данных gfp_t и параметр gfp_mask будут рассмотрены чуть позже. Чтобы определить логический адрес полученной страницы памяти, используется функция

```
void * page_address(struct page *page)
```

Эта функция возвращает указатель, содержащий логический адрес, которому в данный момент соответствует начало указанной страницы физической памяти. Если нет необходимости в соответствующей структуре page, то можно использовать следующую функцию:

```
unsigned long __get_free_pages(gfp_t 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, другие страницы просто следуют за первой.

Если нужна всего одна страница памяти, то для этой цели определены приведенные ниже функции-оболочки, которые позволяют уменьшить количество работы по набору кода программы.

```
struct page * alloc_page(gfp_t gfp_mask)
unsigned long __get_free_page(gfp_t gfp_mask)
```

Эти функции работают так же, как и ранее описанные, но для них в качестве параметра order передается нуль (2^0, т.е. одна страница памяти).

Выделение обнуленных страниц памяти

Если вы хотите, чтобы выделяемая страница памяти была заполнена нулями, то используйте приведенную ниже 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, т.е. все биты всех байтов страницы будут сброшены. Такая возможность пригодится при выделении страниц памяти пользовательским приложениям. Дело в том, что случайный "мусор", находящийся на выделенной странице памяти, может оказаться не таким уж и безобидным и может содержать некоторые (например, секретные) данные. Поэтому с целью повышения безопасности системы следует очищать все блоки памяти, которые выделяются пользовательскому приложению. В табл. 12.2 приведен список всех низкоуровневых средств выделения страниц памяти.

Таблица 12.2. Низкоуровневые средства выделения памяти

Функция	Описание
`alloc_page(gfp_mask)`	Выделяет одну страницу памяти и возвращает указатель на нее
`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, содержащий ее логический адрес
`__get_free_pages(gfp_mask, order)`	Выделяет 2^{order} страниц памяти и возвращает указатель, содержащий логический адрес первой страницы
`get_zeroed_page(gfp_mask)`	Выделяет одну страницу памяти, обнуляет ее содержимое и возвращает указатель, содержащий ее логический адрес

Освобождение страниц памяти

Для освобождения страниц памяти, которые больше не нужны, можно использовать перечисленные ниже 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)
```

Будьте внимательны и освобождайте только те страницы памяти, которые были вам выделены. Передача неправильного значения параметра `page`, `addr` или `order` может привести к потере данных. Не забывайте, что ядро полностью доверяет своему коду. В отличие от пользовательской программы оно с удовольствием зависнет, если вы сделаете что-то не так.

Рассмотрим пример, в котором мы собираемся выделить 8 страниц памяти.

```
unsigned long page;

page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
    /* Недостаточно памяти: эту ошибку нужно обработать! */
    return -ENOMEM;
}
/* В переменной 'page' сейчас хранится логический адрес первой
   из восьми выделенных непрерывных страниц памяти ... */
```

А теперь освободим эти восемь страниц, после того как они стали нам не нужны.

```
free_pages(page, 3);

/*
 * Теперь выделенные нам восемь страниц памяти освобождены.
```

```
 * С этого момента нам нельзя обращаться к памяти по адресу,
 * хранящемуся в переменной 'page'
 */
```

Константа GFP_KERNEL, которая передается в качестве параметра в функцию __get_free_pages(), является примером флага gfp_mask, который скоро мы рассмотрим подробно.

Обратите внимание на проверку ошибок после вызова функции __get_free_pages(). В силу сложившихся обстоятельств иногда ядро *не может* выделить запрошенный объем памяти. Поэтому в коде обязательно нужно учесть эту ситуацию, проверить и по мере необходимости обработать соответствующую ошибку. Иногда такая ошибка приводит к тому, что в коде придется отменить все выполненные ранее операции. В связи с этим часто имеет смысл выделять память в самом начале процедуры, чтобы потом можно было упростить обработку ошибок. В противном случае после попытки выделения памяти отмена ранее выполненных действий может оказаться сложной или даже невозможной (если операция отмены в свою очередь потребует выделения памяти).

Описанные выше низкоуровневые функции выделения памяти полезны, когда нужно выделить участок памяти, кратный размеру страницы, особенно если нужна одна или всего несколько смежных страниц. В остальных случаях, когда необходимо выделить заданное количество байтов памяти, в ядре предусмотрена функция kmalloc().

Функция kmalloc()

Функция kmalloc() аналогична функции malloc(), с которой разработчики пользовательских приложений для Linux хорошо знакомы, за исключением того, что к ней добавлен еще один параметр flags. Функция kmalloc() представляет собой простой интерфейс для выделения в ядре участков памяти размером в заданное количество байтов. Если вам необходимо выделить смежные участки физической памяти, размер которых кратен странице, то лучше использовать интерфейсы, рассмотренные в предыдущих разделах.

В большинстве случаев для выделения памяти в ядре предпочтительнее использовать функцию kmalloc(), которая описана в файле <linux/slab.h>.

```
void * kmalloc(size_t size, gfp_t flags)
```

Данная функция возвращает указатель на участок памяти, размер которого *не менее* size байт[3]. Выделенный участок памяти физически находится на смежных страницах памяти. В случае ошибки функция возвращает значение NULL. Операция по выделению памяти в ядре завершается успешно только в том случае, если доступно достаточное количество памяти. Поэтому после вызова функции kmalloc() всегда проверяйте возвращаемое значение на равенство NULL и соответствующим образом обрабатывайте ошибку!

Рассмотрим пример. Предположим, нам необходимо выделить достаточно памяти для того, чтобы в ней можно было разместить некоторую вымышленную структуру dog.

[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 в области доступной памяти. Ядро никогда не выделит меньше памяти, чем необходимо. Если ядро не в состоянии найти хотя бы указанное количество байтов, то операция завершится неудачно и функция возвратит значение NULL.

```
struct dog *p;
p = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!p)
    /* Здесь нужно обработать ошибку... */
```

Если вызов функции `kmalloc()` завершится успешно, то в переменной `p` будет находиться указатель на область памяти, размер которой больше заданного значения или равен ему. Флаг `GFP_KERNEL` определяет параметры программы распределения памяти, которые будут использоваться при выделении запрошенного участка памяти.

Флаги `gfp_mask`

Выше уже были показаны различные примеры использования флагов, которые изменяют алгоритм работы системы выделения памяти, как при вызове низкоуровневых функций, работающих на уровне страниц, так и при использовании функции `kmalloc()`. А теперь рассмотрим эти флаги подробнее. Флаги имеют тип `gfp_t`, который определен в файле `<linux/types.h>` на основе беззнакового целого числа. Аббревиатура `gfp` происходит от названия `__get_free_pages()` — одной из функций распределения памяти, рассмотренных выше.

Флаги разбиты на три категории: модификаторы операций, модификаторы зон и флаги типов. Модификаторы операций указывают, *каким образом* ядро должно выделять указанную память. При определенных условиях для выделения памяти могут использоваться только некоторые методы. Например, при выделении памяти в обработчике прерывания нужно запретить ядру переводить текущий процесс в состояние ожидания, поскольку его нельзя перепланировать.

Модификаторы зоны указывают, *откуда* нужно выделять память. Как уже было сказано, в ядре физическая память разделена на несколько зон, каждая из которых служит для различных целей. Так вот, этот тип модификаторов как раз и указывает, из какой именно зоны нужно выделить память. Флаги типов представляют собой различные комбинации модификаторов операций и зон, которые необходимы для выполнения определенного *типа* выделения памяти. Флаги типов заменяют собой целый ряд комбинаций других флагов. Например, вместо определения комбинации флагов действий и зон можно указать всего один флаг типа. Рассмотренная выше константа `GFP_KERNEL` как раз и относится к флагу типа, который используется в коде ядра, предназначенного для выполнения в контексте процесса. Рассмотрим флаги по отдельности.

Модификаторы операций

Все флаги, включая модификаторы операций, определены в заголовочном файле `<linux/gfp.h>`. Данный файл подключается также в другом заголовочном файле — `<linux/slab.h>`, поэтому его не часто приходится включать в свой проект явно. На практике обычно лучше использовать только флаги типов, которые будут рассмотрены ниже. Тем не менее полезно иметь представление о назначении отдельных флагов. Список модификаторов операций приведен в табл. 12.3.

Описанные модификаторы можно указывать вместе, как показано в следующем примере:

```
ptr = kmalloc(size, __GFP_WAIT | __GFP_IO | __GFP_FS);
```

Эта комбинация флагов указывает программе распределения памяти (а именно функции `alloc_pages()`), что при выполнении операции можно переводить процесс в состояние ожидания, выполнять операции ввода-вывода и обращаться к файловой системе, если это необходимо. В данном случае ядру предоставляется бо́льшая свобода в отношении того, где именно оно будет искать необходимую память, чтобы удовлетворить запрос.

Таблица 12.3. Модификаторы операций

Флаг	Описание
__GFP_WAIT	При выделении памяти текущий процесс может переводиться в состояние ожидания
__GFP_HIGH	При выделении памяти программа может обращаться к аварийным запасам
__GFP_IO	Программа распределения памяти может инициировать дисковые операции ввода-вывода
__GFP_FS	Программа распределения памяти может использовать операции ввода-вывода файловой системы
__GFP_COLD	При выделении памяти должны использоваться страницы, которые не находятся в кеш-памяти процессора (cache cold)
__GFP_NOWARN	При выделении памяти не должны выводиться предупреждающие сообщения об ошибках
__GFP_REPEAT	Операцию выделения памяти следует повторить в случае ошибки, однако потенциально она может завершиться неудачей
__GFP_NOFAIL	Операцию выделения памяти следует повторять в случае ошибки бесконечное количество раз; такая операция выделения памяти всегда завершается успешно
__GFP_NORETRY	В случае ошибки операцию выделения памяти повторять не нужно
__GFP_NOMEMALLOC	При выделении памяти нельзя обращаться к резервным ресурсам
__GFP_HARDWALL	В программе распределения памяти должны быть задействованы жесткие рамки (hardwall), определенные с помощью команды `cpuset`
__GFP_RECLAIMABLE	Выделенные страницы памяти помечаются в программе распределения как повторно затребованные
__GFP_COMP	В программе распределения памяти добавляются метаданные составной (compound) страницы памяти; они используются в коде поддержки больших страниц памяти (hugetlb)

Приведенные выше флаги указываются при выполнении большинства операций распределения памяти, только не в явном виде, а косвенно, с помощью флагов типа, которые мы скоро рассмотрим. Не нужно волноваться, у вас не будет необходимости каждый раз разбираться, какие именно из этих ужасных флагов использовать при выделении памяти!

Модификаторы зоны

Модификаторы зоны определяют, из какой зоны должна выделяться память. Обычно выделение может происходить из любой зоны. Однако ядро предпочитает зону ZONE_NORMAL, чтобы в других зонах, когда это необходимо, остались свободные страницы.

Существует всего три модификатора зоны, соответствующие трем оставшимся зонам, кроме ZONE_NORMAL (из которой память выделяется по умолчанию). Они описаны в табл. 12.4.

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 которую можно выполнять операции прямого доступа к памяти*". Флаг __GFP_HIGHMEM, наоборот, требует, чтобы выделение памяти было из зоны ZONE_NORMAL или, что предпочтительнее, ZONE_HIGHMEM. Этот флаг эквива-

лентен 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.

Таблица 12.4. Модификаторы зоны

Флаг	Описание
__GFP_DMA	Выделять память только из зоны ZONE_DMA
__GFP_DMA32	Выделять память только из зоны ZONE_DMA32
__GFP_HIGHMEM	Выделять память из зоны ZONE_HIGHMEM или ZONE_NORMAL

Флаг __GFP_HIGHMEM нельзя указывать при вызове функции __get_free_pages() или kmalloc(). Дело в том, что они возвращают логический адрес, а не указатель на структуру page. В результате появляется возможность, что эти функции выделят память, которая в данный момент не отображается в виртуальное адресное пространство ядра и поэтому не имеет логического адреса. Страницы в верхней памяти можно выделить только с помощью функции alloc_pages(). Однако в большинстве случаев в запросах на выделение памяти не нужно указывать модификаторы зоны, так как достаточно того, что по умолчанию используется зона ZONE_NORMAL.

Флаги типов

Флаги типов определяют модификаторы операций и зон, которые необходимы при выполнении определенных запросов на выделение памяти. Поэтому в коде ядра стараются использовать нужный флаг типа вместо большого набора модификаторов. Так проще, и при этом меньше шансов ошибиться. В табл. 12.5 приведен список возможных флагов типов, а в табл. 12.6 показано, какие модификаторы соответствуют какому флагу.

Таблица 12.5. 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, их нижних половинах, при удержании спин-блокировки, а также в других ситуациях, когда в состояние ожидания переходить нельзя
GFP_NOWAIT	Подобен GFP_ATOMIC, за исключением того, что в выделении памяти не задействуются аварийные запасы. Этот флаг увеличивает вероятность того, что операция выделения памяти завершится неудачно
GFP_NOIO	Запрос на выделение памяти может переводить задачу в состояние ожидания, однако при его выполнении нельзя инициировать операции дискового ввода-вывода. Этот флаг предназначен для использования в низкоуровневом коде, выполняющем блочный ввод-вывод, когда нельзя инициировать новые дисковые операции, поскольку это может привести к нежелательной рекурсии
GFP_NOFS	Запрос на выделение памяти может переводить задачу в состояние ожидания, однако при его выполнении можно инициировать операции дискового ввода-вывода и запрещено выполнять операции, связанные с файловыми системами. Этот флаг предназначен для использования в коде файловых систем, когда нельзя начинать выполнение новых файловых операций

Окончание табл. 12.5

Флаг	Описание
GFP_KERNEL	Обычный запрос на выделение памяти, который может переводить задачу в состояние ожидания. Этот флаг предназначен для использования в коде, который выполняется в контексте процесса, когда безопасно переходить в состояние ожидания. При этом ядро будет выполнять свои обычные действия, обрабатывая поступивший запрос на выделение памяти. Этот флаг должен использоваться по умолчанию
GFP_USER	Обычный запрос на выделение памяти, который может переводить задачу в состояние ожидания. Этот флаг используется для выделения памяти пользовательским процессам
GFP_HIGHUSER	Запрос на выделение памяти из зоны ZONE_HIGHMEM, который может переводить задачу в состояние ожидания. Этот флаг используется для выделения памяти пользовательским процессам
GFP_DMA	Запрос на выделение памяти из зоны ZONE_DMA. В драйверах устройств, которым нужна память для выполнения операций по ПДП, нужно использовать этот флаг в комбинации с одним из описанных выше флагов

Таблица 12.6. Список модификаторов, соответствующих каждому флагу типа

Флаг	Модификаторы
GFP_ATOMIC	__GFP_HIGH
GFP_NOWAIT	0
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 часто используемые флаги и объясним, для чего и почему они нужны. В подавляющем большинстве случаев при выполнении операций выделения памяти в ядре используется флаг GFP_KERNEL. В результате операция выделения памяти имеет обычный приоритет и может переводить процесс в состояние ожидания. Поскольку вызов этой функции может заморозить выполнение текущего процесса, его можно использовать только в контексте процесса, выполнение которого может быть безопасно перепланировано (т.е. когда нет удерживаемых блокировок и т.п.). 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.

Использование флага GFP_ATOMIC приводит к совершенно противоположным действиям. Поскольку этот флаг указывает, что операция выделения памяти не может переходить в состояние ожидания, для ее выполнения подходит не любая свободная память. Если под рукой нет непрерывного свободного участка памяти заданного размера, то ядро, скорее всего, не будет дополнительно 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 и вы-

теснить неактивные страницы на диск (swap out), сохранить измененные страницы в дисковый файл (flush dirty pages) и т.п. 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), по сравнению с предыдущим случаем. Тем не менее использование флага GFP_ATOMIC — это единственная возможность выделить память, когда вызывающий код не может переходить в состояние ожидания, как в случае обработчиков прерываний, отложенных прерываний и тасклетов.

По своим свойствам между двумя рассмотренными выше флагами находятся флаги GFP_NOIO и GFP_NOFS. Операции выделения памяти, которые запущены с этими флагами, могут переходить в состояние ожидания, но при этом в них не будут выполняться некоторые действия. При выделении памяти с флагом GFP_NOIO не будут запускаться никакие операции дискового ввода-вывода, предназначенные для удовлетворения запроса. С другой стороны, при использовании флага GFP_NOFS могут запускаться операции дискового ввода-вывода, но не могут инициироваться операции с файловыми системами.

Когда эти флаги могут нам понадобиться? Они используются при написании низкоуровневого кода для выполнения блочного ввода-вывода или фрагментов файловых систем соответственно. Представьте себе, что в некотором общем участке кода файловой системы при выделении памяти *не был указан* флаг GFP_NOFS. В результате при выделении памяти ядро может задействовать некоторые операции с файловой системой, которые в свою очередь вызовут функцию распределения памяти, в которой снова будут задействованы операции с файловой системой, и т.д.! Этот процесс может продолжаться до бесконечности. При разработке кода, в котором используются функции выделения памяти, необходимо гарантировать, что в коде самих этих функций не используется разрабатываемый код, как в рассмотренном выше случае, иначе может возникнуть самоблокировка. Не удивительно, что в ядре рассматриваемые два флага используются лишь в небольшом количестве мест.

Флаг GFP_DMA предписывает системе выделения памяти, что при выполнении запроса нужно выделить память из зоны ZONE_DMA. Этот флаг используется в коде драйверов устройств, для которых необходимо выполнение операций прямого доступа к памяти. Как правило, вместе с этим флагом используется флаг GFP_ATOMIC или GFP_KERNEL.

В подавляющем большинстве случаев при разработке кода вы будете использовать флаг GFP_KERNEL или GFP_ATOMIC. В табл. 12.7 описаны типичные ситуации, связанные с выделением памяти, и используемые при этом флаги. Независимо от типа операции выделения памяти в коде нужно проверить результат выполнения функции и по необходимости обработать ошибку.

Таблица 12.7. Типичные ситуации и используемые при этом флаги

Ситуация	Решение
Контекст процесса, можно переходить в состояние ожидания	Используется флаг GFP_KERNEL
Контекст процесса, нельзя переходить в состояние ожидания	Используется флаг GFP_ATOMIC, либо 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

Окончание табл. 12.7

Ситуация	Решение
Отложенное прерывание	Используется флаг `GFP_ATOMIC`
Тасклет	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)`, либо выделение выполняется в более поздний или более ранний момент времени

Функция `kfree()`

Функция `kfree()` выполняет действия, обратные действиям функции `kmalloc()`. Она определена в файле `<linux/slab.h>` следующим образом:

```
void kfree(const void *ptr)
```

Функция `kfree()` позволяет освободить память, ранее выделенную с помощью функции `kmalloc()`. Не вызывайте эту функцию, если участок памяти не был ранее выделен с помощью функции `kmalloc()` либо если он уже был один раз освобожден. Подобные действия являются ошибкой, которая может привести к освобождению памяти, принадлежащей другой части ядра. Так же как и при разработке пользовательских приложений, следует вести баланс операций выделения и освобождения памяти, который позволит вовремя выявить утечки и другие ошибки, связанные с работой с памятью. Обратите внимание на то, что случай вызова функции `kfree(NULL)` специально проверяется и поэтому является безопасным.

Рассмотрим пример выделения памяти в обработчике прерывания. В данном случае необходимо выделить память под буфер для сохранения поступающих данных. Размер буфера определяется с помощью директивы препроцессора в виде константы `BUF_SIZE`. Очевидно, что размер буфера, скорее всего, будет существенно больше нескольких байт.

```
char *buf;

buf = kmalloc(BUF_SIZE, GFP_ATOMIC);

if (!buf)
   /* Ошибка при выделении памяти! */
```

По окончании работы с буфером не забудьте освободить занимаемую им память, как показано в приведенном ниже фрагменте кода.

```
kfree(buf);
```

Функция `vmalloc()`

Функция `vmalloc()` аналогична функции `kmalloc()`, за исключением того, что она выделяет смежные страницы виртуальной памяти, которые необязательно являются смежными физически. Точно так же работают и функции выделения памяти, используемые в пользовательских приложениях. Страницы памяти, выделяемые с помощью функции `malloc()`, являются смежными в виртуальном адресном пространстве процессора, но нет никакой гарантии, что они являются смежными в физической оперативной памяти.

Функция kmalloc() отличается тем, что выделяет физически (и виртуально) смежные страницы памяти. Функция vmalloc() выделяет смежные страницы только в виртуальном адресном пространстве ядра. Это реализуется путем выделения потенциально несмежных участков физической памяти и изменения информации в таблицах страниц, отображающих эту физическую память в непрерывный участок логического адресного пространства.

В большинстве случаев только аппаратным устройствам необходимо выделение физически непрерывных участков памяти. На многих аппаратных платформах физические устройства напрямую взаимодействуют с оперативной памятью компьютера, минуя модуль управления памятью процессора, и поэтому не могут работать с виртуальными адресами. Следовательно, все области памяти, с которыми работают аппаратные устройства, должны состоять из физически, а не виртуально смежных блоков. Для участков памяти, которые используются только в программном обеспечении, например, буферов памяти, связанных с процессами, прекрасно подходят виртуально непрерывные области памяти. При программировании заметить разницу невозможно, поскольку вся память воспринимается ядром как логически непрерывная.

Несмотря на то что физически смежные страницы памяти необходимы только в определенных случаях, в большей части кода ядра используется для выделения памяти функция kmalloc(), а не vmalloc(). Это делается в основном из соображений производительности. Для того чтобы физически несмежные страницы памяти сделать смежными в виртуальном адресном пространстве, функция vmalloc() должна соответствующим образом сформировать элементы таблицы страниц. Хуже того, страницы виртуальной памяти, которые выделяются с помощью функции vmalloc(), должны отображаться на физические страницы памяти через элементы таблицы страниц по одной, поскольку они физически несмежные. Это приводит к значительно менее эффективному использованию буфера быстрой переадресации, или ББП (Translation Lookaside Buffer, TLB[4]), чем в случае, когда страницы памяти отображаются напрямую. Исходя из этих соображений, функция vmalloc() используется только тогда, когда это абсолютно необходимо, как правило, для выделения очень больших областей памяти. Например, при динамической загрузке модулей ядра они загружаются в память, которая выделяется с помощью функции vmalloc().

Функция vmalloc() объявлена в файле <linux/vmalloc.h> и определена в файле mm/vmalloc.c. Она используется точно так же, как и функция malloc() в пользовательских приложениях.

```
void * vmalloc(unsigned long size)
```

Она возвращает указатель на виртуально непрерывную область памяти, размер которой не менее size байт. В случае ошибки эта функция возвращает значение NULL. При выполнении функции vmalloc() процесс может переходить в состояние ожидания. Поэтому ее нельзя вызывать в контексте прерывания или в других ситуациях, когда блокирование процесса недопустимо.

[4] *Буфер быстрой переадресации* — это аппаратный буфер памяти, который используется на большинстве компьютерных платформ для кеширования отображений виртуальных адресов памяти в физические адреса. Этот буфер позволяет существенно повысить производительность системы, поскольку большинство операций доступа к памяти выполняется с использованием виртуальной адресации и в пределах одной или нескольких страниц.

Для освобождения памяти, выделенной с помощью функции `vmalloc()`, используется функция `vfree()`.

```
void vfree(const void *addr)
```

Эта функция освобождает участок памяти, начинающийся с адреса `addr`, который был ранее выделен с помощью функции `vmalloc()`. Данная функция также может переводить процесс в состояние ожидания и поэтому не может вызываться в контексте прерывания. Функция `vfree()` не возвращает никаких значений.

Использовать рассмотренные выше функции очень просто. Рассмотрим пример.

```
char *buf;

buf = vmalloc(16 * PAGE_SIZE); /* Выделяет 16 страниц */

if (!buf)
   /* Ошибка распределения памяти! */

/*
 * В переменной 'buf' теперь хранится указатель на участок памяти
 * размером не менее 16*PAGE_SIZE байтов виртуально непрерывного
 * блока памяти
 */
```

После того как память больше не нужна, ее нужно освободить с помощью вызова приведенной ниже функции.

```
vfree(buf);
```

Уровень блочного распределения памяти

Выделение и освобождение памяти, занимаемой структурами данных, — одна из самых частых операций, которые выполняются в любом ядре. Для того чтобы упростить процедуру частого выделения и освобождения данных, программисты обычно ведут *списки свободных ресурсов* (free list). В список свободных ресурсов заносится некоторый набор уже выделенных структур данных, готовых к использованию. Как только в коде понадобится новый экземпляр структуры данных, его можно просто извлечь из списка свободных ресурсов и не заниматься при этом выделением достаточного участка памяти, в котором может разместиться структура данных, и ее инициализацией. Позже, когда структура данных больше не нужна, она снова возвращается в список свободных ресурсов, а занимаемая ею память не освобождается. В этом смысле список свободных ресурсов представляет собой кеш объектов, в котором хранятся объекты одного определенного, часто используемого типа. В определенном смысле, список свободных ресурсов представляет собой кеш объектов часто используемого *типа*.

Одна из самых больших проблем, связанных со списками свободных ресурсов в ядре, состоит в том, что над ними нет никакого централизованного управления. При недостатке свободной памяти ядро не может сообщить всем спискам свободных ресурсов, чтобы те уменьшили размер своего кеша и освободили неиспользуемую память. В ядре нет никакой информации о случайно созданных списках свободных ресурсов. Поэтому, чтобы исправить ситуацию и унифицировать программный код, в ядро был введен специальный уровень *блочного распределения памяти* (slab layer), который часто также называют *блочным распределителем памяти* (slab allocator). Блочный распределитель памяти по сути является уровнем кеширования универсальных структур данных.

Концепции блочного распределения памяти впервые были реализованы в операционной системе SunOS 5.4 фирмы Sun Microsystems[5]. Для уровня кеширования структур данных в операционной системе Linux используется такое же название и похожие особенности реализации.

Уровень блочного распределения памяти служит для достижения следующих целей.

- Часто используемые структуры данных, скорее всего, будут часто выделяться и освобождаться, поэтому их следует кешировать.
- Частые операции выделения и освобождения памяти обычно приводят к сильной фрагментации памяти. В результате через некоторое время система не сможет выделить большие участки непрерывной памяти. Для предотвращения этого кешированные списки свободных ресурсов занимают непрерывный участок памяти. Поскольку ненужные структуры данных снова возвращаются в список свободных ресурсов, в результате никакой фрагментации памяти не возникает.
- Список свободных ресурсов обеспечивает улучшенную производительность при частых выделениях и освобождениях объектов, так как ненужные объекты, помещенные в список, сразу же становятся доступными для нового выделения.
- Если в программе блочного распределения памяти можно использовать дополнительную информацию, такую как размер объекта, размер страницы памяти и общий размер кеша, то появляется возможность принимать в критических ситуациях более интеллектуальные решения.
- Если часть кеша связать с определенным процессором (т.е. для каждого процессора в системе используется самостоятельный участок кеш-памяти), то выделение и освобождение структур данных может выполняться без использования SMP-блокировок.
- Если распределитель рассчитан на работу с неравномерной памятью (Non-Uniform Memory Access, или NUMA), то появляется возможность выделения памяти с того же *узла* (node), на котором эта память запрашивается.
- Хранимые в кеш-памяти объекты могут быть "*окрашены*", чтобы предотвратить отображение разных объектов на одни и те же *строки* (lines) системного кеша.

Уровень блочного распределения памяти в ОС Linux был реализован с учетом указанных выше принципов.

Структура уровня блочного распределения памяти

Объекты, хранящиеся на блочном уровне, разделены на группы, называемые *кешами* (*caches*). Разные кеши используются для хранения объектов различных типов. Для каждого типа объектов существует свой уникальный кеш. Например, один кеш может использоваться для хранения дескрипторов процессов (т.е. списка свободных структур типа `task_struct`), а другой — для индексов файловых систем (структура типа `inode`). Интересно, что интерфейс функции `kmalloc()` построен на основе уровня блочного распределения памяти, в котором используется семейство кешей общего назначения.

[5] Позже он был документирован в работе Bonwick J. *The Slab Allocator: An Object-Caching Kernel Memory Allocator*, USENIX, 1994.

Далее кеши делятся на *блоки* (slab) (буквально slab — монолитный блок, отсюда и название всей подсистемы). Блоки занимают одну или несколько физически смежных страниц памяти. Обычно блок занимает только одну страницу памяти. Каждый кеш может содержать несколько блоков.

В каждом блоке находится некоторое количество *объектов*, которые представляют собой кешируемые структуры данных. Каждый блок может находиться в одном из трех состояний: *заполненный* (full), *частично заполненный* (partial) и *пустой* (empty). В заполненном блоке нет свободных объектов, поскольку все они распределены и используются. Соответственно, в пустом блоке нет ни одного распределенного объекта, все его объекты свободны для последующего использования. В частично заполненном блоке есть как выделенные, так и свободные объекты. Как только из некоторой части ядра поступает запрос на новый объект, последний выделяется из частично заполненного блока, если таковой имеется. В противном случае объект выделяется из пустого блока. Если больше не осталось пустых блоков, то они будут созданы по мере необходимости. Очевидно, что из заполненного блока новый объект не может быть выделен, поскольку в нем больше не осталось свободных объектов. Такая стратегия выделения памяти существенно снижает фрагментацию памяти.

В качестве примера рассмотрим структуры `inode`, которые представляют в оперативной памяти индексы дисковых файлов (см. главу 13, “Виртуальная файловая система”). Эти структуры часто создаются и удаляются, поэтому есть смысл управлять ими с помощью блочного распределителя памяти. Структуры `inode` выделяются из кеша `inode_cachep` (такое соглашение по присваиванию имен является стандартом). Этот кеш состоит из одного или нескольких блоков. Вероятнее всего, блоков будет много, так как существует много объектов типа `inode`. В каждом блоке содержится максимально возможное количество объектов данного типа. При поступлении от ядра запроса на новую структуру типа `inode` возвращается указатель на уже распределенную и свободную структуру из частично заполненного блока, либо если такового нет, то из пустого блока. Когда ядро завершит работу с объектом типа `inode`, в блочном распределителе памяти он помечается как свободный. Взаимосвязь между кешами, блоками и объектами показана на рис. 12.1.

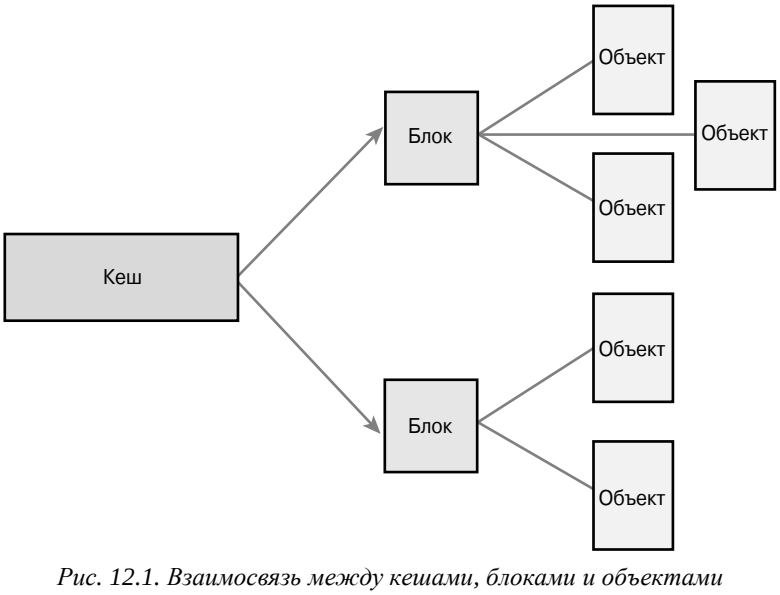


Рис. 12.1. Взаимосвязь между кешами, блоками и объектами

Каждый кеш представляется в виде структуры kmem_cache. В ней содержатся три списка, slabs_full, slabs_partial и slabs_empty, которые хранятся в структуре kmem_list3, определенной в файле mm/slab.c. В этих списках находятся все блоки, связанные с данным кешем. Каждый блок описывается с помощью структуры типа slab, которая является его дескриптором.

```
struct slab {
   struct list_head list; /* Список заполненных, частично заполненных
                             или пустых блоков */
   unsigned      long colouroff; /* Смещение для окрашивания блока */
   void         *s_mem;          /* Указатель на первый объект блока */
   unsigned      int inuse;      /* Количество выделенных объектов */
   kmem_bufctl_t free;           /* Первый свободный объект, если есть */
};
```

Сам дескриптор блока расположен либо за пределами блока в кеше общего назначения, либо в самом начале описываемого им блока. Дескриптор хранится в блоке, если общий размер блока достаточно мал либо если в самом блоке остается достаточно места, чтобы разместить дескриптор.

В распределителе блочного типа новые блоки создаются через специальный интерфейс с низкоуровневой функцией ядра __get_free_pages(), выделяющей страницы памяти, как показано ниже.

```
static void *kmem_getpages(struct kmem_cache *cachep, gfp_t 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;
}
```

В этой функции для выделения памяти, достаточной для хранения кеша, используется функция __get_free_pages(). В первом параметре этой функции указывается конкретный кеш, для которого нужны новые страницы памяти. Во втором параметре указываются флаги, которые передаются в функцию __get_free_pages(). Обратите вни-

мание на то, как значения этих флагов объединяются с другими значениями с помощью логической операции ИЛИ. В данной операции к переданным в параметре `flags` флагам добавляются значения стандартных флагов кеша. В поле `cachep->gfporder` хранится размер выделенных страниц памяти, выраженный в виде степени числа 2. Рассматриваемая функция выглядит более сложной, чем это может показаться сначала, поскольку она также рассчитана на работу с неравномерной памятью (NUMA-системы). Если параметр `nodeid` не равен `-1`, то предпринимается попытка выделить память из 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.

В учебных целях можно убрать код, рассчитанный на NUMA-системы, и получить более простой вариант функции `kmem_getpages()`, как показано ниже.

```
static inline void * kmem_getpages(struct kmem_cache *cachep, gfp_t flags)
{
   void *addr;
   flags |= cachep->gfpflags;
   addr = (void*) __get_free_pages(flags, cachep->gfporder);
   return addr;
}
```

Память освобождается с помощью функции `kmem_freepages()`, в которой вызывается функция `free_pages()` для освобождения заданных страниц кеша. Разумеется, уровень блочного распределения памяти предназначен для упразднения операций по выделению и освобождению страниц. На самом деле функции выделения памяти используются в блочном распределителе только тогда, когда в данном кеше больше нет ни одного пустого или частично заполненного блока. Функция освобождения памяти вызывается только тогда, когда мало доступной памяти и система пытается освободить память или когда кеш полностью аннулируется.

Управление уровнем блочного распределения памяти осуществляется для каждого кеша в отдельности с помощью простого интерфейса, который экспортируется в цельное ядро. Интерфейс позволяет создавать или аннулировать новые кеши, а также выделять и освобождать объекты в этих кешах. Все изощренные средства управления кешами и блоками полностью реализованы внутри уровня блочного распределителя памяти. После того как кеш создан, блочный распределитель памяти работает, как специализированная система создания объектов определенного типа.

Интерфейс блочного распределителя памяти

Новый кеш создается с помощью вызова следующей функции:

```
struct kmem_cache * kmem_cache_create(const char *name,
                                      size_t size,
                                      size_t align,
                                      unsigned long flags,
                                      void (*ctor)(void *));
```

Первый параметр — это строка, содержащая имя кеша. Второй параметр — это размер каждого элемента кеша. Третий параметр — это смещение первого объекта в блоке. Он нужен для того, чтобы обеспечить выравнивание по требуемым границам на страницах памяти. Обычно достаточно указать значение, равное нулю, которое соответствует стандартному выравниванию. В параметре `flags` указываются необязательные флаги, управляющие работой кеша. Он может быть равен нулю, когда не требуется управлять

работой кеша, либо состоять из одного или нескольких значений, показанных ниже и объединенных с помощью логической операции ИЛИ.

- SLAB_HWCACHE_ALIGN. Этот флаг указывает программам уровня блочного распределения памяти, что каждый объект в блоке должен располагаться так, чтобы он был выровнен по строкам процессорного кеша. Это предотвращает так называемое "ошибочное распределение", когда два или более объекта отображаются в одну и ту же строку системного кеша, несмотря на то, что они находятся по разным адресам памяти. При использовании этого флага возрастает производительность за счет увеличения занимаемой памяти, потому что строгое выравнивание приводит к тому, что часть памяти блока не используется. Степень увеличения занимаемой памяти зависит от размера объектов кеша и от того, каким образом происходит их выравнивание по отношению к строкам системного кеша. Для кешей, которые часто используются в критических к скорости выполнения участках кода, этот флаг стоит установить, в остальных случаях следует хорошенько подумать, стоит ли это делать.
- SLAB_POISON. Этот флаг указывает, что неиспользуемые области блока должны быть заполнены предопределенным числовым значением (0xa5a5a5a5). Эта операция называется "загрязнением" (*poisoning*) и служит для отслеживания доступа к неинициализированной области памяти.
- SLAB_RED_ZONE. Этот флаг указывает программам уровня блочного распределения памяти, что вокруг выделенного участка памяти нужно разместить так называемые "красные зоны" (*red zone*) для облегчения выявления ситуаций с переполнением буфера.
- SLAB_PANIC. Этот флаг вызывает состояние *тревоги* (panic) в случае, если выделение памяти было неудачным. При указании данного флага подразумевается, что выделение памяти всегда должно завершаться успешно, как, например, при создании кеша структур VMA (областей виртуальной памяти, см. главу 15, "Адресное пространство процесса") во время начальной загрузки системы.
- SLAB_CACHE_DMA. Этот флаг указывает программам уровня блочного распределения памяти, что все блоки должны выделяться из зоны, в которой возможны операции прямого доступа к памяти. Данный флаг необходим, когда объекты используются в операциях ПДП и должны находиться в зоне ZONE_DMA. В противном случае эта возможность не нужна и данный флаг не нужно устанавливать.

Последний параметр, ctor, определяет конструктор кеша. Конструктор вызывается, когда в кеш добавляются новые страницы памяти. На практике в кешах ядра ОС Linux обычно не используются функции конструктора. На самом деле раньше у функции kmem_cache_create() был также и параметр, определявший деструктор кеша. Однако он был удален, поскольку не использовался в коде ядра. В качестве конструктора можно указывать значение NULL.

В случае успешного выполнения функция kmem_cache_create() возвращает указатель на созданный кеш. В противном случае возвращается NULL. Эта функция не может вызываться в контексте прерывания, так как она может переводить процесс в состояние ожидания.

Для ликвидации кеша вызовите следующую функцию:

```
int kmem_cache_destroy(struct kmem_cache *cachep)
```

Как следует из названия функции kmem_cache_destroy(), она аннулирует кеш, указанный в качестве параметра. Обычно эта функция вызывается при выгрузке модуля из памяти, в котором был создан свой кеш. 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 перечисленные ниже два условия.

- Все блоки в указанном кеше должны быть пустыми. Действительно, если в каком-либо блоке существует объект, который все еще используется, то как можно ликвидировать кеш?
- Во время и особенно после вызова функции kmem_cache_destroy() ни один процесс на должен обращаться к кешу. Об этом нужно позаботиться в вызывающем коде.

В случае успешного выполнения функция возвращает нуль, в других случаях возвращается ненулевое значение.

Размещение объектов в кеше

После создания кеша из него можно получить объект, вызвав приведенную ниже функцию.

```
void * kmem_cache_alloc(struct kmem_cache *cachep, gfp_t flags)
```

Эта функция возвращает указатель на объект, выделенный из кеша, на который указывает параметр cachep. Если ни в одном из блоков нет свободных объектов, то блочный распределитель должен получить новые страницы памяти с помощью функции kmem_getpages(). При этом значение параметра flags передается в функцию __get_free_pages(). Это те же самые флаги, которые были рассмотрены выше. Как правило, здесь указывается флаг GFP_KERNEL или GFP_ATOMIC.

Чтобы освободить ненужный объект и вернуть его в блок, из которого он был выделен, используется следующая функция:

```
void kmem_cache_free(struct kmem_cache *cachep, void *objp)
```

Она помечает объект, на который указывает параметр objp, в кеше cachep как свободный.

Пример использования блочного распределителя памяти

Рассмотрим реальный пример, связанный с работой со структурами дескрипторов процессов task_struct. Показанный ниже код в несколько более сложной форме приведен в файле kernel/fork.c.

Для начала в ядре определяется глобальная переменная, в которой хранится указатель на кеш объектов типа task_struct, как показано ниже.

```
struct kmem_cache *task_struct_cachep;
```

Этот кеш создается во время инициализации ядра, в функции fork_init(), определенной в файле kernel/fork.c, как показано ниже.

```
task_struct_cachep = kmem_cache_create("task_struct",
                                       sizeof(struct task_struct),
                                       ARCH_MIN_TASKALIGN,
                                       SLAB_PANIC | SLAB_NOTRACK,
                                       NULL);
```

В результате будет создан кеш "task_struct", предназначенный для хранения объектов типа task_struct. Объекты создаются с начальным смещением в блоке, равным ARCH_MIN_TASKALIGN байт. Эта константа определяется с помощью директивы препроцессора, и ее значение зависит от используемой аппаратной платформы. Обычно значение выравнивания для каждой аппаратной платформы соответствует размеру процессорного кеша первого уровня в байтах, которое хранится в константе L1_CACHE_BYTES, определяемой с помощью директивы препроцессора. Конструктор кеша отсутствует. Обратите внимание на то, что возвращаемое значение не проверяется на равенство NULL, означающее ошибку, поскольку указан флаг SLAB_PANIC. В случае, когда при выделении памяти произошла ошибка, из блочного распределителя памяти вызывается функция panic(). Если этот флаг не указан, то нужно обязательно проверять возвращаемое значение на равенство NULL, что сигнализирует об ошибке. Флаг SLAB_PANIC указан здесь потому, что данный кеш просто необходим для работы системы (без дескрипторов процессов работать как-то не хорошо).

Каждый раз при вызове из процесса функции fork() должен создаваться новый дескриптор процесса (вспомните главу 3, "Управление процессами"). Это выполняется в функции dup_task_struct(), которая вызывается из функции do_fork(), как показано ниже.

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
   return NULL;
```

Когда процесс завершается и у него больше нет порожденных процессов, которые ожидают завершения родительского процесса, дескриптор освобождается и возвращается обратно в блочный кеш 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);
```

Поскольку дескрипторы процессов относятся к основным компонентам ядра и всегда нужны, то кеш task_struct_cachep никогда не аннулируется. Хотя, если бы его нужно было ликвидировать, делается это так, как показано ниже.

```
int err;

err = kmem_cache_destroy(task_struct_cachep);
if (err)
   /* Ошибка при ликвидации кеша */
```

Достаточно просто, не так ли? Все низкоуровневые операции, связанные с выравниванием, "раскрашиванием", выделением и освобождением памяти, а также ликвидацией "лишней" памяти при нехватке системной памяти выполняются на уровне блочного распределителя и скрыты от пользователя. Если вам необходимо часто создавать много объектов одного типа, то используйте блочный кеш. И уж точно не нужно писать свою реализацию списка свободных ресурсов!

Статическое выделение памяти в стеке

В пользовательских приложениях многие операции выделения памяти, в частности некоторые из рассмотренных ранее примеров, могут быть выполнены с использованием стека, потому что априори известен размер выделяемой области памяти. В пользовательских

приложениях доступна такая роскошь, как очень большой и динамически увеличивающийся стек задачи, тогда как в режиме ядра такой роскоши нет — стек ядра маленький и фиксирован по размеру. Дело в том, что при выделении процессу небольшого фиксированного по размеру стека уменьшается расход памяти, к тому же ядру нет необходимости выполнять дополнительные функции по управлению памятью.

Размер системного стека процесса зависит как от аппаратной платформы, так и от параметров конфигурации, 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, так как при этом используются 4- и 8-килобайтовые страницы памяти соответственно.

Одностраничные стеки ядра

В первых версиях ядер серии 2.6 был введен специальный параметр конфигурации, позволявший уменьшить размер стека ядра каждого процесса до одной страницы памяти. Если этот параметр установлен, то каждому процессу назначается системный стек, размер которого равен всего одной странице памяти: 4 Кбайт на 32-разрядных аппаратных платформах и 8 Кбайт — на 64-разрядных. Это было сделано по двум причинам. Во-первых, для уменьшения расхода памяти на одну страницу для каждого процесса. Во-вторых (и это наиболее важно!), при непрерывной работе системы длительное время становится все труднее найти две свободные физически смежные страницы памяти. Физическая память слишком фрагментируется, и нагрузка на систему управления виртуальной памятью при создании новых процессов становится все более существенной.

Существует и 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 в обработчиках прерываний использовался стек ядра того процесса, выполнение которого было прервано. Поэтому размер такого стека должен был быть достаточно большим, чтобы в него все влезло. Такое решение было достаточно простым и эффективным, но оно накладывало еще бо́льшие ограничения на и так скудный размер стека ядра. А когда перешли на одностраничные стеки ядра, для обработчиков прерываний места в стеке уже не осталось.

Чтобы устранить возникшую проблему, разработчики ядра ввели новую возможность — стеки обработчиков прерываний. Для каждого процессора в системе был выделен один специальный стек, который стал использовать для обработки прерываний. Если при конфигурации установлен одностраничный стек ядра, то в обработчиках прерываний больше не будет использоваться стек ядра прерванных процессов. Вместо этого используются собственные стеки. При таком подходе для размещения стека требуется всего одна страница памяти на процессор.

Итак, подведем итоги. Стек ядра занимает одну или две страницы памяти, в зависимости от параметра конфигурации ядра, заданного во время компиляции. Следовательно, размер стека ядра может изменяться от 4 до 16 Кбайт. Исторически в обработчиках прерываний использовался стек ядра прерванного ими процесса. При переходе на одностраничные стеки ядра обработчикам прерываний были назначены собственные стеки. Однако в любом случае неограниченная рекурсия и использование функций вроде `alloca()` явно не допустимы.

На этом с теорией покончим.

Справедливое использование стека

В любой функции необходимо сократить использование стека до минимума. Хотя здесь не существует каких-либо твердых правил, тем не менее суммарный объем всех локальных переменных (их еще называют автоматическими, или переменными, выделенными в стеке) не должен превышать несколько сотен байтов. Довольно опасно статически выделять память в стеке для больших массивов или структур данных. Во всем остальном выделение памяти в стеке ядра выполняется точно так же, как и в пользовательских приложениях. Переполнение стека происходит незаметно и обычно приводит к большим проблемам. Так как в ядре не выполняется никаких действий по управлению стеком, то при его переполнении новые данные просто перезапишут данные, находящиеся за пределами стека. В первую очередь пострадает структура `thread_info`, которая расположена в самом конце стека процесса (см. главу 3, "Управление процессами"). Кроме того, за пределами стека могут находиться и другие важные данные ядра. В лучшем случае при переполнении стека произойдет аварийный останов компьютера. В худших случаях пострадают данные, и никто этого вовремя не заметит.

В связи с этим для распределения больших объемов памяти необходимо использовать одну из динамических схем выделения памяти, которые были рассмотрены выше.

Отображение верхней памяти

По определению страницы верхней памяти не могут постоянно отображаться в адресное пространство ядра. Поэтому страницы памяти, которые были распределены 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 относится к верхней памяти, и она не может автоматически или постоянно отображаться в адресное пространство ядра, несмотря на то, что процессоры платформы x86 могут адресовать до 4 Гбайт (или до 64 Гбайт при наличии расширения PAE[6]) физической памяти. После выделения эти страницы должны быть отображены в логическое адресное пространство ядра. Для платформы x86 логический адрес таких страниц находится где-то между отметками 3 и 4 Гбайт.

Постоянное отображение

Для отображения заданной структуры типа `page` в адресное пространство ядра используется приведенная ниже функция, описанная в файле `<linux/highmem.h>`.

```
void *kmap(struct page *page)
```

Эта функция работает как со страницами нижней, так и верхней памяти. Если структура типа `page` соответствует странице нижней памяти, то просто возвращается виртуальный адрес этой страницы. Если страница расположена в верхней памяти, то создается постоянное отображение этой страницы памяти и возвращается полученный логический

[6] PAE — Physical Address Extension (расширение физического адреса). Особенность процессоров x86, которая позволяет увеличить разрядность шины адреса с 32 до 36 разрядов. Это позволяет процессорам семейства x86 обращаться к 64 Гбайт физической памяти, несмотря на то, что размер логического адреса остается равным 32-битам, что соответствует максимальному размеру виртуального адресного пространства в 4 Гбайт.

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 в адресное пространство ядра), отображение страниц верхней памяти должно быть отменено, если оно больше не нужно. Это можно сделать, вызвав приведенную ниже функцию.

```
void kunmap(struct page *page)
```

Данная функция отменяет отображение страницы памяти, связанной с параметром типа `page`.

Временное отображение

В тех случаях, когда необходимо создать отображение страниц памяти в адресное пространство ядра, а текущий контекст не может переходить в состояние ожидания, в ядре предусмотрена функция *временного отображения* (temporary mappings), которое также называется *неделимым отображением* (atomic mappings). В ядре существует некоторое количество зарезервированных постоянных отображений, которые могут выполнять временные отображения. Ядро может неделимо отображать страницу верхней памяти в одно из таких зарезервированных отображений. Следовательно, временное отображение используется в коде, который не может переходить в состояние ожидания, как, например, в обработчике прерывания, поскольку в процессе отображения процесс никогда не будет заблокирован.

Создание временного отображения выполняется с помощью следующей функции:

```
void *kmap_atomic(struct page *page, enum km_type type)
```

Параметр `type` соответствует приведенному ниже перечислению, описанному в файле `<asm-generic/kmap_types.h>`, и 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_SYNC_ICACHE,
        KM_SYNC_DCACHE,
        KM_UML_USERCOPY,
        KM_IRQ_PTE,
        KM_NMI,
        KM_NMI_PTE,
        KM_TYPE_NR
};
```

Выполнение данной функции не блокируется, поэтому она может использоваться в контексте прерывания и в других случаях, когда процесс нельзя перепланировать. Эта

функция также отключает мультипрограммный режим работы ядра. Дело в том, что конкретное отображение памяти является уникальным для каждого процессора, а в результате перепланирования выполнение задачи может быть перенесено на другой процессор.

Отменить отображение можно с помощью следующей функции:

```
void kunmap_atomic(void *kvaddr, enum km_type type)
```

Эта функция также не вызывает блокировки процесса. На самом деле на большинстве аппаратных платформ она не делает ничего, кроме разрешения мультипрограммного режима работы ядра. Дело в том, что временное отображение действует только до тех пор, пока не создано новое временное отображение. Поэтому в ядре можно "забыть" о вызванной функции `kmap_atomic()`, и функции `kunmap_atomic()` практически ничего не нужно делать. Следующее неделимое отображение просто заменяет предыдущее.

Выделение памяти для конкретного процессора

В современных мультипроцессорных операционных системах широко используются данные, связанные с определенными процессорами (per-CPU data). Это данные, которые в значительной степени являются уникальными для каждого процессора. Как правило, такие данные хранятся в виде массива. Каждый элемент массива соответствует своему процессору системы. Номер процессора является индексом в этом массиве. Таким образом была реализована работа с данными, связанными с определенными процессорами, в ядрах серии 2.4. В таком подходе нет ничего плохого, поэтому в значительной части кода ядра в серии 2.6 все еще используется этот интерфейс. Данные объявляются следующим образом:

```
unsigned long my_percpu[NR_CPUS];
```

Доступ к этим данным выполняется так, как показано ниже.

```
int cpu;

cpu = get_cpu();  /* Определяем номер текущего процессора и запрещаем
                     мультипрограммный режим работы ядра           */
my_percpu[cpu]++; /* Обращаемся к нужным данным */
printk("my_percpu на cpu=%d равно %lu\n", cpu, my_percpu[cpu]);
put_cpu(); /* Разрешить мультипрограммный режим работы ядра */
```

Обратите внимание на то, 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. Использование мультипрограммного режима работы ядра может вызвать две проблемы.

- Если выполняющийся код вытесняется и позже планируется для выполнения на другом процессоре, то значение переменной `cpu` больше не будет корректным, потому что эта переменная будет содержать номер другого процессора. (По той же причине после получения номера текущего процессора нельзя переходить в состояние ожидания.)
- При вытеснении текущего кода другим кодом в последнем может быть конкурентное обращение к переменной `my_percpu` *на том же процессоре*, в результате чего возникнет конфликт из-за доступа к ресурсу.

Однако все наши опасения напрасны, потому что вызов функции get_cpu(), которая возвращает номер текущего процессора, также запрещает мультипрограммный режим работы ядра. Соответствующий вызов функции put_cpu() возобновляет мультипрограммирование в ядре. Обратите внимание: функция smp_processor_id(), которая также позволяет получить номер текущего процессора, не запрещает мультипрограммный режим работы ядра. Поэтому для безопасной работы следует использовать указанный выше метод.

Новый интерфейс percpu

Для создания и работы с данными, связанными с определенным процессором, в ядрах серии 2.6 предложен новый интерфейс, именуемый percpu. Этот интерфейс обобщает предыдущий пример. При его использовании упрощается работа с данными, связанными с конкретным процессором.

Рассмотренный ранее метод работы с данными, которые связаны с определенным процессором, также вполне допустим и может использоваться. Описанный выше новый интерфейс придумали для того, чтобы на больших многопроцессорных компьютерах, поддерживающих симметричную многозадачность, обеспечить простой и более эффективный способ работы с данными, связанными с процессорами.

Все нужные для этого процедуры объявлены в заголовочном файле <linux/percpu.h>. Собственно реальные определения находятся в файлах mm/slab.c и <asm/percpu.h>.

Работа с процессорными данными на этапе компиляции

Описать переменную, которая связана с определенным процессором, на этапе компиляции достаточно просто, как показано ниже.

```
DEFINE_PER_CPU(type, name);
```

В результате будет создан экземпляр переменной типа type с именем name для каждого процессора в системе. Если необходимо объявить соответствующую переменную и избежать предупреждений компилятора, то используйте следующий макрос:

```
DECLARE_PER_CPU(type, name);
```

Работать с этими переменными можно с помощью функций get_cpu_var() и put_cpu_var(). Функция get_cpu_var() возвращает *левое значение* (l-value) указанной переменной на текущем процессоре. Кроме того, она также отменяет мультипрограммный режим работы ядра, который восстанавливает соответствующая ей функция put_cpu_var().

```
get_cpu_var(name)++; /* Увеличить на единицу значение переменной name,
                        связанное с текущим процессором */
put_cpu_var(name);   /* Восстановить мультипрограммный режим
                        работы ядра */
```

Можно также получить доступ к переменной, связанной с *другим* процессором.

```
per_cpu(name, cpu)++; /* Увеличить значение переменной name
                         на указанном процессоре */
```

Использовать последний вариант вызова функции per_cpu() нужно очень аккуратно, поскольку он не запрещает вытеснение кода в ядре и не обеспечивает никаких блокировок. При работе с данными, связанными с определенным процессором, блокировки не

нужны, только если к этим данным может обращаться всего один процессор. Если процессоры обращаются к данным других процессоров, то необходимо использовать блокировки. Поэтому будьте внимательны! Применение блокировок рассматривается в главе 9, "Общие сведения о синхронизации кода ядра", и главе 10, "Средства синхронизации ядра".

Необходимо сделать еще одно важное замечание относительно создания данных, связанных с процессорами, на этапе компиляции. Рассмотренные выше примеры не будут работать в загружаемых модулях ядра, если данные объявлены не в самом модуле. Дело в том, что компоновщик помещает эти данные в специальные сегменты 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, как описано ниже.

Работа с процессорными данными на этапе выполнения

Для динамического создания данных, связанных с процессорами, в ядре реализован специальный распределитель памяти, который имеет интерфейс, аналогичный функции `kmalloc()`. Он позволяет создать экземпляр участка памяти для каждого процессора в системе. Прототипы его функций объявлены в файле `<linux/percpu.h>` следующим образом:

```
void *alloc_percpu(type); /* Макрос */
void *__alloc_percpu(size_t size, size_t align);
void free_percpu(const void *);
```

Макрос `alloc_percpu()` позволяет создать экземпляр объекта заданного типа (выделить память) для каждого процессора в системе. По сути, он является оболочкой для функции `__alloc_percpu()`. Последней функции передается в качестве параметров количество байтов памяти, которые необходимо выделить, и число байтов, по которому необходимо выполнить выравнивание этой области памяти. Макрос `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);
```

Это аналогично следующему:

```
struct rabid_cheetah = __alloc_percpu(sizeof (struct rabid_cheetah),
                                      __alignof__ (struct rabid_cheetah));
```

Конструкция `__alignof__` является расширением компилятора `gcc`. Она возвращает количество байтов, по границе которого необходимо выполнять выравнивание для заданного типа или левого значения (или рекомендуется выполнять для тех аппаратных платформ, у которых нет жестких требований к выравниванию данных в памяти). Ее синтаксис такой же, как и у оператора `sizeof`. В примере, показанном ниже, для аппаратной платформы x86 будет возвращено значение 4.

```
__alignof__ (unsigned long)
```

Если в качестве параметра используется левое значение (l-value), то возвращается максимально возможное выравнивание, которое может потребоваться для этого значения. Например, левое 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. Вопросы выравнивания более подробно рассмотрены в главе 19, "Переносимость".

Для освобождения памяти, которую занимают соответствующие данные на всех процессорах, используется функция `free_percpu()`.

Макрос `alloc_percpu()` и функция `__alloc_percpu()` возвращают указатель, который используется для косвенной ссылки на динамически созданные данные, связанные с каждым процессором в системе. Для простого доступа к данным в ядре предусмотрены два следующих макроса:

```
get_cpu_var(ptr); /* Возвращает указатель типа void на данные,
                     соответствующие параметру ptr,
                     связанные с текущим процессором */
put_cpu_var(ptr); /* Готово! Возобновляем мультипрограммный режим работы
ядра */
```

Макрос `get_cpu_var()` возвращает указатель на экземпляр данных, связанных с текущим процессором. В результате его вызова также запрещается вытеснение кода в режиме ядра, для возобновления которого нужно вызвать макрос `put_cpu_ptr()`.

А теперь рассмотрим полноценный пример использования этих макросов. Конечно, этот пример не совсем логичный, потому что память обычно выделяется один раз (например, в функции инициализации), используется в разных местах кода, а затем освобождается также один раз (например, в функции, которая вызывается при завершении работы). Тем не менее он позволяет прояснить особенности использования макросов.

```
void *percpu_ptr;
unsigned long *foo;

percpu_ptr = alloc_percpu(unsigned long);
if (!ptr)
       /* Ошибка распределения памяти .. */

foo = get_cpu_var(percpu_ptr);

/* работаем с данными через указатель foo .. */

put_cpu_var(percpu_ptr);
```

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. Однако ничто не может помешать нарушению этого правила.

Во-вторых, данные, связанные с процессорами, позволяют существенно уменьшить недостоверность данных, хранящихся в кеш-памяти. Это происходит потому, что процессоры поддерживают свою кеш-память в синхронизированном состоянии. Если один процессор начинает работать с данными, которые находятся в кеш-памяти другого процессора, то второй процессор должен будет обновить или очистить содержимое своего кеша. Постоянное аннулирование находящихся в кеш-памяти данных, именуемое *перегрузкой кеша* (thrashing the cache), существенно снижает производительность системы. Использование данных, связанных с процессорами, позволяет приблизить эффективность работы с кешем к максимально возможной, потому что в идеале каждый процессор работает только со своими данными. В интерфейсе `percpu` выполняется выравнивание

всех данных в кеш-памяти. В результате гарантируется, что доступ к данным, связанным с одним процессором, никак не повлияет на данные другого процессора, находящиеся в той же строке кеша.

Таким образом, использование данных, связанных с процессорами, часто избавляет от необходимости использования блокировок (или снижает требования, связанные с блокировками). Единственное требование, предъявляемое для безопасной работы с этими данными, — запрещение вытеснения кода, который работает в режиме ядра. Эта операция значительно более эффективна по сравнению с использованием блокировок, к тому же в существующих интерфейсах запрещение и разрешение вытеснения кода выполняется автоматически. Данные, связанные с процессорами, можно безопасно использовать как в контексте прерывания, так и в контексте процесса. Тем не менее следует обратить внимание на то, что при использовании данных, связанных с текущим процессором, нельзя переходить в состояние ожидания (в противном случае выполнение процесса может быть продолжено на другом процессоре).

Сейчас нет строгой необходимости использовать где-либо новый интерфейс работы с данными, которые связаны с процессорами. Вполне можно организовать такую работу вручную (на основании массива, как было рассказано выше), если при этом отключен мультипрограммный режим работы ядра. Тем не менее новый интерфейс проще в использовании, и, возможно, в будущем он будет дополнительно оптимизирован. 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 ядра.

Выбор способа выделения памяти

Поскольку для выделения памяти в ядре существует множество способов и подходов, не всегда очевидно, как же все-таки можно получить память в ядре, хотя этот вопрос действительно очень важен! Если вам необходимы смежные страницы физической памяти, то используйте один из низкоуровневых интерфейсов выделения памяти или функцию `kmalloc()`. Это стандартный способ выделения памяти в ядре, и, скорее всего, в большинстве случаев вы будете 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 актуально для обработчиков прерываний и других случаев, когда нельзя переходить в состояние ожидания. В коде, который может переходить в состояние ожидания, как, например, коде, выполняющемся в контексте процесса и не удерживающем спин-блокировку, необходимо использовать флаг `GFP_KERNEL`. Он указывает, что должна выполняться операция выделения памяти, которая по мере необходимости может перевести текущий процесс в состояние ожидания для получения запрошенного участка памяти.

Если нужно выделить страницы верхней памяти, то используйте функцию `alloc_pages()`. Она возвращает указатель на структуру типа `page`, а не логический адрес участка памяти. Поскольку страницы верхней памяти могут в текущий момент не отображаться в адресное пространство ядра, единственный способ доступа к этой памяти — через структуру типа `page`. Чтобы получить “реальный” указатель на область памяти,

используется функция `kmap()`, которая позволяет отобразить верхнюю память в логическое адресное пространство ядра.

Если нет необходимости в физически смежных страницах памяти, а нужна только виртуально непрерывная область памяти, используйте функцию `vmalloc()`. При этом следует учитывать небольшую потерю производительности этой функции по сравнению с функцией `kmalloc()`. Функция `vmalloc()` выделяет виртуально непрерывный участок памяти, физические страницы которого не обязательно смежные. Это выполняется почти так же, как и в пользовательских приложениях, путем отображения физически несмежных участков памяти в логически непрерывную область памяти.

Если нужно создавать и освобождать много больших структур данных, то лучше всего создать собственный кеш блочного распределителя памяти. Он позволит поддерживать кеш объектов (список свободных объектов), уникальный для каждого процессора, который существенно ускоряет операции выделения и освобождения памяти под объекты. В блочном распределителе память не выделяется и не освобождается все время, вместо этого в его кеше хранятся уже выделенные объекты. Как только вам понадобится выделить память под структуру данных из блочного распределителя, последний вернет указатель на первый свободный элемент, хранящийся в кеше. И только когда в кеше уже не будет свободных элементов, блочный распределитель запросит память у системы под новый кеш.

Резюме

В этой главе были изучены методы управления памятью, используемые в ядре Linux. Мы рассмотрели различные единицы и категории памяти, включая байты страницы и зоны. Четыре категории адресного пространства процесса будут рассмотрены в главе 15, “Адресное пространство процесса”. Далее были рассмотрены различные механизмы выделения памяти, включая распределение на уровне блоков и страниц. Получение памяти внутри ядра связано с дополнительными трудностями, поскольку при этом приходится учитывать текущее состояние процесса, например невозможность его перехода в состояние ожидания или обращение к файловой системе. В связи с этим были рассмотрены флаги `gfp`, различные случаи и необходимые условия для использования каждого флага. Относительная трудность в выделении участка памяти в ядре связана с кардинально другим подходом к проектированию по сравнению с пользовательскими приложениями. Хотя бо́льшая часть этой главы посвящена описанию интерфейсов распределения памяти в ядре, вы не должны забывать и о тех *трудностях*, которые с этим связаны.

Взяв на вооружение материал данной главы, можете смело приступать к изучению материала следующей главы, посвященного виртуальной файловой системе (virtual filesystem, VFS). Речь пойдет о подсистеме ядра, которая управляет файловыми системами и обеспечивает для пользовательских приложений единообразный и логичный интерфейс API. Вперед!

13
Виртуальная файловая система

Виртуальная файловая система (Virtual File System), иногда называемая *виртуальным файловым коммутатором* (*Virtual File Switch*), или просто *VFS*, — это подсистема ядра, в которой реализован интерфейс к файлам и файловым системам, используемый в пользовательских приложениях. VFS положена в основу всех файловых систем. Это позволяет им не только вместе сосуществовать в одной операционной системе, но и взаимодействовать друг с другом. В результате появляется возможность использовать в пользовательских приложениях стандартные системные функции для чтения и записи данных, расположенных в различных файловых системах и находящихся на различных физических носителях, как показано на рис. 13.1.

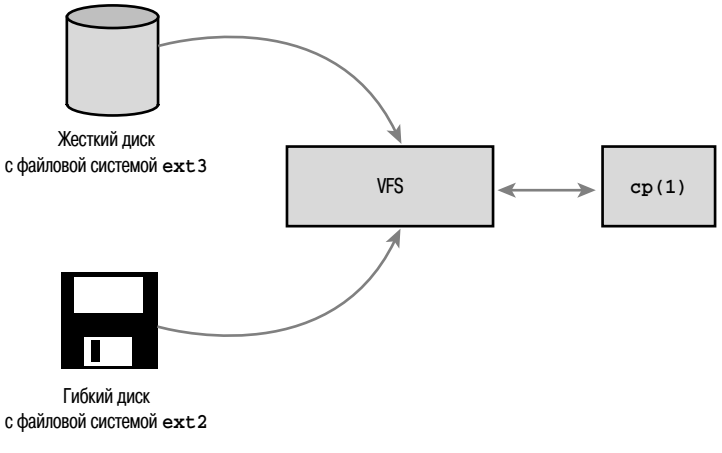


Рис. 13.1. Подсистема VFS в действии: использование команды `cp(1)` для копирования данных с жесткого диска, смонтированного в виде файловой системы `ext3`, на гибкий диск, смонтированный в виде файловой системы `ext2`; две различные файловые системы, два разных носителя, а VFS — одна

Общий интерфейс файловых систем

Подсистема VFS выполняет роль связующего звена, которое позволяет вызовам таких системных функций, как `open()`, `read()` и `write()`, работать независимо от типа файловой системы и физической среды носителя информации. В наши дни такая возможность не особенно впечатляет, поскольку она воспринимается как должное. Тем не менее сделать так, чтобы вызовы универсальных системных функций работали для всех поддерживаемых файловых систем и физических сред хранения данных, — задача не тривиальная. Более того, эти системные функции позволяют выполнять операции *между* различными файловыми системами и различными физическими носителями данных. Так, с помощью вызовов стандартных системных функций мы можем копировать или перемещать данные с одной файловой системы на другую. В устаревших операционных системах (таких, как, например, DOS) таких возможностей не было. Любые операции доступа к "чужеродным" файловым системам требовали использования специальных утилит. Сейчас такие возможности существуют, потому что во всех современных операционных системах, включая Linux, используется абстрактный доступ к файловым системам через виртуальный интерфейс. Он позволяет выполнять операции между файловыми системами и предоставляет универсальные средства 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. В этой главе мы рассмотрим подсистему VFS, с помощью которой организуется абстрактный уровень доступа к данным, позволяющий множеству файловых систем работать как одно целое. В следующей главе мы обсудим уровень блочного ввода-вывода, благодаря которому становится возможной работа различных устройств хранения данных, таких как накопители на компакт-дисках или Blu-ray, жесткие диски или твердотельные устройства типа CompactFlash. Благодаря VFS и уровню блочного ввода-вывода для пользовательских приложений создается своего рода связующее звено, или абстрактная среда, а также универсальный набор интерфейсов. Это позволяет использовать вызовы универсальных системных функций для доступа к файлам через унифицированное соглашение о присвоении имен, расположенных в любой файловой системе, которая в свою очередь может находиться на любом типе носителя.

Абстрактный уровень файловой системы

Подобный обобщенный интерфейс для всех типов файловых систем стал возможен только благодаря тому, что в ядре реализован абстрактный уровень, скрывающий низкоуровневый интерфейс файловых систем. Благодаря ему в операционной системе Linux поддерживаются различные типы файловых систем, даже если эти файловые системы существенно отличаются друг от друга своими функциональными возможностями и особенностями работы. Это в свою очередь становится возможным благодаря тому, что в подсистеме VFS реализована обобщенная файловая модель, которая в состоянии представить общие функции и особенности работы потенциально возможных файловых систем. Разумеется, эта модель имеет уклон в сторону файловых систем в стиле Unix (об этом мы поговорим в следующем разделе). Несмотря на это, в ОС Linux поддерживается довольно большой диапазон различных файловых систем, начиная с FAT системы DOS и NTFS системы Windows и заканчивая специфичными для Unix и Linux файловыми системами.

На абстрактном уровне определяются основные обобщенные интерфейсы и структуры данных, которые нужны для поддержки всех файловых систем. В коде поддержки каждой файловой системы нужно сформировать все концепции своей работы в соответствии с шаблонными требованиями подсистемы VFS, например, "*так открываем файл*", а "*так представляем каталог*". Все детали реализации скрыты в коде конкретной файловой системы. По отношению к уровню VFS и остальным частям ядра все файловые системы выглядят одинаково. Другими словами, все файловые системы должны поддерживать такие объекты, как файлы и каталоги, и такие операции, как создание и удаление файла.

В результате формируется общий абстрактный уровень, который позволяет ядру легко и просто поддерживать множество типов файловых систем. В коде файловых систем должны поддерживаться общие интерфейсы и структуры данных, которые нужны для работы с виртуальной файловой системой. В свою очередь, ядро легко может работать со всеми типами файловых систем, и, соответственно, экспортируемый ядром интерфейс позволяет пользовательским приложениям также легко работать со всеми типами файловых систем.

По сути, 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:

```
ret = write (fd, buf, len);
```

В результате вызова системной функции `write()` в текущую позицию файла, представленного с помощью дескриптора `fd`, будет записано `len` байтов данных, расположенных в памяти по адресу `buf`. В этой системной функции сначала вызывается общая функция ядра `sys_write()`, в которой определяется реальная функция записи в файл для той файловой системы, на которой находится файл, представленный дескриптором `fd`. Далее в общей системной функции вызывается эта найденная функция, которая является частью кода реализации файловой системы и служит для записи данных на физический носитель (или для других действий, которые файловая система выполняет при записи файла). На рис. 13.2 показана схема прохождения данных при выполнении операции записи в файл, начиная с функции пользовательского интерфейса `write()` и заканчивая поступлением данных на физический носитель. С одной стороны системной функции мы видим обобщенный интерфейс VFS, который предоставляет внешний интерфейс для пользовательских приложений. А с другой стороны находится код, специфичный для файловой системы, в котором и скрыты все подробности ее реализации. В оставшейся части этой главы будет показано, как в подсистеме VFS удалось добиться необходимого уровня абстракции и какие для этого используются интерфейсы.

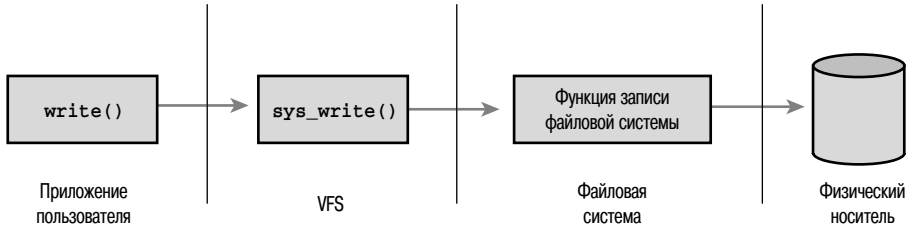


Рис. 13.2. Схема прохождения данных из пользовательского приложения, в котором вызывается функция `write()`, через вызов общей системной функции VFS, к специфической функции записи данных файловой системы и поступления их в конечном счете на физический носитель

Файловые системы Unix

Исторически так сложилось, что в ОС Unix используются четыре абстракции, связанные с файловыми системами: файлы, элементы каталогов (directory entry), индексы (inode) и точки монтирования (mount point).

Файловая система (filesystem) — это иерархическое хранилище данных определенной структуры. В файловой системе находятся файлы, каталоги и соответствующая управляющая информация. Типичными операциями, выполняемыми с файловыми системами, являются создание (create), удаление (delete) и монтирование (mount). В ОС Unix файловые системы подключаются к определенной точке монтирования, находящейся в общей иерархической структуре[1], которая называется *пространством имен* (namespace). Это позволяет расположить все смонтированные файловые системы в виде одной древовидной структуры. В отличие от этого единого и универсального дерева в таких системах, как DOS и Windows, используется другой подход. В нем все пространство имен файлов разбивается на отдельные логические устройства, для обозначения которых используется специальная буква, например `C:`. В результате все пространство имен оказывается разбитым на устройства и разделы, которые скорее относятся к физическим параметрам диска, а не к абстрактному уровню файловой системы. Поскольку такое разделение может выполняться произвольным образом и иногда сбивает пользователя с толку, его можно считать не самым удачным вариантом по сравнению с унифицированным пространством имен системы Linux.

Файл (file) — это упорядоченная последовательность байтов. Ее первый байт соответствует началу файла, а последний байт — концу файла. Каждому файлу присваивается удобочитаемое имя, по которому файл идентифицируется как пользователями, так и системой. Типичными для файлов являются операции чтения (read), записи (write), создания (create) и удаления (delete). Концепция файла в системе Unix кардинально отличается от файловых систем, ориентированных на записи, типа Files-11 операционной системы OpenVMS. Последние обеспечивают широкое и более структурированное представление файлов, чем та простая абстракция, основанная на потоке байтов, принятая в системе Unix, правда, за счет простоты и гибкости.

Файлы размещаются в каталогах (directory). *Каталог* — это некий аналог папки, в которой обычно содержатся связанные между собой файлы. Каталоги могут также содержать и другие каталоги, которые называются подкаталогами, или вложенными каталогами (subdirectories). Таким образом, вложенные друг в друга каталоги образуют *пути* (path). Каждый компонент пути называется *элементом каталога* (directory entry). Примером пути может служить `/home/wolfman/butter`, где символ `/` соответствует корневому каталогу, а каталоги `home`, `wolfman` и файл `butter` являются элементами каталогов, которые называются *dentries*. В операционной системе Unix каталоги представляют собой обычные файлы, в которых просто содержится список файлов каталога. Поскольку по отношению к VFS каталог — это файл, то с каталогами можно выполнять те же операции, что и с файлами.

В системах Unix различается сам файл и связанная с ним информация наподобие прав доступа, размера, владельца, времени создания и т.п. Последнюю иногда называют *ме-*

[1] В современной 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 только одно глобальное пространство имен.

таданными (metadata) файла (т.е. данные о данных файла). Метаданные хранятся 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 хранятся в *суперблоке* (superblock). Суперблок — это структура данных, которая содержит информацию о файловой системе в целом. Иногда эти общие данные называются *метаданными файловой системы* (filesystem metadata). Метаданные файловой системы содержат информацию об отдельных файлах и о файловой системе в целом.

Традиционно в файловых системах ОС Unix перечисленные выше элементы реализованы в виде структур данных, которые определенным образом расположены на физических дисках. Например, информация о файлах хранится в индексе, занимающем отдельный блок диска; каталоги являются файлами; служебная информация файловой системы хранится централизованно в суперблоке и т.д. Концепция файла в системах Unix подразумевает его *физическое отображение* на устройство хранения данных. Подсистема VFS операционной системы Linux рассчитана на работу с файловыми системами, в которых поддерживаются аналогичные концепции. Другие, отличные от Unix, файловые системы, такие как FAT и NTFS, также могут работать в Linux, но в их программном коде нужно обеспечить поддержку описанных выше концепций. Например, если в файловой системе не поддерживаются отдельные индексные узлы, то в коде нужно создать в оперативной памяти структуры данных таким образом, чтобы казалось, что такая поддержка работает. Или если в файловой системе каталоги являются объектами специального типа, то для VFS они должны представляться как обычные файлы. Часто чтобы удовлетворить требования VFS и соблюсти все принципы построения систем Unix, в коде чужеродных файловых систем приходится выполнять некоторую дополнительную обработку данных прямо на лету. Поэтому работа таких файловых систем поддерживается за счет привлечения дополнительных ресурсов системы (часто несущественных).

Объекты VFS и их структуры данных

Виртуальная файловая система (VFS) является объектно-ориентированной[2]. Общая файловая модель представлена набором структур данных. Эти структуры данных очень похожи на объекты. Поскольку программы для ядра Linux пишутся исключительно на языке C, то при отсутствии прямой поддержки средств ООП в этом языке программирования структуры данных объектов представляются структурами языка C. В них хранятся указатели как на элементы данных, так и на функции, которые работают с этими данными.

Ниже перечислены четыре основных типа объектов VFS.

- Объект *суперблок* (*superblock*), который представляет определенную смонтированную файловую систему.
- Объект *файловый индекс* (*inode*), который представляет определенный файл.

[2] Часто многие этого не замечают и даже отрицают, но тем не менее в ядре существует довольно много примеров использования объектно-ориентированного подхода в программировании. Хотя разработчики ядра и сторонятся языка C++ и других явно выраженных объектно-ориентированных языков программирования (ООП), иногда очень полезно мыслить в терминах объектов. Подсистема VFS — это хороший пример того, как просто и эффективно объектно-ориентированное программирование реализуется на языке C, в котором нет объектно-ориентированных средств.

- Объект *элемент каталога* (*dentry*), который представляет определенный элемент, являющийся компонентом пути.
- Объект *файл* (*file*), который представляет открытый файл, связанный с процессом.

Обратите внимание на то, что, поскольку в подсистеме VFS каталоги рассматриваются как обычные файлы, не существует специальных объектов для каталогов. Как уже было сказано выше, с помощью объекта `dentry` представляется отдельный компонент пути, который может быть и обычным файлом. Другими словами, `dentry` — это не то же самое, что каталог, а каталог — это всего лишь еще одна разновидность файла. Вам все понятно?

В каждом из рассмотренных выше основных объектов содержится объект *операций* (operations). С помощью объектов этого типа описываются методы, которые ядро может применять для работы с основными объектами. В частности, существуют перечисленные ниже 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 `inode_operations` (операции с файловыми индексами) содержит методы, которые ядро может вызывать для определенного файла, как, например, `create()` 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 выполняют операции над родительским объектом (т.е. объектом, к которому относится данный объект операций). Для многих методов объекты операций могут унаследовать общую функцию, если для работы достаточно базовой функциональности. В противном случае в каждой файловой системе в объекте операций указываются адреса своих специальных методов.

Повторим еще раз: под *объектами* мы будем понимать структуры данных, которые явно не связаны с соответствующими классами или объектными типами (в отличие от языков программирования C++ и Java). Тем не менее эти структуры представляют определенные экземпляры объектов, данные, связанные с объектами, и методы, которые ими оперируют. Это практически то же самое, что и объектные типы.

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 представлена в виде структуры `vfsmount`. В этой структуре содержится информация о точке монтирования, такая, как ее положение и флаги, с которыми выполнена операция монтирования.

И наконец, каждый процесс имеет две структуры, которые описывают файловую систему и файлы, связанные с процессом. Это структуры `fs_struct` и `file`.

Далее в главе мы рассмотрим перечисленные выше объекты и их роль в функционировании уровня VFS.

Объект *суперблок*

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). Для тех файловых систем, которые не располагаются на дисках, а, например, находятся в виртуальной памяти, как `sysfs`, информация для суперблока должна создаваться на лету и храниться в памяти.

Объект *суперблок* представляется с помощью структуры `super_block`, которая определена в файле `<linux/fs.h>`. Эта структура вместе с описанием полей в виде комментариев приведена ниже.

```
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 char             s_blocksize_bits; /* Размер блока в битах */
    unsigned char             s_dirt;      /* Признак изменения */
    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_need_sync; /* Признак отсутствия
                                              синхронизации */
    atomic_t                  s_active;    /* Счетчик активных ссылок */
    void                     *s_security;  /* Модуль безопасности */
    struct xattr_handler    **s_xattr;     /* Обработчики расширенных
                                              атрибутов */
    struct list_head          s_inodes;    /* Список индексов */
    struct list_head          s_dirty;     /* Список измененных индексов */
    struct list_head          s_io;        /* Список отложенной записи */
    struct list_head          s_more_io;   /* Дополнительный список
                                              отложенной записи */
    struct hlist_head         s_anon;      /* Анонимные элементы каталога
                                              для экспортирования */
    struct list_head          s_files;     /* Список связанных файлов */
    struct list_head          s_dentry_lru; /* Список неиспользуемых dentries
*/
    int                       s_nr_dentry_unused; /* Число dentries в списке
*/
    struct block_device      *s_bdev;      /* Связанное блочное устройство */
    struct mtd_info          *s_mtd;       /* Информация о memory disk */
    struct list_head          s_instances; /* Экземпляры этой fs */
    struct quota_info         s_dquot;     /* Параметры квот */
```

```
    int                         s_frozen;    /* Состояние замораживания */
    wait_queue_head_t           s_wait_unfrozen; /* Очередь ожидания
                                                    приостановленных задач */
    char                        s_id[32];    /* Текстовое имя */
    void                       *s_fs_info;   /* Специфическая информация
                                                файловой системы */
    fmode_t                     s_mode;      /* Права доступа для монтирования
*/
    struct semaphore            s_vfs_rename_sem; /* Семафор переименования */
    u32                         s_time_gran; /* Точность временной метки */
    char                       *s_subtype;   /* Имя подтипа */
    char                       *s_options;   /* Сохраненные параметры
                                                монтирования */
};
```

Код для создания, управления и ликвидации объектов *суперблок* находится в файле `fs/super.c`. Объект *суперблок* создается и инициализируется в функции `alloc_super()`, которая вызывается при монтировании файловой системы. Она считывает суперблок файловой системы с диска и заполняет все поля его объекта.

Операции суперблока

Самым важным элементом суперблока является поле `s_op`, в котором содержится указатель на его таблицу операций. Последняя представляется с помощью структуры `super_operations` и определяется 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 (*dirty_inode) (struct inode *);
   int (*write_inode) (struct inode *, int);
   void (*drop_inode) (struct inode *);
   void (*delete_inode) (struct inode *);
   void (*put_super) (struct super_block *);
   void (*write_super) (struct super_block *);
   int (*sync_fs)(struct super_block *sb, int wait);
   int (*freeze_fs) (struct super_block *);
   int (*unfreeze_fs) (struct super_block *);
   int (*statfs) (struct dentry *, struct kstatfs *);
   int (*remount_fs) (struct super_block *, int *, char *);
   void (*clear_inode) (struct inode *);
   void (*umount_begin) (struct super_block *);
   int (*show_options)(struct seq_file *, struct vfsmount *);
   int (*show_stats)(struct seq_file *, struct vfsmount *);
   ssize_t (*quota_read)(struct super_block *, int, char *, size_t, loff_t);
   ssize_t (*quota_write)(struct super_block *, int, const char *, size_t,
loff_t);
   int (*bdev_try_to_free_page)(struct super_block*, struct page*, gfp_t);
};
```

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.

Как только для файловой системы требуется выполнить операции с собственным суперблоком, сначала из объекта *суперблок* считывается адрес таблицы операций, а затем определяется указатель на нужный метод. Например, запись суперблока выполняется с помощью приведенной ниже функции.

```
sb->s_op->write_super(sb);
```

В этом фрагменте кода переменная sb является указателем на суперблок файловой системы. Следуя по указателю s_op, получаем таблицу операций суперблока и, наконец, адрес необходимой функции write_super(), которая и вызывается. Обратите внимание на то, что функции write_super() передается в качестве параметра указатель на суперблок, несмотря на то, что сам метод связан с этим суперблоком. Дело в том, что язык программирования C не является объектно-ориентированным. В C++ аналогичный вызов функции выполняется следующим образом:

```
sb.write_super();
```

В языке C нет простого способа получить указатель на объект, для которого вызван метод, поэтому его необходимо передавать явно.

А теперь рассмотрим некоторые из операций суперблока, описанные с помощью структуры super_operations.

- `struct inode * alloc_inode(struct super_block *sb)`

  Создает и инициализирует новый объект файлового индекса, связанного с данным суперблоком.

- `void destroy_inode(struct inode *inode)`

  Уничтожает указанный объект индекса файла.

- `void dirty_inode(struct inode *inode)`

  Вызывается подсистемой VFS, если в индекс были внесены изменения. В файловых системах с ведением журнала (как, например, в ext3 или ext4) эта функция используется для обновления журнала.

- `void write_inode(struct inode *inode, int wait)`

  Записывает указанный индекс на диск. Параметр wait указывает, должна ли данная операция выполняться синхронно.

- `void drop_inode(struct inode *inode)`

  Вызывается подсистемой VFS, когда исчезает последняя ссылка на индекс. В обычных файловых системах Unix эта функция никогда не определяется. В таких случаях подсистема VFS просто удаляет индекс.

- `void delete_inode(struct inode *inode)`

  Удаляет указанный индекс файла с диска.

- `void put_super(struct super_block *sb)`

  Вызывается подсистемой VFS при размонтировании файловой системы для освобождения указанного суперблока. В вызывающем коде должна удерживаться блокировка s_lock.

- `void write_super(struct super_block *sb)`

  Обновляет суперблок на диске данными из указанного суперблока, находящимися в памяти. В подсистеме VFS эта функция вызывается для синхронизации измененного суперблока в памяти с данными суперблока на диске. В вызывающем коде должна удерживаться блокировка s_lock.

- `int sync_fs(struct super_block *sb, int wait)`

  Синхронизирует метаданные файловой системы с данными на диске. Параметр wait указывает, что операция должна выполняться синхронно.

- `void write_super_lockfs(struct super_block *sb)`

  Запрещает модификацию файловой системы и затем обновляет данные суперблока на диске данными из указанного суперблока в памяти. В настоящее время она используется диспетчером логических томов (LVM, Logical Volume Manager).

- `void unlockfs(struct super_block *sb)`

  Разблокирует файловую систему после выполнения функции `write_super_lockfs()`.

- `int statfs(struct super_block *sb, struct statfs *statfs)`

  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)`

  Вызывается подсистемой VFS, когда файловая система монтируется с другими параметрами монтирования. В вызывающем коде должна удерживаться блокировка `s_lock`.

- `void clear_inode(struct inode *inode)`

  Вызывается подсистемой VFS для освобождения индекса и очистки всех страниц памяти, связанных с индексом.

- `void umount_begin(struct super_block *sb)`

  Вызывается подсистемой VFS для прерывания операции монтирования. Она используется сетевыми файловыми системами, такими как NFS.

Все рассмотренные выше функции вызываются подсистемой VFS в контексте процесса. Все они, кроме функции `dirty_inode()`, могут переводить процесс в состояние ожидания.

Часть этих функций не является обязательной. В подобных случаях в структуре операций суперблока конкретной файловой системы вместо указателя на функцию находится значение `NULL`. Если соответствующий указатель на функцию равен `NULL`, то подсистема VFS или вызывает обобщенный вариант функции, или не делает ничего, в зависимости от выполняемой операции.

Объект `inode`

В объекте `inode` содержится вся информация, необходимая ядру для работы с файлами и каталогами. Для традиционных файловых систем Unix эта информация просто считывается с дисковых индексов и помещается в объект `inode` подсистемы VFS. Если в файловых системах индексные узлы отсутствуют, то необходимая информация считывается из других дисковых структур. Как правило, она хранится в самом файле. В отличие от файловых систем Unix в таких файловых системах не разделяются данные файла и служебная информация о файле. В некоторых современных файловых системах метаданные файла хранятся в базе данных, расположенной на диске. В любом случае объект индекса создается в оперативной памяти тем способом, который лучше всего подходит для конкретной файловой системы.

Объект индекса файла представляется с помощью структуры inode, которая определена в файле <linux/fs.h>. Эта структура с комментариями, описывающими назначение каждого поля, приведена ниже.

```
struct inode {
   struct   hlist_node i_hash;    /* Хешированный список */
   struct   list_head  i_list;    /* Список объектов inodes */
   struct   list_head  i_sb_list; /* Список объектов суперблока */
   struct   list_head  i_dentry;  /* Список объектов dentry */
   unsigned long       i_ino;     /* Номер индекса */
   atomic_t            i_count;   /* Счетчик ссылок */
   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;    /* Узел реального устройства */
   u64                 i_version; /* Номер версии */
   loff_t              i_size;    /* Размер файла в байтах */
   seqcount_t          i_size_seqcount; /* Сериализатор для i_size */
   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 */
   blkcnt_t            i_blocks;  /* Размер блока в байтах */
   unsigned short      i_bytes;   /* Количество использованных байтов */
   umode_t             i_mode;    /* Права доступа */
   spinlock_t          i_lock;    /* Спин-блокировка */
   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; /* Стандартные операции с индексом */
   struct super_block *i_sb;      /* Связанный суперблок */
   struct file_lock   *i_flock;   /* Список блокировок файлов */
   struct address_space *i_mapping; /* Связанное отображение */
   struct address_space  i_data;  /* Отображение для устройства */
   struct dquot *i_dquot[MAXQUOTAS]; /* Дисковые квоты для индекса */
   struct list_head    i_devices; /* Список блочных устройств */
   union {
            struct pipe_inode_info *i_pipe; /* Информация конвейера */
            struct block_device    *i_bdev; /* Драйвер блочного
                                               устройства */
            struct cdev            *i_cdev; /* Драйвер символьного
                                               устройства */
         };
   unsigned long       i_dnotify_mask; /* Маска уведомления о
                                          событиях каталога */
   struct dnotify_struct *i_dnotify;   /* Уведомления о событиях
                                          каталога */
   struct list_head    inotify_watches; /* Наблюдателя событий */
   struct mutex        inotify_mutex;   /* Защищает поле
                                           inotify_watches */
   unsigned long       i_state;    /* Флаги состояния */
   unsigned long  dirtied_when;    /* время первого изменения */
   unsigned int        i_flags;    /* Флаги файловой системы */
   atomic_t            i_writecount; /* Счетчик использования
                                        по записи */
   void               *i_security; /* модуль безопасности */
   void               *i_private;  /* Приватный указатель файловой системы */
};
```

Для каждого файла в файловой системе существует представляющий его объект inode. Однако сам файловый индекс создается в памяти только в момент доступа к файлу. Это справедливо и для специальных файлов, таких как файлы устройств или конвейеры. Следовательно, некоторые из полей структуры inode связаны с этими специальными файлами. Например, поле i_pipe указывает на структуру данных именованного конвейера, поле i_bdev — на структуру данных блочного устройства, а поле i_cdev — на символьное устройство. Эти три поля сохраняются в структуре inode в виде объединения (union), поскольку один конкретный индекс в данный момент времени может быть связан только с одним из перечисленных выше типов устройств (либо ни с одним из них).

Может оказаться, что та или иная файловая система не поддерживает всех тех свойств, которые присутствуют в объекте inode. Например, в некоторых файловых системах не фиксируется время последнего обращения к файлу. В подобных случаях в файловой системе это свойство может быть реализовано произвольным образом. Например, в поле i_atime может храниться нулевое значение, или оно может быть равным значению поля i_mtime. При этом поле i_atime будет обновляться в оперативной памяти, но оно никогда не будет сбрасываться на диск или на другой носитель, связанный с конкретной файловой системой.

Операции с файловыми индексами

По аналогии с операциями суперблока для операций с файловыми индексами важным является поле inode_operations. В нем содержится указатель на таблицу функций, реализуемых для конкретной файловой системы, которые могут быть вызваны подсистемой VFS для объекта файлового индекса. Как и для суперблока, операции с файловыми индексами могут вызываться следующим образом:

```
i->i_op->truncate(i)
```

В этом фрагменте кода в переменной i содержится указатель на определенный объект файлового индекса. В данном случае для индекса i выполняется операция truncate(), определенная для той 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 nameidata *);
   struct dentry * (*lookup) (struct inode *,struct dentry *,
                              struct nameidata *);
   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 __user *,int);
   void * (*follow_link) (struct dentry *, struct nameidata *);
   void (*put_link) (struct dentry *, struct nameidata *, void *);
   void (*truncate) (struct inode *);
   int (*permission) (struct inode *, int);
   int (*setattr) (struct dentry *, struct iattr *);
   int (*getattr) (struct vfsmount *mnt, 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 *);
    void (*truncate_range)(struct inode *, loff_t, loff_t);
    long (*fallocate)(struct inode *inode, int mode,
                      loff_t offset, loff_t len);
    int (*fiemap)(struct inode *, struct fiemap_extent_info *,
                  u64 start, u64 len);
};
```

Ниже описаны интерфейсы разных функций, которые могут вызываться из подсистемы VFS. При этом VFS, по сути, "просит" указанную файловую систему выполнить нужную операцию с заданным индексом.

- `int create(struct inode *dir, struct dentry *dentry, int mode)`

  Вызывается подсистемой VFS из системных функций `create()` и `open()` для создания нового файлового индекса с указанным в параметре `mode` первоначальным режимом доступа, который связан с заданным в параметре `dentry` элементом каталога.

- `struct dentry * lookup(struct inode *dir, struct dentry *dentry)`

  Выполняет поиск файлового индекса в указанном каталоге. Файловый индекс должен соответствовать имени файла, хранящемуся в указанном объекте элемента каталога.

- `int link(struct dentry *old_dentry, struct inode *dir, struct dentry *dentry)`

  Вызывается из системной функции `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)`

  Вызывается из системной функции `unlink()` для удаления файлового индекса из каталога `dir`, соответствующего элементу каталога `dentry`.

- `int symlink(struct inode *dir, struct dentry *dentry, const char *symname)`

  Вызывается из системной функции `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)`

  Вызывается из системной функции `mkdir()` для создания нового каталога с указанным первоначальным режимом доступа `mode`.

- `int rmdir(struct inode *dir, struct dentry *dentry)`

  Вызывается из системной функции `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)`

Вызывается из системной функции 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 первоначальным режимом доступа mode.

- int rename(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry)

  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)

  Вызывается из системной функции readlink() для копирования не более buflen байт полного пути, связанного с символьной ссылкой, соответствующей указанному элементу каталога dentry, в указанный буфер.

- int follow_link(struct dentry *dentry, struct nameidata *nd)

  Вызывается подсистемой VFS для трансляции символьной ссылки в индекс файла, на который эта ссылка указывает. Ссылка задается указателем dentry, а результат сохраняется в структуре nameidata, на которую указывает параметр nd.

- int put_link(struct dentry *dentry, struct nameidata *nd)

  Вызывается подсистемой VFS после вызова функции follow_link() для выполнения очистки.

- void truncate(struct inode *inode)

  Вызывается подсистемой VFS для изменения размера заданного файла. Перед вызовом поле i_size указанного индекса файла должно содержать требуемый размер.

- int permission(struct inode *inode, int mask)

  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 поле указателя на данную функцию содержит значение NULL. При этом используется общий метод VFS, который просто сравнивает биты поля режима доступа файлового индекса с указанной маской. В более сложных файловых системах, в которых поддерживаются списки контроля доступа (ACL), реализован свой метод permission().

- int setattr(struct dentry *dentry, struct iattr *attr)

  Вызывается из системной функции 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)

  Вызывается подсистемой VFS после получения уведомления, что индекс должен быть обновлен с диска.

- `int setxattr(struct dentry *dentry, const char *name, const void *value, size_t size, int flags)`

  Вызывается подсистемой VFS для установки одного из *расширенных атрибутов* (extended attributes) с именем `name` в значение `value` для файла, соответствующего элементу каталога `dentry`. Расширенные атрибуты — это новая возможность, которая появилась в ядре серии 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 еще используются не достаточно широко.

- `ssize_t getxattr(struct dentry *dentry, const char *name, void *value, size_t size)`

  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)`

  Копирует список всех атрибутов для указанного файла в буфер, соответствующий параметру `list`.

- `int removexattr(struct dentry *dentry, const char *name)`

  Удаляет указанный атрибут для указанного файла.

Объект элемента каталога (dentry)

Как уже отмечалось выше, в подсистеме VFS каталоги обрабатываются как особый тип файла. В пути `/bin/vi` и элемент `bin`, и элемент `vi` — это файлы, только `bin` — это особый файл, который является каталогом, а `vi` — обычный. Оба этих типа файлов представляются в виде объекта файлового индекса (inode). Несмотря на такую полезную унификацию, в подсистеме VFS также необходимо выполнять операции, специфичные для каталогов, такие как поиск компонента пути по его имени. При этом нужно проанализировать каждый компонент пути, проверить, что он существует, и перейти к следующему компоненту.

Для решения этой задачи в подсистеме VFS реализована концепция элемента каталога (directory entry, или dentry). *Элемент каталога* является особым компонентом пути. Например, все компоненты из предыдущего примера (`"/"`, `"bin"` и `"vi"`) являются элементами каталога, или объектами dentry. Причем первые два — это каталоги, а последний — обычный файл. Важный момент заключается в том, что все элементы пути, включая файлы, являются объектами dentry. Анализ компонентов пути и их обход являются довольно нетривиальной задачей, требующей больших затрат времени и активного использования операций со строками. Для реализации последних требуется довольно большой объем громоздкого кода. Использование объекта dentry позволяет упростить весь процесс.

Среди элементов каталога также могут находиться точки монтирования. Например, в пути `/mnt/cdrom/foo` элемент корневого каталога `"mnt"` является точкой монтирования, при этом все компоненты (`"/"`, `"mnt"`, `"cdrom"` и `"foo"`) являются объектами `dentry`. Они создаются в подсистеме VFS на лету по мере необходимости во время выполнения операций с каталогами.

Объекты элементов каталога представляются в виде структуры dentry и определены в файле <linux/dcache.h>. Эта структура с комментариями, поясняющими назначение каждого поля, приведена ниже.

```
struct dentry {
   atomic_t          d_count;   /* Счетчик использования */
   unsigned int      d_flags;   /* Флаги объекта dentry */
   spinlock_t        d_lock;    /* Поэлементная блокировка */
   int               d_mounted; /* Является ли объект точкой монтирования */
   struct inode     *d_inode;   /* Связанный inode */
   struct hlist_node d_hash;    /* Список хеширования */
   struct dentry    *d_parent;  /* Объект dentry родительского каталога */
   struct qstr       d_name;    /* Имя dentry */
   struct list_head  d_lru;     /* Список неиспользуемых элементов */
   union {
      struct list_head d_child; /* Список dentries в каталоге */
      struct rcu_head  d_rcu;   /* Блокировка RCU */
   } d_u;
   struct list_head  d_subdirs; /* Список подкаталогов */
   struct list_head  d_alias;   /* Список альтернативных индексов */
   unsigned long     d_time;    /* Время проверки правильности */
   struct dentry_operations *d_op; /* Таблица операций элемента каталога */
   struct super_block       *d_sb; /* Суперблок файла */
   void             *d_fsdata;  /* Специфические данные файловой системы */
   unsigned char     d_iname[DNAME_INLINE_LEN_MIN]; /* короткое имя файла */
};
```

В отличие от описанных выше двух предыдущих 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, в структуре dentry нет никаких флагов, которые указывают на то, изменен ли объект (т.е. должен ли он быть записан снова на диск).

Состояние элементов каталога

Корректный объект элемента каталога может находиться в одном из трех состояний: *используемый* (used), *неиспользуемый* (unused) и *противоречивый* (negative).

Используемый элемент каталога соответствует существующему файловому индексу (т.е. поле d_inode указывает на связанный объект типа inode), и в поле d_count содержится положительное значение (т.е. он используется один или более раз). Используемый элемент каталога задействован в подсистеме VFS, кроме того, он также указывает на существующие данные, поэтому не может быть удален.

Неиспользуемый элемент каталога также 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). Поскольку элемент каталога продолжает указывать на существующий объект, он сохраняется на случай, если вдруг окажется нужным. Если элемент каталога не будет сразу ликвидирован, то его и не нужно будет создавать заново, если вдруг он понадобится в будущем. Очевидно, что поиск элемента в пути будет выполняться быстрее, когда объект dentry создан заранее. Когда же появляется необходимость освободить память, такой объект элемента каталога может быть удален, потому что он никем не используется.

Элемент каталога в противоречивом состоянии 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. Тем не менее такие объекты элементов каталогов сохраняются, чтобы в будущем поиск элемента в пути по имени проходил быстрее. В качестве примера рассмотрим фоновый серверный процесс (daemon), в котором постоянно выполняются попытки открытия и чтения несуществующего файла конфигурации. При этом из системной функции `open()` будет сразу же возвращаться код ошибки `ENOENT`. Ядру больше не нужно будет каждый раз создавать элементы пути, анализировать дисковые структуры и проверять, существует ли файл. Поскольку даже такая операция поиска несуществующего файла требует достаточно большого количества ресурсов, запоминание "отрицательных" результатов имеет определенный смысл. Хотя противоречивые элементы каталога и могут пригодиться, но при отсутствии достаточного количества оперативной памяти их можно удалить, поскольку на самом деле они нигде не используются.

Несмотря на то что объект элемента каталога может быть освобожден, при этом он остается в кеше блочного распределителя памяти, как обсуждалось в предыдущей главе. В подобных случаях все ссылки на него из любой части подсистемы VFS или кода любой файловой системы являются некорретными.

Кеш объектов dentry

После того как в подсистеме VFS были преодолены все трудности, связанные с переводом всех элементов пути в объекты элементов каталогов, и был достигнут конец пути, было бы просто расточительным выбрасывать на ветер всю проделанную работу. Поэтому объекты элементов каталога сохраняются в ядре в специальной кеш-памяти, которую называют *dcache*.

Кеш объектов dentry состоит из трех частей, описанных ниже.

- Список "используемых" объектов dentry, которые связаны с определенным файловым индексом через поле `i_dentry` объекта inode. Поскольку указанный файловый индекс может иметь несколько ссылок, ему может соответствовать несколько объектов `dentry`, а следовательно, используется связанный список.
- Двухсвязный список неиспользуемых и противоречивых объектов dentry, упорядоченный по времени использования (last recently used, LRU). Вставка элементов в этот список выполняется с головы, поэтому самые новые элементы находятся в его начале. Как только ядру нужно удалить элементы каталогов для освобождения памяти, эти элементы берутся из конца списка. Там находятся элементы, которые использовались очень давно и для которых меньше шансов, что они понадобятся в ближайшем будущем.
- Хеш-таблица и хеш-функция, которые позволяют быстро преобразовать заданный путь в связанный объект dentry.

Указанная хеш-таблица представлена с помощью массива `dentry_hashtable`. Каждый элемент массива — это указатель на список тех объектов dentry, которые соответствуют одному значению ключа. Размер этого массива зависит от объема физической памяти в системе.

Значение ключа вычисляется с помощью функции `d_hash()`. Это позволяет для каждой файловой системы использовать свою хеш-функцию.

Поиск в хеш-таблице выполняется с помощью функции d_lookup(). Если в кеше найден соответствующий объект, то возвращается указатель на него. В противном случае возвращается значение NULL.

В качестве примера предположим, что вы редактируете файл с исходным кодом программы, расположенный в вашем рабочем каталоге, полный путь которого /home/dracula/src/the_sun_sucks.c. Каждый раз при доступе к этому файлу (например, при первом открытии, при последующей записи, при компиляции и т.д.) в подсистеме VFS нужно последовательно проанализировать каждый элемент пути: "/", "home", "dracula", "src" и только затем the_sun_sucks.c. Для того чтобы каждый раз при доступе к этому (и любому другому) файлу избежать выполнения данной операции, которая требует довольно больших затрат времени и обращения к диску, в подсистеме VFS вначале выполняется поиск этого пути в dentry-кеше. Если поиск выполнен успешно, то необходимый конечный элемент каталога нужного пути находится очень быстро. Если же данного элемента каталога нет в dentry-кеше, то в подсистеме VFS нужно проанализировать все элементы пути, обратившись к файловой системе. После завершения поиска найденные объекты dentry помещаются в кеш для ускорения поиска в будущем.

Кеш dcache также является *внешним интерфейсом* (front end) к кешу файловых индексов *icache*. Объекты inode, связанные с объектами dentry, не освобождаются до тех пор, пока значение их счетчика использования больше нуля. Это, в свою очередь, позволяет объектам dentry удерживать связанные с ними объекты inode в памяти. Иными словами, если элемент каталога находится в кеш-памяти, то соответствующий ему файловый индекс также находится в кеше. Следовательно, если поиск в кеше некоторого имени пути прошел успешно, соответствующие файловые индексы также уже будут находиться в кеш-памяти.

Кеширование элементов каталога и файловых индексов крайне желательно, поскольку доступ к файлу — это операция, выполняемая как в пространстве, так и во времени. Мы говорим, что доступ к файлу выполняется во времени, подразумевая, что разные программы могут получать доступ к одним и тем же файлам много раз. Поэтому после первого обращения к файлу существует очень высокая вероятность того, что находящиеся в кеш-памяти элементы каталога и связанные с ними файловые индексы будут там находиться на протяжении некоторого интервала времени. Говоря о том, что доступ к файлу выполняется в пространстве, мы подразумеваем, что программы могут получать доступ к разным файлам, хранящимся в одном и том же каталоге. Поэтому если сохранить в кеше все элементы пути для одного файла, то это существенно ускорит процесс доступа к файлу, находящемуся с ним в одном каталоге.

Операции с элементами каталогов

Методы, которые подсистема VFS может вызывать для элементов каталогов определенной файловой системы, определяются в структуре dentry_operations. Последняя определена в файле <linux/dcache.h> и имеет следующий вид:

```
struct dentry_operations {
   int (*d_revalidate) (struct dentry *, struct nameidata *);
   int (*d_hash) (struct dentry *, struct qstr *);
   int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
   int (*d_delete) (struct dentry *);
   void (*d_release) (struct dentry *);
   void (*d_iput) (struct dentry *, struct inode *);
   char *(*d_dname) (struct dentry *, char *, int);
};
```

Ниже приведено описание методов.

- `int d_revalidate(struct dentry *dentry, struct nameidata *)`

  Определяет, является ли указанный объект элемента каталога действительным. Подсистема VFS вызывает эту функцию, когда она пытается использовать объект dentry из кеша dcache. Для большинства файловых систем в поле указателя на этот метод содержится значение `NULL`, потому что объекты `dentry`, которые находятся в кеше, всегда действительны.

- `int d_hash(struct dentry *dentry,struct qstr *name)`

  Вычисляет значение хеш-ключа на основании указанного объекта dentry. В подсистеме VFS эта функция вызывается каждый раз, когда объект элемента каталога добавляется в хеш-таблицу.

- `int d_compare(struct dentry *dentry, struct qstr *name1, struct qstr *name2)`

  Вызывается подсистемой VFS для сравнения двух имен файлов `name1` и `name2`. В большинстве файловых систем для этой цели используется стандартная процедура VFS, в которой выполняется обычное сравнение двух строк. Для некоторых файловых систем, таких как FAT, такого сравнения строк недостаточно. Дело в том, что в файловой системе FAT имена файлов не зависят от регистра символов и на диске всегда представляются в виде прописных букв. В подобных случаях нужно реализовать функцию, в которой бы при сравнении двух строк не учитывался регистр символов. Эта функция вызывается при захваченной блокировке `dcache_lock`.

- `int d_delete (struct dentry *dentry)`

  Вызывается подсистемой VFS, когда количество ссылок `d_count` указанного объекта dentry становится равным нулю. Функция вызывается при захваченной блокировке `dcache_lock`, а также `dentry->d_lock`.

- `void d_release(struct dentry *dentry)`

  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)`

  Вызывается подсистемой VFS, когда элемент каталога теряет связь со своим файловым индексом (например, когда этот элемент каталога удаляется с диска). По умолчанию в подсистеме VFS просто вызывается функция `iput()`, чтобы освободить соответствующий объект inode. Если в файловой системе функция `d_iput()` переопределена, то, кроме выполнения специфичной для конкретной файловой системы работе, также нужно вызвать функцию `iput()`.

Файловый объект

Нам осталось рассмотреть последний из основных объектов подсистемы VFS — файловый объект, который используется для представления файла, открытого процессом. Рассматривая подсистему VFS с точки зрения пользовательской программы, первым приходит в голову файловый объект. Процессы непосредственно работают с файлами, а не с суперблоками, индексами или элементами каталогов. Нет ничего удивительного в том,

что информация, содержащаяся в файловом объекте, вам хорошо знакома (например, режим доступа или текущее смещение), а файловые операции очень похожи на знакомые вызовы системных функций, таких как `read()` и `write()`.

Файловый объект представляет открытый файл в памяти компьютера. Этот объект (а не сам файл) создается в результате вызова системной функции `open()` и уничтожается в результате вызова функции `close()`. На самом деле все функции для работы с файлами являются методами, которые определены в таблице файловых операций. Поскольку один и тот же файл одновременно может быть открыт в нескольких процессах, для одного файла может существовать несколько файловых объектов. С точки зрения процесса файловый объект просто представляет открытый файл. В этом объекте содержится указатель на соответствующий элемент каталога (который, в свою очередь, указывает на файловый индекс), представляющий открытый файл. Разумеется, соответствующие объекты файлового индекса и элемента каталога являются уникальными.

Файловый объект представляется с помощью структуры `file`, которая определена в файле `<linux/fs.h>`. Эта структура приведена ниже, а в комментариях описано назначение каждого поля.

```
struct file {
   union {
   struct list_head fu_list;    /* Список файловых объектов */
   struct rcu_head  fu_rcuhead; /* Список RCU после освобождения */
         } f_u;
   struct path      f_path;     /* Содержит объект dentry */
   struct file_operations *f_op; /* Таблица файловых операций */
   spinlock_t       f_lock;     /* Блокировка на уровне файла */
   atomic_t         f_count;    /* Счетчик ссылок на файловый объект */
   unsigned int     f_flags;    /* Флаги, указанные при вызове функции open
*/
   mode_t           f_mode;     /* Режим доступа к файлу */
   loff_t           f_pos;      /* Текущий указатель файла (смещение) */
   struct fown_struct f_owner;  /* Информация о владельце для обработки сиг-
налов */
   const struct cred *f_cred;   /* Параметры доступа к файлу */
   struct file_ra_state f_ra;   /* Состояние упреждающей загрузки */
   u64              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 кеше */
   unsigned long    f_mnt_write_state; /* Состояние при отладке */
};
```

Так же как и объект элемента каталога, файловый объект не соответствует никакой структуре, хранящейся на жестком диске. Поэтому в приведенной выше структуре нет никакого флага, который бы указывал на то, что объект изменен (dirty) и требует сохранения на диск. В структуре `f_path` (поле `dentry`) файлового объекта хранится указатель на связанный с ним объект элемента каталога. Объект dentry в свою очередь содержит указатель на связанный с ним файловый индекс, в котором хранится информация о том, изменен ли файл.

Файловые операции

Как и для других объектов подсистемы VFS, таблица файловых операций является важной структурой. Операции, связанные со структурой `file`, вам должны быть хорошо

знакомы, поскольку эти функции положены в основу стандартных системных функций 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 __user *, size_t, loff_t *);
   ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
   ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                                        unsigned long, loff_t);
   ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                                        unsigned long, 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);
   long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
   long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
   int (*mmap) (struct file *, struct vm_area_struct *);
   int (*open) (struct inode *, struct file *);
   int (*flush) (struct file *, fl_owner_t id);
   int (*release) (struct inode *, struct file *);
   int (*fsync) (struct file *, struct dentry *, int datasync);
   int (*aio_fsync) (struct kiocb *, int datasync);
   int (*fasync) (int, struct file *, int);
   int (*lock) (struct file *, int, struct file_lock *);
   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);
   int (*flock) (struct file *, int, struct file_lock *);
   ssize_t (*splice_write) (struct pipe_inode_info *,
                                       struct file *,
                                            loff_t *,
                                            size_t,
                                       unsigned int);
   ssize_t (*splice_read) (struct file *,
                                  loff_t *,
                 struct pipe_inode_info *,
                                  size_t,
                            unsigned int);
   int (*setlease) (struct file *, long, struct file_lock **);
};
```

В зависимости от файловой системы каждая из перечисленных выше операций может иметь свою уникальную реализацию либо использовать универсальный метод, если последний существует. Универсальные методы отлично работают в стандартных Unix-подобных файловых системах. Разработчики файловых систем не обязаны реализовать все перечисленные выше операции, хотя основные методы должны быть реализованы. Если какой-либо метод не представляет интереса, то вместо ссылки на него в структуре `file_operations` задается значение `NULL`.

Рассмотрим каждую операцию подробнее.

- `loff_t llseek(struct file *file, loff_t offset, int origin)`

  Изменяет значение указателя текущей позиции в файле (file pointer), определяемое значением параметра `offset`. Функция вызывается из системной функции `lseek()`.

- `ssize_t read(struct file *file, char *buf, size_t count, loff_t *offset)`

  Считывает `count` байт данных из указанного файла, начиная с позиции, заданной параметром `offset`, в буфер памяти, на который указывает параметр `buf`. После этого обновляется значение указателя текущей позиции в файле. Данная функция вызывается из системной функции `read()`.

- `ssize_t aio_read(struct kiocb *iocb, char *buf, size_t count, loff_t offset)`

  Запускает асинхронную операцию считывания `count` байтов данных из файла, который описывается параметром `iocb`, в буфер памяти, описанный параметром `buf`. Эта функция вызывается из системной функции `aio_read()`.

- `ssize_t write(struct file *file, const char *buf, size_t count, loff_t *offset)`

  Записывает `count` байтов данных в указанный файл, начиная с позиции `offset`. Данная функция вызывается из системной функции `write()`.

- `ssize_t aio_write(struct kiocb *iocb, const char *buf, size_t count, loff_t offset)`

  Запускает асинхронную операцию записи `count` байтов данных в файл, описываемый параметром `iocb`, из буфера памяти, на который указывает параметр `buf`. Данная функция вызывается из системной функции `aio_write`.

- `int readdir(struct file *file, void *dirent, filldir_t filldir)`

  Возвращает следующий элемент из списка содержимого каталога. Данная функция вызывается из системной функции `readdir()`.

- `unsigned int poll(struct file *file, struct poll_table_ struct *poll_table)`

  Переводит вызывающий процесс в состояние ожидания до завершения выполнения операций с указанным файлом. Она вызывается из системной функции `poll()`.

- `int ioctl(struct inode *inode, struct file *file, unsigned int cmd, unsigned long arg)`

  Используется для отправки управляющих данных устройству в виде пар значений команда–аргумент. Для работы функции должен быть открыт специальный файл устройства. Функция вызывается из системной функции `ioctl()`. В вызывающем коде нужно захватить большую блокировку ядра BKL.

- `int unlocked_ioctl(struct file *file, unsigned int cmd, unsigned long arg)`

  Аналогична по функциям `ioctl()`, но при этом не требуется захватывать большую блокировку ядра BKL. В подсистеме VFS функция `unlocked_ioctl()`

вызывается (если она существует) вместо функции `ioctl()`, в ответ на вызов из пользовательского приложения системной функции `ioctl()`. Таким образом, в файловой системе требуется реализовать только одну из функций, предпочтительнее `unlocked_ioctl()`.

- `int compat_ioctl(struct file *file, unsigned int cmd, unsigned long arg)`

  Представляет собой переносимый вариант функции `ioctl()` для использования 32-разрядными приложениями на 64-разрядных системах. Эта функция должна быть спроектирована так, чтобы ее можно было безопасно вызывать из 32-разрядных приложений даже на 64-разрядных системах (при этом должны выполняться все необходимые преобразования размеров). В новых драйверах устройств должны быть реализованы собственные переносимые функции `ioctl()`. Поэтому методы `compat_ioctl()` и `unlocked_ioctl()` могут соответствовать одной и той же функции. Как и `unlocked_ioctl()`, для работы функции `compat_ioctl()` не требуется удерживать большую блокировку ядра BKL.

- `int mmap(struct file *file, struct vm_area_struct *vma)`

  Отображает указанный файл на область памяти в указанном адресном пространстве и вызывается из системной функции `mmap()`.

- `int open(struct inode *inode, struct file *file)`

  Создает новый файловый объект и связывает его с указанным файловым индексом. Она вызывается из системной функции `open()`.

- `int flush(struct file *file)`

  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)`

  Вызывается подсистемой VFS, когда исчезает последняя ссылка на файл. Например, такое может произойти, когда в последнем процессе, в котором использовался соответствующий файловый дескриптор, вызывается функция `close()` или этот процесс завершается. Назначение этой функции также зависит от файловой системы.

- `int fsync(struct file *file, struct dentry *dentry, int datasync)`

  Вызывается из системной функции `fsync()` для записи на диск всех данных файла, хранящихся в буфере в оперативной памяти.

- `int aio_fsync(struct kiocb *iocb, int datasync)`

  Вызывается из системной функции `aio_fsync()` для записи на диск всех данных файла, хранящихся в буфере в оперативной памяти, связанного с параметром `iocb`.

- `int fasync(int fd, struct file *file, int on)`

  Разрешает или запрещает отправку сигнала для уведомлении о событиях, возникающих при асинхронном вводе-выводе.

- `int lock(struct file *file, int cmd, struct file_lock *lock)`

Управляет файловыми блокировками для данного файла.

- `ssize_t readv(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)`

  Вызывается из системной функции `readv()` для считывания данных из указанного файла в `count` буферов, которые описываются параметром `vector`. После этого соответствующим образом обновляется значение указателя текущей позиции в файле.

- `ssize_t writev(struct file *file, const struct iovec *vector, unsigned long count, loff_t *offset)`

  Вызывается из системной функции `writev()` для записи в указанный файл буферов, описанных параметром `vector`; количество буферов определяется параметром `count`. После этого соответствующим образом обновляется значение указателя текущей позиции в файле.

- `ssize_t sendfile(struct file *file, loff_t *offset, size_t size, read_actor_t actor, void *target)`

  Вызывается из системной функции `sendfile()` и предназначена 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)`

  Используется для пересылки 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)`

  Определяет неиспользуемое пространство адресов для отображения данного файла.

- `int check_flags(int flags)`

  Используется для проверки корректности флагов, которые передаются системной функции `fcntl()`, при использовании команды `SETFL`. Как и в случае многих операций подсистемы VFS, для файловой системы нет необходимости реализовывать свою функцию `check_flags()`. Сейчас это сделано только для файловой системы NFS. Эта функция позволяет файловой системе ограничить некорректные значения флагов команды `SETFL`, которые допускаются в обобщенной системной функции `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)`

  Используется для реализации системной функции `flock()`, которая служит для выполнения *рекомендательных* (advisory) блокировок (т.е. блокировок, не взаимодействующих с подсистемой ввода-вывода).

Зачем столько вариантов метода `ioctl()`?

Не так давно существовал только один метод `ioctl()`, а на сегодняшний день существуют три варианта этого метода. Метод `unlocked_ioctl()` аналогичен `ioctl()` за исключением того, что он выполняется без захвата большой блокировки ядра BKL. Таким образом, все

вопросы достижения корректной синхронизации лежат на совести автора этой функции. Поскольку блокировка BKL является крупномасштабной и неэффективной, в драйверах должна использоваться функция `unlocked_ioctl()`, а не `ioctl()`.

Метод `compat_ioctl()` также выполняется без захвата большой блокировки ядра BKL. Основной целью его создания было предоставление 32-разрядного, совместимого с функцией `ioctl()` интерфейса для 64-разрядных систем. Конкретная реализация этого метода зависит от существующих ioctl-команд. В старых драйверах, где использовались не строго определенные размеры типов (например, `long`), должен быть реализован метод `compat_ioctl()`, предназначенный для использования в 32-разрядных приложениях. Как правило, это означает, что нужно преобразовать 32-разрядные значения в соответствующие 64-разрядные типы, использующиеся в ядре. В новых драйверах, где доступна такая роскошь, как создание собственных ioctl-команд с нуля, нужно позаботиться о том, чтобы размеры всех аргументов и передаваемых данных были четко определены и ими могли без проблем пользоваться 32-разрядные приложения на 32-разрядных системах, 32-разрядные приложения на 64-разрядных системах и 64-разрядные приложения на 64-разрядных системах. В таких драйверах указатель на функцию `compat_ioctl()` может соответствовать функции `unlocked_ioctl()`.

Структуры данных, связанные с файловыми системами

Кроме основных объектов подсистемы VFS, в ядре используются и другие стандартные структуры данных для управления данными, связанными с файловыми системами. Первый объект используется для описания конкретного типа файловой системы, как, например, `ext3`, `ext4` или `UDF`. Вторая структура данных используется для описания каждого экземпляра смонтированной файловой системы.

Поскольку в операционной системе Linux поддерживается множество файловых систем, в ядре должна быть предусмотрена специальная структура данных для описания возможностей и характеристик каждой файловой системы. Для этой цели используется структура `file_system_type`, определенная в файле `<linux/fs.h>` и приведенная ниже.

```
struct file_system_type {
   const char *name;     /* Название файловой системы */
   int         fs_flags; /* Флаги типа файловой системы */

  /* Следующая функция используется для чтения 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; /* Модуль, владеющий файловой системой */
   struct file_system_type *next;  /* Следующая файловая система в списке */
   struct list_head     fs_supers; /* Список объектов типа суперблок */

  /* Оставшиеся поля используются для проверки блокировок во время выполнения
программы */

   struct lock_class_key   s_lock_key;
   struct lock_class_key   s_umount_key;
   struct lock_class_key   i_lock_key;
   struct lock_class_key   i_mutex_key;
   struct lock_class_key   i_mutex_dir_key;
```

```
   struct lock_class_key   i_alloc_sem_key;
};
```

Функция `get_sb()` служит для считывания суперблока с диска и заполнения объекта суперблока соответствующими данными при монтировании файловой системы. Остальные функции позволяют определить 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.

Ситуация становится значительно интереснее после монтирования файловой системы. При этом создается структура `vfsmount`, которая 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;     /* Список потомков, связанных с родителем
*/
   int mnt_flags;                  /* Флаги монтирования */
   char           *mnt_devname;    /* Имя смонтированного устройства */
   struct list_head  mnt_list;      /* Список дескрипторов */
   struct list_head  mnt_expire;    /* Список элементов с истекшим сроком хра-
нения */
   struct list_head  mnt_share;     /* Список общих элементов монтирования */
   struct list_head  mnt_slave_list; /* Список подчиненных точек монтирования
*/
   struct list_head  mnt_slave;     /* Список подчиненных элементов монтирова-
ния */
   struct vfsmount  *mnt_master;    /* Указатель на основную структуру
vfsmount */
   struct mnt_namespace *mnt_namespace; /* Связанное пространство имен */
   int              mnt_id;        /* Идентификатор монтирования */
   int              mnt_group_id;   /* Идентификатор группы одного уровня */
   atomic_t          mnt_count;     /* Счетчик использования */
   int              mnt_expiry_mark;/* Отметка о устаревании */
   int              mnt_pinned;     /* Счетчик фиксации */
   int              mnt_ghosts;     /* Счетчик ghosts */
   atomic_t        __mnt_writers;   /* Счетчик записи */
};
```

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`.

В структуре `vfsmount` также предусмотрено поле флажков `mnt_flags`, в которое заносятся значения флагов, указанных во время монтирования. Список стандартных флагов монтирования приведен в табл. 13.1.

Эти флаги полезны в основном для сменных носителей, которым администратор не доверяет. Эти и другие менее используемые флаги определены в файле `<linux/mount.h>`.

Таблица 13.1. Список стандартных флагов монтирования

Флаг	Описание
MNT_NOSUID	Запрещает использование флагов setuid и setgid для бинарных файлов данной файловой системы
MNT_NODEV	Запрещает доступ к файлам устройств в данной файловой системе
MNT_NOEXEC	Запрещает запуск двоичных файлов с программами, расположенных в данной файловой системе

Структуры данных, связанные с процессом

С каждым процессом в системе связаны список открытых им файлов, корневая файловая система, текущий рабочий каталог, точки монтирования и т.д. Перечисленные ниже 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/fdtable.h>. Адрес этой структуры хранится в поле files дескриптора процесса. В данной структуре хранится вся информация об открытых процессом файлах и файловых дескрипторах. Эта структура с комментариями приведена ниже.

```
struct files_struct {
   atomic_t        count; /* Счетчик использования */
   struct fdtable *fdt;   /* Указатель на другую таблицу файловых
                             дескрипторов */
   struct fdtable  fdtab; /* Основная таблица файловых дескрипторов */
   spinlock_t      file_lock; /* Блокировка для защиты данной структуры */
   int             next_fd;   /* Кеш следующего доступного файлового
                                 дескриптора */
   struct embedded_fd_set close_on_exec_init; /* Файловые дескрипторы,
                         которые должны закрываться при вызове exec() */
   struct embedded_fd_set open_fds_init /* Первоначальный набор
                                       открытых файловых дескрипторов */
   struct file *fd_array[NR_OPEN_DEFAULT]; /* Массив файловых объектов */
};
```

Список указателей на открытые файловые объекты хранится в массиве fd_array. Поскольку стандартное значение константы NR_OPEN_DEFAULT равно BITS_PER_LONG, что соответствует числу 64 на 64-разрядных машинах, в этом массиве может находиться максимум 64 указателя на файловые объекты. Если во время работы процесса было открыто больше 64 файловых объектов, то ядро выделяет память под новый массив и присваивает полю fdt указатель на него. При таком подходе доступ к небольшому количеству файловых объектов осуществляется очень быстро, потому что они хранятся в статическом массиве. В случае, когда в процессе открывается аномально большое количество файлов, ядро может создать новый массив дескрипторов. Если в подавляющем количестве процессов в системе открывается больше 64 файлов, то для получения оптимальной производительности администратор может увеличить значение константы NR_OPEN_DEFAULT с помощью директивы препроцессора.

Следующей структурой данных, связанной с процессом, является fs_struct. Указатель на нее содержится в поле fs дескриптора процесса. В этой структуре хранится информация, связанная с процессом. Данная структура определена в файле <linux/fs_struct.h> и приведена ниже с поясняющими комментариями.

```
struct fs_struct {
   int      users; /* Счетчик ссылок */
   rwlock_t lock;  /* Блокировка для защиты структуры */
   int      umask; /* Стандартные права доступа к файлу */
   int      in_exec; /* Текущий выполняемый файл */
   struct path root; /* Корневой каталог */
   struct path pwd;  /* Текущий рабочий каталог */
};
```

В этой структуре содержится путь к текущему рабочему каталогу (он выводится в ответ на команду `pwd`) и путь к корневому каталогу данного процесса.

Третьей, и последней, структурой является `namespace`, которая определена в файле `<linux/mnt_namespace.h>`. Указатель на экземпляр данной структуры содержится в поле `mnt_namespace` дескриптора процесса. Индивидуальные для каждого процесса пространства имен были введены в ядрах Linux серии 2.4, что 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. Эта структура данных с поясняющими комментариями приведена ниже.

```
struct mnt_namespace {
   atomic_t          count; /* Счетчик использования */
   struct vfsmount  *root;  /* Объект монтирования корневого каталога */
   struct list_head  list;  /* Список точек монтирования */
   wait_queue_head_t poll;  /* Опрашиваемая очередь ожидания */
   int               event; /* Счетчик событий */
};
```

Поле `list` представляет собой двухсвязный список смонтированных файловых систем, которые составляют пространство имен.

Каждый дескриптор процесса имеет связанные с ним рассмотренные выше структуры данных. В большинстве случаев в дескрипторе процесса хранятся указатели на уникальные для текущего процесса структуры `files_struct` и `fs_struct`. Однако для процессов, при создании которых были указаны флаги `CLONE_FILES` и `CLONE_FS`, эти структуры являются совместно используемыми[3]. Отсюда следует, что ссылки на один и тот же экземпляр структур `files_struct` или `fs_struct` могут содержаться в нескольких дескрипторах процессов. В поле `count` каждой структуры находится счетчик использования, который предотвращает уничтожение структуры данных, если она используется хотя бы в одном процессе.

Структура `namespace` используется несколько по-другому. По умолчанию во всех процессах используется общее пространство имен, т.е. для всех процессов существует одна и та же иерархия файловых систем, монтируемая с помощью общей таблицы монтирования. И только в случае, когда при вызове системной функции `clone()` указан флаг `CLONE_NEWNS`, для процесса создается уникальная копия структуры `namespace`. Поскольку при создании большинства процессов этот флаг *не* указывается, процессы обычно наследуют пространство имен родительского процесса. Следовательно, в большинстве систем существует только одно пространство имен, несмотря на то, что в них

[3] Флаги `CLONE_FILES` и `CLONE_FS` обычно указываются при создании потоков, поэтому в потоках совместно используются структуры `files_struct` и `fs_struct`. С другой стороны, при создании обычных процессов эти флаги не указываются, поэтому для каждого процесса существует своя информация о файловой системе и своя таблица открытых файлов.

может поддерживаться множество пространств имен. Для этого при создании процесса достаточно лишь указать единственный флаг `CLONE_NEWNS`.

Резюме

В операционной системе Linux поддерживается большой набор файловых систем, от "родных" ext3 и ext4 до сетевых файловых систем, таких как NFS или Coda. Сейчас в официальном ядре ОС Linux поддерживается более 60 файловых систем. Благодаря уровню VFS для всех разнообразных файловых систем создается общая база для их реализации и общий интерфейс для работы со стандартными системными функциями. Таким образом, уровень виртуальной файловой системы позволяет совершенно ясно реализовывать поддержку новых файловых систем в операционной системе Linux, а также дает возможность работать с этими файловыми системами с помощью вызовов стандартных системных функций Unix.

В этой главе было описано назначение подсистемы VFS и рассмотрены соответствующие структуры данных, включая такие важные объекты, как файловый индекс (inode), элемент каталога (dentry) и суперблок. В главе 14, "Уровень блочного ввода-вывода", речь пойдет о физической организации данных файловых систем.

14
Уровень блочного ввода-вывода

Блочные устройства относятся к физическим устройствам компьютера и позволяют выполнять произвольный (а не последовательный) доступ к фрагментам данных фиксированного размера, называемых *блоками*. Самым популярным блочным устройством является жесткий диск, хотя существуют и другие блочные устройства, например устройства для работы с гибкими дисками, оптическими дисками (Blu-ray, DVD или CD-ROM) и флеш-памятью. Обратите внимание: мы перечислили все устройства, с которых обычно монтируются файловые системы. Таким образом, можно сказать, что файловая система — это своего рода универсальный язык общения с блочными устройствами различного типа.

К другому типу основных устройств относится *символьное устройство*, или устройство с посимвольным вводом-выводом данных. Доступ к данным таких устройств осуществляется в виде последовательного потока, т.е. байт за байтом. В качестве примера символьных устройств можно привести последовательный порт и клавиатуру. Таким образом, если доступ к данным устройства выполняется в виде потока данных, оно относится к классу символьных. С другой стороны, если к данным устройства можно обращаться произвольным образом (не последовательно), то оно относится к классу блочных устройств.

Различие между описанными выше типами устройств заключается в способности устройства выполнять произвольный доступ к данным, т.е. в возможности устройства выполнять *поиск* данных, перемещаясь из одной позиции в другую. В качестве примера рассмотрим клавиатуру. С точки зрения драйвера устройства клавиатура выдает поток данных. Если вы наберете на клавиатуре слово *wolf*, то драйвер клавиатуры вернет поток из этих четырех символов, расположенных в указанном порядке. При этом считывание символов в другом порядке или считывание какого-нибудь другого символа, кроме следующего в потоке, не имеет особого смысла. Поэтому драйвер клавиатуры — это устройство посимвольного ввода-вывода, позволяющие на выходе получить поток символов, которые пользователь вводит на клавиатуре. При чтении данных с устройства возвращается поток символов “w”,

“o”, “l” и в конце — “f”. Если никакая клавиша не была нажата, то этот поток будет пустой. Жесткий диск работает совсем иначе. Драйвер жесткого диска может потребовать прочитать содержимое определенного блока, а затем содержимое другого блока, причем эти блоки не обязательно должны следовать друг за другом. Поэтому доступ к данным жесткого диска может выполняться произвольным образом, а не в виде последовательного потока. Таким образом, жесткий диск относится к блочным устройствам.

По сравнению с символьными устройствами при управлении блочными устройствами в ядре им требуется уделять больше внимания, а также выполнять дополнительную подготовку. Связано это с тем, что символьные устройства имеют всего одну позицию для считывания данных, причем она всегда текущая. В отличие от него блочные устройства могут перемещаться в любую позицию на физическом носителе информации. В самом деле, не нужно создавать в ядре целую подсистему для обслуживания символьных устройств, тогда как для блочных устройств это просто необходимо. Такая подсистема необходима отчасти из-за сложности блочных устройств. Однако основная причина такой мощной поддержки в ядре состоит в том, что скорость работы блочных устройств напрямую зависит от скорости их обслуживания со стороны процессора. Действительно, выжать максимум производительности из жесткого диска значительно важнее, чем получить некоторое увеличение скорости при работе с клавиатурой. Более того, как будет показано ниже, сложность блочных устройств обеспечивает большой простор для таких оптимизаций. В этой главе будет описано, как ядро управляет работой блочных устройств и запросами к этим устройствам. Рассматриваемая часть ядра называется *уровнем блочного ввода-вывода* (block I/O layer). Интересно отметить, что усовершенствование подсистемы блочного ввода-вывода было одной из целей при разработке ядер серии 2.5. В данной главе рассматриваются все новые возможности уровня блочного ввода-вывода, которые появились в ядрах серии 2.6.

Структура блочного устройства

Наименьший адресуемый элемент блочного устройства называется *сектором*. Сектора могут иметь разный размер, который кратен 2^n, однако чаще всего размер сектора равен 512 байтам. Размер сектора определяется физическими параметрами устройства. Сектор является основным элементом любого блочного устройства. Устройства не могут обращаться к блокам данных, размер которых меньше размера сектора, однако большинство блочных устройств при выполнении одной команды позволяет оперировать несколькими секторами сразу. Хотя большинство блочных устройств и имеет размер сектора, равный 512 байтам, все же существуют и другие стандартные размеры сектора (например, у большинства накопителей на компакт-дисках размер сектора составляет 2 Кбайт).

При разработке программного обеспечения преследуются разные цели, исходя из которых выбирается минимально адресуемая единица данных, которая называется *блоком*. Блок — это абстракция файловой системы, т.е. все обращения к файловым системам могут выполняться только с данными, кратными размеру блока. Несмотря на то что физические устройства сами по себе работают на уровне секторов, ядро выполняет все дисковые операции на уровне блоков. Так как наименьший возможный адресуемый элемент — это сектор, 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 кратен 2^n. Ядро также требует, чтобы максимальный размер блока не превышал размер страницы памяти (см. главу 12,

"Управление памятью" и главу 19, "Переносимость")[1]. Поэтому размер блока равен размеру сектора, умноженному на число, кратное 2^n, и не может быть больше размера страницы. Чаще всего используются размеры блоков 512 байтов, 1 и 4 Кбайт.

Часто сбивает с толку то, что некоторые люди под секторами и блоками подразумевают разные вещи. Секторы являются минимальными адресуемыми элементами устройства, и их иногда называют "аппаратными секторами" (hardware sector) или "блоками аппаратного устройства" (device block). Между тем блоки являются наименьшими адресуемыми элементами файловых систем, которые иногда называют "блоками файловой системы" (filesystem 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. 14.1 схематически показана взаимосвязь секторов и блоков.

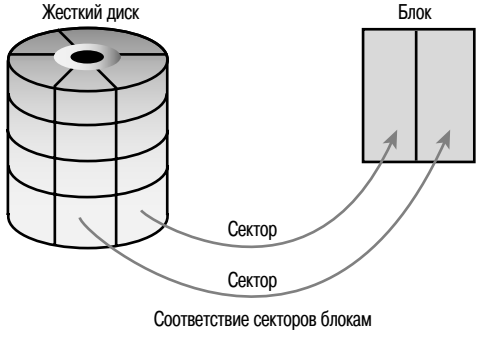


Рис. 14.1. Взаимосвязь секторов и блоков

Существует и другая часто встречающаяся терминология. По крайней мере, для жестких дисков используются такие понятия, как *кластеры* (clusters), *цилиндры* (cylinder) и *головки* (head). Эти термины являются специфическими только для некоторых типов блочных устройств. Поскольку по большей части они не используются в пользовательских приложениях, то в этой главе мы их рассматривать не будем. Так как все операции ввода-вывода на устройстве должны выполняться на уровне секторов, последние являются исключительно важными для ядра. В свою очередь, в ядре используется высокоуровневая концепция блоков, в основу которой положены физические секторы.

Буферы и их заголовки

При сохранении блока в памяти, скажем, после выполнения операций чтения или отложенной записи, он записывается в буфер (buffer). В каждый буфер записывается только один блок. По сути, буфер играет роль объекта, представляющего дисковый блок в оперативной памяти. Напомним: блок состоит из одного или нескольких секторов и по размеру не может быть больше одной страницы памяти. Поэтому в одной странице памяти может находиться один или несколько блоков. Поскольку для ядра требуется некоторая управляющая информация, связанная с данными (например, какому блочному устройству и какому блоку соответствует буфер), то с каждым буфером связан свой деск-

[1] Это ограничение является искусственным и в будущем может быть отменено. Тем не менее требование, чтобы размер блока был меньше или равен размеру страницы памяти, позволяет значительно упростить код ядра.

риптор. Этот дескриптор называется *заголовком буфера* (*buffer head*) и представляется с помощью структуры `buffer_head`, которая определена в файле `<linux/buffer_head.h>`. В ней содержится вся информация, которая необходима ядру для управления буферами. Эта структура вместе с поясняющими комментариями приведена ниже.

```
struct buffer_head {
   unsigned long       b_state;     /* Флаги состояния буфера */
   struct buffer_head *b_this_page; /* Список буферов, находящихся
                                       в текущей странице памяти */
   struct page        *b_page;      /* Ссылка на страницу памяти */
   sector_t            b_blocknr;   /* Начальный номер блока */
   size_t              b_size;      /* Размер отображения */
   char               *b_data;      /* Указатель на данные внутри страницы */
   struct block_device *b_bdev;     /* Связанное блочное устройство */
   bh_end_io_t        *b_end_io;    /* Метод завершения ввода-вывода */
   void               *b_private;   /* Зарезервировано для b_end_io */
   struct list_head    b_assoc_buffers; /* Список связанных отображений */
   struct address_space *b_assoc_map; /* список связанных адресных про-
странств */
   atomic_t            b_count;       /* Счетчик использования */
};
```

В поле `b_state` хранится состояние определенного буфера. Это значение может содержать один или несколько флагов, которые приведены в табл. 14.1. Возможные значения флагов описаны в виде перечисления `bh_state_bits`, которое определено в файле `<linux/buffer_head.h>`.

Таблица 14.1. Значения флагов поля `b_state`

Флаг состояния	Описание
`BH_Uptodate`	Буфер содержит корректные данные
`BH_Dirty`	Буфер изменен (содержимое буфера новее соответствующих данных на диске, и поэтому буфер в конечном счете должен быть записан на диск)
`BH_Lock`	Для буфера выполняется операция ввода-вывода с дискового устройства, поэтому он заблокирован для доступа со стороны других процессов
`BH_Req`	Буфер включен в запрос на выполнение операции ввода-вывода
`BH_Mapped`	Буфер содержит корректные данные и отображен на дисковый блок
`BH_New`	Буфер только что отображен с помощью функции `get_block()`, и к нему еще не было доступа
`BH_Async_Read`	Для буфера выполняется асинхронная операция чтения с помощью функции `end_buffer_async_read()`
`BH_Async_Write`	Для буфера выполняется асинхронная операция записи с помощью функции `end_buffer_async_write()`
`BH_Delay`	С буфером еще не связан дисковый блок (так называемое отложенное распределение)
`BH_Boundary`	Буфер является последним в последовательности смежных блоков — следующий за ним блок не является смежным с этой серией
`BH_Write_EIO`	При записи буфера произошла ошибка ввода-вывода
`BH_Ordered`	Упорядоченная запись
`BH_Eopnotsupp`	Операция на заданном буфере не поддерживается

Окончание табл. 14.1

Флаг состояния	Описание
BH_Unwritten	Место под данные буфера было выделено на устройстве, но сами данные еще не записаны
BH_Quiet	Подавить ошибки при выполнении операций ввода-вывода для этого буфера

В качестве последнего элемента перечисления bh_state_bits указан флаг BH_PrivateStart. Но это значение флага использовать нельзя, так как оно соответствует первому биту, который можно использовать в других целях по усмотрению разработчиков кода. Все биты, номер которых больше или равен значению BH_PrivateStart, не используются в подсистеме блочного ввода-вывода. Они без проблем могут использоваться в драйверах, разработчикам которых необходимо хранить информацию в поле b_state. Флаги, которые используются в драйверах, могут быть определены на основании значения флага BH_PrivateStart. Это позволяет избежать перекрытия с битами, которые официально используются уровнем блочного ввода-вывода.

В поле b_count хранится счетчик использования буфера. Значение этого поля увеличивается и уменьшается с помощью двух встраиваемых функций, которые определены в файле <linux/buffer_head.h>, как показано ниже.

```
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 обращаться к полям заголовка буфера, необходимо увеличить значение счетчика использования с помощью функции get_bh(). Это гарантирует, что во время работы с буфером он не будет освобожден другим процессом. Когда работа с заголовком буфера будет закончена, необходимо уменьшить значение счетчика ссылок с помощью функции put_bh().

Данному буферу соответствует физический блок с логическим номером b_blocknr, который находится на устройстве b_bdev.

В поле b_page хранится указатель на физическую страницу памяти, в которой располагаются данные буфера. В поле b_data хранится указатель на данные блока, расположенные на 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).

Заголовок буфера предназначен для описания соответствия между дисковым блоком и буфером, находящимся в оперативной памяти компьютера и представляющим собой последовательность байтов, которые расположены в указанной странице памяти. Единственное назначение этой структуры данных ядра — выполнение роли дескриптора отображения “буфер–блок”.

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 страниц физической памяти в дисковые бло-

ки, но и являлся контейнером для всех операций блочного ввода-вывода. Это приводило к двум основным 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. С точки зрения ядра предпочтительнее работать со страницами памяти, что одновременно и проще, и позволяет получить бо́льшую производительность. Использовать большой заголовок буфера для описания отдельного буфера, размер которого может быть меньше страницы памяти, неэффективно. Поэтому в ядрах серии 2.6 была проделана большая работа, которая позволила выполнять операции непосредственно со страницами памяти и пространствами адресов, а не с буферами. Некоторые из этих операций будут описаны в главе 16, "Страничный кеш и отложенная запись страниц", где также рассматриваются структура `address_space` и демоны `pdflush`.

Вторая проблема, связанная с заголовками буферов, заключается в том, что один заголовок применяется для описания только одного буфера. Когда заголовок буфера используется в качестве контейнера для операций ввода-вывода, это требует, чтобы в ядре потенциально большая операция блочного ввода-вывода (например, записи) разбивалась на множество мелких структур `buffer_head`. В результате это приводит к снижению производительности системы ввода-вывода и ненужным затратам памяти для хранения структур данных. Поэтому основной целью при создании ядер серии 2.5 была разработка нового гибкого и быстрого контейнера для операций блочного ввода-вывода. Так появилась структура `bio`, которая будет рассмотрена в следующем разделе.

Структура `bio`

Основным контейнером для блочных операций ввода-вывода в ядре является структура `bio`, которая определена в файле `<linux/bio.h>`. Она представляет активные операции блочного ввода-вывода в виде списка *сегментов* (segment). Сегмент представляет собой непрерывный участок буфера в памяти. Таким образом, отдельные буферы не обязательно должны быть непрерывными в памяти. Благодаря тому что буфер может представляться в виде нескольких сегментов, структура `bio` дает возможность выполнять операции блочного ввода-вывода, даже если данные одного буфера хранятся в разных местах памяти. Векторные операции ввода-вывода, подобные описанным выше, называются также *распределенным вводом-выводом* (scatter-gather I/O). Структура `bio` с комментариями, описывающими назначение каждого поля, приведена ниже.

```
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 */
   unsigned int   bi_size;      /* Объем данных для ввода-вывода */
   unsigned int   bi_seg_front_size; /* Размер первого сегмента */
   unsigned int   bi_seg_back_size;  /* Размер последнего сегмента */
   unsigned int   bi_max_vecs; /* Максимально возможное количество струк-
тур bi_io_vec */
```

```
    unsigned int    bi_comp_cpu; /* CPU, на котором завершена операция */
    atomic_t        bi_cnt;      /* Счетчик использования */
    struct bio_vec *bi_io_vec;   /* Список векторов bi_io_vec */
    bio_end_io_t   *bi_end_io;   /* Метод завершения ввода-вывода */
    void           *bi_private;  /* Закрытое поле владельца */
    bio_destructor_t *bi_destructor; /* Метод деструктора */
    struct bio_vec bi_inline_vecs[0];/* Встроенные векторы bio */
};
```

Основное назначение структуры `bio` — это представление активной (выполняющейся) операции блочного ввода-вывода. В связи с этим большинство полей этой структуры являются служебными. Самыми важными являются поля `bi_io_vec`, `bi_vcnt` и `bi_idx`. Взаимосвязь структуры `bio` с другими используемыми структурами данных показана на рис. 14.2.

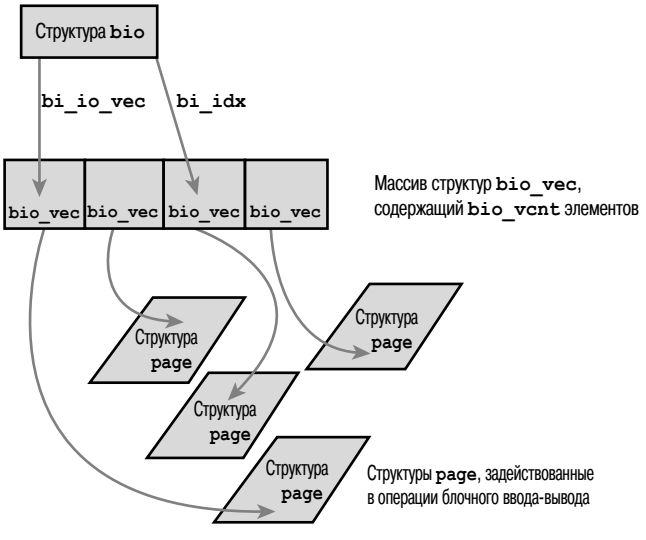


Рис. 14.2. Взаимосвязь структур `bio`, `bio_vec` и `page`

Векторы ввода-вывода

В поле `bi_io_vec` хранится указатель на массив 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. В этой структуре находится указатель на физическую страницу памяти, в которой находится сегмент, смещение блока относительно начала страницы и длина блока. Полный массив рассмотренных выше векторов описывает весь буфер полностью. Структура `bio_vec` определена в файле `<linux/bio.h>` и приведена ниже.

```
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-вывода создается массив из bi_vcnt элементов типа bio_vec, адрес которого содержится в поле bi_io_vec. В процессе выполнения операции блочного ввода-вывода поле bi_idx используется в качестве указателя на текущий элемент массива.

В общем, каждый запрос на выполнение блочного ввода-вывода представляется с помощью структуры bio. Он состоит из одного или нескольких блоков, которые хранятся в массиве структур bio_vec. Каждая из этих структур представляет собой вектор, который описывает положение в физической памяти каждого сегмента запроса. В поле bi_io_vec хранится указатель на первый сегмент операции ввода-вывода. Каждый следующий сегмент следует сразу за предыдущим, и всего в массиве расположено bi_vcnt сегментов. После обработки текущего сегмента операции блочного ввода-вывода обновляется поле bi_idx, чтобы его значение соответствовало номеру текущего сегмента.

В поле bi_idx хранится индекс текущей структуры bio_vec в массиве, что позволяет уровню блочного ввода-вывода отслеживать частично выполненные операции ввода-вывода. Однако важнее всего то, что подобный подход позволяет разбивать структуры типа bio на несколько частей. В результате в драйвере устройств типа RAID[2] появляется возможность использования одной структуры типа bio, которая изначально была предназначена для одного устройства с несколькими жесткими дисками, входящими в RAID-массив. Все, что нужно сделать в драйвере RAID-устройства, — это создать необходимое количество копий структуры bio, которая предназначалась для одного устройства, и изменить для каждой копии значение поля bi_idx, чтобы оно указывало на ту часть массива, откуда должна начинаться операция ввода-вывода для каждого диска.

В поле bi_cnt структуры bio хранится счетчик использования. Когда значение этого счетчика становится равным нулю, структура уничтожается и занятая ею память освобождается. Значение счетчика использования можно изменить с помощью двух функций:

```
void bio_get(struct bio *bio)
void bio_put(struct bio *bio)
```

В первой функции значение счетчика использования увеличивается на единицу, а во второй — уменьшается на единицу, и если это значение становится равным нулю, то соответствующая структура bio аннулируется. Перед обращением к полям активной структуры bio необходимо увеличить значение счетчика использования. Это гарантирует, что во время работы с ним экземпляр данной структуры не будет удален другим процессом. После окончания работы необходимо уменьшить значение счетчика использования.

И наконец, bio_private — это закрытое поле владельца (т.е. создателя) структуры. Как правило, записывать и считывать информацию из этого поля может только тот процесс, который 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, в которой может быть задействована одна или несколько страниц в памяти. В противоположность этому заголовок буфера связан с одним дисковым блоком, который занимает не более одной страницы памяти.

[2] Redundant Array of Inexpensive Disks — избыточный массив недорогих/независимых жестких дисков. Специальный способ использования жестких дисков, при котором один логический том может быть распределен по нескольким физическим дискам для увеличения надежности и/или производительности.

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. Обработка структур `bio` выполняется быстрее, она может описывать несмежные блоки памяти и не требует без необходимости разбивать операции ввода-вывода на части.

Переход от структуры `buffer_head` к структурам `bio` позволяет получить также и другие преимущества, перечисленные ниже.

- С помощью структуры `bio` можно легко задействовать в операции ввода-вывода верхнюю память, так как эта структура работает только со страницами физической памяти, а не с указателями на объекты, расположенные в виртуальном адресном пространстве.
- С помощью структуры `bio` можно представлять как обычные страничные операции ввода-вывода, так и операции непосредственного (direct) ввода-вывода (т.е. те, которые не проходят через страничный кеш; страничный кеш обсуждается в главе 16, “Страничный кеш и отложенная запись страниц”).
- Структура `bio` позволяет легко выполнять операции рассеянного (векторного) ввода-вывода, в которых данные находятся в нескольких несмежных страницах физической памяти.
- Структура `bio` значительно проще заголовка буфера, потому что в ней содержится только минимум информации, необходимой для представления операции блочного ввода-вывода, а не лишняя информация, которая связана с самим буфером.

Несмотря на все недостатки, заголовки буферов все еще используются в качестве дескрипторов отображения дисковых блоков на страницы физической памяти. В структуре `bio` не содержится никакой информации о состоянии буфера. Это просто массив векторов, которые описывают один или несколько сегментов данных, задействованных в одной операции блочного ввода-вывода, плюс соответствующая дополнительная информация. На сегодняшний день от структуры `buffer_head` все еще нельзя отказаться, поскольку в ней содержится информация о состоянии буфера, тогда как с помощью структуры `bio` описывается активная операция ввода-вывода. 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 из кода ядра более высокого уровня, например, из файловых систем. Драйвер блочного устройства, связанный с очередью, извлекает запросы из головы очереди и отправляет их на соответствующее блочное устройство до тех пор, пока очередь запросов не будет пуста. Каждый элемент списка очереди запросов соответствует одному запросу и представляется с помощью структуры `request`, которая также определена в файле `<linux/blkdev.h>`.

Каждый запрос может состоять из нескольких структур `bio`, поскольку в одном запросе может выполняться обращение к нескольким смежным дисковым блокам. Обратите

внимание на то, что, хотя блоки на диске и должны быть смежными, данные этих блоков, хранящиеся в оперативной памяти, не обязательно должны быть смежными. В каждой структуре `bio` может быть описано несколько сегментов (напомним: сегменты — это непрерывные участки памяти, в которых хранятся данные блока), а запрос может состоять из нескольких структур `bio`.

Планировщики ввода-вывода

Простая отправка запросов блочному устройству в том же порядке, в котором эти запросы поступают из ядра, приводит к резкому снижению производительности подсистемы ввода-вывода в целом. В современном компьютере самой медленной операцией является позиционирование данных на жестком диске. При выполнении этой операции головки жесткого диска должны переместиться к указанному блоку. В зависимости от их текущего положения относительно пластин диска на это может уйти от 1 до 10–15 мск. Следовательно, для того чтобы поднять производительность системы ввода-вывода в целом, нужно минимизировать (или оптимизировать) перемещение головок жесткого диска.

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 после их получения. Вместо этого оно выполняет так называемые операции *слияния* (*объединения*, merging) и *сортировки* (sorting), позволяющие значительно увеличить производительность всей системы[3]. Подсистема ядра, в которой выполняются указанные операции, называется *планировщиком ввода-вывода* (I/O scheduler).

Планировщик ввода-вывода распределяет дисковые ресурсы между ожидающими в очереди запросами на ввод-вывод. Не путайте его с системным планировщиком (см. главу 4, “Системный планировщик и диспетчеризация процессов“), который распределяет ресурсы процессора между всеми процессами в системе. Хотя два планировщика и выполняют похожие задачи, но они не одно и то же. Оба планировщика выполняют виртуализацию ресурсов между несколькими объектами. В случае планировщика процессов выполняется виртуализация процессора, который совместно используется всеми процессами в системе. Это создает иллюзию того, что процессы выполняются одновременно, и является основой многозадачных операционных систем с режимом разделения времени, таких как Unix. В отличие от него планировщик ввода-вывода выполняет виртуализацию блочных устройств для ожидающих выполнения запросов на ввод-вывод. Это делается с целью минимизации перемещения головок по жесткому диску и получения оптимальной производительности дисковых подсистем ввода-вывода.

Задачи планировщика ввода-вывода

Работа планировщика ввода-вывода заключается в управлении очередью запросов блочного устройства. Он определяет, в каком порядке должны обрабатываться запросы, находящиеся в очереди, и в какое время каждый запрос должен передаваться на блочное устройство. Запросы из очереди выбираются с учетом минимизации перемещения головок по жесткому диску, что значительно повышает производительность системы ввода вывода *в целом*. Дополнение “в целом” здесь существенно. Планировщик ввода-вывода

[3] Этот момент стоит выделить 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.

откровенно несправедлив по отношению к некоторым запросам, однако за счет этого ему удается повысить производительность системы в целом.

Для минимизации перемещения головок по жесткому диску планировщик ввода-вывода выполняет две основные операции: объединение и сортировку. Объединение — это слияние нескольких запросов в один. В качестве примера рассмотрим запрос, который поступил от файловой системы, например, чтобы прочитать порцию данных из файла. Разумеется, на данном этапе вся работа уже выполняется на уровне секторов и блоков, а не файлов. Однако предположим, что в запрошенном файловой системой блоке находится фрагмент данных файла. Если в очереди уже есть полученный ранее запрос на чтение данных из соседнего сектора диска (например, другого фрагмента того же файла), то два запроса могут быть объединены в один. Тогда чтение нескольких расположенных рядом секторов диска можно будет выполнить за одно обращение к устройству. Путем слияния запросов планировщик ввода-вывода уменьшает затраты ресурсов, связанные с обработкой нескольких запросов, до уровня, необходимого на обработку одного запроса. Важнее всего то, что в результате для чтения данных объединенного запроса контроллеру диска посылается всего одна команда, а обработка нескольких запросов может быть выполнена без перемещения головок либо с их минимальным перемещением. Следовательно, слияние запросов уменьшает накладные расходы и минимизирует перемещение головок.

А теперь предположим, что наш запрос на чтение помещается в очередь запросов, но там нет других запросов на чтение соседних секторов. Поэтому нет возможности объединить этот запрос с другими запросами, находящимися в очереди. В таком случае запрос просто помещается в конец очереди. Но что если в очереди уже есть запросы к расположенным рядом секторам диска? Не лучше ли будет поместить новый запрос сразу за (или перед) запросом к физически близко расположенным секторам диска? На самом деле планировщики ввода-вывода именно так и поступают. Все запросы в очереди упорядочиваются (т.е. сортируются) по номерам секторов так, чтобы последовательность запросов к диску (насколько это возможно) соответствовала линейному перемещению головок жесткого диска. Цель состоит в том, чтобы не только минимизировать количество перемещений головок в каждом конкретном случае, но и уменьшить общее количество операций позиционирования на диске за счет линейного перемещения головок диска. Это чем-то напоминает алгоритм перемещения обычного лифта, который не начинает хаотичное движение между этажами в ответ на нажатие кнопок пассажирами. Вместо этого он всегда старается плавно двигаться в одном направлении. Когда лифт доходит до последнего этажа, он начинает движение в противоположном направлении. Из-за такой аналогии алгоритм сортировки планировщика ввода-вывода иногда называют *лифтовым*, а сам планировщик — *лифтом*.

Лифт имени Линуса

А теперь рассмотрим некоторые планировщики ввода-вывода, применяемые в реальной жизни. Первый планировщик ввода-вывода, который мы рассмотрим, *называется лифтом Линуса* (Linus Elevator). Да, это не опечатка. Действительно, существует лифтовой планировщик, разработанный Линусом Торвальдсом и названный в его честь! Он был стандартным планировщиком ввода-вывода в ядрах серии 2.4. В ядрах серии 2.6 его заменили другими планировщиками, которые будут рассмотрены ниже. Однако поскольку этот алгоритм значительно проще новых и в то же время позволяет выполнять почти те же функции, то он заслуживает нашего внимания.

В лифтовом алгоритме Линуса выполняется как объединение, так и сортировка запросов. При добавлении запроса в очередь вначале проверяются находящиеся там запросы и выявляются кандидаты на предмет возможного объединения. В алгоритме Линуса выполняются два типа объединения: *добавление в начало запроса* (front merging) и *добавление в конец запроса* (back merging). Тип объединения зависит от того, с какой стороны расположены соседние секторы. Если секторы нового запроса расположены перед существующими, то они добавляются в начало запроса. Соответственно, если секторы нового запроса расположены после существующих, то они добавляются в конец запроса. В связи с тем что файлы обычно располагаются на диске в секторах, номера которых увеличиваются, а операции ввода-вывода чаще всего выполняются от начала и до конца файла (т.е. данные обычно прочитываются в порядке от начала и до конца файла, а не наоборот), то при обычной работе вставка в начало запроса встречается значительно реже, чем вставка в конец запроса. Тем не менее в алгоритме Линуса проверяются оба случая и выполняются оба типа объединения.

Если попытка объединения была неудачной, то определяется возможное место вставки нового запроса в очередь (положение в очереди, в котором новый запрос наилучшим образом вписывается по номеру сектора между окружающими запросами). Если такое место найдено, то новый запрос помещается туда. Если подходящего места не найдено, то новый запрос помещается в конец очереди. Кроме того, если в очереди найден запрос, который является достаточно старым (т.е. время его нахождения в очереди превышает заданный предел), новый запрос также добавляется в конец очереди, несмотря на то, что это может нарушить порядок сортировки. Это предотвращает ситуацию, при которой поступление большого количества запросов к близко расположенным секторам приводит к увеличению на неопределенное время обслуживания запросов к другим секторам диска. К сожалению, такая проверка “на старость” не очень эффективна. В рассмотренном алгоритме не предпринимается никаких попыток обслуживания запросов в заданных временны*х* рамках, а просто прекращается процесс сортировки-вставки при наличии заданной задержки. Это, в свою очередь, вызывает неопределенную задержку при выполнении операций ввода-вывода, что было веской причиной для доработки планировщика ввода-вывода ядра серии 2.4.

Итак, когда запрос добавляется в очередь, возможны четыре варианта действий.

1. Если в очереди находится запрос к соседнему сектору, то существующий запрос и новый объединяются в один.
2. Если в очереди существует достаточно старый запрос, то новый запрос помещается в конец очереди, чтобы предотвратить отказ в обслуживании для других запросов, которые долгое время находятся в очереди.
3. Если для нового запроса найдено подходящее положение в очереди, которое соответствует рациональному перемещению головок, то он помещается туда. Это позволяет поддерживать очередь в упорядоченном по физическим номерам диска состоянии.
4. И наконец, если такое положение не найдено, запрос помещается в конец очереди.

Реализация алгоритма лифта Линуса находится в файле `block/elevator.c`

Планировщик ввода–вывода с ограничением по времени

Планировщик ввода-вывода с ограничением по времени (Deadline I/O scheduler) разработан с целью предотвращения задержек в обслуживании, которые могут возникать при использовании алгоритма лифта Линуса. Если задаться целью минимизировать только перемещение головок по диску, то при большом количестве операций ввода-вывода, направленных в одну область диска, могут возникать задержки в обслуживании на неопределенное время при выполнении операций с другими областями диска. Очевидно, что при таком подходе непрерывный поток запросов, направленных к одной и той же области диска, может привести к тому, что запросы к другой области диска, которая находится далеко от первой, так никогда и не будут обработаны. Такой алгоритм не может обеспечить равноправный доступ ко всем областям диска.

Ситуация ухудшается еще и тем, что общая проблема в задержке обслуживания запросов приводит к другой специфической проблеме — *откладыванию операций чтения при выполнении операций записи* (*writes-starving-reads*). Как правило, операции физической записи на диск инициируются ядром, как только в этом возникает необходимость. Они выполняются полностью асинхронно по отношению к пользовательской программе, которая сгенерировала запрос на запись в файл. Операции чтения обрабатываются совсем иначе. Обычно, когда пользовательская программа отправляет запрос на чтение данных, она переводится в состояние ожидания до тех пор, пока запрос не будет выполнен, т.е. запросы на чтение возникают синхронно по отношению к приложению, которое их генерирует. Хотя задержка по записи (время, необходимое для завершения запроса на запись) практически не влияет на время реакции системы, задержка по чтению (время, необходимое для завершения запроса на чтения) очень важна. Задержка по записи мало сказывается на производительности пользовательских приложений[4], однако эти программы должны ожидать завершения выполнения каждого запроса на чтение. При этом пользователь будет сидеть у экрана монитора и нервно постукивать пальцами по столу, пока программа выдаст нужные ему результаты. Следовательно, задержка по чтению сильно влияет на производительности системы в целом.

Проблему усугубляет еще и то, что запросы на чтение обычно взаимозависимы. В качестве примера рассмотрим чтение большого количества файлов. Каждая операция чтения выполняется небольшими порциями, которые соответствуют размеру буфера. Приложение не станет считывать следующую порцию данных (или, в данном случае, следующий файл), пока предыдущая порция данных не будет считана с диска и доставлена приложению. Хуже того, во время выполнения как чтения, так и записи требуется прочитать с диска различные служебные данные, такие как индексные дескрипторы. Чтение блоков с диска, содержащих эти данные, приводит к еще большей взаимозависимости операций ввода-вывода. Следовательно, если существует задержка в обслуживании одного запроса на чтение, то для пользовательской программы эти задержки складываются и общая задержка может стать недопустимой. Принимая во внимание то, что синхронность и взаимозависимость запросов на чтение приводят к большим задержкам в обработке этих запросов (что в свою очередь сильно влияет на производительность системы), в планировщик ввода-вывода с ограничением по времени был добавлен ряд функций, ко-

[4] Однако все же не желательно задерживать операции записи на неопределенное время. Дело в том, что ядро должно гарантировать, что данные в конечном счете все-таки будут записаны на диск. В противном случае это может привести к нежелательному увеличению размера внутреннего буфера либо к устареванию в них данных.

торые позволяют гарантированно минимизировать задержки в обработке запросов вообще и в обработке запросов на чтение в частности.

Следует обратить внимание на то, что уменьшение времени задержки в обслуживании может быть выполнено только за счет снижения общей производительности системы. Даже для алгоритма лифта Линуса такой компромисс существует, хотя и в более мягкой форме. При использовании этого алгоритма можно было бы и увеличить общую производительность системы ввода-вывода путем радикального уменьшения количества перемещений головок по диску. Для этого нужно было бы *всегда* помещать запросы в очередь в соответствии с номерами секторов, отменив проверку на наличие старых запросов и последующего за ним помещения нового запроса в конец очереди. Хотя минимизация перемещения головок по диску и важна, тем не менее неопределенное время задержки при выполнении запроса тоже не очень хорошая вещь. Поэтому алгоритм планировщика ввода-вывода с ограничением по времени был тщательно проработан, чтобы величины задержек не превышали заранее заданных значений и при этом общая производительность системы оставалась на довольно высоком уровне. Со всей уверенностью можно заявить, что достаточно сложно одновременно обеспечить равнодоступность всех секторов диска и при этом максимизировать общую производительность системы.

В планировщике ввода-вывода с ограничением по времени каждому запросу назначается *максимальный интервал ожидания* (expiration time). По умолчанию он равен 500 мс для запросов на чтение и 5 с для запросов на запись. Этот планировщик работает аналогично лифту 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, планировщик ввода-вывода выполняет объединение и вставку запросов так же, как это делается в лифтовом алгоритме Линуса[5]. Кроме того, в зависимости от типа запроса новый планировщик помещает каждый запрос еще и во вторую очередь. Запросы на чтение помещаются в свою специальную FIFO-очередь, а запросы на запись — в свою. Несмотря на то что запросы в обычной очереди упорядочены по номерам секторов диска, запросы в этих двух FIFO-очередях упорядочены по времени поступления. Поэтому новые запросы всегда будут добавляться в конец очереди. При нормальной работе планировщик ввода-вывода выбирает запросы из головы основной отсортированной очереди и помещает их в очередь диспетчеризации, откуда они отправляются на выполнение жесткому диску. В результате минимизируется количество перемещений головок диску.

Если время ожидания запроса, находящегося в голове одной из FIFO-очередей (на чтение или запись), истекло (т.е. текущее системное время становится больше, чем предельное время ожидания запроса, вычисленное на основе максимального интервала ожидания), планировщик ввода-вывода начинает выбирать запросы из этой очереди. В результате гарантируется, что ни один из запросов не будет находиться в очереди на обслуживание дольше максимального интервала ожидания (рис. 14.3).

Следует заметить, что планировщик ввода-вывода с ограничением по времени не дает строгих гарантий в отношении величины задержки при обработке запроса. Однако в общем он позволяет отправлять запросы на обработку до или вскоре после того, как истек их период ожидания. Это позволяет предотвратить ситуацию, когда обработка запросов на ввод-вывод откладывается на неопределенное время. Поскольку максимальный ин-

[5] В планировщике ввода-вывода с ограничением по времени операция объединения со вставкой в начало запроса может и не выполняться. Дело в том, что количество таких запросов на общем фоне незначительно и кардинально они не влияют на быстродействие всей системы.

тервал ожидания для запросов на чтение значительно меньше, чем для запросов на запись, планировщик ввода-вывода с ограничением по времени также позволяет гарантировать, что обслуживание запросов на чтение не будет откладываться на неопределенное время из-за обработки запросов на запись. Больший приоритет запросов на чтение позволяет минимизировать задержку при выполнении операций чтения.

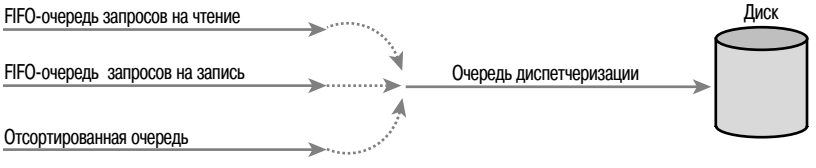


Рис. 14.3. Три очереди планировщика ввода-вывода с ограничением по времени

Код планировщика ввода-вывода с ограничением по времени находится в файле `block/deadline-iosched.c`.

Прогнозирующий планировщик ввода-вывода

Несмотря на то что планировщик ввода-вывода с ограничением по времени позволяет минимизировать задержки при обработке запросов на чтение, делается это за счет снижения общей производительности системы. Рассмотрим систему с большой активностью запросов на запись. При получении запроса на чтение планировщик ввода-вывода очень быстро начинает его обработку. Это вызывает перемещение головок диска к считываемому сектору, после чего головки снова возвращаются в то место, где должна выполняться следующая операция записи. В результате линейное перемещение головок вдоль поверхности пластин диска будет прерываться при получении каждого запроса на чтение. Как следствие, отдавая больший приоритет запросам по чтению, мы в результате получаем два лишних перемещения головок (в точку чтения и обратно) на каждую операцию чтения, что плохо сказывается на общей производительности дисковой подсистемы ввода-вывода. При создании прогнозирующего планировщика ввода-вывода (anticipatory I/O scheduler) преследовалась цель обеспечить минимальные задержки обработки запросов на чтение и в то же время обеспечить высокую общую производительность системы ввода-вывода.

Поскольку прогнозирующий планировщик построен на базе планировщика ввода-вывода с ограничением по времени, он не особо от него отличается. В прогнозирующем планировщике используются три очереди (плюс очередь диспетчеризации) и учитывается максимальный интервал ожидания для каждого запроса, как и в случае планировщика с ограничением по времени. Основное отличие состоит во введении дополнительного модуля *эвристического прогнозирования* (*anticipation heuristic*).

В прогнозирующем планировщике ввода-вывода попытались уменьшить “хаос перемещения головок”, который неизбежно возникает после получения очередного запроса на чтение во время выполнения других дисковых операций. После того как запрос поступает на чтение, он обрабатывается как обычно в рамках назначенного ему интервала ожидания. Однако, после того как запрос отправлен жесткому диску, прогнозирующий планировщик не возвращается сразу же к выполнению следующих запросов, находящихся в очереди, поэтому головки диска остаются на месте. Он абсолютно ничего не делает в течение нескольких миллисекунд (значение этого периода времени можно изменять, по умолчанию оно равно 6 мс). Существует большой шанс, что за эти несколько миллисе-

кунд приложение отправит еще один запрос на чтение. Все запросы к соседним секторам диска будут выполнены немедленно. Когда интервал ожидания истекает, прогнозирующий планировщик возвращается к выполнению оставшихся запросов в очереди и головки жесткого диска перемещаются в точку выполнения нового запроса.

Важно заметить, что те несколько миллисекунд, в течение которых планировщик *ожидает* поступления новых запросов, полностью окупаются, даже если это позволяет минимизировать всего лишь небольшой процент операций чтения, для которых требуется переместить головки, на фоне большого количества других запросов. Если во время ожидания приходит запрос на чтение секторов из соседней области диска, то это позволяет избежать двух лишних перемещений головок. Чем больше за это время приходит запросов на чтение соседних областей диска, тем большего количества операций перемещений головок можно избежать.

Конечно, если в течение периода ожидания не было никакой активности, то эти несколько миллисекунд будут потрачены зря в прогнозирующем планировщике. Поэтому, чтобы получить максимальную производительность при использовании прогнозирующего планировщика, необходимо точно предсказывать дальнейшие действия приложений и файловых систем. Это выполняется на основе эвристических алгоритмов и сбора статистических данных. В прогнозирующем планировщике ведется статистика операций блочного ввода-вывода по каждому процессу, которая с большой степенью вероятности позволяет предсказать дальнейшие действия приложений. При достаточно высоком проценте точных предсказаний прогнозирующий планировщик способен значительно снизить потери времени на перемещение головок при выполнении операций чтения и в то же время уделить достаточно внимания тем запросам, от которых зависит общая производительность системы. Это позволяет прогнозирующему планировщику минимизировать задержки при обслуживании запросов на чтение за счет уменьшения количества и времени перемещения головок. В результате возрастает общая производительность системы и уменьшается ее время реакции.

Код прогнозирующего планировщика находится в файле `block/as-iosched.c` дерева исходных кодов ядра. Этот планировщик хорошо работает при обслуживании большинства типовых запросов в системе. Он идеален для серверов, однако работает очень плохо при поступлении запросов определенного типа, которые встречаются не так часто, но очень критичны к скорости обслуживания. В качестве примера можно привести выполнение операций активного поиска информации в базах данных, когда результаты одного запроса зависят от результатов выполнения другого запроса.

Планировщик ввода–вывода с полностью равноправными очередями

Планировщик ввода-вывода с полностью равноправными очередями (Complete Fair Queuing, CFQ) разработан для обслуживания в системе запросов определенного типа. Но на практике оказалось, что он обеспечивает хорошую производительность для широкого типа запросов, несмотря на то, что по структуре он в корне отличается от всех ранее рассмотренных планировщиков ввода-вывода.

Планировщик CFQ распределяет все входящие запросы ввода-вывода по определенным очередям в зависимости от процесса, приславшего этот запрос. Например, запросы от процесса `foo` идут в очередь `foo`, а запросы от процесса `bar` — в очередь `bar`. В пределах каждой очереди запросы объединяются со смежными и сортируются. Таким образом, запросы в очереди отсортированы по номерам секторов, так же как и в случае других плани-

ровщиков ввода-вывода. Отличие планировщика CFQ состоит в том, что в нем поддерживается отдельная очередь для каждого процесса, выполняющего операции ввода-вывода.

После этого планировщик CFQ выбирает запросы из разных очередей по круговому алгоритму, обрабатывая заданное 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. Планировщик CFQ разрабатывался для обработки запросов, поступающих из мультимедийных приложений. Он гарантирует, что, например, аудиопроигрыватель всегда будет успевать вовремя считывать данные с диска и помещать их в аудиобуферы для последующего воспроизведения. Однако на практике оказалось, что планировщик CFQ хорошо справляется с запросами, поступающими от приложений разного типа.

Код планировщика CFQ находится в файле `block/cfq-iosched.c`. Этот планировщик рекомендуется использовать для офисных компьютеров, хотя он хорошо работает практически со всеми остальными типами приложений, не вызывая патологических узких мест. По этой причине планировщик CFQ стал стандартным планировщиком ввода-вывода системы Linux.

Планировщик ввода-вывода с отсутствием операций (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 уменьшения числа перемещений головок по диску. В нем не нужно реализовывать практически ничего, включая алгоритмы, минимизирующие время обслуживания запроса, которые были рассмотрены выше для других 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, близком к FIFO.

В том, что из планировщика Noop убрана вся сложная логика, есть определенный смысл. Он предназначен для работы с блочными устройствами, обеспечивающими настоящий произвольный доступ к данным, таким как карты флеш-памяти или твердотельные диски. Если для доступа к данным устройству не требуется время на перемещение головок, то не нужно выполнять сортировку и объединение вновь приходящих запросов. Поэтому в подобных случаях лучшим выбором будет именно планировщик Noop.

Код планировщика Noop находится в файле `block/noop-iosched.c`. Он предназначен только для обслуживания запросов, поступающих для устройств с настоящим произвольным доступом.

Выбор планировщика ввода-вывода

Выше были рассмотрены четыре типа планировщиков ввода-вывода, которые используются в ядрах серии 2.6. Каждый из них может быть активизирован и встроен в ядро на этапе компиляции. По умолчанию для всех блочных устройств используется планировщик CFQ. Эту ситуацию можно изменить во время загрузки системы, указав в командной строке ядра параметр `elevator=`*`foo`*, где *`foo`* — корректный и активизированный планировщик ввода-вывода (табл. 14.2).

Таблица 14.2. Возможные значения параметра `elevator`

Значение	Тип планировщика
`as`	Прогнозирующий
`cfq`	С полностью равноправными очередями
`deadline`	С ограничением по времени
`noop`	С отсутствием операций (Noop)

Например, указание параметра `elevator=as` в командной строке ядра при загрузке системы означает, что для всех блочных устройств будет использоваться прогнозирующий планировщик. Этот параметр отменяет назначенный по умолчанию планировщик CFQ.

Резюме

В этой главе были рассмотрены принципы работы блочных устройств ввода-вывода, а также структуры данных, используемые на уровне блочного ввода-вывода. Вы ознакомились со структурой `bio`, представляющей активную (т.е. выполняемую) в данный момент операцию ввода-вывода; со структурой `buffer_head`, представляющей отображение блоков на страницы памяти; и со структурой `request`, которая представляет собой отдельный запрос ввода-вывода. После запросов ввода-вывода был описан их короткий, но важный путь, кульминацией которого является прохождение через планировщик ввода-вывода. Были рассмотрены основные противоречия, возникающие при планировании операций ввода-вывода, а также четыре типа планировщиков, которые на данный момент существуют в ядре Linux. Кроме того, мы описали также старый планировщик ввода-вывода из ядра 2.4, называемый лифтом имени Линуса.

В следующей главе речь пойдет об адресном пространстве процесса.

15
Адресное пространство процесса

В главе 12, “Управление памятью”, рассматривалось управление физической памятью в ядре. Кроме управления собственной памятью, ядро должно управлять еще и памятью пользовательских программ. Эта память называется *адресным пространством процесса* (process address space) и является образом памяти, выделяемой операционной системой каждому пользовательскому процессу. Операционная система Linux является системой с поддержкой виртуальной памяти, т.е. в ней выполняется виртуализация ресурсов памяти среди всех процессов в системе. Для каждого процесса создается иллюзия того, что он один использует всю физическую память в системе. Еще более важно то, что адресное пространство даже одного процесса может значительно превышать объем физической памяти компьютера. В этой главе речь пойдет об управлении адресным пространством процесса в ядре.

Адресные пространства

Адресное пространство процесса состоит из виртуальной памяти, адресуемой процессом, и диапазона адресов в этой виртуальной памяти, которые разрешено использовать процессу. Каждому процессу назначается 32- или 64-разрядное *линейное* (flat) адресное пространство, размер которого зависит от используемого типа аппаратной платформы. Термин *линейное* обозначает, что адресное пространство состоит из одного диапазона адресов (например, 32-разрядное адресное пространство занимает диапазон адресов от 0 до 4294967295 в десятичной системе счисления). В некоторых операционных системах используется *сегментная модель адресного пространства* (segmented address space), в которой может существовать несколько диапазонов адресов, относящихся к разным сегментам памяти. В современных операционных системах с поддержкой виртуальной памяти обычно используется линейная, а не сегментная модель памяти. Как правило, для каждого процесса создается свое уникальное линейное адресное пространство. При этом один и тот же адрес памяти в адресном пространстве одного процесса никак не связан с аналогичным адресом в адресном пространстве другого процесса.

Однако возможны случаи, когда нескольким процессам назначается одно и то же адресное пространство. Такие процессы называются *потоками* (*threads*).

Адрес памяти, например `4021f000`, определяется числом, значение которого должно находиться в допустимых для данного адресного пространства диапазоне. Приведенное выше значение идентифицирует байт данных в 32-разрядном адресном пространстве. Несмотря на то что в 32-разрядном адресном пространстве процесса можно адресовать до 4 Гбайт памяти, процессу разрешено обращаться к данным, расположенным не по всем возможным значениям адресов. Важной частью адресного пространства являются диапазоны адресов памяти, к которым процесс имеет право доступа, как, например, `08048000-0804c000`. Эти диапазоны разрешенных адресов называются *областями памяти* (memory area). С помощью ядра процесс может динамически добавлять и удалять области памяти своего адресного пространства.

Процесс может обращаться только к разрешенным областям памяти. Каждой области памяти назначаются определенные права доступа, такие как чтение, запись или выполнение, которые процесс должен неукоснительно соблюдать. Если процесс обращается по адресу, который не относится к разрешенной области памяти, или если доступ к разрешенной области памяти выполняется некорректным образом, ядро уничтожает такой процесс со страшным сообщением “Segmentation Fault” (Ошибка сегментации).

В областях памяти может содержаться вся нужная процессу информация, которая описана ниже.

- Машинный код, загруженный из исполняемого файла в область памяти процесса, которая называется *сегментом кода* (text section).
- Инициализированные переменные, загруженные из исполняемого файла в область памяти процесса, которая называется *сегментом данных* (data section).
- Страницы памяти, заполненные нулями, в которых содержатся неинициализированные глобальные переменные программы. Эта область памяти называется *сегментом bss*[1] (bss section).
- Страницы памяти, заполненные нулями, в которых находится пользовательский стек процесса. Его не нужно путать со стеком ядра процесса, который является отдельной структурой данных и управляется и используется ядром.
- Дополнительные сегменты кода, данных и BSS для каждой совместно используемой библиотеки, такой как библиотека `libc` и динамический компоновщик, которые загружаются в адресное пространство процесса.
- Все файлы, содержимое которых отображено в память.
- Все области совместно используемой памяти.

[1] Термин *BSS* имеет исторические корни и означает Block Started by Symbol (блок, начинающийся с символа). Неинициализированные переменные не хранятся в исполняемом файле, поскольку с ними не связано никакое значение. Тем не менее в стандарте языка C требуется, чтобы неинициализированным переменным присваивались определенные стандартные значение (обычно нули). Поэтому ядро распределяет в памяти блок неинициализированных переменных, описанный в исполняемом файле, и обнуляет его. В результате всем неинициализированным переменным присваиваются нулевые значения и экономится место в объектном файле, поскольку отпадает необходимость хранить в нем блоки нулевых значений.

- Все анонимные отображения в память, как, например, связанные с функцией `malloc()`[2].

Каждое корректное значение адреса памяти в адресном пространстве процесса принадлежит одной и только одной области памяти (области памяти не перекрываются). Как будет показано ниже, для каждого отдельного участка памяти в выполняющемся процессе существует своя область: стек, объектный код, глобальные переменные, отображенный в память файл и т.д.

Дескриптор памяти

Адресное пространство процесса представляется в ядре в виде структуры данных, которая называется *дескриптором памяти* (memory descriptor). В этой структуре содержится вся информация, относящаяся к адресному пространству процесса. Дескриптор памяти представляется с помощью структуры `mm_struct`, которая определена в файле `<linux/mm_types.h>`. Эта структура вместе с поясняющими комментариями по каждому полю приведена ниже.

```
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 па-
мяти */
   unsigned long          free_area_cache; /* Первый незанятый участок адрес-
ного
                                              пространства */
   pgd_t                 *pgd;       /* Глобальный каталог страниц */
   atomic_t               mm_users;  /* Счетчик использования 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 распределенных
                                        физических страниц памяти */
   unsigned long          total_vm;  /* Общее количество страниц памяти */
   unsigned long          locked_vm; /* Количество заблокированных
                                        страниц памяти */
   unsigned long saved_auxv[AT_VECTOR_SIZE]; /* Сохраненный вектор auxv */
```

[2] В более новых версиях библиотеки `glibc` функция `malloc()` реализована не только через вызов системной функции `brk()`, но и через вызов системной функции `mmap()`.

```
    cpumask_t               cpu_vm_mask;/* Маска отложенного переключения
                                           буфера TLB */
    mm_context_t            context;    /* Данные, специфичные для
                                           аппаратной платформы */
    unsigned long           flags;      /* Флаги состояния */
    int                     core_waiters;/* количество потоков, ожидающих
                                            создания файла дампа */
    struct core_state      *core_state; /* Поддержка дампа */
    spinlock_t              ioctx_lock; /* Блокировка списка асинхронного
                                           ввода-вывода (AIO) */
    struct hlist_head       ioctx_list; /* Список асинхронного
                                           ввода-вывода (AIO) */
};
```

В поле `mm_users` хранится количество процессов, в которых используется данное адресное пространство. Например, если одно и то же адресное пространство используется в двух потоках, значение поля `mm_users` равно `2`. В поле `mm_count` хранится основной счетчик использования структуры `mm_struct`. Все потоки, использующие данное адресное пространство (их число хранится в поле `mm_users`), соответствуют одному использованию структуры `mm_struct`. Так, в предыдущем примере значение поля `mm_count` равно единице. Если адресное пространство будет использоваться в девяти потоках, то значение поля `mm_users` будет равно `9`, а значение поля `mm_count` будет снова равно единице. И только когда значение поля `mm_users` становится равным нулю (т.е. все потоки, в которых используется данное адресное пространство, завершились), значение счетчика `mm_count` уменьшается на единицу. Как только оно становится равным нулю (т.е. на структуру `mm_struct` в системе больше не осталось ссылок), структура `mm_struct` освобождается. Когда ядру нужно выполнить какие-либо операции над адресным пространством и обновить связанный с ним счетчик использования, оно увеличивает на единицу значение поля `mm_count`. Поддержка двух счетчиков позволяет ядру отличать счетчик использования собственно структуры (`mm_count`) от количества процессов, в которых используется данное адресное пространство (поле `mm_users`).

В полях `mmap` и `mm_rb` хранятся ссылки на две различные структуры данных, содержащие одну и ту же информацию: информацию обо всех областях памяти в соответствующем адресном пространстве. В первой структуре эта информация хранится в виде связанного списка, а во второй — в виде красно-черного дерева. Поскольку красно-черное дерево — это разновидность двоичного дерева, то, как и для всех типов двоичных деревьев, количество операций поиска заданного элемента в нем подчиняется закону `O(log(n))`. Более подробно красно-черные деревья будут описаны ниже, 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 оправдана. Структура данных `mmap` представляет собой связанный список, позволяющий очень быстро проходить по всем его элементам. С другой стороны, структура `mm_rb` представляет собой красно-черное дерево, которое очень хорошо подходит для поиска заданного элемента в нем. Подробнее области памяти будут рассмотрены ниже в главе. Структуры `mm_struct` не дублируются в ядре, дублируются только содержащиеся в них объекты. Наложение связанного списка на двоичное дерево и использование обеих структур данных для доступа к одной и той же информации иногда называют *переплетением деревьев* (*threaded tree*).

Все структуры mm_struct объединены в двухсвязный список с помощью полей mmlist. Первым элементом этого списка является дескриптор памяти init_mm, который описывает адресное пространство процесса init. Этот список защищен от конкурентного доступа с помощью блокировки mmlist_lock, которая определена в файле kernel/fork.c.

Выделение дескриптора памяти

Указатель на дескриптор памяти, выделенный для какой-либо задачи, хранится в поле mm дескриптора процесса этой задачи. Напомним, что дескриптор процесса представляется с помощью структуры task_struct, которая определена в файле <linux/sched.h>. Следовательно, выражение current->mm позволяет получить указатель на дескриптор памяти текущего процесса. Для копирования дескриптора родительского процесса в дескриптор порожденного процесса во время выполнения функции fork() используется функция copy_mm(). Память под структуру mm_struct выделяется из кеша блочного распределителя mm_cachep с помощью макроса allocate_mm(), определенного в файле kernel/fork.c. Обычно каждому процессу назначается уникальный экземпляр структуры mm_struct и соответственно уникальное адресное пространство.

Одно и то же адресное пространство может совместно использоваться как в главном процессе, так и в порожденных им процессах, если при вызове функции clone() был указан флаг CLONE_VM. Такие процессы называются *потоками* (thread). В этом и состоит *единственное* существенное отличие между обычными процессами и потоками в операционной системе Linux (подробнее об этом см. в главе 3, "Управление процессами"). Больше никаким другим образом в ядре Linux они не различаются. С точки зрения ядра потоки являются обычными процессами, совместно использующими некоторые общие ресурсы.

В случае, если 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 в функции copy_mm() с помощью приведенного ниже условного оператора.

```
if (clone_flags & CLONE_VM) {
   /*
    * current — это родительский процесс,
    * tsk — это процесс, порожденный с помощью функции fork()
    */
    atomic_inc(&current->mm->mm_users);
    tsk->mm = current->mm;
}
```

Удаление дескриптора памяти

После завершения процесса, связанного с определенным адресным пространством, вызывается функция exit_mm(), определенная в файле kernel/exit.c. В этой функции выполняются некоторые служебные действия и обновляется ряд статистической информации. Далее вызывается функция mmput(), в которой значение счетчика количества пользователей mm_users для дескриптора памяти уменьшается на единицу. Когда значение этого счетчика становится равным нулю, вызывается функция mmdrop(), в которой значение основного счетчика использования mm_count уменьшается на единицу. Когда и *этот* счетчик использования наконец достигнет нулевого значения, вызывается

функция free_mm(). Поскольку данный дескриптор памяти больше нигде не используется, то в ней экземпляр структуры mm_struct возвращается в кеш блочного распределителя памяти mm_cachep с помощью вызова функции kmem_cache_free().

Структура mm_struct и потоки ядра

Потоки, выполняющиеся в пространстве ядра, не имеют своего адресного пространства и, следовательно, связанного с ним дескриптора памяти. Поэтому значение поля mm для потока пространства ядра равно NULL. Потоки ядра *определяются* как процессы, не имеющие пользовательского контекста.

Поскольку потоки ядра вообще не обращаются к памяти в пространстве пользователя (действительно, зачем им вообще куда-либо обращаться?), то собственное адресное пространство им не нужно. Так как для потоков ядра вообще не выделяются страницы памяти, принадлежащие пространству пользователя, то им также не нужен собственный дескриптор памяти и таблицы страниц, которые описаны ниже в главе. Тем не менее потокам ядра все же нужны некоторые структуры данных, такие как таблицы страниц, чтобы обращаться к памяти ядра. Поэтому, чтобы обеспечить потоки ядра всеми необходимыми данными и при этом не расходовать зря память под ненужные дескрипторы и таблицы страниц, а также сэкономить процессорное время, требуемое для переключения на новое адресное пространство при запуске потока ядра, для потоков ядра используются дескрипторы памяти задания, которое выполнялось перед этим.

Когда процесс запланирован на выполнение, загружается адресное пространство, на которое указывает поле mm дескриптора этого процесса. Далее обновляется значение поля active_mm дескриптора процесса, чтобы оно указывало на новое адресное пространство. Поскольку потоки ядра не имеют своего адресного пространства, значение поля mm для них равно NULL. Поэтому, когда поток ядра планируется на выполнение, ядро определяет, что значение поля mm равно NULL, и оставляет загруженным предыдущее адресное пространство. После этого ядро обновляет значение поля active_mm дескриптора процесса для потока ядра, чтобы он указывал на дескриптор памяти предыдущего процесса. По мере необходимости в потоке ядра могут использоваться таблицы страниц предыдущего процесса. Так как потоки ядра не обращаются к памяти в пространстве пользователя, они используют только ту информацию об адресном пространстве ядра, которая связана с памятью ядра и является общей для всех процессов.

Области виртуальной памяти

Области памяти (memory areas) представляются с помощью структуры vm_area_struct, которая определена в файле <linux/mm_types.h>. В ядре Linux области памяти часто называются *областями виртуальной памяти* (*virtual memory area*, или *VMA*).

Структура vm_area_struct используется для описания одной непрерывной области памяти в данном адресном пространстве. В ядре каждая область памяти считается уникальным объектом. Для каждой области памяти определены некоторые общие свойства, такие как права доступа и набор соответствующих операций. Таким образом, каждая структура VMA может представлять различный тип области памяти, например файлы, отображаемые в память, или стек пользовательского приложения. Это аналогично объектно-ориентированному подходу, который используется в подсистеме VFS (см. главу 13, “Виртуальная файловая система”). Эта структура данных с комментариями, описывающими назначение каждого поля, приведена ниже.

```
struct vm_area_struct {
   struct mm_struct *vm_mm;   /* Соответствующая структура mm_struct */
   unsigned long     vm_start;/* Начало диапазона адресов (включительно) */
   unsigned long     vm_end;  /* Конец диапазона адресов (исключая) */
   struct vm_area_struct *vm_next; /* Список областей VMA */
   pgprot_t          vm_page_prot; /* Права доступа */
   unsigned long     vm_flags;/* Флажки */
   struct rb_node    vm_rb;   /* Узел текущей области VMA в дереве */
   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; /* Элемент анонимной области */
   struct anon_vma  *anon_vma;      /* Объект анонимной VMA */
   struct vm_operations_struct *vm_ops; /* Связанные операции */
   unsigned long     vm_pgoff; /* Смещение в файле */
   struct file      *vm_file;  /* Отображенный файл (если есть) */
   void *vm_private_data;      /* Частные данные */
};
```

Как уже было сказано, каждый дескриптор памяти связан с уникальным диапазоном адресов в адресном пространстве процесса. В поле `vm_start` хранится начальный (т.е. самый младший) адрес этого диапазона, а в поле `vm_end` — адрес первого байта, расположенного после описываемого диапазона (т.е. самый старший адрес диапазона плюс `1`). Таким образом, в поле `vm_start` хранится начальный адрес диапазона памяти (включительно), а в поле `vm_end` — его конечный адрес (исключая последний байт). Поэтому выражение `(vm_end – vm_start)` позволяет определить размер в байтах области памяти, расположенной в диапазоне адресов `[vm_start, vm_end)`. Диапазоны адресов разных областей памяти одного адресного пространства не могут перекрываться.

В поле `vm_mm` хранится указатель на структуру `mm_struct`, связанную с данной областью VMA. Заметим, что в области памяти, с которой связана структура `mm_struct`, создается уникальная область виртуальной памяти VMA. Поэтому, даже если один и тот же файл отображается в адресные пространства двух разных процессов, для каждого процесса создается своя структура `vm_area_struct`, чтобы идентифицировать уникальные области памяти каждого процесса. Следовательно, если адресное пространство совместно используется в двух потоках, то также совместно используются и все структуры `vm_area_struct` в этом адресном пространстве.

Флаги областей VMA

В поле `vm_flags` находятся 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, в поле `vm_flags` содержится информация, относящаяся к каждой странице в области памяти или, что то же самое, ко всей области памяти в целом, а не к отдельным указанным страницам. Возможные значения флагов `vm_flags` и их действие на область VMA и ее страницы памяти приведены в табл. 15.1.

Таблица 15.1. Флаги областей VMA

Флаг	Описание
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 общей (совместно используемой) памяти
VM_DENYWRITE	В область отображается файл, в который нельзя выполнять запись
VM_EXECUTABLE	В область отображается исполняемый 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 произвольный доступ
VM_DONTCOPY	Область памяти не должна копироваться при вызове функции fork()
VM_DONTEXPAND	Область памяти не может быть увеличена с помощью вызова функции mremap()
VM_RESERVED	Область памяти не должна вытесняться 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

А теперь рассмотрим подробнее назначение самых интересных и важных флагов. Флаги VM_READ, VM_WRITE и VM_EXEC определяют обычные права доступа на чтение-запись и выполнение кода для страниц памяти, которые принадлежат *данной конкретной области памяти*. По мере необходимости их можно комбинировать для формирования требуемых прав доступа, которые должен соблюдать процесс, обращающийся к данной области VMA. Например, для областей памяти, содержащих исполняемый код процесса, должны быть указаны флаги VM_READ и VM_EXEC, но никак не VM_WRITE. С другой стороны, для области памяти, содержащей сегмент данных из исполняемого файла, должны быть указаны флаги VM_READ и VM_WRITE, указывать при этом флаг VM_EXEC не имеет особого смысла. Если в область данных отображается файл, предназначенный только для чтения, то нужно указать только флаг VM_READ.

Флаг VM_SHARED указывает на то, что в области памяти содержатся отображенные данные, которые могут совместно использоваться в нескольких процессах. 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 на область памяти отображена область ввода-вывода аппаратного устройства. Этот флаг обычно устанавливается драйверами устройств при вызове функции mmap() для отображения в память области ввода-вывода аппаратного устройства. Кроме всего прочего, этот флаг указывает, что область памяти не должна включаться в файл дампа памяти процесса. Флаг VM_RESERVED указывает, что область памяти не должна вытесняться на диск. Этот флаг также указывается при отображении в память областей ввода-вывода в драйверах аппаратных устройств.

Флаг VM_SEQ_READ является подсказкой ядру, что приложение выполняет последовательное (т.е. линейное и непрерывное) чтение данных из соответствующего отображения. При этом ядро может повысить скорость 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 вызова системной функции madvice(), которой передаются соответственно флаги MADV_SEQUENTIAL и MADV_RANDOM. 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 произвольным образом, то режим упреждающего чтения не эффективен.

Операции с областями VMA

В поле vm_ops структуры vm_area_struct содержится указатель на таблицу операций, связанных с данной областью памяти, которые может вызывать ядро для манипуляций с ней. Структура vm_area_struct служит обобщенным объектом, служащим 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 *);
   int (*fault) (struct vm_area_struct *, struct vm_fault *);
   int (*page_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
   int (*access) (struct vm_area_struct *, unsigned long ,
                                  void *, int, int);
};
```

Ниже приведено описание каждого метода.

- `void open(struct vm_area_struct *area)`

  Вызывается, когда соответствующая область памяти добавляется к адресному пространству.

- `void close(struct vm_area_struct *area)`

Вызывается, когда соответствующая область памяти удаляется из адресного пространства.

- `int fault(struct vm_area_sruct *area, struct vm_fault *vmf)`

  Вызывается обработчиком прерывания из-за отсутствия страницы (page fault), которое имеет место при доступе к странице, отсутствующей в физической памяти.

- `int page_mkwrite(struct vm_area_sruct *area, struct vm_fault *vmf)`

  Вызывается обработчиком прерывания из-за отсутствия страницы (page fault), которое имеет место при записи данных в страницу, предназначенную только для чтения.

- `int access(struct vm_area_struct *vma, unsigned long address, void *buf, int len, int write)`

  Вызывается из функции `access_process_vm()`, когда функция `get_user_pages()` завершается неудачей.

Списки и деревья областей памяти

Как уже отмечалось выше, к областям памяти осуществляется доступ с помощью двух структур данных, указатели на которые содержатся в полях `mmap` и `mm_rb` дескриптора памяти. В этих двух структурах данных независимо друг от друга содержатся ссылки на все области памяти, связанные с данным дескриптором памяти. По сути, в них содержатся указатели на одни и те же структуры `vm_area_struct`, просто эти указатели представлены в разном виде.

В первом поле `mmap` содержится указатель на односвязный список, в котором собраны все объекты областей памяти. В каждой структуре типа `vm_area_struct` имеется поле `vm_next`, в котором находится указатель на следующую структуру типа `vm_area_struct`. Области памяти в списке отсортированы в порядке увеличения адресов (от наименьшего до наибольшего). В поле `mmap` дескриптора памяти находится указатель на первую структуру типа `vm_area_struct`. В поле `vm_next` последней структуры типа `vm_area_struct` находится значение `NULL`.

Во втором поле `mm_rb` находится указатель на корневой элемент красно-черного дерева, содержащего все объекты областей памяти. Каждая структура `vm_area_struct` текущего адресного пространства связывается с деревом через поле `vm_rb`.

Красно-черное дерево является сбалансированным двоичным деревом. Каждый элемент красно-черного дерева называется *узлом* (node). Начальный узел называется *корневым* (root) элементом дерева. У большинства узлов имеются два дочерних узла: левый и правый. У некоторых узлов есть только один дочерний узел, а у заключительных узлов дерева, называемых *листьями* (leave), вообще нет потомков. Для любого узла дерева справедливо следующее: 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 данного узла, всегда больше по значению, чем значение этого узла. Более того, каждому узлу назначается свой цвет (красный или черный, отсюда и название этого типа деревьев) в соответствии с перечисленными ниже двумя правилами. Дочерние элементы красного узла являются черными, и любой путь по дереву от узла к листьям должен проходить через одинаковое количество черных узлов. Корневой

узел дерева всегда красный. Для поиска, вставки и удаления элементов из такого дерева требуется выполнить порядка O(log(n)) операций.

Связанный список используется, когда необходимо пройти по всем его элементам. Красно-черное дерево используется, когда в адресном пространстве необходимо найти определенную область памяти. Таким образом, в ядре используются избыточные структуры данных для обеспечения оптимальной производительности, независимо от того, какие операции выполняются с областями памяти.

Области памяти в реальных приложениях

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 приведенную ниже простую 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 будет немного. Мы уже знаем, что среди них есть сегмент кода, сегмент данных, сегмент bss. Предположим, эта программа динамически скомпонована с библиотекой функций языка C. Поэтому соответствующие области памяти должны существовать также для модуля libc.so и для модуля ld.so. И наконец, среди областей памяти должен быть также стек процесса.

Ниже приведен результат вывода списка областей адресного пространства этого процесса из файла /proc/<pid>/maps.

```
rlove@wolf:~$ cat /proc/1426/maps
00e80000-00faf000 r-xp 00000000 03:01 208530    /lib/tls/libc-2.5.1.so
00faf000-00fb2000 rw-p 0012f000 03:01 208530    /lib/tls/libc-2.5.1.so
00fb2000-00fb4000 rw-p 00000000 00:00 0
08048000-08049000 r-xp 00000000 03:03 439029    /home/rlove/src/example
08049000-0804a000 rw-p 00000000 03:03 439029    /home/rlove/src/example
40000000-40015000 r-xp 00000000 03:01 80276     /lib/ld-2.5.1.so
40015000-40016000 rw-p 00015000 03:01 80276     /lib/ld-2.5.1.so
4001e000-4001f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Информация об областях памяти выводится в следующем формате:

```
начало-конец   права доступа   смещение   старший:младший   индекс   файл
```

Чтобы вывести эту информацию в более удобочитаемом виде, воспользуемся утилитой pmap(1)[3].

```
rlove@wolf:~$ pmap 1426
example[1426]
00e80000 (1212 KB) r-xp (03:01 208530)   /lib/tls/libc-2.5.1.so
00faf000 (12 KB)   rw-p (03:01 208530)   /lib/tls/libc-2.5.1.so
```

[3] Утилита pmap(1) отображает отформатированный 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.

```
00fb2000 (8 KB)    rw-p (00:00 0)
08048000 (4 KB)    r-xp (03:03 439029)   /home/rlove/src/example
08049000 (4 KB)    rw-p (03:03 439029)   /home/rlove/src/example
40000000 (84 KB)   r-xp (03:01 80276)    /lib/ld-2.5.1.so
40015000 (4 KB)    rw-p (03:01 80276)    /lib/ld-2.5.1.so
4001e000 (4 KB)    rw-p (00:00 0)
bfffe000 (8 KB)    rwxp (00:00 0) [ stack ]
mapped: 1340 KB writable/private: 40 KB shared: 0 KB
```

В первых трех строках выводится информация о сегментах кода, данных и `bss` модуля `libc.so` библиотеки языка C. В следующих двух стоках приведена информация о сегментах кода и данных нашей программы. В следующих трех строках выводится информация о сегментах кода, данных и `bss` модуля `ld.so` динамического компоновщика. В последней строке (перед статистическими данными) содержится информация о стеке процесса.

Обратите внимание на то, что всем областям памяти, содержащим сегменты кода, присвоены атрибуты, разрешающие чтение данных и выполнение кода, что и можно было ожидать для исполняемого кода программы. С другой стороны, областям памяти, содержащим сегменты данных и `bss` (в обоих из них содержатся глобальные переменные), присвоены атрибуты, разрешающие чтение и запись данных, но никак не выполнение кода. Областям памяти, содержащим стек, разумеется, присвоены атрибуты, разрешающие чтение и запись данных, а также выполнение кода. Столь обширными правами не обладает ни одна область памяти.

Общий размер адресного пространства составляет 1340 Кбайт, но только 40 Кбайт из него доступно для записи и размещения данных программы. Если область памяти является совместно используемой или доступна только для чтения, ядро сохраняет в памяти только одну копию отображаемого файла. Такой подход очевиден для областей памяти, содержащих совместно используемые данные, но если область доступна только для чтения, вас это может немного смутить. Но если вспомнить, что данные в таких областях памяти никогда не изменяются, то станет очевидно, что в таких случаях будет достаточно загрузить в память только одну копию образа файла. Поэтому динамически загружаемая библиотека функций языка C занимает в памяти всего 1212 Кбайт, а не 1212 Кбайт, умноженное на количество процессов, в которых эта библиотека используется. Поскольку для нашего процесса доступно 1340 Кбайт памяти для размещения кода и данных и при этом реально используется всего 40 Кбайт физической памяти, в результате использования общих областей получается довольно существенная экономия памяти.

Обратите внимание на области памяти, в которые не отображаются файлы. Они находятся на устройстве с номерами `00:00` и имеют номер файлового индекса, равный нулю. В эти области памяти отображаются страницы, заполненные нулями. Если отобразить такую страницу на область памяти, которая имеет права на запись, то в результате “побочного эффекта” все находящиеся в ней переменные будут проинициализированы нулевыми значениями. Это важно, поскольку в таком случае получается область памяти, заполненная нулями, которая нужна для размещения сегмента `bss`. Поскольку данная область памяти не является совместно используемой, то при первой же записи в нее данных со стороны процесса на лету создается копия страницы (так называемое копирование при записи), и ненулевые значения записываются уже в новую страницу памяти.

Каждой области памяти, связанной с процессом, соответствует структура `vm_area_struct`. Поскольку процесс не является потоком (thread), то для него существует отдельная структура `mm_struct`, на которую есть ссылка из структуры `task_struct`.

Работа с областями памяти

Операции с областями памяти выполняются в ядре довольно часто. Например, ядру может потребоваться выяснить, существует ли в заданном адресном пространстве некоторая область виртуальной памяти (VMA). Эти операции положены в основу функции `mmap()`, которая будет рассмотрена в следующем разделе. Для удобства на основе этой функции создан ряд вспомогательных функций, которые определены в файле `<linux/mm.h>`.

Функция `find_vma()`

Для поиска VMA, в котором расположен указанный адрес памяти, в ядре предусмотрена функция `find_vma()`. Она определяется в файле `mm/mmap.c`, как показано ниже.

```
struct vm_area_struct * find_vma(struct mm_struct *mm, unsigned long
addr);
```

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 виртуальной памяти, в которой содержится адрес `addr` или которая 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 на следующее: найденная область VMA может начинаться с адреса, большего адреса `addr`, т.е. указанный адрес *может не принадлежать* найденной области памяти. Результат выполнения функции `find_vma()` кешируется в поле `mmap_cache` дескриптора памяти. Поскольку очень велика вероятность того, что после одной операции с областью памяти последуют еще операции с ней же, то эффективность кеширования получается достаточно высокой (на практике получаются значения порядка 30–40%). Проверка кешированных результатов выполняется очень быстро. Если нужный адрес в кеше 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 `mmap_cache` на предмет того, содержит ли кешированная область VMA необходимый адрес. Однако 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 кеш в этой ситуации оказался полезным, проверяемый адрес должен принадлежать кешированной области памяти. К счастью, это как раз и соответствует случаю выполнения последовательных операций с одной и той же областью VMA.

Если нужная область VMA не содержится в кеше, то в функции выполняется поиск по красно-черному дереву 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, содержащая заданный адрес `addr`. Если такая область VMA не найдена, то поиск по дереву продолжается и функция возвращает первую найденную область памяти, которая начинается после адреса `addr`. Если не будет найдено ни одной области памяти, то возвращается значение `NULL`.

Функция `find_vma_prev()`

Функция `find_vma_prev()` аналогична функции `find_vma()`, но дополнительно еще возвращает последнюю область VMA, заканчивающуюся перед адресом `addr`. Эта функция также определяется в файле `mm/mmap.c`, а ее прототип описан в файле `<linux/mm.h>`, как показано ниже.

```
struct vm_area_struct * find_vma_prev(struct mm_struct *mm, unsigned long addr,
                            struct vm_area_struct **pprev)
```

После возвращения из функции в параметре `pprev` будет находиться указатель на предыдущую область VMA.

Функция `find_VMA_intersection()`

Функция `find_vma_intersection()` возвращает первую область VMA, которую перекрывает указанный диапазон адресов. Эта функция определена в файле `<linux/mm.h>`, поскольку она является встроенной.

```
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;
}
```

В первом параметре указывается адресное пространство, в котором выполняется поиск. Начальный и конечный адреса диапазона указываются в параметрах `start_addr` и `end_addr` соответственно.

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 диапазона адресов

Функция `do_mmap()` используется в ядре для создания нового линейного диапазона адресов. Говорить, что эта функция создает новую область VMA, — технически не корректно, поскольку если создаваемый диапазон адресов является смежным с существующим диапазоном адресов и у этих диапазонов одинаковые права доступа, то два диапазона объединяются в один. Если это невозможно, то создается новая область VMA. В любом случае функция `do_mmap()` используется для добавления диапазона адресов к адресному пространству процесса, независимо от того, создается ли при этом новая область VMA или расширяется существующая.

Функция `do_mmap()` объявляется 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)
```

Данная функция отображает в память содержимое файла `file`, начиная с позиции в файле `offset`; размер отображаемого участка равен `len` байт. Если значение параметра `file` равно `NULL`, а `offset` — нуль, то в этом случае содержимое памяти отображения не будет сохраняться в файле. Такое отображение называется *анонимным* (anonymous mapping). Если указан и файл, и смещение, то выполняется *отображение файла* в память (file-backed mapping).

Параметр `addr` дополнительно определяет начальный адрес, откуда будет выполняться поиск свободного диапазона адресов.

В параметре `prot` указываются права доступа для страниц памяти, расположенных в данной области. Возможные значение флагов зависят от аппаратной платформы и описываются в файле `<asm/mman.h>`. Однако на практике для всех аппаратных платформ определены флаги, описанные в табл. 15.2.

Таблица 15.2. Права доступа для страниц памяти VMA

Флаг	Влияние на страницы памяти в созданном диапазоне адресов
`PROT_READ`	Соответствует флагу `VM_READ`
`PROT_WRITE`	Соответствует флагу `VM_WRITE`

Окончание табл. 15.2

Флаг	Влияние на страницы памяти в созданном диапазоне адресов
PROT_EXEC	Соответствует флагу VM_EXEC
PROT_NONE	К страницам памяти нет доступа

В параметре flags указываются все остальные флаги области VMA (табл. 15.3). Они определяют тип и режим работы отображенной области памяти и также описаны в файле <asm/mman.h>.

Таблица 15.3. Флаги, определяющие тип отображения

Флаг	Влияние на созданный диапазон адресов
MAP_SHARED	Отображение может быть совместно используемым
MAP_PRIVATE	Отображение не может быть совместно используемым
MAP_FIXED	Создаваемый диапазон адресов *должен* начинаться с указанного адреса addr
MAP_ANONYMOUS	Отображение является анонимным и не связано с файлом
MAP_GROWSDOWN	Соответствует флагу VM_GROWSDOWN
MAP_DENYWRITE	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 противном случае в области виртуальной памяти создается необходимый диапазон адресов. Если это возможно, то данный диапазон адресов объединяется с соседней областью памяти. Если это невозможно, то создается новая структура vm_area_struct, которая выделяется из кеша блочного распределителя 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 диапазона адресов.

К функции do_mmap() можно обращаться из пользовательских приложений с помощью вызова системной функции mmap(), которая определена следующим образом:

```
void * mmap2(void *start,
             size_t length,
             int prot,
             int flags,
             int fd,
             off_t pgoff)
```

Данная функция названа mmap2(), так как это второй вариант функции mmap(). В первоначальном варианте функции, mmap(), в последнем параметре передавалось смещение в байтах, а в текущем варианте, mmap2(), — смещение, выраженное в страницах

памяти. Это позволяет отображать файлы большего размера с бо́льшим значением смещения. Первоначальный вариант функции, `mmap()`, соответствующий стандарту POSIX, реализован в виде одноименной функции библиотеки языка C, но в ядре он больше не поддерживается, несмотря на то, что доступна новая версия функции — `mmap2()`. В обеих библиотечных функциях используется вызов системной функции `mmap2()`, только в функции `mmap()` выполняется пересчет смещения из байтов в страницы.

Функции `munmap()` и `do_munmap()`: удаление диапазона адресов

Функция `do_munmap()` удаляет диапазон адресов из указанного адресного пространства процесса. Она описана в файле `<linux/mm.h>`, как показано ниже.

```
int do_munmap(struct mm_struct *mm, unsigned long start, size_t len)
```

В первом параметре указывается адресное пространство, из которого удаляется диапазон адресов, начинающийся с адреса `start` и длиной `len` байтов. При успешном завершении возвращается нулевое значение, а в случае ошибки — отрицательное значение кода ошибки.

Системная функция `munmap()` экспортируется в пространство пользователя и позволяет процессу самостоятельно удалить диапазон адресов из своего адресного пространства. Эта функция выполняет действия, противоположные функции `mmap()`:

```
int munmap(void *start, size_t length)
```

Данная системная функция реализована в файле `mm/mmap.c` 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);
   up_write(&mm->mmap_sem);
   return ret;
}
```

Таблицы страниц

Несмотря на то что пользовательские программы обращаются к памяти по виртуальным адресам, которые затем преобразовываются в физические адреса, процессоры при доступе к памяти используют только физические адреса. Следовательно, чтобы процессор мог выполнить запрос прикладной программы на доступ к памяти, сначала нужно каким-то образом преобразовать виртуальный адрес в физический. Такое преобразование выполняется с помощью таблиц страниц. При этом виртуальный адрес разбивается на несколько частей, каждой из которых соответствует своя таблица страниц. Каждая часть виртуального адреса на самом деле является индексом в одной из таблиц страниц. Информация, хранящаяся в элементе таблицы страниц, соответствующем заданному индексу, является либо адресом другой таблицы страниц, либо адресом физической страницы памяти, в которой хранятся нужные данные.

В операционной системе Linux таблицы страниц являются трехуровневыми. Несколько уровней позволяют эффективно поддерживать неравномерно заполненные адресные пространства даже для 64-разрядных машин. Если бы таблицы страниц были реализованы 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, в которых аппаратно не поддерживаются трехуровневые таблицы страниц. Например, для некоторых аппаратных платформ поддерживается только два уровня таблиц страниц или хеширование, реализованное на аппаратном уровне. Три уровня соответствуют своего рода "наибольшему общему знаменателю". Для аппаратных платформ с менее сложной реализацией работа с таблицами страниц в ядре по необходимости может быть упрощена с помощью параметров оптимизации компилятора.

Таблица страниц самого верхнего уровня называется *глобальным каталогом страниц* (page global directory, PGD). Он представляет собой массив элементов типа `pgd_t`. Для большинства аппаратных платформ тип `pgd_t` соответствует типу `unsigned long`. Элементы в таблице PGD содержат указатели на каталоги страниц более низкого уровня, PMD.

Каталоги страниц второго уровня называются *каталогами страниц среднего уровня* (page middle directory, PMD). Каждый каталог PMD представляет собой массив элементов типа `pmd_t`. Элементы таблиц PMD содержат указатели на таблицы PTE (page table entry — запись таблицы страниц).

На последнем уровне находятся обычные таблицы страниц, состоящие из элементов типа `pte_t`. Записи таблиц страниц содержат указатели на страницы физической памяти.

Для большинства аппаратных платформ просмотр элементов таблиц страниц выполняется специальным блоком процессора (по крайней мере, частично). Обычно бо́льшая часть работы по использованию таблиц страниц выполняется аппаратным обеспечением. Однако перед этим ядро должно все настроить так, чтобы аппаратное обеспечение могло нормально работать. На рис. 15.1 показана схема преобразования виртуального адреса в физический с помощью таблиц страниц.

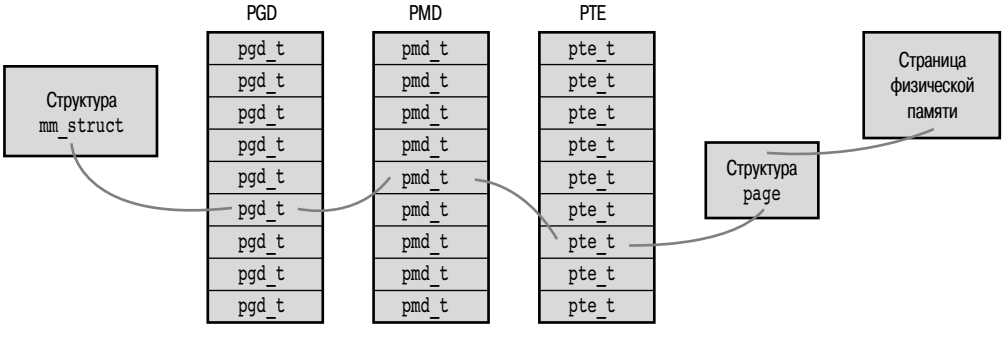


Рис. 15.1. Схема преобразования виртуального адреса в физический

Для каждого процесса создается свой набор таблиц страниц. Разумеется, все потоки пользуются общим набором таблиц страниц родительского процесса. Указатель на глобальный каталог страниц текущего процесса хранится в поле `pgd` дескриптора памяти. При выполнении операций с таблицами страниц и прохождении по ним нужно захватить блокировку `page_table_lock`, которая также находится в соответствующем поле дескриптора памяти.

Структуры данных, связанные с таблицами страниц, полностью зависят от используемой аппаратной платформы и определяются в файле <asm/page.h>.

Поскольку при каждом обращении к странице виртуальной памяти требуется определить соответствующий физический адрес в памяти, скорость выполнения операций с таблицами страниц является очень критичной. К сожалению, преобразование всех этих адресов должно всегда выполняться очень быстро. Поэтому для ускорения процесса преобразования в большинстве процессоров предусмотрен специальный *буфер быстрого преобразования адреса (ББП)* (translation lookaside buffer, *TLB*), который, по сути, является аппаратным кешем, в котором сохраняются соответствия виртуальных адресов физическим адресам. При обращении программы по виртуальному адресу процессор вначале просматривает ББП. Если соответствующий виртуальный адрес там найден, то физический адрес определяется сразу же. В противном случае выполняется полный цикл преобразования виртуального адреса в физический с помощью таблиц страниц.

Несмотря на это, управление таблицами страниц все же остается критичной и развивающейся частью ядра. В ядре серии 2.6 был изменен механизм размещения элементов таблиц страниц, которые были вынесены из верхней памяти. В будущем планируется создать механизм совместного использования таблиц страниц с копированием при записи. При этом можно будет пользоваться общими таблицами страниц в родительском и порожденных процессах сразу после вызова функции `fork()`. Если же некоторый элемент таблицы страниц изменяется в родительском или порожденном процессе, то создается копия страницы памяти, содержащей таблицу страниц, и с этого момента процессы больше не будут совместно использовать общую таблицу страниц. Совместное использование таблиц страниц позволит сэкономить память, которая расходуется на копирование таблиц страниц при вызове функции `fork()`.

Резюме

В этой чрезвычайно переполненной новой информацией главе мы рассмотрели абстракцию виртуальной памяти, которая выделяется каждому процессу. Было описано, как в ядре представляются адресное пространство процесса (с помощью структуры `mm_struct`) и области виртуальной памяти в этом пространстве (с помощью структуры `vm_area_struct`). Кроме того, вы ознакомились с процессом создания (с помощью функции `mmap()`) и удаления (с помощью функции `munmap()`) этих областей памяти. В конце главы были рассмотрены таблицы страниц. Поскольку Linux является операционной системой с поддержкой виртуальной памяти, то все эти понятия очень важны для понимания работы системы и используемой модели процессов.

В следующей главе будет рассмотрен страничный кеш — универсальный кеш данных в памяти, который используется для выполнения страничных операций ввода-вывода. Кроме того, вы узнаете о том, как в ядре выполняется отложенная запись данных из измененных страниц.

16
Страничный кеш и отложенная запись страниц

В ядре Linux реализована дисковая кеш-память, которая называется *страничным кешем* (page cache). Он предназначен для уменьшения количества дисковых операций ввода-вывода за счет хранения часто используемых данных в физической памяти компьютера. В этой главе описаны страничный кеш и процесс, благодаря которому изменения, внесенные в станичный кеш, сохраняются на диске. Этот процесс называется *отложенной записью страниц* (page writeback).

Дисковый кеш является важным компонентом любой современной операционной системы по двум причинам. Во-первых, доступ к данным, хранящимся на диске, осуществляется на несколько порядков медленнее, чем доступ к оперативной памяти (миллисекунды против наносекунд). С точки зрения процессора доступ к данным, расположенным в оперативной памяти компьютера, выполняется намного быстрее, чем на диске, а выборка данных из кешей первого и второго уровня (L1 и L2) процессора выполняется еще быстрее. Во-вторых, если к некоторым данным осуществлялся доступ, то с достаточно большой степенью вероятности 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 *временной локализацией* (temporal locality). Этот принцип гарантирует, что если данные кешируются при первом доступе к ним, то существует большая вероятность того, что при следующем обращении к ним в ближайшем будущем они будут находиться в кеш-памяти. Тот факт, что доступ к памяти выполняется намного быстрее, чем к диску, а также высокая вероятность повторного использования данных, находящихся в кеше, позволяют добиться существенного увеличения производительности системы в целом.

Методики кеширования

Страничный кеш состоит из страниц физической памяти, содержимое которых соответствует физическим блокам на диске. Размер страничного кеша может динамически изменяться. При наличии свободной памяти он увеличивается, а при недостатке физической памяти — уменьшается. Устройство хранения данных, которое кешируется, называется *внешней памятью* (backing store), поскольку при этом диск как бы расположен за пределами кеша и служит первоисточником данных, хранящихся в нем. Перед тем как начать операцию чтения, например, если из приложения вызывается функция `read()`, ядро сначала проверяет, находятся ли нужные данные в страничном кеше. Если они там есть, то дисковая операция не выполняется, а данные копируются прямо из памяти. Этот процесс называется *удачным обращением в кеш* (*cache hit*). Если нужных данных в кеше нет (это называется *неудачным обращением в кеш* (cache miss)), ядро должно выполнять ряд блочных операций ввода-вывода, чтобы прочитать нужные данные с диска. После того как данные будут прочитаны с диска, ядро помещает их в станичный кеш, и все последующие чтения тех же самых данных будут выполняться прямо из кеша. При этом кешировать полностью все данные файла не нужно. В зависимости от ситуации в страничном кеше могут находиться все данные из нескольких файлов и только несколько страниц данных из других файлов. Кешируются только те данные, к которым выполняется доступ.

Кеширование при записи

Выше был описан процесс чтения данных с диска и помещения их в страничный кеш. Но что произойдет в случае, если процессу нужно записать данные на диск, например, через вызов системной функции `write()`? По сути, при реализации кеш-памяти может использоваться одна из трех стратегий. В первой стратегии, которая называется *без кеширования* (nowrite), записываемые на диск данные просто не кешируются. Если в кеше находится фрагмент данных, соответствующий тому, который в настоящий момент выводится процессом на диск, то этот фрагмент в кеш-памяти аннулируется. В результате при следующей операции считывания этого фрагмента с диска он будет снова загружен в кеш-память. Первая стратегия очень редко используется при реализации кеш-памяти, так как при этом не только не кешируются записываемые на диск данные, но еще и аннулируется содержимое кеша. В результате снижается производительность дисковой подсистемы и увеличиваются накладные расходы.

При использовании второй стратегии кеширования данные, записываемые на диск, одновременно обновляются как в дисковом файле, так и в памяти. Этот подход называется *кешем со сквозной записью* (write-through cache), поскольку данные после обновления кеш-памяти незамедлительно сбрасываются на диск. При его использовании содержимое кеш-памяти и внешней памяти находится в согласованном состоянии. При этом не требуется аннулировать содержимое кеш-памяти, и, кроме того, данная стратегия очень проста.

Третья стратегия, которая взята на вооружение в ОС Linux, называется *с отложенной записью* (write-back)[1]. При использовании этой стратегии данные при записи помещаются

[1] В некоторых учебниках по операционным системам данная стратегия называется с *отложенным копированием* (copy-back) или с *обратной записью* (write-behind). Все три названия являются синонимами.

непосредственно в страничный кеш. Незамедлительное или непосредственное обновление внешней памяти не выполняется. Вместо этого страницы с обновленными данными в кеше помечаются как *не сохраненные* (dirty) и помещаются в *список не сохраненных страниц* (dirty list). В системе существует специальный процесс, называемый `write-back`, который периодически сохраняет страницы, находящиеся в этом списке, на диск. В результате данные, находящиеся в памяти и на диске, синхронизируются. После этого признак не сохранения страниц сбрасывается. Таким образом, при использовании кеша с отложенной записью актуальность данных поддерживается только в оперативной памяти компьютера. До сохранения на диске находятся устаревшие данные, т.е. происходит так называемая *рассинхронизация* данных. Стратегия с отложенной записью считается более прогрессивной по сравнению со стратегией со сквозной записью. Из-за переноса операции сохранения на более поздний срок система может собрать все измененные до этого момента данные и записать их на диск, что называется, одним махом. Безусловно, производительность дисковой подсистемы при такой стратегии резко возрастает, однако недостатком является сложность ее реализации.

Вытеснение данных из кеша

Нам осталось рассмотреть заключительный момент процесса кеширования — удаление данных из кеша. Эта операция может понадобиться либо для освобождения памяти под более подходящие порции данных, либо когда нужно уменьшить размер страничного кеша при недостатке оперативной памяти в системе, чтобы ею могли воспользоваться другие процессы. Под *вытеснением данных из кеша* (cache eviction) будем понимать процесс удаления данных, а также используемую при этом стратегию, определяющую, что именно требуется удалить. Вытеснение данных в Linux выполняется путем простой выборки синхронизированных (т.е. тех, что уже сохранены на диске) страниц памяти и замещения их содержимого чем-нибудь другим. Если таких страниц недостаточно в кеше, ядро запускает процесс отложенной записи, чтобы можно было освободить дополнительные страницы. Самое сложное здесь решить, что именно нужно вытеснять. В идеале, конечно, нужно избавляться от тех страниц памяти, к данным которых не будут обращаться в ближайшем будущем. Разумеется, чтобы знать, какие именно страницы нам не понадобятся в будущем, нужно как-то заглянуть в это будущее. По этой причине такую стратегию часто называют *алгоритмом предсказания*. Очевидно, что данная стратегия просто идеальна, но ее невозможно реализовать.

Стратегия минимального использования

На основе имеющихся статистических данных в стратегиях, применяемых при вытеснении содержимого кеш-памяти, выполняется аппроксимация алгоритма предсказания. Одним из самых удачных алгоритмов аппроксимации, особенно для кеш-памяти общего назначения, считается алгоритм замещения элемента с самым старым временем последнего обращения (least recently used, LRU). Для реализации стратегии вытеснения по алгоритму LRU требуется, чтобы с каждой страницей кеш-памяти была связана временн*а*я метка, фиксирующая время последнего обращения к ней, либо чтобы список всех страниц кеш-памяти был упорядочен по времени последнего обращения. Тогда можно легко удалить страницы с самым старым временем последнего обращения или удалить страницы, находящиеся в начале списка (при условии, что страницы с недавним временем обращения находятся в его конце). Эта стратегия работает очень хорошо, поскольку чем дольше к фрагменту данных, находящихся в кеш-памяти, никто не обращается, тем

меньше шансов на то, что к нему кто-то обратится в ближайшем будущем. Стратегия замещения минимально используемого элемента является очень хорошей аппроксимацией алгоритма предсказания, которая применяется в большинстве реальных приложений. Однако и она имеет один недостаток, который проявляется при однократном доступе к множеству файлов на диске. В таком случае помещение всех страниц с самым старым временем последнего обращения в начало списка не является оптимальным. Разумеется, как и прежде, ядро не может знать заранее, сколько раз будут обращаться к каждому конкретному файлу. Однако ядру известно, сколько было обращений к файлу за прошедший интервал времени.

Стратегия с двумя списками

Из-за приведенных в предыдущем разделе причин в ОС Linux используется модифицированная версия алгоритма LRU, которая называется *стратегией с двумя списками* (two-list strategy). Вместо поддержки одного списка элементов, упорядоченных по времени последнего обращения (так называемый список LRU), в Linux используются два списка: *активный* и *пассивный*. Страницы, находящиеся в активном списке, рассматриваются как "необходимые", и по этой причине их нельзя вытеснять. Страницы из пассивного списка могут быть вытеснены из кеша. Страницы помещаются в активный список, только если при обращении к ним *они находятся в пассивном списке*. При обслуживании обоих списков используется псевдо-LRU подход: элементы добавляются в конец списка и удаляются из его начала так же, как и в очередях. Размеры этих списков поддерживаются примерно одинаковыми. Если активный список становится слишком длинным по сравнению с неактивным списком, часть его элементов переносится в неактивный список и становится доступной для вытеснения. Стратегия с использованием двух списков позволяет избавиться от недостатка классического алгоритма LRU, связанного с однократным доступом к большому числу элементов. Как и классический алгоритм, она позволяет применить простой псевдо-LRU подход для управления элементами списков. Описанный выше подход с двумя списками часто называют *LRU/2*. Его можно обобщить для случая с *n*-списками, и тогда он будет называться *LRU/n*.

Теперь, когда нам известно, как данные попадают в страничный кеш (при выполнении операций чтения и записи), как они синхронизируются с данными на диске (через механизм отложенной записи) и как старые данные замещаются в кеше новыми (с использованием стратегии с двумя списками), рассмотрим на примере реального сценария все те преимущества, которые дает нам страничный кеш. Предположим, вы задействованы в большом программном проекте наподобие ядра Linux, состоящего из огромного числа исходных файлов. Ваша задача — ознакомиться с содержимым и структурой этих файлов. По мере открытия и чтения содержимого этих файлов их данные сохраняются в страничном кеше. При этом переход от файла к файлу будет осуществляться практически мгновенно, поскольку их данные находятся в кеше. В случае, если вам потребуется изменить содержимое файлов, процесс сохранения также не займет много времени, поскольку данные при этом записываются только в память, а не на диск. При компиляции проекта страничный кеш позволит резко уменьшить количество обращений к диску и выполнить его намного быстрее. Если же все дерево исходного кода проекта очень велико и не помещается целиком в кеш-память, то часть его данных должна быть вытеснена другими данными. Благодаря использованию стратегии с двумя списками вытесняются только данные файлов, находящихся в пассивном списке. Поэтому велика вероятность того, что эти данные не будут относиться к файлам, содержащим исходный код, который

вы непосредственно редактируете. Позднее, вероятнее всего, по окончании процесса компиляции, в ядре будет запущен процесс отложенной записи, который обновит данные на диске и синхронизирует внесенные вами изменения в исходные файлы с их копиями на диске. В результате применения кеширования производительность системы существенно возрастает. Чтобы оценить разницу, попробуйте сравнить время полной компиляции проекта в случае пустой кеш-памяти (например, сразу после загрузки системы), и когда нужные данные находятся в кеше.

Реализация страничного кеша в ОС Linux

Как следует из названия, страничный кеш предназначен для кеширования страниц данных в оперативной памяти компьютера. Эти данные загружаются в память из обычных файлов файловой системы при выполнении операций чтения или записи, из специальных файлов блочных устройств и из файлов, отображаемых в память. Таким образом, в страничном кеше содержатся блоки данных из часто используемых файлов. Перед выполнением операции страничного ввода-вывода, как, например, `read()`[2], ядро проверяет, есть ли те данные, которые нужно считать, в страничном кеше. Если данные находятся в кеше, то ядро может быстро возвратить требуемую страницу, скопировав ее из памяти, а не считывая с относительно медленного диска. В оставшейся части этой главы будут рассмотрены структуры данных и средства ядра, задействованные для реализации страничного кеша в ОС Linux.

Объект `address_space`

В одной странице кеш-памяти могут находиться данные из нескольких несмежных физических дисковых блоков[3]. Из-за этого проверка наличия определенных данных в страничном кеше может быть затруднена. Помимо того, невозможно проиндексировать данные, находящиеся в страничном кеше, используя только имя устройства и номер блока, что было бы наиболее простым решением.

Более того, в страничном кеше ОС Linux могут храниться самые разные данные, полученные из самых разных источников. Первоначально страничный кеш был предложен в операционной системе System V (SVR4) только для кеширования данных, полученных из файловых систем. Следовательно, для управления страничным кешем в операционной системе SVR4 использовался эквивалент объекта файлового индекса (`inode`), который назывался `vnode`. В операционной системе Linux страничный кеш разрабатывался с целью кеширования *любых* объектов, размещающихся в страницах памяти, к которым могут относиться файлы разных типов и отображений в памяти.

Хотя для поддержки операций ввода-вывода в страничном кеше ОС Linux можно было бы просто расширить структуру `inode` (она описана в главе 13, “Виртуальная файловая система”), такое решение привело бы к ограничению использования страничного

[2] Как было показано в главе 13 ,“Виртуальная файловая система”, операции страничного ввода-вывода непосредственно выполняются не в системных функциях `read()` и `write()`, а в специфичных для файловых системах методах `file->f_op->read()` и `file->f_op->write()`.

3 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.

кеша только с файлами. Поэтому для создания страничного кеша общего назначения (т.е. такого, который не был бы привязан к файлам или структуре inode) потребовалось создать новый объект, предназначенный для управления элементами кеша и страничными операциями ввода-вывода. Этот объект представляется в виде структуры address_space. Его можно считать физическим аналогом области виртуальной памяти, представляющейся с помощью структуры vm_area_struct, описанной в главе 15, "Адресное пространство процесса". Несмотря на то что с одним файлом может быть связано десять структур vm_area_struct (если, например, в пяти процессах по два раза использовалась функция mmap()), с этим файлом связывается только одна структура address_space. Дело в том, что файл может отображаться в несколько адресных пространств, в физической памяти располагается только одна его копия. Как и многое другое в ядре Linux, название структуры address_space полностью сбивает нас с толку. Возможно, лучшим названием для нее было бы page_cache_entity или physical_pages_of_a_file.

Структура address_space определена в файле <linux/fs.h> и приведена ниже вместе с комментариями к каждому полю.

```
struct address_space {
   struct inode    *host;           /* Файловый индекс, которому
                                       принадлежит объект */
   struct radix_tree_root page_tree; /* Базисное дерево всех страниц */
   spinlock_t      tree_lock;       /* Блокировка для защиты дерева страниц */
   unsigned int    i_mmap_writable; /* Количество областей памяти
                                       с флагом VM_SHARED */
   struct prio_tree_root i_mmap;     /* Список всех отображений */
   struct list_head i_mmap_nonlinear;/* Список областей памяти
                                       с флагом VM_NONLINEAR */
   spinlock_t      i_mmap_lock;     /* Блокировка поля i_mmap */
   atomic_t        truncate_count;  /* Счетчик запросов truncate */
   unsigned long   nrpages;         /* Общее количество страниц */
   pgoff_t         writeback_index; /* Смещение начала отложенной 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 для
                                                 упреждающего чтения */
   spinlock_t      private_lock;    /* Блокировка для частных отображений */
   struct list_head private_list;    /* Список частных отображений */
   struct address_space *assoc_mapping; /* Связанные буферы */
};
```

В поле i_mmap находится указатель на приоритетное дерево поиска для всех совместно используемых и частных отображений. Приоритетное дерево поиска — это хитрая смесь базисных и частично упорядоченных бинарных деревьев[4]. Выше мы уже упоминали о том, что хотя с файлом, находящимся в кеше, связывается одна структура address_space, с ним может быть связано несколько структур vm_area_struct. Так создается отображение типа "один ко многим", связывающим одну физическую страницу памяти с множеством страниц виртуальной памяти. Благодаря полю i_mmap ядро может очень быстро найти все отображения, связанные с конкретным файлом, находящемся в кеше.

Всего в адресном пространстве существует nrpages страниц памяти.

[4] Реализация в ядре основана на базисном приоритетном дереве поиска, предложенном в работе Эдварда М. Мак-Крейга (Edward M. McCreight), опубликованной в *SIAM Journal of Computing*, май 1985, vol. 14. № 2, с. 257–276.

Структура address_space связывается с некоторым другим объектом ядра, обычно с файловым индексом. Если это так, то в поле host находится указатель на соответствующую структуру inode. 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).

Операции объекта address_space

Указатель на таблицу операций объекта address_space находится в поле a_ops по аналогии с таблицей операций виртуальной файловой системы VFS. Эта таблица представляется с помощью структуры address_space_operations, которая также определяется в файле <linux/fs.h> и приведена ниже.

```
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 *,
                      struct writeback_control *);
   int (*set_page_dirty) (struct page *);
   int (*readpages) (struct file *, struct address_space *,
                     struct list_head *, unsigned);
   int (*write_begin)(struct file *, struct address_space *mapping,
                      loff_t pos, unsigned len, unsigned flags,
                      struct page **pagep, void **fsdata);
   int (*write_end)(struct file *, struct address_space *mapping,
                    loff_t pos, unsigned len, unsigned copied,
                    struct page *page, void *fsdata);
   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);
   int (*get_xip_mem) (struct address_space *, pgoff_t, int,
                       void **, unsigned long *);
   int (*migratepage) (struct address_space *,
                       struct page *, struct page *);
   int (*launder_page) (struct page *);
   int (*is_partially_uptodate) (struct page *,
                                 read_descriptor_t *,
                                 unsigned long);
   int (*error_remove_page) (struct address_space *,
                             struct page *);
};
```

В элементах таблицы операций находятся указатели на функции, с помощью которых реализуется система ввода-вывода страниц конкретного объекта, находящегося в кеше. У каждого объекта внешней памяти создается своя таблица операций, с помощью которой выполняется его взаимодействие со страничным кешем. Например, таблица операций файловой системы ext3 определена в файле fs/ext3/inode.c. В нем определены также функции, управляющие работой страничного кеша. Среди них есть часто используемые функции, предназначенные для загрузки страниц в кеш и обновления данных, находящихся в нем. Поэтому самыми важными являются методы readpage() и writepage(). Рассмотрим последовательность действий, выполняемую в каждом из них, начиная с операции чтения станиц. Сначала в ядре Linux выполняется попытка поиска нужных данных в страничном кеше. Для этого вызывается функция find_get_page(),

которой передаются объект address_space и смещение в страницах. По этим параметрам выполняется поиск нужных данных в кеш-памяти.

```
page = find_get_page(mapping, index);
```

Здесь в параметре mapping передается указатель на объект address_space, а в параметре index — нужное смещение в файле, выраженное в страницах. (Разумеется, имя параметра mapping, содержащее указатель на структуру address_space, еще больше запутывает дело. Чтобы не нарушать логику изложения, я просто привел имена так, как они используются в ядре, однако это совсем не означает, что я со всем согласен.) Если требуемой страницы в кеше нет, то функция find_get_page() возвращает значение NULL, после чего выделяется память под новую страницу, и она добавляется в страничный кеш, как показано ниже.

```
struct page *page;
int error;

/* Выделяем память под страницу ... */
page = page_cache_alloc_cold(mapping);
if (!page)
   /* Ошибка при выделении памяти */

/* ... теперь добавим страницу в кеш */
error = add_to_page_cache_lru(page, mapping, index, GFP_KERNEL);
if (error)
   /* Ошибка при добавлении страницы в кеш */
```

После выполнения всего этого нужные данные считываются с диска, помещаются в страничный кеш и возвращаются пользователю.

```
error = mapping->a_ops->readpage(file, page);
```

Операции записи выполняются немного иначе. Для отображаемых в память файлов при изменении страницы памяти система управления виртуальной памятью просто вызывает приведенную ниже функцию.

```
SetPageDirty(page);
```

Запись этой страницы памяти выполняется ядром чуть позже с помощью вызова метода writepage(). Операции записи для файлов, открытых обычным образом (без отображения в память), выполняются чуть сложнее. Универсальная последовательность действий, выполняемая при записи, реализована в файле mm/filemap.c и описана ниже.

```
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);
```

Сначала выполняется поиск требуемой страницы памяти в страничном кеше. Если ее там нет, выделяется память под новую страницу и последняя добавляется в кеш. После этого ядро формирует запрос на запись и данные копируются из пространства пользователя в буфер, расположенный в ядре. В конечном итоге данные записываются на диск.

Поскольку все описанные выше действия выполняются при всех операциях страничного ввода-вывода, то все операции страничного ввода-вывода выполняются исключительно через страничный кеш. Таким образом, при поступлении всех запросов на чтение ядро сначала пытается искать нужные данные в страничном кеше. Если их там нет, требуемая страница загружается с диска и помещается в кеш. При выполнении операций за-

писи страничный кеш выполняет роль приемника данных. Поэтому все записываемые страницы с данными также попадают в кеш.

Базисное дерево

Поскольку перед началом любой операции страничного ввода-вывода ядро должно проверять наличие страниц в кеш-памяти, то эта проверка должна выполняться быстро. В противном случае время, затраченное на поиск в кеше, может свести на нет все выгоды от использования кеширования. По крайней мере, в случае незначительного количества удачных обращений в кеш 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 кеше выполняется на основании информации, полученной из объекта `address_space`, и значения смещения. Для каждого объекта `address_space` создается свое уникальное *базисное дерево* (radix tree), указатель на которое хранится в поле `page_tree`. Базисное дерево относится к одному из типов двоичного дерева. Оно позволяет выполнять очень быстрый поиск необходимой страницы только на основании значения смещения в файле. Функции поиска в страничном кеше, такие как `find_get_page()` и `radix_tree_lookup()`, выполняют поиск заданного объекта с использованием указанного дерева.

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

До появления ядер серии 2.6 поиск в страничном кеше не выполнялся с помощью базисных деревьев. Вместо этого поддерживалась глобальная хеш-таблица для всех страниц памяти в системе. Специальная хеш-функция возвращала двухсвязный список значений, связанных с одним значением ключа. Если нужная страница находится в кеше, то ей будет соответствовать один из элементов этого списка. Если страница в кеше 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 при средних нагрузках. В результате страдала производительность системы.
- Размер хеш-таблицы был очень большим, потому что в ней содержалась информация обо всех страницах памяти в страничном кеше, в то время как важными являются лишь страницы, связанные с одним конкретным файлом.
- Производительность системы в случае неудачного обращения в кеш (когда искомая страница памяти не находится в кеше) 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 кеша на основании базисных деревьев позволило решить эти проблемы.

Буферный кеш

Отдельные дисковые блоки также связаны со страничным кешем через буферы блоков ввода-вывода. В главе 14, "Уровень блочного ввода-вывода", мы уже говорили о том, что буфер является представлением в памяти компьютера одного физического дискового блока. По сути, буфера выполняют роль дескрипторов, связывающих дисковые блоки и страницы отображаемой памяти. В результате страничный кеш позволяет также уменьшить количество обращений к диску при выполнении блочных операций ввода-вывода за счет кеширования как дисковых блоков, так и буферизации блоков при выполнении операций ввода-вывода. Такой вид кеша часто называют *буферным кешем* (buffer cache), несмотря на то, что он реализован не в виде отдельной кеш-памяти, а является частью страничного кеша.

В блочных операциях ввода-вывода обрабатывается один дисковый блок за раз. Чаще всего выполняются операции чтения и записи индексов. Для выполнения низкоуровневой операции чтения одного блока с диска в ядре предусмотрена функция `bread()`. Дисковые блоки отображаются в память через буферы, которые кешируются в страничном кеше.

Буферная кеш-память и страничная не всегда были объединены. Это произошло только в версии 2.4 ядра Linux и было одним из основных изменений. В ранних версиях ядра Linux существовали два отдельных дисковых кеша: страничный и буферный. В первом кешировались страницы памяти, а во втором — дисковые буфера. Эти две кеш-памяти не были унифицированы. Так, один и тот же дисковый блок мог находиться в двух кешах одновременно. Это существенно затрудняло синхронизацию данных в двух кешах и приводило к напрасному расходу памяти, которая была занята одними и теми же данными. В современной версии ядра Linux существует только одна дисковая кеш-память — страничный кеш. Однако для представления дисковых блоков в памяти ядру по-прежнему нужны буфера. Просто буфера описывают отображение дисковых блоков на страницы памяти, которые находятся в страничном кеше.

Потоки синхронизатора

Выше уже говорилось о том, что при выполнении операций записи данные из приложения попадают в страничный кеш, а сама команда записи данных на диск откладывается "на потом". Когда данные в страничном кеше становятся новее, чем во внешней памяти, страницы с ними помечаются как *измененные* (dirty). Такие страницы в конечном итоге должны быть сохранены на диске. Процесс отложенной записи измененных страниц запускается в системе в трех случаях, которые перечислены ниже.

- Если объем свободной памяти в системе становится ниже определенного предела, ядро записывает все измененные страницы на диск, чтобы потом освободить часть памяти, занимаемой станичным кешем. Дело в том, что только неизмененные страницы могут безболезненно вытесняться из кеша. После вытеснения страниц можно уменьшить размер страничного кеша и освободить память.
- Если истек установленный срок давности находящихся в страничном кеше измененных данных, то они записываются на диск. В результате предотвращается ситуация, когда измененные данные могут находиться в кеше бесконечно долгое время.

- Если в пользовательском процессе вызываются системные функции `sync()` и `fsync()`, ядро по этому требованию выполняет отложенную запись измененных данных на диск.

Задачи в перечисленных выше трех случаях преследуют совершенно разные цели. По сути, в более старых версиях ядра они обрабатывались двумя разными потоками пространства ядра (см. следующий раздел). Однако в ядрах серии 2.6 все эти три случая обрабатываются группой (gang[5]) потоков ядра, которые называются *потоками синхронизатора* (flusher threads).

Прежде всего потоки синхронизатора должны сбросить измененные данные на диск, если количество свободной памяти в системе уменьшается ниже определенного предела. Целью такой фоновой записи является освобождение памяти, занимаемой измененными страницами, в случае недостатка физических страниц памяти в системе. Минимальное значение свободной памяти, при достижении которого запускается процесс отложенной записи, определяется `sysctl`-параметром `dirty_background_ratio`. Как только объем свободной памяти становится меньше этого порога, из ядра вызывается функция `wakeup_flusher_threads()`, которая переводит в состояние выполнения один или несколько потоков синхронизатора. В них запускается функция `bdi_writeback_all()`, которая начинает отложенную запись измененных страниц. В качестве параметра ей передается число страниц, которые должны быть записаны на диск. Функция продолжает запись страниц на диск 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`.

При выполнении этих условий гарантируется, что потоки синхронизатора выполнили свою часть работы по предотвращению проблемы нехватки памяти в системе. Завершение процесса отложенной записи может произойти и *до* выполнения этих условий, если будут сброшены на диск все страницы, и для потоков синхронизатора не останется больше никакой работы.

Для выполнения второй задачи поток синхронизатора периодически переводится в состояние выполнения (без привязки к условию нехватки памяти) для записи измененных страниц, долго находящихся в кеше, на диск. Это делается для того, чтобы измененные страницы не находились в кеше бесконечно долго. Поскольку оперативная память является энергозависимой и ее содержимое не сохраняется при перезагрузке компьютера, то в случае непредвиденного сбоя системы содержимое измененных страниц, находящихся в памяти и еще не записанных на диск, будет потеряно. Следовательно, очень важно выполнять периодическую синхронизацию страничного кеша с данными, находящимися на диске. При загрузке системы инициализируется таймер, по сигналу которого и будет периодически запускаться на выполнение поток синхронизатора. В этом потоке вызывается функция `wb_writeback()`, выполняющая отложенную запись на диск всех данных, после модификации которых прошло больше чем `dirty_expire_interval` миллисекунд. Далее выполняется повторная инициализация таймера так, чтобы он сработал через `dirty_writeback_interval` миллисекунд. Таким образом, периодиче-

[5] Слово “gang” не является жаргонным. Этот термин часто используется в информатике для обозначения группы чего-либо, что может выполняться параллельно.

ский запуск потоков синхронизатора позволяет сохранить на диске все измененные страницы, находящиеся в кеше дольше указанного интервала времени.

Системный администратор может установить указанные значения либо с помощью каталога /proc/sys/vm, либо утилиты sysctl. В табл. 16.1 приведен список всех соответствующих переменных.

Таблица 16.1. Параметры настройки процесса отложенной записи

Переменная	Описание
dirty_background_ratio	Количество свободных страниц, выраженное в процентах от общего объема памяти, при достижении которого в потоках синхронизатора запускается процесс отложенной записи измененных данных на диск
dirty_expire_interval	Время, выраженное в сотых долях секунды, определяющее, как долго измененные данные могут оставаться в памяти до того, как они будут записаны на диск при следующем вызове потока синхронизатора
dirty_ratio	Количество измененных страниц одного процесса, выраженное в процентах от общего объема оперативной памяти, при достижении которого запускается процесс отложенной записи измененных данных на диск
dirty_writeback_interval	Время, выраженное в сотых долях секунды, определяющее, как часто должны запускаться потоки синхронизатора для отложенной записи измененных данных на диск
laptop_mode	Переменная булева типа, которая включает или выключает *режим ноутбука* (см. следующий раздел)

Код реализации потоков синхронизатора находится в файлах mm/page-writeback.c и mm/backing-dev.c, а механизма отложенной записи — в файле fs/fs-writeback.c.

Режим ноутбука

Режим ноутбука (laptop mode) — это специальная стратегия отложенной записи, предназначенная для увеличения времени работы аккумуляторной батареи за счет снижения активности работы жесткого диска и удлинения, на сколько это возможно, времени его отключения. Данный режим включается через файл /proc/sys/vm/laptop_mode. По умолчанию в этом файле содержится значение нуль и режим ноутбука отключен. Для включения режима ноутбука в этот файл нужно записать единицу.

В режиме ноутбука в стратегию отложенной записи вносится всего одно изменение. Кроме сохранения измененных страниц, долгое время находившихся в кеш-памяти, в потоках синхронизатора отслеживается любая другая дисковая активность, и на диск сбрасываются *все* измененные буфера. Таким образом, учитывается тот факт, что к началу процесса отложенной записи пластины диска находятся в раскрученном состоянии и после этого диск на некоторое время будет отключен.

Такое изменение алгоритма работы имеет смысл только в том случае, когда значения переменных dirty_expire_interval и dirty_writeback_interval установлены достаточно большими, например 10 минут. При столь длительной задержке по записи пластины диска будут раскручиваться нечасто, однако, после того как диск вклю-

чился в работу, операционная система воспользуется этой ситуацией по полной программе. Поскольку отключение электродвигателя привода пластин диска существенно снижает потребляемую компьютером мощность, режим ноутбука позволяет заметно увеличить время работы аккумуляторной батареи до очередной подзарядки. Обратной стороной является риск потери данных, если операционная система "зависнет" или произойдет какой-либо сбой в ее работе.

В большинстве дистрибутивов ОС Linux режим ноутбука включается и отключается автоматически, а также автоматически изменяются другие параметры настройки отложенной записи при переходе на автономное питание и при подключении к электросети. В результате при питании от аккумулятора система переходит в режим ноутбука и за счет нечастого обращения к диску экономит энергию, а при подключении компьютера к электрической сети нормальное функционирование процесса отложенной записи восстанавливается.

Экскурс в историю: `bdflush`, `kupdated` и `pdflush`

До появления ядер серии 2.6 работа потоков синхронизатора выполнялась двумя другими потоками ядра: `bdflush` и `kupdated`.

Поток ядра `bdflush` выполнял фоновую отложенную запись измененных страниц памяти в случае небольшого количества доступной памяти. Так же как и для потоков синхронизатора, для него определялся ряд пороговых параметров. Сам поток `bdflush` переводился в состояние выполнения с помощью функции `wakeup_bdflush()` в случае, когда количество свободной памяти в системе становилось ниже установленного порога.

Между современными потоками синхронизатора и потоком ядра `bdflush` существуют два основных отличия. Первое из них (подробнее оно будет описано в следующем разделе) заключается в том, что в системе всегда существовал только один фоновый процесс (демон) `bdflush`, тогда как количество потоков синхронизатора может изменяться динамически в зависимости от числа физических дисков. Второе отличие состоит в том, что демон `bdflush` работал с буферами ввода-вывода; он сохранял на диске данные измененных буферов. Потоки синхронизатора работают со страницами; они сохраняют на диске целые страницы. Разумеется, страницы могут соответствовать буферам ввода-вывода, однако единицей ввода-вывода является именно страница, а не один буфер. Это дает преимущество, поскольку работать со страницами памяти проще, чем с буферами, так как страница памяти — более универсальный и часто используемый объект.

Поскольку демон `bdflush` выполнял отложенную запись буферов только тогда, когда количество свободной памяти становилось меньше заданного предела либо когда количество буферов становилось очень большим, то был введен поток ядра `kupdated`, который периодически выполнял отложенную запись измененных страниц памяти. Его действия были аналогичны функции `wb_writeback()`.

В ядре серии 2.6 потоки `bdflush` и `kupdated` были заменены *потоками* `pdflush`. Это название происходит от сокращения слов *page dirty flush* (сброс измененных страниц) и является еще одним примером неудачного выбора имен в ядре Linux. Потоки `pdflush` выполняли действия, аналогичные тем, которые выполняют в современных версиях ядра потоки синхронизатора. Основное отличие между ними заключалось в том, что число потоков `pdflush` могло динамически изменяться (по умолчанию от 2 до 8) в зависимости загрузки системы ввода-вывода. Потоки `pdflush` не были привязаны к какому-либо конкретному диску; они были общими для всех физических дисков системы.

Такой подход упрощал реализацию. Однако такая простота оборачивалась тем, что работу потока pdflush мог легко застопорить любой перегруженный диск, а для современного компьютерного оборудования вызвать перегрузку диска не составляет особого труда. Привязка потоков отложенной записи к физическим дискам позволила синхронно выполнять операции ввода-вывода, упростить логику управления перегрузкой и увеличить производительность системы. Поэтому в ядрах версии начиная с 2.6.32 потоки pdflush были заменены потоками ядра. Основное отличие между ними заключается в том, что последние привязаны к физическим дискам. В остальном материал оставшейся части этой главы применим и к потокам pdflush, а также к любой версии ядра серии 2.6.

Предотвращение перегрузки с помощью нескольких потоков

Один из главных недостатков решения на основе bdflush состоял в том, что демон bdflush имел всего один поток выполнения. При большом количестве операций отложенной записи на перегруженном устройстве это иногда приводило к медленной работе демона bdflush. Дело в том, что его единственный поток оказывался все время заблокированным из-за единственной и длинной очереди запросов к устройству ввода-вывода. Напомним: очередь запросов — это список запросов на ввод-вывод, которые должны быть выполнены устройством. При этом очереди запросов к другим устройствам ввода-вывода были сравнительно свободными. Если в системе присутствует несколько дисков и имеется в избытке соответствующая процессорная мощность, то ядро должно по возможности загрузить работой все диски. К сожалению, даже при большом количестве данных, для которых требовалось выполнить отложенную запись, демон bdflush мог заниматься обработкой только одной очереди. При этом другие диски могли простаивать. Это происходило потому, что скорость работы дисков конечна и, к сожалению, очень низкая. Если процесс отложенной записи страниц выполнялся только в одном потоке, то этот поток мог довольно долго находиться в состоянии ожидания завершения операций ввода-вывода на одном диске. Для решения этой проблемы пришлось реализовать в ядре многопоточный механизм отложенной записи измененных страниц. В таком случае ни одна очередь запросов к устройству не может стать узким местом.

В ядрах серии 2.6 эта проблема решается путем введения нескольких потоков синхронизатора. В каждом потоке независимо от других потоков выполняется отложенная запись страниц памяти на диск. Это позволяет различным потокам синхронизатора работать с разными очередями запросов устройств. В случае с pdflush число потоков изменялось динамически так, чтобы для каждого потока находилась работа. При этом очереди измененных страниц были привязаны к суперблокам файловых систем. Использование нескольких потоков pdflush позволяло устранить проблему, когда один загруженный диск мог повлиять на процессы отложенной записи, выполняемые на других дисках. Все это замечательно, но что произойдет, если все запущенные потоки pdflush перейдут в состояние ожидания обработки своих запросов на запись, которые попали в одну и ту же перегруженную очередь? В таком случае производительность нескольких потоков pdflush будет ничем не лучше производительности одного потока. Правда, при этом количество израсходованной памяти будет существенно выше. Для решения этой проблемы в потоках pdflush принято правило избегать перегрузок — процесс активной отложенной записи выполняется только в случае, если очереди к устройствам не перегружены. В результате вся работа потоков pdflush равномерно распределяется и они не загружают одно и то же и так перегруженное устройство.

Подобный подход работал достаточно хорошо, однако принцип избежания перегруженных устройств был далеко не идеален. В современных компьютерах перегрузка диска вызывается достаточно легко. Все дело в том, что технологии развития каналов ввода-вывода данных отстают от всего остального компьютерного оборудования. Несмотря на то что увеличение скорости работы процессоров соответствует закону Мура[6], скорость работы жестких дисков стала ненамного быстрее той, что была двадцать лет тому назад. Более того, кроме потоков `pdflush`, ни в какой другой части системы ввода-вывода не предпринимались попытки избежания перегрузок. В результате в определенных случаях потоки `pdflush` начинали процесс отложенной записи на некоторых дисках с существенным опозданием. Поэтому начиная с ядра версии 2.6.32 они были заменены современными потоками синхронизатора, привязанными к блочному устройству. В результате в каждом потоке обрабатывался свой список измененных страниц, связанный с конкретным диском, и данные записывались на этот диск. Таким образом, процесс отложенной записи стал выполняться синхронно на всех дисках. Поскольку для каждого диска существует свой поток отложенной записи, запутанные правила избежания перегрузок стали не нужны. При таком подходе улучшилась равномерность выполнения процессов отложенной записи и удалось избежать ненужных задержек.

Из-за внесения улучшений в механизм отложенной записи, которые начались после введения потоков `pdflush`, а затем трансформировались в потоки синхронизатора, ядро серии 2.6 может поддерживать одновременную работу на большем количестве дисков, чем все ранее выпущенные ядра. В случае возникновения перегрузки потоки синхронизатора могут обеспечить бо́льшую производительность при использовании нескольких дисков.

Резюме

В этой главе был рассмотрен страничный кеш системы Linux и его механизм отложенной записи. Было показано, что все операции страничного ввода-вывода в ядре сначала проходят через страничных кеш. Мы увидели, как этот кеш (за счет сохранения данных в оперативной памяти компьютера) позволяет существенно увеличить общую производительность системы и уменьшить число операций ввода-вывода, выполняемых с дисками. Мы обсудили также процесс записи данных программами в страничный кеш и их последующую запись на диск с помощью механизма отложенной записи. Было показано, что при выполнении функций записи в приложении данные не записываются на диск сразу, а сначала попадают в страничный кеш, и содержащие их страницы помечаются как измененные. Далее через определенные интервалы времени активизируется группа потоков синхронизатора, которые и выполнят отложенную запись на диск.

На основании материала последних нескольких глав вы получили устойчивое представление о том, как выполняется управление памятью и файловыми системами. Теперь мы плавно перейдем к теме драйверов устройств и модулей и посмотрим, как в ядре Linux обеспечивается модульная и динамическая инфраструктура, позволяющая загрузить и выгрузить код ядра прямо во время работы системы.

[6] В 1965 году Гордон Мур заметил, что степень интеграции микросхем (число транзисторов на кристалле) удваивалась каждые 24 месяца.

17
Устройства и модули

В данной главе будут рассмотрены четыре компонента ядра, имеющих отношение к драйверам устройств и управлению самими устройствами.

- **Типы устройств.** Классификация, которая используется во всех системах Unix с целью унификации работы устройств общего назначения.
- **Модули.** Механизм, с помощью которого ядро Linux может загружать и выгружать объектный код по мере необходимости.
- **Объекты ядра.** Позволяют добавить к структурам данных ядра поддержку простых форм объектно-ориентированного взаимодействия и отношений типа "родитель–потомок".
- **Sysfs.** Файловая система, с помощью которой представляется дерево системных устройств.

Типы устройств

В Linux, как и во всех остальных системах Unix, все устройства относятся к одному из трех типов:

- блочные устройства;
- символьные устройства;
- сетевые устройства.

Блочные устройства, которые часто обозначают как *blkdevs*, позволяют адресовать фрагменты данных, находящиеся на устройстве и называемые *блоками*. Как правило, такие устройства поддерживают операцию *позиционирования*, благодаря которой осуществляется произвольный доступ к данным. В качестве примеров блочных устройств можно привести жесткие диски, накопители на DVD- и Blu-ray-дисках, а также устройства внешней памяти, такие как флешки. Для доступа к блочным устройствам используется специальный файл, называемый *узлом блочного устройства (block device node)*, который, как правило, смонтирован в виде отдельной файловой системы. Файловые системы были рассмотрены в главе 13, "Виртуальная файловая система", а блочные устройства — в главе 14, "Уровень блочного ввода-вывода".

Символьные устройства, которые часто обозначают как *cdevs*, обычно не позволяют адресовать отдельные блоки данных. Они предоставляют доступ к данным только в виде непрерывного потока символов (байтов). В качестве примеров символьных устройств можно привести клавиатуру, мышь, принтеры, а также большинство псевдоустройств. Для доступа к символьным устройствам используется специальный файл, называемый *узлом символьного устройства (character device node)*. В отличие от блочного устройства приложение взаимодействует с символьным устройством напрямую через узел этого устройства.

Сетевые устройства часто называют *устройствами Ethernet*, поскольку технология Ethernet является самой популярной на сегодняшний день сетевой технологией. Они предоставляют доступ к сети (например, такой, как Интернет) с помощью физического адаптера (такого, как, например, плата беспроводного сетевого интерфейса 802.11n ноутбука) и набора специфических протоколов (например, протокола IP). Несмотря на используемый в системах Unix подход к проектированию, заключающийся в том, что "все в системе является файлом", сетевые устройства ему не подчиняются. Обращение к ним выполняется не через соответствующий узел файловой системы, а через специальный интерфейс, называемый *API-сокетом (socket API)*.

В системе Linux используются также и другие типы устройств, но они не являются универсальными и относятся только к одной специализированной задаче. Исключением здесь являются *смешанные устройства (miscellaneous devices)*, которые часто обозначают как *miscdevs*. По сути, они являются упрощенной формой символьных устройств. Смешанные устройства позволяют авторам драйверов с легкостью работать с простыми устройствами и сосредоточить всю функциональность на общей инфраструктуре.

Однако не все драйверы устройств в системе Linux работают с физическими устройствами. Часть драйверов устройств являются *виртуальными* и обеспечивают доступ к функциональным возможностям ядра. Такие виртуальные устройства называют *псевдоустройствами*. Самые известные из них — это *генератор случайных чисел ядра*, обращение к которому выполняется через файлы `/dev/random` и `/dev/urandom`; *фиктивное устройство*, доступное через `/dev/null`; *нулевое устройство*, доступное через `/dev/zero`; *полное устройство*, доступное через `/dev/full`, и *устройство памяти*, доступное через `/dev/mem`. Тем не менее большинство драйверов устройств взаимодействует с реальными физическими устройствами.

Модули

Несмотря на то что ядро является "монолитным", в том смысле, что оно выполняется в общем защищенном адресном пространстве, ядро Linux также является модульным, что позволяет выполнять динамическую загрузку 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 создать минимальное базовое ядро операционной системы, а все дополнительные возможности и драйверы скомпилировать в качестве загружаемых модулей, т.е. самостоятельных объектов. Модули также позволяют удалять и повторно загружать код ядра, что облегчает отладку, 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 рассмотрены, разработка модулей подобна созданию новой прикладной программы, по крайней мере, в том, что модули имеют точки входа и 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, вашему вниманию представляется 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");
MODULE_DESCRIPTION("A Hello, World Module");
```

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 для текущего модуля. Все функции инициализации должны соответствовать прототипу

```
int my_init(void);
```

Поскольку функция инициализации редко вызывается за пределами модуля, ее обычно не нужно экспортировать за пределы видимости файла и можно объявить с ключевым словом `static`.

Функции инициализации возвращают значение типа `int`. Если инициализация (или то, что делает функция инициализации) прошла успешно, то функция должна возвратить

нулевое значение. В случае ошибки функция должна прекратить выполняемые действия и возвратить ненулевое значение.

В рассмотренном нами случае эта функция просто выводит сообщение и возвращает нулевое значение. В реальных модулях функция инициализации регистрирует ресурсы, инициализирует работу оборудования, выделяет структуры данных и т.д. Если рассматриваемый файл будет статически скомпилирован в образ ядра, то функция инициализации помещается в этот образ и будет вызвана при загрузке ядра.

Точка выхода из модуля регистрируется с помощью макроса `module_exit()`. В данном примере мы зарегистрировали в качестве точки выхода функцию `hello_exit()`. Ядро вызывает эту функцию перед удалением модуля из памяти. В функции завершения нужно освободить все используемые ресурсы, остановить работу и выполнить аппаратный сброс оборудования, а также выполнить другие необходимые при завершении работы модуля действия. Обычно в функции завершения выполняются действия, обратные тем, что были сделаны в функции инициализации и в процессе работы модуля, особенно это относится к освобождению всех ресурсов, оставшихся после работы модуля. После возврата из функции завершения модуль выгружается из памяти.

Функция завершения должна соответствовать прототипу

```
void my_exit(void);
```

Так же как и в случае функции инициализации, ее можно объявить как `static`.

Если этот файл будет скомпилирован в статический образ 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 без лицензии GPL приведет к установке в ядре флага `tainted` (буквально "запорченный"). Лицензия на право копирования модуля служит двум целям. Во-первых, для информационных целей. Большинство разработчиков ядра считают отчеты об ошибках, в которых установлен флаг `tainted`, не заслуживающими доверия, поскольку это предполагает, что в ядро был загружен бинарный модуль (т.е. такой модуль, который невозможно отлаживать). Во-вторых, в модулях без лицензии GPL нельзя использовать символы, которые служат "только для GPL". Об этих символах речь пойдет ниже, в разделе "Экспортируемые символы".

Наконец, макросы `MODULE_AUTHOR()` и `MODULE_DESCRIPTION()` позволяют указать автора модуля и его короткое описание соответственно. Назначение этих макросов чисто информационное.

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 того, можно компилировать исходный код модуля отдельно от исходных кодов ядра.

Использование дерева каталогов исходных кодов ядра

В идеальном случае разрабатываемый вами модуль является частью официального ядра Linux и находится в каталоге исходных кодов ядра. Введение модуля непосредст-

венно в ядро может вначале потребовать больше работы, но обычно такое решение более предпочтительно. Дело в том, что код, включенный в ядро, может сопровождаться и отлаживаться всем сообществом разработчиков (по крайней мере, они могут оказать вам неоценимую помощь!).

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. В этом каталоге драйверы делятся на классы, типы и собственно на отдельные драйверы. Например, символьные устройства находятся в каталоге `drivers/char/`, блочные — в каталоге `drivers/block/`, устройства USB — в каталоге `drivers/usb/`. Эти правила не являются жесткими, поскольку многие устройства принадлежат к разным категориям. Например, большинство USB-устройств являются символьными, но они располагаются в каталоге `drivers/usb/`, а не в `drivers/char/`. Несмотря на подобные сложности, такая организация является понятной и четкой, после того, как вы в ней разберетесь.

Предположим, вы хотите создать драйвер символьного устройства и сохранить его файлы в подкаталоге `drivers/char/`. В этом каталоге находится большое количество исходных файлов на языке C, а также ряд других подкаталогов. Если драйвер имеет всего один или два исходных файла, их можно поместить непосредственно в этот каталог. Если код драйвера состоит из нескольких исходных или других вспомогательных файлов, то лучше всего создать отдельный подкаталог. Здесь не существует каких-то жестких правил. Предположим, вы хотите создать отдельный подкаталог. Для определенности предположим, что вы хотите написать драйвер для удочки с числовым программным управлением, которая имеет интерфейс Fish Master XL 3000 для подключения к компьютеру. Следовательно, необходимо создать подкаталог `fishing` в каталоге `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 построения, что при компиляции модулей необходимо войти в подкаталог `fishing/`. Скорее всего, при компиляции драйвера вам потребуется указать отдельный параметр конфигурации, например `CONFIG_FISHING_POLE` (создание новых параметров рассматривается ниже, в разделе “Поддержка параметров конфигурации”). В этом случае в файл `Makefile` необходимо добавить строку вида

```
obj-$(CONFIG_FISHING_POLE) += fishing/
```

И наконец, в каталоге `drivers/char/fishing` необходимо создать файл `Makefile`, содержащий следующую строку:

```
obj-m += fishing.o
```

Теперь система построения перейдет в каталог `fishing/` и создаст модуль `fishing.ko` из исходного файла `fishing.c`. Здесь вас может сбить с толку указанное расширение объектного файла `.o`, но в результате будет создан модуль с расширением `.ko`. Как уже говорилось, скорее всего, при компиляции модуля драйвера удочки с числовым программным управлением вам нужно будет указать параметр конфигурации. В таком случае в новый файл `Makefile` необходимо добавить следующую строку:

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
```

В один прекрасный день ваш драйвер удочки может стать довольно сложным. Введение функции автоматического определения наличия лески может привести к тому, что модуль станет очень большим и будет занимать больше одного файла исходного кода. Никаких проблем! Просто нужно внести в файл `Makefile` приведенные ниже изменения.

```
obj-$(CONFIG_FISHING_POLE) += fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

Теперь файлы `fishing-main.c` и `fishing-line.c` будут скомпилированы и скомпонованы в файл модуля `fishing.ko` при условии, что будет установлен параметр конфигурации `CONFIG_FISHING_POLE`.

Наконец, вам может понадобиться передать компилятору языка C дополнительные параметры командной строки во время построения исключительно вашего файла модуля. Для этого в файл `Makefile` необходимо добавить строку

```
EXTRA_CFLAGS += -DTITANIUM_POLE
```

Если не хотите создавать отдельный каталог для файлов модуля и собираетесь поместить их в каталог `drivers/char/`, то нужно ввести приведенные выше строки (т.е. те, которые вы должны были указать в файле `drivers/char/fishing/Makefile`) в файл `drivers/char/Makefile`.

Для компиляции модуля запустите процесс сборки ядра, как обычно. Если для построения вашего модуля нужно указать параметр конфигурации (в данном случае `CONFIG_FISHING_POLE`), перед запуском процесса сборки убедитесь, что он определен.

Компиляция вне дерева исходных кодов ядра

Если вы предпочитаете разрабатывать и поддерживать ваш модуль отдельно от дерева исходных кодов ядра и жить жизнью аутсайдера, создайте файл `Makefile` в том каталоге, где находятся файлы с исходным кодом модуля, и поместите в него приведенную ниже строку.

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

В результате файл `fishing.c` будет скомпилирован в файл `fishing.ko`. Если ваш исходный код находится в нескольких файлах, то необходимо добавить две строки.

```
obj-m := fishing.o
fishing-objs := fishing-main.o fishing-line.o
```

В этом примере сначала будут скомпилированы файлы `fishing-main.c` и `fishing-line.c`, а затем скомпонованы в файл модуля `fishing.ko`.

Главное отличие от случая, когда модуль находится в дереве исходного кода ядра, состоит в запуске его процесса сборки. Поскольку модуль находится за пределами дерева исходных кодов ядра, необходимо указать утилите `make` расположение исходных файлов ядра и основного `make`-файла ядра. Это также делается с помощью команды

```
make -C /kernel/source/location SUBDIRS=$PWD modules
```

В данном примере `/kernel/source/location` — это путь к сконфигурированному дереву исходных кодов ядра. Напомним, что *не нужно* хранить копию дерева исходных кодов ядра, с которой вы работаете, в каталоге `/usr/src/linux`, — она должна находиться в другом легкодоступном месте, скажем, где-нибудь в вашем домашнем каталоге.

Установка модулей

Скомпилированные модули должны быть инсталлированы в каталог /lib/modules/*версия*/kernel. Здесь каждый подкаталог каталога kernel/ соответствует расположению файлов модуля в дереве исходных кодов ядра. Например, для версии ядра 2.6.34 скомпилированный модуль драйвера удочки будет находиться в файле /lib/modules/2.6.34/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 с правами 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 дистрибутивов ОС Linux эта информация генерируется автоматически и обновляется при загрузке системы. Для генерации информации о зависимостях между модулями необходимо, обладая правами пользователя root, выполнить приведенную ниже команду.

```
depmod
```

Для быстрого обновления и генерации информации только о более новых модулях, чем сам файл зависимостей, необходимо, обладая правами пользователя root, выполнить другую команду:

```
depmod -A
```

Информация о зависимостях между модулями хранится в файле

/lib/modules/*версия*/modules.dep.

Загрузка модулей

Проще всего загрузить модуль в память, воспользовавшись утилитой insmod. Эта утилита очень простая: она “просит” ядро загрузить в память указанный вами модуль. Утилита insmod не отслеживает зависимости и не выполняет никакой интеллектуальной обработки ошибок. Использовать ее очень просто. Обладая правами пользователя root, нужно ввести команду

```
insmod module.ko
```

где вместо *module.ko* необходимо указать имя файла модуля, который требуется загрузить. Так, для загрузки модуля управления удочкой выполните следующую команду:

```
insmod fishing.ko
```

Удалить модуль можно аналогичным образом с помощью утилиты rmmod. Для этого, обладая правами пользователя root, нужно выполнить приведенную ниже команду, в которой вместо параметра module укажите имя загруженного в память модуля.

```
rmmod module
```

Например, для удаления модуля управления удочкой введите следующую команду:

```
rmmod fishing
```

Несмотря на все сказанное выше, эти утилиты очень просты по своей природе и не обладают каким-либо "интеллектуальным" поведением. Поэтому была создана другая утилита `modprobe`, которая позволяет загрузить модуль с учетом зависимостей, выполняет проверку и обработку ошибочных ситуаций, позволяет передать модулям параметры конфигурации и обладает другими полезными функциями. Для управления загрузкой и выгрузкой модулей мы настоятельно рекомендуем использовать именно ее.

Для загрузки модуля в ядро с помощью утилиты `modprobe`, обладая правами пользователя `root`, запустите команду

```
modprobe module [ параметры модуля ]
```

где вместо параметра *`module`* необходимо указать имя загружаемого модуля. Все следующие далее аргументы интерпретируются как параметры, которые передаются модулю при загрузке. Параметры модулей обсуждаются ниже, в разделе "Параметры модулей".

Утилита `modprobe` пытается загрузить не только указанный модуль, но и все модули, от которых он зависит. Следовательно, это наиболее предпочтительный механизм загрузки модулей ядра.

Утилита `modprobe` также может использоваться для удаления модулей из ядра. Для этого, обладая правами пользователя `root`, запустите команду

```
modprobe -r modules
```

где вместо параметра *`modules`* нужно указать один или несколько модулей, которые необходимо удалить. В отличие от `rmmod`, утилита `modprobe` также удаляет и все модули, от которых зависит указанный модуль, если последние не используются. В восьмом разделе справочного руководства операционной системы Linux приведен список других, менее распространенных параметров этой команды.

Поддержка параметров конфигурации

В предыдущих разделах этой главы рассматривалась компиляция модуля управления удочкой при условии, что установлен параметр конфигурации `CONFIG_FISHING_POLE`. В начальных главах книги мы уже рассматривали параметры конфигурации, а теперь рассмотрим, как они добавляются на практике, используя в качестве примера драйвер устройства, управляющий удочкой.

Благодаря новой системе компиляции ядра "kbuild", которая появилась в серии ядер 2.6, добавление нового параметра конфигурации 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 нем, то необходимо указать на него ссылку в существующем файле `Kconfig`. Это можно сделать, если добавить в этот файл приведенную ниже строку.

```
source "drivers/char/fishing/Kconfig"
```

В данном примере эту строку нужно добавить в файл `drivers/char/Kconfig`.

Добавить конфигурационные записи в файл Kconfig очень просто. Для нашего модуля управления удочкой эта запись может выглядеть следующим образом:

```
config FISHING_POLE
   tristate "Поддержка Fish Master 3000"
   default n
   help
     Если вы введете символ Y, то в ядро будет включена поддержка
     компьютерного интерфейса Fish Master 3000, который будет доступен
     через узел устройства. Вы можете также ввести символ M, чтобы драйвер
     был создан в виде модуля с именем fishing.ko.


     Если вы не уверены, введите N.
```

В первой строке указывается, какому именно параметру конфигурации соответствует текущее определение. Обратите внимание: префикс CONFIG_ указывать не нужно, он добавляется автоматически.

Во второй строке указывается, 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 в виде модуля кода, который соответствует конфигурационному параметру (допустим, этот параметр определяет не драйвер, а просто некоторую дополнительную функцию), необходимо указать директиву bool вместо tristate. Текст в кавычках, который следует после этой директивы, определяет название конфигурационного параметра и будет отображаться различными утилитами конфигурации.

В третьей строке указывается стандартное значение этого параметра, который соответствует в данном случае запрещению компиляции модуля (n). В качестве стандартного значения можете также указать значение (y), тогда драйвер будет встроен в ядро; либо значение (m), чтобы был скомпилирован модуль. Однако для драйверов устройств в качестве стандартного значения обычно выбирается значение (n), соответствующее запрещению компиляции кода.

Директива help указывает на то, что остальная часть текста будет интерпретироваться как есть и соответствует текстовому описанию данного элемента конфигурации. Этот текст может по мере необходимости отображаться в различных конфигурационных утилитах. Так как этот текст предназначен для разработчиков, которые будут компилировать собственный вариант ядра, он должен быть коротким и ясным. Обычные пользователя, скорее всего, не будут компилировать ядро, а если будут, то тогда они должны понимать, что сказано в этом описании.

Существуют также и другие директивы файла конфигурации. В директиве depends перечисляются параметры конфигурации, которые должны быть активизированы перед тем, как может быть установлен текущий параметр. Если зависимости не будут удовлетворены, то текущий параметр отключается. Например, если в файл Kconfig будет добавлена директива

```
depends on FISH_TANK
```

то драйвер устройства нельзя будет активизировать (ввести символы y или m) до того, как будет активизирован параметр CONFIG_FISH_TANK.

Директива select аналогична директиве depends, за исключением того, что она принудительно активизирует указанный параметр конфигурации, если активизируется текущий параметр. Она не должна использоваться так же часто, как директива depends,

потому что она автоматически активизирует другие параметры конфигурации. В приведенной ниже строке файла конфигурации автоматически активизируется параметр CONFIG_BAIT, если будет активизирован параметр CONFIG_FISHING_POLE.

```
select BAIT
```

В обеих директивах, select и depends, можно указывать несколько параметров конфигурации, разделив их символами &&. В директиве depends можно задать, что некоторый параметр конфигурации не должен быть активизирован, если перед его именем указать восклицательный знак, например

```
depends on EXAMPLE_DRIVERS && !NO_FISHING_ALLOWED
```

Эта строка говорит системе построения модулей о том, что для компиляции текущего драйвера должен быть установлен параметр CONFIG_EXAMPLE_DRIVERS и не установлен CONFIG_NO_FISHING_ALLOWED.

После директив tristate и bool можно указать директиву if, что позволяет сделать соответствующий параметр зависимым от другого параметра конфигурации. Если условие не выполняется, то параметр конфигурации не только отключается, но и не будет отображаться утилитами конфигурации. Например, в приведенной ниже строке указывается, что система конфигурации ядра будет отображать параметр "Режим глубокого моря", только если параметр CONFIG_OCEAN будет активизирован.

```
bool "Режим глубокого моря" if OCEAN
```

Директива if также может быть указана после директивы default. Это означает, что стандартное значение будет установлено, только если выполняется условие, указанное в директиве if.

Для упрощения процесса настройки параметров в системе конфигурации экспортируется несколько метапараметров. Параметр CONFIG_EMBEDDED устанавливается только тогда, когда пользователь указывает, что он хочет видеть все параметры, отвечающие за запрещение некоторых ключевых возможностей ядра (обычно с целью сохранения памяти на встраиваемых системах). Параметр CONFIG_BROKEN_ON_SMP используется, чтобы указать, что драйвер не рассчитан на работу в системах с поддержкой симметричной многопроцессорности. Обычно этот параметр не устанавливается, при этом от пользователя требуется, чтобы он сам убедился в возможности компиляции драйвера для SMP. Разумеется, в новых драйверах этот флаг использоваться не должен. Параметр CONFIG_DEBUG_KERNEL позволяет задать установки, связанные с процессом отладки. И наконец, параметр CONFIG_EXPERIMENTAL используется для указания экспериментальных или не очень хорошо протестированных возможностей. По умолчанию этот параметр отключен, что требует от пользователя лично убедиться в степени риска при разрешении компиляции того или иного драйвера.

Параметры модулей

В ядре Linux реализована простая возможность определения параметров драйверов, значение которых должен указать пользователь 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 (о ней речь пойдет ниже). По этой причине создавать параметры модуля и управлять ими очень просто, несмотря на то, что их значения могут быть указаны буквально десятками разных способов.

Параметр модуля определяется с помощью макроса `module_param()` следующим образом:

```
module_param(name, type, perm);
```

где вместо аргумента *`name`* нужно указать реальное имя параметра, которое будет отображаться пользователю, а также совпадать с именем переменной в модуле. В аргументе *`type`* указывается тип данных, которые содержит параметр. Можно указать следующие значения типа: `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `charp`, `bool` и `invbool`. Они соответствуют байту, короткому целому, беззнаковому короткому целому, целому, беззнаковому целому, длинному целому, длинному беззнаковому целому, указателю на тип `char`, булеву значению и инверсному булеву (т.е. инвертированному значению, которое ввел пользователь). Данные типа `byte` хранятся в переменной типа `char`, а данные булевых типов — в переменных типа `int`. Остальные типы соответствуют аналогичным основным типам языка C.

В последнем аргументе *`perm`* указываются права доступа к соответствующему файлу в файловой системе `sysfs`. Права доступа можно указывать как в обычном восьмеричном формате, например `0644` (т.е. владелец имеет права на чтение и запись, а группа и остальные пользователи имеют право только на чтение), так и в виде обычных определений препроцессора (`S_I`*`что-то`*), 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.

Макрос `module_param` не определяет переменную вместо вас. *Перед* тем как использовать макрос, вы должны определить соответствующую переменную. В связи с этим типичный пример использования макроса может быть следующим:

```
/* Параметр модуля, который управляет возможностью использования
   живой приманки в удочке */
static int allow_live_bait = 1; /* По умолчанию активизирован */
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 по отношению к файлу.

Существует возможность дать внешнему параметру модуля имя, отличное от имени внутренней переменной. Это можно сделать с помощью макроса `module_param_named()`:

```
module_param_named(name, variable, type, perm);
```

где аргумент *`name`* соответствует имени внешнего параметра модуля, а *`variable`* — имени внутренней глобальной переменной, как показано в примере ниже.

```
static unsigned int max_test = DEFAULT_MAX_LINE_TEST;
module_param_named(maximum_line_test, max_test, int, 0);
```

Обычно тип `charp` используется для определения параметра модуля, которому должна быть передана строка. Ядро копирует переданную пользователем строку символов в память и присваивает переменной указатель на эту строку, как показано в следующем примере:

```
static char *name;
module_param(name, charp, 0);
```

По мере необходимости ядро может скопировать строку в заранее определенный массив символов, который должен указать разработчик. Это делается с помощью макроса `module_param_string()`:

```
module_param_string(name, string, len, perm);
```

где аргумент *name* соответствует имени внешнего параметра, а *string* — внутренней переменной модуля. В аргументе *len* указывается длина буфера *string* или некоторое меньшее его число, что обычно не имеет особого смысла. Вместо аргумента *perm* указываются права доступа к файлу файловой системы sysfs (нулевое значение запрещает доступ к параметру через sysfs).

```
static char species[BUF_LEN];
module_param_string(specifies, species, BUF_LEN, 0);
```

В качестве параметров модулю также можно передавать список значений, разделенные запятой, а в коде модуля они будут записаны в массив данных с помощью макроса module_param_array(), как показано ниже.

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

В данном случае в аргументе *name* указывается имя внешнего параметра и внутренней переменной. В аргументе *type* определяется тип данных одного значения, а в *perm* — права доступа к файлу файловой системы sysfs. В новом аргументе *nump* содержится указатель на целочисленную переменную, в которой ядро сохраняет количество элементов, записанных в массив. Обратите внимание на то, что массив, который передается в качестве параметра *name*, должен быть выделен статически. Размер этого массива определяется на этапе компиляции, и в ядре гарантируется, что он не будет переполнен. Пользоваться данным макросом очень просто, как показано ниже.

```
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);
```

Эти 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, "Размер удочки в дюймах.");
```

Для работы со всеми перечисленными в этом разделе макросами нужно включить заголовочный файл <linux/module.h>.

Экспортируемые символы

При загрузке модули динамически компонуются с ядром. Так же как и в случае динамически загружаемых бинарных файлов в пространство пользователя, в коде модулей могут вызываться только те функции ядра (основного образа или других модулей), которые явно *экспортированы* для использования. В ядре экспортирование осуществляется с помощью специальных директив EXPORT_SYMBOL() и EXPORT_SYMBOL_GPL().

Функции, которые экспортируются, можно использовать в модулях. 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. В коде ядра можно вызывать любые интерфейсы ядра (кроме тех, которые определены с ключевым словом static), потому что код ядра компонуется в один исполняемый образ. Разу-

меется, экспортируемые символы также не должны определяться как static. Набор символов ядра, которые экспортируются, называется *экспортируемым интерфейсом ядра*.

Экспортировать символы очень легко. После того как функция определена, необходимо вызвать директиву EXPORT_SYMBOL(), как показано ниже.

```
/*
 * get_pirate_beard_color - возвращает значение цвета бороды текущего пирата.
 * @pirate - это указатель на структуру типа pirate,
 * цвета определяются в <linux/beard_colors.h>.
*/
int get_pirate_beard_color(struct pirate *p)
{
   return p->beard.color;
}
EXPORT_SYMBOL(get_pirate_beard_color);
```

Предположим, 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 директивы MODULE_LICENSE(). Если хотите, чтобы рассматриваемая функция была доступна только для модулей, которые соответствуют лицензии GPL, то экспортировать функцию нужно следующим образом:

```
EXPORT_SYMBOL_GPL(get_pirate_beard_color);
```

Если ваш код сконфигурирован для компиляции в виде модуля, то необходимо гарантировать, что все используемые интерфейсы экспортируются. В противном случае будут возникать ошибки компоновки и загружаемый модуль работать не будет.

Модель представления устройств

В ядрах серии 2.6 появилась новая важная особенность — унифицированная *модель представления устройства*. Модель устройства — это единый механизм для представления устройств и описания их топологии в системе. Использование единого представления устройств позволяет получить ряд преимуществ.

- Уменьшается дублирование кода.
- Задействуется механизм для выполнения общих, часто встречающихся функций, таких как счетчики использования.
- Появляется возможность определения общего числа устройств в системе, просмотра их состояний и определения, к какой шине подключено то или другое устройство.
- Появляется возможность генерации полной и корректной информации о древовидной структуре всех устройств в системе, включая все шины и соединения.
- Обеспечивается возможность привязки устройств к их драйверам, и наоборот.
- Появляется возможность разделения устройств на категории в соответствии с различными классификациями, например устройства ввода, без знания физической топологии устройств.
- Обеспечивается возможность просмотра иерархии устройств от листьев к корню и выключения питания устройств в правильном порядке.

Отправной точкой в создании модели представления устройств стал последний пункт приведенного выше списка: создание корректной иерархии устройств для облегчения управления питанием. Для того чтобы реализовать в ядре управление электропитанием на уровне отдельного устройства, необходимо построить дерево, которое представляет топологию устройств в системе, — какое устройство подключено к какому контроллеру и какое из устройств подключено к какой из шин. Для выключения питания устройств, которые организованы в виде древовидной топологии, ориентированной сверху вниз, ядро должно сначала 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 корректно для всей системы, ядру необходимо отслеживать топологию дерева всех устройств в системе.

Объекты `kobject`

Основой модели представления устройств являются объекты *kobject*, которые представляются с помощью структуры `kobject`, определенной в файле `<linux/kobject.h>`. Объект `kobject` аналогичен классу `Object` таких объектно-ориентированных языков программирования, как C# и Java. В нем определяются самые общие функциональные возможности, такие как счетчик ссылок, название и указатель на родительский объект, что позволяет создавать иерархию объектов.

Структура, с помощью которой реализованы объекты `kobject`, имеет следующий вид:

```
struct kobject {
   const char          *name;
   struct list_head     entry;
   struct kobject      *parent;
   struct kset         *kset;
   struct kobj_type    *ktype;
   struct sysfs_dirent *sd;
   struct kref          kref;
   unsigned int         state_initialized:1;
   unsigned int         state_in_sysfs:1;
   unsigned int         state_add_uevent_sent:1;
   unsigned int         state_remove_uevent_sent:1;
   unsigned int         uevent_suppress:1;
};
```

В поле `name` хранится указатель на строку, содержащую имя объекта `kobject`. В поле `parent` хранится указатель 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 показано ниже, фактически эта иерархия объектов составляет основу файловой системы `sysfs`. С помощью последней в пространстве пользователя выполняется представление иерархии объектов `kobject`, которая существует в ядре.

В поле `sd` хранится указатель на структуру типа `sysfs_dirent`, с помощью которой представляется текущий объект `kobject` в файловой системе `sysfs`. Для этой цели внутри этой структуры находится структура типа `inode`.

Структура типа `kref` обеспечивает поддержку счетчика ссылок. Структуры `ktype` и `kset` позволяют описать и сгруппировать объекты `kobjects`. Они будут рассмотрены в двух следующих подразделах.

Обычно структуры kobject встраиваются в другие структуры данных и сами по себе не используются. Например, такая важная структура, как cdev, определенная в файле <linux/cdev.h>, имеет поле kobj.

```
/* Структура cdev - объект, представляющий символьное устройство */
struct cdev {
   struct kobject                kobj;
   struct module                *owner;
   const struct file_operations *ops;
   struct list_head              list;
   dev_t                         dev;
   unsigned int                  count;
};
```

При встраивании структур kobject в другие структуры данных последние будут поддерживать те же стандартные возможности, которые обеспечиваются структурами kobject. Более важно то, что структуры, которые содержат в себе объекты kobject, становятся частью иерархии объектов. Например, структура cdev представляется в иерархии объектов с помощью указателя на родительский объект cdev->kobj.parent и списка cdev->kob.entry.

Типы ktype

Объекты kobjects связаны с определенным типом, который называется ktype — сокращение от *kernel object type* (тип объекта ядра). Типы ktype представляются с помощью структуры kobj_type, определенной в файле <linux/kobject.h>, как показано ниже.

```
struct kobj_type {
   void (*release) (struct kobject *);
   const struct sysfs_ops *sysfs_ops;
   struct attribute      **default_attrs;
};
```

Тип ktype выполняет простую работу — он предназначен для описания стандартного поведения некоторого семейства объектов kobject. Чтобы не задавать особенности поведения для каждого отдельного объекта kobject, они сохраняются в структуре типа ktype. Таким образом, объекты kobject, имеющие один и тот же тип поведения, указывают на одну и ту же структуру типа ktype и в результате характеризуются одинаковым поведением.

В поле release находится указатель на деструктор, который вызывается, когда количество ссылок на объект kobject становится равным нулю. В этой функции выполняется освобождение памяти, связанной с объектом kobject, и другие завершающие операции.

В поле sysfs_ops содержится указатель на структуру sysfs_ops. С ее помощью определяется поведение файловой системы sysfs при выполнении операций чтения и записи файлов. Более подробно она рассматривается ниже, в разделе “Добавление файлов в файловую систему sysfs”.

И наконец, в поле default_attrs содержится указатель на массив структур attribute. В этих структурах определяются стандартные *атрибуты*, связанные с текущим объектом kobject. Атрибуты соответствуют свойствам данного объекта. Если некоторый объект kobject экспортируется через файловую систему sysfs, то атрибуты

будут экспортированы в виде отдельных файлов. В последнем элементе этого массива должно содержаться значение NULL.

Множества объектов kset

Множества kset (сокращение от *kernel object sets*, или множества объектов ядра) представляют собой совокупность коллекций объектов kobject. Множество kset выполняет функции базового контейнерного класса для набора объектов ядра. Оно позволяет собрать в одном месте все связанные объекты kobjects, например, соответствующие всем блочным устройствам. Множества kset очень похожи на типы ktype, поэтому возникает законный вопрос: "Для чего нужны два разных обобщения?" Множество kset объединяет несколько объектов kobject, тогда как типы ktype позволяют объектам ядра (функционально связанным друг с другом или нет) использовать общие операции. Это различие позволяет сгруппировать объекты ядра одного типа ktype в различные множества kset. Другими словами, в ядре Linux может существовать только небольшое количество типов ktype и довольно много множеств kset.

В поле kset объекта kobject хранится указатель на связанное с данным объектом множество kset. Множество объектов kset представляется с помощью структуры kset, которая определена в файле <linux/kobject.h> следующим образом:

```
struct kset {
   struct list_head        list;
   spinlock_t              list_lock;
   struct kobject          kobj;
   struct kset_uevent_ops *uevent_ops;
};
```

Все объекты kobject, входящие в одно множество, связываются через поле list этой структуры в двухсвязный список. Для защиты текущего элемента списка используется поле спин-блокировки list_lock (спин-блокировки были описаны в главе 10, "Средства синхронизации ядра"). В поле kobj находится объект kobject, который представляет базовый класс для всех объектов данного множества. В поле uevent_ops хранится указатель на структуру, определяющую поведение объектов kobject при "горячем" подключении устройств, связанных с данным множеством. Для передачи информации пользовательским программам о подключаемых и отключаемых устройствах в системе придуман механизм пользовательских событий *uevent* (User Events).

Взаимосвязь kobject, ktype и kset

Те несколько структур, которые только что были рассмотрены, приводят к путанице не потому, что их много (всего три) или они сложные (все они достаточно просты), а потому, что они сильно переплетаются друг с другом. При использовании объектов kobject достаточно сложно рассказать об одной структуре, не упоминая другие. Тем не менее на основании рассмотренных особенностей этих структур можно воссоздать четкую структуру их взаимоотношений.

Самым важным является объект kobject, который представляется с помощью структуры kobject. Структура kobject используется для представления наиболее общих объектных свойств структур данных ядра, таких как счетчик ссылок, взаимоотношения "родитель–потомок" и имя объекта. С помощью структуры kobject эти свойства можно обеспечить одинаковым для всех стандартным способом. Сами по себе

структуры kobject не очень полезны: обычно они встраиваются в другие структуры данных, наделяя последние свойствами объектов kobject.

С каждым объектом kobject связан один определенный тип данных — ktype, который представляется с помощью структуры kobj_type. Ссылка на экземпляр такой структуры указывается в 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 стандартные атрибуты объекта. Структуре ktype присвоено не очень удачное имя. Ее нужно рассматривать не с точки зрения группировки, а с точки зрения обеспечения поддержки общих операций.

Объекты kobject группируются в множества, которые называются kset и представляются с помощью структур данных 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, позволяют объединять вместе несколько связанных между собой объектов kobject. В файловой системе sysfs объекты kobject представляются в виде отдельных каталогов. Связанные между собой каталоги, например, все подкаталоги одного каталога, могут быть включены в одно множество kset. Взаимоотношения между этими структурами данных показаны на рис. 17.1.

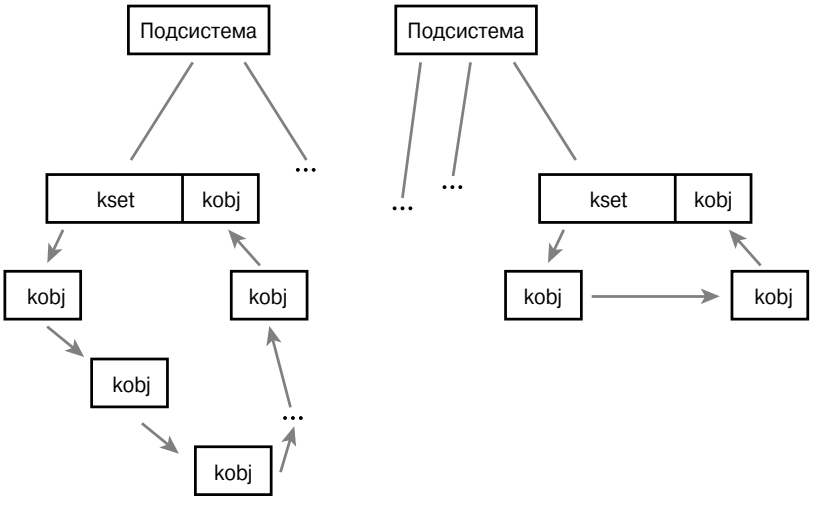


Рис. 17.1. Взаимоотношения между объектами kobject, множествами kset и подсистемами

Управление и работа 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 действия. 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 подсис-

темы 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.

Чтобы начать работу с объектами kobject, их необходимо описать и проинициализировать. Для инициализации объектов kobject используется функция kobject_init(), которая определена в файле <linux/kobject.h> следующим образом:

```
void kobject_init(struct kobject *kobj, struct kobj_type *ktype);
```

В качестве первого параметра функции передается указатель на объект kobject, который нужно проинициализировать. Перед вызовом этой функции область памяти, в которой хранится объект kobject, должна быть заполнена нулевыми значениями. Обычно это делается при инициализации большой структуры данных, в которую встраивается объект kobject. В других случаях просто вызовите функцию memset(), как показано ниже.

```
memset(kobj, 0, sizeof (*kobj));
```

После заполнения нулями можно смело проинициализировать значения полей parent и kset, как показано в следующем примере:

```
struct kobject *kobj;

kobj = kmalloc(sizeof (*kobj), GFP_KERNEL);
if (!kobj)
   return -ENOMEM;
memset(kobj, 0, sizeof (*kobj));
kobj->kset = my_kset;
kobject_init(kobj, my_ktype);
```

Приведенная выше последовательность действий выполняется автоматически в функции kobject_create(), которая возвращает только что распределенный объект kobject.

```
struct kobject * kobject_create(void);
```

Использовать эту функцию очень просто.

```
struct kobject *kobj;

kobj = kobject_create();
if (!kobj)
   return -ENOMEM;
```

В большинстве случаев для создания объектов kobject следует использовать функцию kobject_create() либо связанную с ними вспомогательную функцию и не задавать значения полей этой структуры вручную.

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 *закреплен*. Перед работой с объектом в любом коде следует вначале увеличить значение счетчика ссылок. После окончания работы с объектом в коде нужно уменьшить значение счетчика ссылок на единицу. Увеличение значения счетчика называют *захватом* (*getting*), а уменьшение — *освобождением* (*putting*) ссылки на объект. Когда значение счетчика становится равным нулю, объект может быть уничтожен, а занимаемая им память освобождена.

Увеличение и уменьшение значения счетчика ссылок

Увеличение значения счетчика ссылок выполняется с помощью функции kobject_get(), описанной в файле <linux/kobject.h>, следующим образом:

```
struct kobject * kobject_get(struct kobject *kobj);
```

Эта функция возвращает указатель на объект kobject в случае успешного выполнения и значение NULL в случае ошибки.

Уменьшение значения счетчика ссылок выполняется с помощью функции kobject_put(), также описанной в файле <linux/kobject.h>.

```
void kobject_put(struct kobject *kobj);
```

Если значение счетчика ссылок объекта kobject, передаваемого в качестве параметра, становится равным нулю, то вызывается функция-деструктор, на которую указывает указатель release поля ktype этого объекта. При этом занимаемая объектом память освобождается, и все ссылки на него становятся некорректными.

Структуры *kref*

Для внутреннего представления счетчика ссылок используется структура kref, которая определена в файле <linux/kref.h>, как показано ниже.

```
struct kref {
   atomic_t refcount;
};
```

В единственном поле этой структуры находится неделимая переменная, в которой хранится значение счетчика ссылок. Структура используется просто для того, чтобы можно было выполнять проверку типов. Перед использованием структуры kref ее необходимо инициализировать с помощью функции kref_init().

```
void kref_init(struct kref *kref)
{
   atomic_set(&kref->refcount, 1);
}
```

Как видите, эта функция просто присваивает неделимой переменной типа atomic_t значение, равное единице. Следовательно, структура kref является захваченной сразу же после инициализации, поскольку значение ее счетчика равно единице. Так же ведут себя и объекты kobject.

Для того чтобы захватить ссылку на структуру kref, необходимо использовать функцию kref_get(), описанную в файле <linux/kref.h>.

```
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(), также описанную в файле <linux/kref.h>.

```
int kref_put(struct kref *kref, void (*release) (struct kref *kref))
{
   WARN_ON(release == NULL);
   WARN_ON(release == (void (*)(struct kref *))kfree);
```

```
    if (atomic_dec_and_test(&kref->refcount)) {
        release(kref);
        return 1;
    }
    return 0;
}
```

Эта функция уменьшает значение счетчика ссылок на единицу и вызывает функцию `release()`, которая передается ей в качестве параметра, когда значение счетчика ссылок становится равным нулю. Как видно из использованного выражения `WARN_ON()`, функция `release()` не может просто совпадать с функцией `kfree()`, а должна быть специальной функцией, которой передается указатель на структуру `kref` в качестве единственного параметра, и не возвращает никаких значений. Эта функция возвращает нулевое значение, если освобождаемая ссылка не была последней ссылкой на этот объект (тогда возвращается значение `1`). Обычно в программе, вызвавшей функцию `kref_put()`, ее код возврата не проверяется.

Вместо того чтобы разрабатывать свои функции управления счетчиками ссылок на основании типа данных `atomic_t` и простых интерфейсных функций для захвата и освобождения ссылок, настоятельно рекомендуется использовать тип данных `kref` и соответствующие вспомогательные функции, которые обеспечивают общий и правильно работающий механизм поддержки счетчиков ссылок в ядре.

Все описанные выше 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 пользователям 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 по необходимости 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 представления устройств решил экспортировать дерево устройств в виде файловой системы. Такое решение быстро показало свою целесообразность 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 Linux, на которой работает ядро 2.6, имеет поддержку файловой системы `sysfs`, и в большинстве случаев ее точкой монтирования является каталог `/sys`.

Основная идея работы файловой системы `sysfs` — это привязка объектов `kobject` к структуре каталогов с помощью поля `dentry`, которое есть в каждой структуре `kobject`. Вспомните из материала главы 13, “Виртуальная файловая система”, что структура `dentry` используется для представления элементов каталогов. Связывание объектов `kobject`

с элементами каталогов dentry 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.

В корневом каталоге файловой системы sysfs находится как минимум десять каталогов: block, bus, class, dev, devices, firmware, fs, kernel, module и power. В каталоге block содержатся подкаталоги для каждого зарегистрированного в системе устройства блочного ввода-вывода. В каждом из этих подкаталогов в свою очередь содержатся 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 dev позволяет просматривать зарегистрированные узлы устройств. Каталог devices содержит информацию о топологии устройств в системе. Она отображается непосредственно на иерархию структур устройств ядра. Каталог firmware содержит специфичное для данного компьютера дерево низкоуровневых подсистем, таких как ACPI, EDD, EFI и т.д. Каталог fs позволяет просматривать список зарегистрированных файловых систем. В каталоге kernel находится список параметров настройки ядра, а также информация о состоянии. В каталоге module представлен список загруженных модулей ядра. В каталоге power содержатся данные по управлению электропитанием всех устройств системы. Перечисленные выше каталоги реализованы не во всех ОС Linux, кроме того, в некоторых системах имеются каталоги, не описанные выше.

Наиболее важным является каталог devices, с помощью которого экспортируется 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 упорядочена в соответствии с высокоуровневым представлением зарегистрированных сетевых интерфейсов. 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

```
|-- block
|   |-- loop0 -> ../devices/virtual/block/loop0
|   |-- md0 -> ../devices/virtual/block/md0
|   |-- nbd0 -> ../devices/virtual/block/nbd0
|   |-- ram0 -> ../devices/virtual/block/ram0
|   `-- xvda -> ../devices/vbd-51712/block/xvda
|-- bus
|   |-- platform
|   |-- serio
|-- class
|   |-- bdi
|   |-- block
|   |-- input
|   |-- mem
|   |-- misc
|   |-- net
|   |-- ppp
|   |-- rtc
|   |-- tty
|   |-- vc
|   `-- vtconsole
|-- dev
|   |-- block
|   `-- char
|-- devices
|   |-- console-0
|   |-- platform
|   |-- system
|   |-- vbd-51712
|   |-- vbd-51728
|   |-- vif-0
|   `-- virtual
|-- firmware
|-- fs
|   |-- ecryptfs
|   |-- ext4
|   |-- fuse
|   `-- gfs2
|-- kernel
|   |-- config
|   |-- dlm
|   |-- mm
|   |-- notes
|   |-- uevent_helper
|   |-- uevent_seqnum
|   `-- uids
`-- module
    |-- ext4
    |-- i8042
    |-- kernel
    |-- keyboard
    |-- mousedev
    |-- nbd
    |-- printk
    |-- psmouse
    |-- sch_htb
    |-- tcp_cubic
    |-- vt
    `-- xt_recent
```

Рис. 17.2. Содержимое части каталога `/sys`

доступны, являются побочным эффектом обслуживания иерархии устройств в ядре и описывают все то, что происходит внутри ядра[1].

Добавление и удаление объектов файловой системы sysfs

Проинициализированные 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, struct kobject *parent,
                                          const char *fmt, ...);
```

Положение конкретного объекта в файловой системе sysfs зависит от его положения в иерархии объектов ядра. Если установлен указатель на родительский объект, находящийся в поле parent, то текущий объект будет отображен в каталоге, соответствующем объекту, на который указывает указатель parent. Если указатель на родительский объект не установлен, то текущий 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. В большинстве случаев практического использования одно из полей, parent или kset (либо сразу оба), должно быть корректно установлено перед вызовом функции kobject_add(). Несмотря ни на что, имя каталога, который представляет объект kobject в файловой системе sysfs, берется из аргумента fmt этой функции. Формат этого аргумента совпадает со строкой форматирования функции printf().

Существует еще одна вспомогательная функция kobject_create_and_add(), которая выполняет работу двух функций — kobject_create() и kobject_add().

```
struct kobject * kobject_create_and_add(const char *name, struct kobject
*parent);
```

Обратите внимание: имя каталога, представляющего объект kobject, передается функции kobject_create_and_add() в виде прямой ссылки на строку (аргумент name), тогда как для функции kobject_add() используется система форматирования имени, принятая в функции printf().

Удаление объекта из файловой системы sysfs выполняется с помощью функции kobject_del().

```
void kobject_del(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, или уровень абстракции аппаратных средств), информация о котором доступна по адресу http://www.freedesktop.org/wiki/Software/hal. Подсистема HAL позволяет создать в оперативной памяти базу данных на основании информации файловой системы sysfs, объединяя вместе концепции классов, устройств и драйверов. На основании этих данных уровень HAL предоставляет API, которое позволяет разрабатывать более интеллектуальные программы.

Добавление файлов в файловую систему sysfs

Мы уже знаем, что объекты kobject отображаются на каталоги файловой системы sysfs, причем иерархия объектов ядра полностью соответствует структуре ее каталогов. Но как быть с самими файлами, составляющими эти каталоги? Файловая система sysfs представляет собой удобное средство для отображения структуры объектов в виде дерева, в которую не входят файлы, соответствующие реальным данным.

Стандартные атрибуты

Стандартный набор файлов, которые создаются в каталоге, 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 соответствуют этим объектам. В структуре kobj_type содержится поле default_attrs, которое представляет собой массив структур attribute. С помощью атрибутов данные ядра отображаются в файлы файловой системы sysfs.

Структура attribute определена в файле <linux/sysfs.h>.

```
/* структура attribute - атрибуты позволяют отобразить данные ядра
                         в файлы файловой системы sysfs */
struct attribute {
   const char    *name;  /* Имя атрибута */
   struct module *owner; /* Модуль, которому принадлежат данные
                            (если таковой имеется) */
   mode_t         mode;  /* Права доступа к файлу */
};
```

Имя атрибута определяется с помощью поля name этой структуры. Такое же имя будет иметь и соответствующий файл в файловой системе sysfs. В поле owner хранится 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 в файловой системе sysfs. Если атрибут предназначен для чтения всеми, то флаг прав доступа должен быть установлен в значение S_IRUGO, если атрибут имеет право читать только сам владелец, то нужно использовать S_IRUSR. Атрибуты с правом на запись, скорее всего, будут иметь права доступа S_IRUGO | S_IWUSR. Все файлы и каталоги в файловой системе sysfs принадлежат пользователю и группе, идентификаторы которых равны нулю.

Несмотря на то что в поле default_attrs находится список стандартных атрибутов, способы работы с ними описываются с помощью поля sysfs_ops. В нем находится указатель на одноименную структуру, которая определена в файле <linux/sysfs.h>.

```
struct sysfs_ops {
   /* Метод вызывается при чтении файла файловой системы sysfs */
   ssize_t (*show) (struct kobject   *kobj,
                    struct attribute *attr,
                    char *buffer);
   /* Метод вызывается при записи файла файловой системы sysfs */
   ssize_t (*store) (struct kobject   *kobj,
                     struct attribute *attr,
                     const char *buffer,
                     size_t size);
};
```

Метод show() вызывается при чтении файла файловой системы sysfs из пространства пользователя. Он должен скопировать значение атрибута, который передается в качестве параметра attr, в буфер, на который указывает параметр buffer. Размер буфера равен PAGE_SIZE байт. Для аппаратной платформы x86 значение PAGE_SIZE равно 4096 байтам. Функция должна возвратить количество байтов данных, которые записаны в буфер при успешном завершении, и отрицательный код в случае возникновения ошибки.

Метод store() вызывается при записи. Он должен скопировать size байт данных из буфера buffer в переменную, представляемую атрибутом attr. Размер буфера всегда равен PAGE_SIZE или менее. Функция должна возвратить количество байтов данных, которые прочитаны из буфера при успешном выполнении, и отрицательный код ошибки в случае неудачного завершения.

Поскольку этот набор функций должен выполнять операции ввода-вывода для *всех* атрибутов, необходимо выполнить некоторые дополнительные действия, чтобы вызвать обработчик, специфичный для каждого атрибута.

Создание нового атрибута

Обычно стандартных атрибутов, предоставляемых типом ktype и связанных с объектом kobject, оказывается достаточно. Действительно, целью введения типа ktype была реализация общих операций в объектах kobject. Использование общих типов ktype в нескольких объектах kobject не только упрощает сам процесс программирования, 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 объектов может оказаться желательным (или необходимым) создать атрибуты, которых нет у общего типа данного объекта. Возможно, вы захотите оперировать данными или использовать функциональные возможности, которые не предусмотрены в более общем типе ktype. Для такого случая в ядре предусмотрена функция sysfs_create_file() для добавления атрибута к существующему набору.

```
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 нулевое значение в случае нормального завершения и отрицательное значение в случае ошибки.

Для обработки указанного атрибута используется структура sysfs_ops, соответствующая типу ktype объекта. Иными словами, существующие стандартные методы show() и store() должны иметь возможность обработать вновь созданный атрибут.

Кроме создания реальных файлов, существует возможность создавать символьные ссылки. Создать символьную ссылку в файловой системе sysfs очень легко.

```
int sysfs_create_link(struct kobject *kobj, struct kobject *target, char *name);
```

Данная 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 нормального завершения и отрицательное значение в случае ошибки.

Удаление атрибутов

Для удаления атрибута используется функция 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 name удаляется из каталога, на который отображается объект kobj.

Все описанные выше 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 файловой системы sysfs

Файловая система sysfs — это *место*, где должна реализовываться функциональность, для которой раньше использовалась системная функция ioctl(), вызванная для специальных файлов устройств, или файловая система procfs. Вместо использования устаревших интерфейсов ядра современные разработчики реализуют подобные функциональные возможности через атрибуты файловой системы sysfs, помещаемые в соответствующий каталог. Например, вместо того чтобы реализовывать новые директивы 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) и непонятных аргументов функции 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, в виде файла должно экспортироваться только одно значение атрибута sysfs. Значения должны быть представлены в текстовом формате и соответствовать базовым типам языка программирования C. Целью такого представления является необходимость избежать чрезвычайно запутанного и плохо структурированного представления информации, которое мы сегодня имеем в файловой системе /proc. Экспортирование одной переменной в 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 C. В случаях, когда экспортирование одного значения в 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 простой 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, чтобы они формировали интуитивно понятную древовидную структуру файловой системы sysfs. 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 двоичным интерфейсом ядра к прикладным программам (Application Binary Interface — ABI). Пользовательские программы должны разрабатываться в соответствии с наличием, положением, содержимым и поведением каталогов и файлов файловой системы sysfs. Изменение положения существующих файлов крайне не желательно, 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 смогут создавать простой и понятный код, а также иметь мощный и интуитивно понятный интерфейс с ядром.

Уровень событий ядра

Уровень событий ядра (Kernel Event Layer) — это подсистема, которая позволяет передавать информацию о различных событиях из ядра в пользовательские приложения и реализована, как вы уже, наверное, догадываетесь, на основе объектов kobject. После выпуска ядра версии 2.6.0 стало ясно, что необходим механизм для отправки сообщений из ядра пользовательским приложениям, 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 должно помещать возникающие события в стек. Например, “Жесткий диск переполнен!”, “Процессор перегрелся!”, “Раздел диска смонтирован!”.

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. Поскольку эти объекты отображаются на элементы каталогов файловой системы sysfs, *источниками* событий являются определенные элементы пути файловой системы sysfs. Например, если поступившее событие связано с первым жестким диском, адресом источника события является каталог /sys/block/hda. Внутри же ядра источником данного события будет соответствующий объект kobject.

Каждому событию присваивается определенная строка символов, которая представляет сигнал и называется *командой* (*verb*) или *действием* (*action*). В этой строке символов содержится информация о том, *что именно* произошло, например *изменение* (*modified*) или *размонтирование* (*unmounted*).

И наконец, для каждого события может существовать некоторая дополнительная информация. Однако строка, описывающая эту полезную информацию, не передается непосредственно в пользовательские приложения, поскольку она может иметь произволь-

ный формат. Вместо этого она представляется с помощью атрибутов, отображаемых в файлы файловой системы sysfs.

В ядре события поступают из пространства ядра в пространство пользователя через интерфейс netlink. Интерфейс netlink — это специальный тип высокоскоростного сетевого сокета многоадресатной 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. Чтобы эти события можно было перенаправить в пространство пользователя, необходимо 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, основана на привязке событий к демону D-BUS[2], с помощью которого также реализуется и глобальная системная шина сообщений между приложениями пользователя. В результате ядро может подавать сигналы так же, как это делают все остальные компоненты системы.

Для отправки события в пространство пользователя в коде ядра нужно вызвать функцию kobject_uevent().

```
int kobject_uevent(struct kobject *kobj, enum kobject_action action);
```

В первом параметре указывается объект kobject, который является источником сигнала. Соответствующее событие ядра будет содержать элемент пути файловой системы sysfs, связанный с объектом, сгенерировавшим сигнал.

Во втором параметре указывается *команда* или *событие*, описывающее сигнал. Сгенерированное событие ядра будет содержать строку, которая соответствует номеру, передаваемому в качестве значения параметра enum kobject_action. Как видите, вместо передачи строки здесь передается 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.h> и имеют имена в формате KOBJ_*foo*. На момент написания этой книги были определены следующие события: KOBJ_MOVE, KOBJ_ONLINE, KOBJ_OFFLINE, KOBJ_ADD, KOBJ_REMOVE и KOBJ_CHANGE. Эти значения соответствуют строкам “move” (перемещение), “online” (готов), “offline” (не готов), “add” (добавление), “remove” (удаление) и “change” (изменение) соответственно. Допускается добавление новых значений событий, если существующих значений недостаточно.

Использование объектов kobject и их атрибутов не только дает возможность описать события в терминах файловой системы sysfs, но и стимулирует создание новых объектов и их атрибутов, которые еще не представлены через файловую систему sysfs.

Функция kobject_uevent() и связанные с ней функции определены в файле lib/kobject_uevent.c и описаны в файле <linux/kobject.h>.

[2] Более подробную информацию о демоне D-BUS можно найти на сайте http://www.freedesktop.org/wiki/Software/dbus.

Резюме

В данной главе были рассмотрены функциональные возможности ядра, которые используются для реализации драйверов устройств и поддержки их иерархии в виде древовидной структуры. Были описаны модули, объекты ядра `kobject` и связанные с ними множества `kset` и типы `ktype`, а также файловая система `sysfs`. Приведенные сведения очень важны для разработчиков драйверов устройств, поскольку они позволяют писать модульный, компактный и мощный код.

В последних трех главах мы уйдем от рассмотрения специфичных для ядра Linux подсистем и остановимся на общих вопросах, связанных с ошибками ядра. В следующей главе речь пойдет о процессе отладки ядра Linux.

18
Отладка

Один из самых существенных факторов, который отличает разработку ядра от разработки пользовательских приложений, — это сложность отладки. Отлаживать код ядра сложно, по крайней мере, по сравнению с кодом пользовательских приложений. 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 разработке ядра вообще в основном зависит от вашего опыта и понимания принципов работы операционной системы в целом. Разумеется, в этом деле вам также помогут наблюдательность и проницательность, но для успешного поиска ошибок в ядре необходимо понимать, как работает ядро. Поэтому в данной главе будут рассмотрены методы отладки кода ядра.

Начало работы

Поиск ошибок в ядре часто представляет собой длительный и мучительный процесс. Некоторые ошибки ставили в тупик все сообщество разработчиков ядра на несколько месяцев. К счастью, на каждую из таких трудных ошибок приходится довольно много простых, которые легко исправить. Если вам повезет, то все проблемы, с которыми вы столкнетесь, будут простыми. Однако вы об этом никогда и не узнаете, если не начнете свои исследования. Для этого вам понадобится следующее.

- Сама проблема. Как это ни странно звучит, но ошибка в ядре должна быть точно идентифицирована и иметь характерные признаки проявления. Если хотя бы кто-нибудь может ее устойчиво воспроизвести, то процесс поиска существенно облегчается. Но, к сожалению, ошибки в ядре не всегда можно точно идентифицировать и воспроизвести условия, при которых они возникают.
- Номер версии ядра, в которой существует ошибка. Лучше всего, если известна версия, в которой эта ошибка *впервые* появилась. Если такой информации нет, то в текущей главе будут описаны методы, позволяющие установить это.
- Хорошие знания кода ядра, в котором, предположительно, может возникать ошибка и, конечно, немного удачи. Поиск ошибок в ядре очень сложен, и чем лучше вы знаете код, тем лучше.

В большинстве описанных в этой главе методик отладки предполагается, что ошибка может быть устойчиво воспроизведена. От этого зависит дальнейший успех отладки. Если же ошибку нельзя воспроизвести, то вам остается только переосмыслить проблему и попытаться отыскать ее визуально путем анализа исходного кода. На самом деле так случается достаточно часто, но очевидно, что шансы добиться успеха становятся более весомыми, если появляется возможность устойчиво воспроизвести ошибку.

Вам может показаться странным, что существуют ошибки в ядре, которые у кого-то никак не проявляются. Ведь в пользовательских программах ошибки чаще всего проявляются стабильно, например *вызов функции* `foo` *приводит к созданию файла* `core`. Ошибки в ядре чаще всего не так очевидны. Процессы взаимодействия ядра, пространства пользователя и оборудования могут быть достаточно тонкими. Конфликт при доступе к ресурсу может возникать всего один раз за миллион итераций алгоритма. Плохо спроектированный или не правильно скомпилированный код может обеспечивать удовлетворительную производительность на одной системе, но совсем неудовлетворительную на другой. Очень часто происходит так, что на какой-то случайной машине, при очень специфическом характере загрузке, начинают проявляться ошибки, которые больше нигде не проявляются. Чем больше доступно дополнительной информации при локализации ошибки, тем лучше. Во многих случаях, как только удалось устойчиво воспроизвести проблему, можно считать, что б*о*льшая половина работы сделана.

Ошибки ядра

Ошибки в ядре могут быть самыми разными. Они возникают по различным причинам и проявляются в разнообразных формах. Одной из причин является некорректный исходный код (например, правильное значение было записано в неподходящее место в памяти). Другой причиной являются ошибки синхронизации, когда при обращении к совместно используемой переменной она не была должным образом заблокирована. Ошибки в ядре возникают также и при некорректной работе с аппаратным обеспечением компьютера (например, отправка контроллеру ошибочного кода операции или запись значения в некорректный управляющий регистр). Ошибки в ядре могут проявиться в любой форме: от потери производительности системы до некорректного ее функционирования и даже до потери данных и зависания.

Часто между тем моментом, когда в ядре возникла ошибка, и тем моментом, когда пользователь ее заметил, происходит большая цепь событий. Например, при совместном использовании структуры данных, у которой нет счетчика использования, может возникнуть конфликтная ситуация из-за доступа к ресурсу. Если не принять необходимых мер, то один процесс может освободить память, в которой хранится структура, использующаяся в другом процессе. Через какое-то время второму процессу может потребоваться обратиться к полям этой (уже несуществующей!) структуры, воспользовавшись ставшим уже некорректным указателем. В результате будет выполнено обращение к памяти по указателю `NULL`, либо из памяти будут считаны некорректные данные (так называемый “мусор”), либо вообще не произойдет ничего плохого, если к этому моменту область памяти, в которой хранилась структура, еще не была перезаписана. Обращение к памяти по указателю `NULL` вызовет выдачу на консоль сообщения `oops`, тогда как считывание некорректных данных приведет к дальнейшему искажению данных, что вызовет неправильную работу системы и в конечном счете может вызвать ее аварийный останов. При этом пользователь заметит только сообщение `oops` либо некорректное функционирова-

ние системы. Разработчик ядра при этом должен пойти по 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 конфликта при доступе к ресурсу, и исправить ошибку путем правильного учета количества ссылок на совместно используемую структуру данных.

Процесс отладки 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 ограничение по времени выполнения участков кода и возможность возникновения конфликтов при доступе к ресурсам. Все это является результатом параллельного выполнения множества потоков в ядре.

Отладка с помощью вывода диагностических сообщений

Функция форматированного вывода сообщений ядра `printk()` аналогична библиотечной функции `printf()` языка C. Действительно, до этого момента мы не видели никаких существенных отличий в ее использовании. Для большинства задач это то, что нужно. Функция `printk()` — это просто функция ядра, выполняющая форматированный вывод сообщений. Однако некоторые различия все же имеются.

Устойчивость

Одно из проверенных и часто используемых свойств функции `printk()` — ее устойчивость. Функцию `printk()` можно вызывать практически *в любое время* и *в любом месте* ядра. Ее можно вызывать из контекста прерывания и из контекста процесса. Ее можно вызывать во время захвата любой блокировки. Ее можно также вызывать одновременно на нескольких процессорах, и она не требует при этом захвата какой-нибудь блокировки.

Данная функция очень устойчива, и это очень важно, потому что полезность функции `printk()` основывается на том факте, что она всегда доступна и всегда работает.

Однако слабое место у функции `printk()` в плане устойчивости все же существует. При загрузке ядра ее нельзя использовать до некоторого момента, пока еще не прошла инициализация консоли. Действительно, если консоли еще нет, то куда будут выводиться сообщения? Обычно это не является большой проблемой, если только вы не выполняете отладку кода, который выполняется на очень ранних стадиях процесса загрузки (например, функции `setup_arch()`, выполняющей инициализацию, специфичную для аппаратной платформы). Отладка такого рода является очень сложной задачей, а отсутствие каких-либо способов вывода сообщений только усугубляет проблему.

В подобных 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 особой радости. Одно из решений проблемы — вариант функции `printk()`, который может выводить информацию на консоль на очень ранних стадиях процесса загрузки — `early_printk()`. Поведение этой функции аналогично поведению функции `printk()`, за исключением имени и возможности работать на очень ранних стадиях процесса загрузки. Однако такое решение не является

переносимым, поскольку данная функция реализована не для всех поддерживаемых аппаратных платформ. Тем не менее вам остается только надеяться, что эта функция реализована на используемой вами аппаратной платформе, как это сделано для большинства аппаратных платформ, включая x86.

Кроме ситуаций, когда необходимо выводить на консоль информацию на очень ранних стадиях процесса загрузки системы, для отладки кода ядра можно использовать функцию `printk()`, которая работает практически всегда.

Уровни вывода сообщений ядра

Главное отличие между функциями `printk()` и `printf()` — это возможность указывать в первой *уровень важности сообщений ядра* (*loglevel*). Последняя используется ядром при принятии решения о том, стоит ли выводить данное сообщение на консоль или нет. На консоли будут отображены все сообщения, уровень важности которых ниже заданного значения.

Ниже приведено несколько примеров указания уровня важности.

```
printk(KERN_WARNING "This is a warning!\n");
printk(KERN_DEBUG "This is a debug notice!\n");
printk("I did not specify a loglevel!\n");
```

Строки `KERN_WARNING` и `KERN_DEBUG` определены с помощью директив препроцессора в заголовочном файле `<linux/kernel.h>`. Вместо них подставляется строка символов наподобие `"<4>"` или `"<7>"`, которая добавляется в самое начало сообщения, выводимого функцией `printk()`. На основании этого префикса и установленного уровня важности сообщений консоли (значение переменной `console_loglevel`) ядро может принять решение о выводе данного конкретного сообщения на консоль. В табл. 18.1 приведен полный список возможных значений уровня важности сообщений.

Таблица 18.1. Уровни важности сообщений ядра (`loglevel`)

Значение `loglevel`	Описание
`KERN_EMERG`	Аварийная ситуация; система, скорее всего, находится в нерабочем состоянии
`KERN_ALERT`	Проблема, на которую требуется немедленно обратить внимание
`KERN_CRIT`	Критическая ситуация
`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 содержащее избыточную информацию

Если уровень важности сообщений ядра не указан, то используется стандартное значение `DEFAULT_MESSAGE_LOGLEVEL`, которое в данный момент соответствует `KERN_WARNING`. Поскольку стандартное значение может со временем измениться, для своих сообщений необходимо всегда указывать уровень важности.

Наиболее важный уровень сообщения — `KERN_EMERG` — определен как `"<0>"`, а наименее важный — `KERN_DEBUG` — как `"<7>"`. Например, после обработки препроцессором кода из предыдущего примера получается следующее:

```
printk("<4>This is a warning!\n");
printk("<7>This is a debug notice!\n");
printk("<4>I did not specify a loglevel!\n");
```

Назначение уровней важности сообщений, выводимых с помощью функции `printk()`, зависит только от вас. Разумеется, обычные сообщения, которые нужно вывести на экран консоли, должны иметь соответствующий уровень важности. Отладочные сообщения, вывод которых в большом количестве встраивается в самые разные места кода с целью разобраться с проблемой, — "допустим, ошибка здесь", "пробуем", "работает" — могут иметь любой уровень важности. Один из вариантов — не изменять стандартный уровень вывода сообщений на консоль, а всем вашим отладочным сообщениям назначить уровень важности типа `KERN_CRIT`. Можно поступить и наоборот, т.е. установить для отладочных сообщений уровень `KERN_DEBUG` и поднять уровень, при котором эти сообщения выводятся на консоль. Каждый из вариантов имеет свои положительные и отрицательные стороны, поэтому решать вам.

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`. Стандартный размер этого буфера для однопроцессорной машины составляет 16 Кбайт. Другими словами, в ядре одновременно может храниться до 16 Кбайт системных сообщений. Если размер очереди сообщений достигает установленного максимума, то при новом вызове функции `printk()` самое старое сообщение будет затерто новым сообщением. Буфер сообщений ядра называется *кольцевым* потому, что запись и считывание сообщений выполняется по круговому принципу.

Использование кольцевого буфера предоставляет определенные преимущества. Поскольку одновременные операции чтения и записи в кольцевом буфере выполняются достаточно просто, функцию `printk()` можно использовать даже из контекста прерывания. Более того, это позволяет организовать управление системными сообщениями достаточно просто. Если сообщений оказывается слишком много, то новые сообщения просто затирают старые. Если в системе возникает проблема, которая проявляется в генерации большого количества сообщений, то буфер будет все время перезаписываться новыми сообщениями, а не бесконтрольно расширяться, занимая всю свободную память. Единственный недостаток кольцевого буфера заключается в возможной потере устаревших сообщений. Но это не такая уж и большая плата за ту устойчивость, которую предоставляет такое решение.

Демоны `syslogd` и `klogd`

В стандартных системах Linux для извлечения системных сообщений из буфера сообщений ядра и перенаправления их в системный журнал через демон `syslogd` используется специальный демон `klogd`, запущенный в пространстве пользователя. Для чтения системных сообщений из буфера программа `klogd` может считывать данные из файла `/proc/kmsg` или использовать вызов системной функции `syslog()`. По умолчанию используется подход на основе файловой системы `/proc`. Если сообщений нет, то при выполнении операции чтения демон `klogd` переходит в состояние ожидания до поступления нового сообщения. Когда приходит новое сообщение, демон возвращается к вы-

полнению, считывает все сообщения и обрабатывает их. По умолчанию сообщения отправляются демону `syslogd`.

Демон `syslogd` добавляет полученные сообщения в конец файла журнала, стандартное имя которого `/var/log/messages`. Однако это имя можно изменить, отредактировав файл конфигурации `/etc/syslog.conf`.

Изменить уровень вывода сообщений на консоль (console loglevel) можно при запуске демона `klogd` с помощью флага `-c`.

Взаимозаменяемость функций `printf()` и `printk()`

Начинающие разработчики кода ядра часто по ошибке вместо функции `printk()` используют функцию `printf()`. Это вполне естественно, учитывая их многолетний опыт по написанию пользовательских приложений и использованию в них функции `printf()`. Однако по счастливой случайности вы не сможете допускать подобную ошибку слишком долго, поскольку вам начнут слишком быстро надоедать одни и те же ошибки компоновщика.

Сообщения `Oops`

Сообщения *oops* — обычный для ядра способ сообщить пользователю, что произошло что-то нехорошее. Поскольку ядро управляет всей системой, то оно не может само себя исправить или завершить свою работу, как это возможно в пользовательских приложениях, когда они делают что-то не так. Вместо этого ядро выводит сообщение `oops`, которое включает вывод информации об ошибке на консоль, вывод дампа содержимого всех регистров и обратной трассировки вызовов функций (back trace). Сбои в работе ядра очень трудно обработать. Поэтому ядро должно обойти все препятствия и постараться вывести сообщение `oops`, а также 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 находиться в процессе обработки важных данных. В этот момент может удерживаться блокировка или выполняться сеанс взаимодействия с оборудованием. При этом ядро должно аккуратно выйти из сложившегося текущего контекста 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 сможет продолжить работу и переходит в состояние паники. В результате состояния паники система полностью останавливается. Если сообщение `oops` возникает в холостом процессе (idle task, идентификатор `pid` равен нулю) или при выполнении процесса `init` (идентификатор `pid` равен единице), то ядро также переходит в состояние паники, потому что оно не может продолжать выполнение без этих важных процессов. Однако, если сообщение `oops` появляется при выполнении любого другого процесса, ядро завершает этот процесс и пытается продолжить свою работу.

Сообщение `oops` может выдаваться по многим причинам, включая нарушение прав доступа к памяти (memory access violation) и выполнение недопустимой машинной команды. Как разработчику ядра, вам придется иметь дело с сообщениями `oops` и даже, несомненно, быть причиной их появления.

Ниже приведен пример сообщения `oops` для машины аппаратной платформы PPC, которое появилось в 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: 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] tulip_timer+0x128/0x1c4
[c0020744] run_timer_softirq+0x10c/0x164
[c001b864] do_softirq+0x88/0x104
[c0007e80] timer_interrupt+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 (целых 32!). Сообщения `oops` для аппаратной платформы x86, с которыми вам, возможно, приходилось сталкиваться, имеют несколько более простой вид. Тем не менее важная информация будет идентична для всех аппаратных платформ: в ней приводится содержимое всех регистров общего назначения и обратная трассировка вызовов функций.

Обратная трассировка показывает последовательность вызовов функций, которая привела к проблеме. В данном случае можно точно определить, что же произошло. Компьютер находился в состоянии простоя, выполняя холостой процесс. Для этого была вызвана функция `cpu_idle()`, в которой в цикле вызывалась функция `default_idle()`. Далее поступило прерывание от системного таймера, в котором вызываются обработчики таймеров ядра. Среди них вызывается обработчик таймера сетевой платы — функция `tulip_timer()`, в которой почему-то выполнено обращение к памяти по указателю `NULL`. Можно даже воспользоваться значением смещения (шестнадцатеричные числа вроде `0x128/0x1c4`, которые указаны справа от имени функции) для точного нахождения команды, в которой возникла ошибка.

Содержимое регистров также может быть 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 источник проблемы. В данном случае можно увидеть, в каких регистрах содержится значение `NULL` (все разряды нулевые), и определить, какая из переменных функции содержит некорректное значение. В ситуациях, похожих на данную, источник проблемы, скорее всего, заключается в возникновении конфликта при доступе к ресурсу со стороны таймера и другой частью драйвера сетевого адаптера. Выявить состояние конфликта всегда не просто!

Утилита `ksymoops`

Выше мы рассмотрели сообщение `oops` в так называемом *декодированном* виде, поскольку вместо адресов памяти в нем приведены имена функций, которые им соответствуют. В не декодированном виде предыдущее сообщение выглядело бы так:

```
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]
```

Здесь адреса обратной трассировки должны быть переведены в символические имена функций. Это можно сделать с помощью утилиты `ksymoops` при наличии файла плана `System.map`, который был сгенерирован во время компиляции данного ядра. Если используются загружаемые модули ядра, то необходима также информация о плане модулей. Утилита `ksymoops` пытается самостоятельно определить всю необходимую информацию, поэтому обычно ее можно просто вызывать следующим образом:

```
ksymoops saved_oops.txt
```

При этом утилита выведет декодированную информацию о сообщении `oops`. Если информация, которая используется по умолчанию, недоступна или нужно указать альтернативное положение соответствующих информационных файлов, то для этой цели в программе предусмотрены различные параметры. Перед использованием утилиты `ksymoops` обязательно ознакомьтесь с ее справочной страницей, где вы найдете довольно много полезной информации. Утилита `ksymoops` включена в большинство поставок операционной системы Linux.

Функция `kallsyms`

К счастью, больше нет необходимости использовать утилиту `ksymoops`. Это большой прогресс, потому что, хотя у разработчиков обычно не было проблем с ее использованием, пользователи часто указывали неправильный файл `System.map` или не могли корректно декодировать сообщение `oops`.

При разработке ядра серии 2.5 была введена новая возможность `kallsyms`, которая активизируется с помощью параметра конфигурации `CONFIG_KALLSYMS`. При этом в исполняемый образ ядра включается информация о соответствии адресов памяти именам функций, что дает возможность ядру самостоятельно декодировать информацию обратной трассировки. Следовательно, для декодирования сообщений `oops` больше не требуется файл `System.map` или утилита `ksymoops`. Как недостаток такого подхода следует отметить некоторое увеличение объема памяти, используемой ядром, поскольку таблица соответствия адресов памяти именам функций загружается в постоянно отображаемую память ядра. Однако на такое увеличение объемов используемой памяти все же стоит пойти, по крайней мере, не только на этапе разработки ядра, но и развертывания. При активизации параметра `CONFIG_KALLSYMS_ALL` дополнительно в образ ядра включается также информация о расположении всех символических имен переменных, а не только имен функций. Как правило, она нужна только для специализированных отладчиков.

Активизация параметра `CONFIG_KALLSYMS_EXTRA_PASS` приводит к тому, что при построении образа ядра будет выполнен еще один проход по объектному коду ядра. Этот параметр полезен только в случае отладки самой системы `kallsyms`.

Параметры конфигурации для отладки ядра

Существует несколько параметров конфигурации, которые помогают в отладке и тестировании кода ядра и включаются во время компиляции. Эти параметры собраны в виде меню Kernel hacking редактора конфигурации ядра и зависят от параметра CONFIG_DEBUG_KERNEL. При разработке кода ядра следует включать только те параметры, которые необходимы.

Некоторые из этих параметров достаточно полезны, поскольку позволяют проводить отладку распределителя памяти блочного типа, контролировать распределение верхней памяти и отображение файлов в память, а также спин-блокировки и проверять, не переполнился ли стек. Однако одной из самых полезных возможностей является *проверка перехода системы в состояние ожидания при захваченной спин-блокировке*, которая на самом деле выполняет значительно больше работы.

Начиная с серии 2.5 в ядре появилась отличная инфраструктура для определения всех типов нарушения неделимости операций. Как уже говорилось в главе 9, “Общие сведения о синхронизации кода ядра”, под *неделимостью* (*atomic*) подразумевается возможность выполнения операции без ее прерывания (т.е. без деления на подоперации). При этом неделимый участок кода выполняется полностью и без прерывания либо не выполняется вовсе. Код, который выполняется после захвата спин-блокировки или при отключенном режиме мультипрограммирования ядра, является неделимым. Задача, в которой выполняется неделимый код, не должна переходить в состояние ожидания, поскольку это часто приводит к возникновению взаимоблокировок.

Поскольку в ядре используется режим мультипрограммирования, в него был введен глобальный счетчик неделимости. Ядро можно настроить так, что если выполняется переход в состояние ожидания или даже выполняется код, который *потенциально* может переходить в состояние ожидания при выполнении неделимой операции, то ядро выводит предупреждающее сообщение и обратную трассировку. В результате можно выявить потенциальные ошибки, связанные с вызовом функции schedule() при захваченной 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 довольно много ошибок, и ее настоятельно рекомендуется использовать.

Для полноценного использования данной возможности необходимо активизировать перечисленные ниже параметры конфигурации.

```
CONFIG_PREEMPT=y
CONFIG_DEBUG_KERNEL=y
CONFIG_KALLSYMS=y
CONFIG_DEBUG_SPINLOCK_SLEEP=y
```

Объявление об ошибках и выдача информации

В ядре предусмотрено несколько подпрограмм, позволяющих легко сигнализировать о наличии ошибок в коде, обеспечивать объявления об ошибках и выводить необходимую информацию. Чаще всего используются функции BUG() и BUG_ON(). При вызове эти функции генерируют ситуацию oops, в результате чего выводится обратная трассировка стека ядра и сообщение об ошибке. Конкретный способ генерации ситуации oops зависит от аппаратной платформы. На многих аппаратных платформах функции BUG()

и BUG_ON() определяются в виде некоторой недопустимой машинной команды, которая приводит к выводу желаемого сообщения oops. Обычно вызовы этих функций используются для объявления о наличии ошибки (assertion), чтобы сигнализировать о ситуации, которая не должна произойти.

```
if (bad_thing)
    BUG();
```

Или даже так:

```
BUG_ON(bad_thing);
```

Большинство разработчиков ядра считают, что функция BUG_ON() не загромождает код и лучше говорит сама за себя, чем функция BUG(). Кроме того, функция BUG_ON() помещает объявление об ошибке в оператор unlikely(). Следует заметить, что часть разработчиков обсуждала идею введения опции немедленной компиляции операторов BUG_ON(). Это означает, что все объявления, сделанные в функции BUG_ON(), не должны иметь никаких побочных эффектов. Для этих же целей служит и макрос BUILD_BUG_ON(), правда, он работает только на этапе компиляции. Если во время компиляции указанное в этом макросе выражение становится истинным, процесс компиляции завершается с ошибкой.

О более критичной ошибке можно сигнализировать с помощью функции panic(). Вызов этой функции выводит сообщение об ошибке и останавливает работу ядра. Очевидно, что эту функцию следует использовать только в самом худшем случае.

```
if (terrible_thing)
   panic("terrible_thing is %ld!\n", terrible_thing);
```

Иногда необходимо просто вывести на консоль трассировку стека, чтобы облегчить отладку. Для этого предназначена функция dump_stack(), позволяющая вывести на консоль содержимое регистров процессора и обратную трассировку вызовов функций.

```
if (!debug_check) {
   printk(KERN_DEBUG "provide some information...\n");
   dump_stack();
}
```

"Магическая" клавиша <SysRq>

Использование "магической" клавиши <SysRq>, которую можно активизировать с помощью параметра конфигурации CONFIG_MAGIC_SYSRQ на этапе компиляции ядра, часто позволяет значительно облегчить жизнь. Клавиша <SysRq> (System Request, или запрос к системе) имеется практически на всех стандартных клавиатурах. На аппаратных платформах i386 и PPC ей соответствует комбинация клавиш <ALT+PrintScreen>. Если указанный параметр конфигурации задан, то нажатие специальных комбинаций клавиш позволяет напрямую взаимодействовать с ядром, независимо от того, чем в данный момент занимается ядро. Это, в свою очередь, позволяет выполнять некоторые полезные операции даже на неработоспособной системе.

Кроме параметра конфигурации, описываемое средство можно включать и отключать, установив соответствующее значение переменной sysctl, как показано ниже. Для включения поддержки "магической" клавиши <SysRq> запустите приведенную ниже команду, а для отключения — вместо 1 используйте значение 0.

```
echo 1 > /proc/sys/kernel/sysrq
```

Список возможных комбинаций клавиш можно получить с консоли, нажав комбинацию клавиш <SysRq+h>. Для сброса не сохраненных буферов файловых систем на диск нажмите <SysRq+s>, для размонтирования всех файловых систем — <SysRq+u>, а для перезагрузки машины — <SysRq+b>. Последовательное использование этих трех комбинаций клавиш позволяет более безопасно перегрузить машину, которая зависла, чем простое нажатие кнопки сброса.

Если система заблокирована очень сильно, то она может не отвечать на нажатие "магических" комбинаций клавиш <SysRq>, либо запрошенная 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 справку 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

Более подробную информацию по данному вопросу вы сможете получить, ознакомившись с файлом `Documentation/sysrq.txt`, находящимся в дереве каталогов исходных кодов ядра. 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. Поскольку он предоставляет большие возможности для любого пользователя при работе с консолью, следует использовать его с осторожностью на особо ответственных компьютерах. Если же машина используется для разработок, то польза от этих команд огромная.

Сага об отладчике ядра

Многие разработчики уже давно высказывали мнение о том, что в ядро должен быть встроен отладчик. К сожалению, Линус не желает видеть отладчик ядра в своем дереве исходного кода. Он уверен, что использование программ-отладчиков приводит к плохому исправлению ошибок введенными в заблуждение 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, взамен было разработано несколько заплат, которые добавляют поддержку отладчика в стандартном ядре. Несмотря на то что это внешние и неофициальные заплаты, они являются мощными инструментами с высокими функциональными возможностями. Перед тем как обращаться к этим решениям, посмотрим, насколько нам может помочь стандартный отладчик ОС Linux — gdb.

Отладчик gdb

Для того чтобы мельком заглянуть в работающее ядро, можно использовать стандартный отладчик GNU. Запуск отладчика для работы с ядром почти ничем не отличается от отладки запущенного процесса.

```
gdb vmlinux /proc/kcore
```

Файл vmlinux — это разархивированный исполняемый образ ядра, который хранится в корне каталога исходных кодов, где выполнялась сборка. Сжатые файлы zImage и bzImage использовать нельзя.

Необязательный параметр /proc/kcore играет роль файла core и позволяет отладчику "заглянуть" в память выполняющегося ядра. Чтобы иметь возможность прочитать этот файл, необходимо иметь права пользователя root.

Для чтения информации можно пользоваться практически всеми командами программы gdb. Например, чтобы вывести значение переменной, можно воспользоваться командой

```
p global_variable
```

Для того чтобы дизассемблировать код функции, выполните следующую команду:

```
disassemble function
```

Если при компиляции ядра был указан флаг -g (его необходимо добавить к значению переменной CFLAGS в файле Makefile построения ядра), то отладчик gdb сможет вывести намного больше информации. Например, можно выводить дампы структур данных и переходить по значению указателя. В результате размер файла ядра значительно возрастает, 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 изменить данные ядра. Нет возможности пошагово выполнять код ядра или устанавливать точки останова (breakpoint). То, что с помощью gdb нельзя изменять структуры данных ядра, является большим недостатком. 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 выполнять отладку ядра на удаленном терминале, подключенном к последовательному порту компьютера. Для этого потребуются два компьютера. На первом запускается ядро с заплатой kgdb. На втором, подключенном к последовательному порту первого через нуль-модемный кабель, запускается программа Терминал, в окне которой работает отладчик gdb. Благодаря заплате kgdb становится полностью доступен весь набор средств

отладчика gdb: чтение и запись любых переменных, установка точек останова, установка точек слежения (watch points), пошаговое исполнение и т.д.! Специальные версии kgdb даже позволяют вызывать функции.

У вас может вызвать определенное затруднение настройка kgdb для работы по последовательному каналу связи, но если ее выполнить, то отладка ядра значительно упрощается. При установке заплаты ядра также записывается большое количество документации в каталог Documentation/. Прочитайте ее!

Поддержка заплаты kgdb для различных аппаратных платформ и версий ядра выполняется разными людьми. Поиск в Интернете — наилучший способ найти необходимую заплату для вашей версии ядра.

Исследование и тестирование системы

По мере приобретения опыта в отладке ядра у вас будет появляться все больше маленьких хитростей, которые помогают в исследовании и тестировании ядра для получения ответов на интересующие вопросы. Поскольку отладка ядра требует больших усилий, то каждый маленький совет или хитрость могут оказаться полезными. Рассмотрим несколько таких хитростей.

Использование идентификатора UID в качестве условия

Если разрабатываемый код связан с контекстом процесса, то иногда требуется выполнить его альтернативную реализацию, не "ломая" существующий код. Это важно, если требуется переписать код важной системной функции и при этом необходима полностью функционирующая система, на которой эту функцию нужно отладить.

В качестве примера предположим, что нам нужно изменить алгоритм работы системной функции fork(), чтобы она могла поддерживать новую замечательную возможность, только что реализованную в ядре. Если сразу не получится все сделать так, как нужно, то будет очень тяжело отлаживать ядро, поскольку неработающая системная функция fork() скорее всего приведет к неработоспособности системы. Но, как и всегда, надежда умирает последней.

Часто проще и безопаснее всего будет сохранить старый алгоритм, а его новую реализацию выполнить отдельно. Это можно сделать, если задействовать идентификатор пользователя (UID) в качестве условия, какой алгоритм следует использовать.

```
if (current->uid != 7777) {
   /* Старый алгоритм.. */
} else {
   /* Новый алгоритм.. */
}
```

При этом все пользователи, кроме того, у которого идентификатор UID равен 7777, будут использовать старый алгоритм. Для тестирования нового алгоритма можно создать новую учетную запись пользователя с идентификатором 7777. Это позволяет более просто протестировать критические участки кода, связанные с запуском 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 иден-

тификатора пользователя. Просто создайте в коде глобальную переменную и используйте ее значение при проверке 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 глобальной переменной может быть изменено с помощью отладчика или специального экспортируемого интерфейса.

Использование статистики

Иногда необходимо получить представление о том, насколько часто происходит некоторое событие, или сравнить несколько событий и вычислить характеристики для их сравнения. Это очень легко сделать путем создания механизма статистки и экспортирования значений соответствующих параметров.

В качестве примера предположим, что необходимо выяснить, как часто происходит событие `foo` и событие `bar`. В файле исходного кода, в идеале там, где возникают соответствующие события, нужно ввести две глобальные переменные.

```
unsigned long foo_stat = 0;
unsigned long bar_stat = 0;
```

Как только наступает интересующее нас событие, значение соответствующей переменной увеличивается на единицу. Значения этих переменных могут быть экспортированы как угодно. Например, можно создать интерфейс к ним через файловую систему `/proc` или написать свою системную функцию. Однако проще всего прочитать их значения с помощью отладчика.

Следует обратить внимание на то, что такой подход чрезвычайно не устойчив при использовании его на SMP-машине. В идеале нужно использовать неделимые переменные. Однако для ведения временной статистики, которая необходима только для отладки, никакой защиты обычно не требуется.

Ограничение частоты следования и общего количества событий при отладке

Зачастую с целью отладки необходимо встроить в код ряд проверок (с соответствующими операторами вывода информации), чтобы можно было визуально обнаружить проблему. Однако в ядре некоторые функции вызываются много раз в секунду. Если в такую функцию будет помещен вызов функции `printk()`, то системная консоль будет перегружена выводом отладочных сообщений и ее невозможно станет использовать.

Для предотвращения такой проблемы существуют два сравнительно простых приема. Первый — *ограничение частоты следования событий* (rate limiting) — очень полезен, когда необходимо наблюдать, как развивается событие, но частота возникновения события очень велика. Чтобы ограничить поток отладочных сообщений, эти сообщения выводятся только раз в несколько секунд, как 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 данном примере отладочные сообщения выводятся не чаще одного раза в две секунды. Это предотвращает перегрузку консоли сообщениями, и системой можно будет

нормально пользоваться. Частота вывода может быть большей или меньшей в зависимости от ваших потребностей.

Если вы используете *только* функцию `printk()`, то для ограничения частоты вывода отладочных сообщений можно использовать специальную функцию, как показано ниже.

```
if (error && printk_ratelimit())
   printk(KERN_DEBUG "error=%d\n", error);
```

Функция `printk_ratelimit()` возвращает значение `0`, если установленный интервал времени между сообщениями еще не истек, и ненулевое значение в противном случае. По умолчанию эта функция позволяет выводить только одно сообщение не чаще, чем раз в 5 секунд. Кроме того, она разрешает выводить пакетом не более десяти первых сообщений. Указанные параметры настройки функции можно изменить, воспользовавшись `sysctl`-переменными `printk_ratelimit` и `printk_ratelimit_burst` соответственно.

Еще одна неприятная ситуация возникает в случае, если вам нужно определить, правильно ли выполняется ваша программа (т.е. сколько раз управление попадает в некоторую ветку кода). В отличие от предыдущего примера, здесь не нужно выводить сообщения в реальном масштабе времени. Такая неприятная ситуация может возникнуть в случае, если какой-то участок кода выполняется многократно, тогда как вы точно уверены, что управление в него должно попадать всего один или несколько раз. В данном случае нужно ограничивать не частоту, а *общее количество повторений*, как показано ниже.

```
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 они используются. Тогда после завершения работы функции значения статических переменных будут сохраняться до следующего вызова функции.

Ни один из приведенных выше примеров не является устойчивым ни к многопроцессорной обработке, ни к мультипрограммному режиму работы ядра. Данную проблему можно решить достаточно просто, перейдя к использованию неделимых переменных. Однако стоит ли это делать во временном отладочном коде, решать вам.

Поиск методом половинного деления изменений, приводящим к ошибкам

Обычно полезно знать, в какой версии исходных кодов ядра появилась ошибка. Если известно, что ошибка появилась в версии 2.6.33, но ее не было в версии 2.6.29, то сразу возникает ясная картина изменений, которые привели к появлению ошибки. Устранение ошибки сводится к выполнению обратных изменений или внесению исправлений в измененный код.

Однако во многих случаях точно нельзя сказать, в какой из версий ядра появилась ошибка. Известно только, что проблема проявилась в *текущей* версии ядра, и всегда каза-

лось, что она была только 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 рукой подать.

Нельзя начинать поиск, если ошибка стабильно не проявляется. Желательно, чтобы она проявлялась сразу же после загрузки системы. Далее вам понадобится гарантированно работающее ядро. Вероятно, оно вам должно быть 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. Скорее всего, найти ядро без ошибки будет не сложно, если, конечно, она не существовала всегда.

Далее, необходимо ядро, в котором гарантированно есть ошибка. Чтобы облегчить поиск, воспользуйтесь самой ранней версией ядра, в которой была ошибка.

Теперь можно начинать поиск методом половинного деления версии ядра, в которой впервые появилась ошибка. Этот поиск нужно вести в направлении от гарантированно дефектной версии к гарантированно хорошей. Рассмотрим пример. Предположим, последнее ядро, в котором не было ошибки, — 2.6.11, а последнее с ошибкой — 2.6.20. Сначала выбираем версию ядра, которая находится где-то посередине, — скажем, 2.6.15. Проверяем версию 2.6.15 на наличие ошибки. Если версия 2.6.15 работает, значит, ошибка появилась в более поздней версии. Тогда нужно попробовать протестировать версию между 2.6.15 и 2.6.20, скажем, версию 2.6.17. С другой стороны, если версия 2.6.15 не работает, то тогда ясно, что ошибка появилась в более ранней версии, и следует попробовать, к примеру, версию 2.6.13. И так далее.

В конечном итоге проблема сужается до двух последовательных версий ядер — одно с дефектом, другое — без. В таком случае появляется ясная картина изменений, которые привели к проблеме. Описанный выше метод поиска подходит для любой версии ядра!

Поиск с помощью `git`

Полезный механизм для выполнения поиска методом половинного деления реализован в утилите управления исходными кодами `git`. Если вы используете `git` для управления своей копией дерева исходных кодов ядра Linux, то процесс поиска методом половинного деления можно автоматизировать. Более того, в `git` такой поиск выполняется на уровне *номера редакции* и позволяет точно определить дату и время операции фиксации в хранилище, которая привела к проблеме. В отличие от многих других задач, выполняемых с помощью `git`, операция поиска методом половинного деления в нем выполняется довольно просто. Прежде всего нужно сообщить `git` о начале операции поиска, как показано ниже.

```
$ git bisect start
```

После этого следует указать программе `git` самый ранний номер редакции, в которой обнаружена проблема.

```
$ git bisect bad <редакция>
```

Если проблема обнаружена в самой последней версии ядра, то номер редакции указывать не нужно.

```
$ git bisect bad
```

Затем нужно указать программе `git` самый последний номер редакции, в которой проблемы еще не было.

```
$ git bisect good v2.6.28
```

После этого git автоматически извлекает из хранилища исходных кодов редакцию ядра Linux, номер которой расположен посередине между указанными вами номерами плохой и хорошей редакции. Далее вы должны скомпилировать, запустить и протестировать эту редакцию ядра. Если она работает нормально, введите приведенную ниже команду.

```
$ git bisect good
```

Если в этой редакции ядра есть ошибка и она не работает как нужно, введите команду

```
$ git bisect bad
```

После ввода любой из приведенных выше двух команд git снова извлечет следующую доступную редакцию ядра, номер которой определяется по методу половинного деления. Описанный выше процесс повторяется до тех пор, пока не будут перебраны все возможные редакции ядра. В завершение git сообщает номер редакции ядра, в которой появилась проблема.

Процесс поиска ошибки довольно длительный, хотя git максимально облегчает его. Если вам кажется, что источник ошибки известен, например, ошибка может вкрасться в загрузочный код, специфичный только для платформы x86, сообщите об этом git. Тогда эта программа будет извлекать код из хранилища методом половинного деления только для указанного списка каталогов, как показано ниже.

```
$ git bisect start - arch/x86
```

Если ничто не помогает — обратитесь к сообществу

Возможно, вы уже испробовали все известные вам методы. Вы просидели за клавиатурой несчетное количество часов и даже дней, а решение все еще не найдено. Если проблема находится в основном ядре Linux, то всегда можно обратиться за помощью к сообществу разработчиков ядра.

Короткое, но достаточно детальное описание проблемы вместе с вашими находками, посланное в список рассылки разработчиков ядра по электронной почте, поможет отыскать решение. В конце концов, ошибок никто не любит.

В главе 20, “Заплаты, хакерство и сообщество”, мы рассмотрим способы обращения к сообществу разработчиков и на их основной форум — список рассылки разработчиков ядра Linux (Linux Kernel Mailing List, LKML).

Резюме

В этой главе были рассмотрены способы отладки ядра Linux — процесса, позволяющего определить, *почему* всегда наши намерения расходятся с их реализацией. Мы ознакомились с несколькими методиками, включая встроенную в ядро инфраструктуру поддержки отладчиков, отладочный вывод сообщений и поиск методом половинного деления с помощью git. Поскольку отлаживать ядро Linux может быть значительно труднее, чем пользовательское приложение, материал данной главы является ключевым для всех, кто собирается писать реальный код ядра.

В следующей главе будет рассмотрена еще одна важная тема, посвященная переносимости ядра Linux. Так что, вперед!

19
Переносимость

Linux является *переносимой* операционной системой, которая может работать на большом количестве различных компьютерных аппаратных платформ. Под *переносимостью* будем понимать свойство кода, которое указывает, насколько легко (если вообще это возможно) можно перенести программный код с одной аппаратной платформы на другую. Известно, что ОС Linux является переносимой операционной системой, поскольку ее уже *перенесли* (портировали) на большое количество различных аппаратных платформ. Тем не менее переносимость не возникает вдруг сама по себе. Для ее реализации от программиста требуется большое усердие и постоянный контроль над создаваемым переносимым кодом. Поэтому 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.

Переносимые операционные системы

Ряд операционных систем специально разрабатывали с учетом требований переносимости как их главной отличительной черты. Количество машинно-зависимого кода в них сведено к минимуму (насколько это возможно). При разработке таких систем стараются не использовать низкоуровневый язык программирования ассемблер, а интерфейсы и возможности ОС выполняются принципиально общими и абстрактными, чтобы их легко можно было реализовать 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 портированы 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 вручную оптимизирован под конкретную машину. В данном случае переносимость ставится выше оптимальности. Примером операционных систем с высокой переносимостью могут быть Minix, OpenBSD и многие другие исследовательские системы.

Существует и противоположный подход к проектированию операционных систем, когда оптимизация кода выполняется в ущерб переносимости. Код, по возможности, пишется на ассемблере или любом другом языке, специально предназначенном для конкретных аппаратных платформ. В таких системах возможности ядра разрабатываются с учетом свойств аппаратной платформы. Поэтому перенос такой операционной системы на другую аппаратную платформу по трудозатратам эквивалентен написанию ядра с нуля. И даже если такой перенос технически осуществим, портированная операционная система может плохо работать на новой аппаратной платформе. При подобном подходе к проектированию переносимость приносится в жертву оптимальности кода. Часто в подобных системах поддержка кода требует бо́льших трудозатрат, чем в переносимых ОС. Разумеется, целесообразность создания таких систем по сравнению с переносимыми ОС не может быть высокой. Тем не менее используемый в них бескомпромиссный подход к проектированию, связанный с отказом от переносимости, позволяет создавать довольно эффективные системы, работающие на весьма слабом оборудовании. В качестве примера подобных системы можно привести Microsoft DOS и Windows 95, которые могли работать на компьютерах с тактовой частотой 100 Мгц, оснащенных всего 4 Мбайт ОЗУ.

Операционная система Linux в плане переносимости занимает промежуточное положение. Исходя из практической целесообразности, интерфейсы и код сохраняются независимыми от аппаратной платформы и пишутся на языке C. Однако функции ядра, которые критичны к производительности, оптимизируются под конкретную аппаратную платформу. Например, низкоуровневый код и код, который должен выполняться очень быстро, разрабатываются зависимыми от аппаратной платформы и обычно пишутся на языке ассемблера. Такой подход позволяет сохранить переносимость ОС Linux и при этом воспользоваться преимуществами оптимизации под конкретную аппаратную платформу. В тех случаях, когда переносимость влияет на производительность, преимущество всегда отдается последней. В остальных случаях сохраняется переносимость кода.

Обычно экспортируемые интерфейсы ядра независимы от аппаратной платформы. Если какая-либо из частей одной подпрограммы должна зависеть от используемой аппаратной платформы (из соображений производительности или по необходимости), то код выполняется в виде нескольких функций, которые вызываются, в нужных местах. При этом для каждой поддерживаемой аппаратной платформы реализуются свои функции, которые затем компонуются в общий исполняемый образ ядра.

Хорошим примером здесь может служить системный планировщик. Бо́льшая часть планировщика является независимой от аппаратной платформы, написана на языке C и находится в файле `kernel/sched.c`. Часть функций планировщика, такие как сохранение и восстановление состояния процессора или переключение адресного пространства, зависит от аппаратной платформы. Поэтому в функции `context_switch()`, которая выполняет переключение процессов, вызываются два метода, `switch_to()` и `switch_mm()`, для переключения состояния процессора и адресного пространства соответственно.

Код функций `switch_to()` и `switch_mm()` реализуется отдельно для каждой поддерживаемой аппаратной платформы. Когда операционная система Linux портируется на новую аппаратную платформу, для последней нужно реализовать эти функции.

Аппаратно-зависимые файлы хранятся в каталоге `arch/`*`architecture`*`/`, где вместо *`architecture`* нужно подставить сокращенное название аппаратной платформы, принятой в Linux. Например, аппаратной платформе Intel x86 присвоено короткое имя `x86`. Она поддерживает как 32-, так и 64-разрядный режимы работы — x86-32 и x86-64 соответственно. Аппаратно-зависимые файлы для этого типа платформ хранятся в ката-

логе arch/x86. В ядрах серии 2.6 поддерживаются следующие типы аппаратных платформ: alpha, arm, avr32, blackfin, cris, frv, h8300, ia64, m32r, m68k, m68knommu, mips, mn10300, parisc, powerpc, s390, sh, sparc, um, x86 и xtensa. Их более полное описание приведено ниже, в табл. 19.1.

История переносимости Linux

Когда Линус Торвальдс впервые выпустил операционную систему Linux в еще ничего не подозревающий мир, эта ОС работала только на компьютерах Intel i386. Хотя данная операционная система и была в значительной степени обобщена и хорошо написана, переносимость для нее не была основным требованием. Однажды Линус даже сказал, что операционная система Linux никогда не будет работать ни на какой другой аппаратной платформе, кроме i386! Тем не менее в 1993 году началась работа по портированию ОС Linux на компьютеры Digital Alpha. На то время Digital Alpha была новой высокопроизводительной RISC-платформой с поддержкой 64-разрядной адресации памяти. Она очень сильно отличалась от аппаратной платформы i386, о которой говорил Линус. Однако первоначальный перенос на аппаратную платформу Alpha занял около года, и она стала первой официально поддерживаемой аппаратной платформой после x86. Этот процесс портирования был, наверное, самым сложным, потому что был первым. Вместо простого переписывания ядра для поддержки новой аппаратной платформы части ядра были переписаны с учетом возможной переносимости[1]. Хотя при этом и был выполнен большой объем работы, в результате получился более ясный для понимания код, и в будущем перенос стало выполнять много проще.

Хотя в первых выпусках ОС Linux поддерживалась только платформа i386, в серии ядер 1.2 уже была введена поддержка платформ Digital Alpha, MIPS и SPARC, хотя она и была отчасти экспериментальной.

В ядро версии 2.0 была добавлена официальная поддержка платформ Motorola 68k и PowerPC. Кроме того, поддержка объявленных ранее в ядрах серии 1.2 аппаратных платформ стала официальной и стабильной.

В серию ядер 2.2 была введена поддержка еще большего количества аппаратных платформ: добавлены ARM, IBM S/390 и UltraSPARC. А всего через несколько лет в серии ядер 2.4 количество поддерживаемых аппаратных платформ было почти удвоено и доведено до 15! Была добавлена поддержка платформ CRIS, IA-64, 64-разрядная MIPS, HP PA-RISC, 64-разрядная IBM S/390 и Hitachi SH.

В текущей серии 2.6 количество поддерживаемых аппаратных платформ было доведено до 21 за счет добавления платформ AVR, FR-V, Motorola 68k без блока MMU, M32xxx, H8/300, IBM POWER, Xtensa, а также версии ядра, которое работает на виртуальной машине под управлением ОС Linux, называемой Usermode Linux. Поддержка 64-разрядной s390 была объединена с 32-разрядной платформой s390, чтобы избежать дублирования.

Необходимо заметить, что каждая из упомянутых выше аппаратных платформ поддерживает различные типы компьютеров и микросхем системной логики. Некоторые из поддерживаемых аппаратных платформ, такие как ARM и PowerPC, поддерживают очень большое количество типов микросхем и компьютеров. Другие же платформы, та-

[1] Это обычная 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.

кие как x86 и SPARC, поддерживают только 32- и 64-разрядные режимы работы процессоров. Поэтому, хотя ОС Linux и поддерживает 21 тип популярных аппаратных платформ, она может работать 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. В качестве других количественных характеристик данных здесь уместно применить аналогию бумажного документа, состоящего из *символов* (*character*), обычно размером 8 бит, и *страниц* (pages) размером 4 или 8 Кбайт, содержащих много слов. Слово состоит из определенного количества байтов, например, одного, двух, четырех или восьми. Когда говорят о n-разрядной машине, то чаще всего имеют в виду размер машинного *слова*. Например, когда говорят, что процессор Intel Core i7 является 64-разрядным, то обычно имеют в виду размер машинного слова, равный 64 битам, или 8 байтам.

Размер регистров общего назначения процессора соответствует размеру машинного слова этого процессора. Обычно разрядность остальных компонентов этой же аппаратной платформы, например шины памяти, в точности равна размеру машинного слова. Кроме того, по крайней мере, для аппаратных платформ, которые поддерживаются в ОС Linux, размер виртуального адресного пространства соответствует размеру машинного слова, хотя размер адресуемой физической памяти при этом гораздо меньше[2]. Следовательно, размер указателя равен размеру машинного слова. В дополнение к этому размер типа `long` языка C также равен размеру машинного слова, тогда как размер типа `int` иногда может быть меньше размера машинного слова. Например, для аппаратной платформы Alpha размер машинного слова равен 64 битам. Следовательно, регистры, указатели и тип `long` имеют размер 64 бита. Однако тип `int` для этой платформы имеет размер 32 бита. Компьютеры платформы Alpha могут с помощью одной операции обрабатывать 64 бита данных, т.е. одно машинное слово.

> **Слова, двойные слова и путаница в терминологии**
>
> В некоторых операционных системах и процессорах стандартную порцию данных не называют машинным *словом*. Вместо этого *словом* называется 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 учетверенные слова (quad word — 64 бит), несмотря на то, что на самом деле система является 32-разрядной. Подобная система наименования была принята в системах на основе Windows NT, а также в современной Windows 7. В этой книге и вообще в контексте операционной системы Linux под машинным словом понимают стандартную порцию данных, которую может обработать процессор (см. выше).

Для каждой аппаратной платформы, поддерживаемой операционной системой Linux, в файле `<asm/types.h>` определяется константа `BITS_PER_LONG`, которая равна размеру типа `long` языка C и совпадает с размером машинного слова системы. Полный

[2] Например, для 64-разрядных аппаратных платформ размер указателя равен 64 битам, однако только 48 бит можно использовать для адресации физической памяти. В дополнение к этому общее количество физической памяти может быть больше максимального значения машинного слова, как, например, при использовании расширения Intel PAE.

список всех поддерживаемых аппаратных платформ и размеры их машинного слова приведены в табл. 19.1.

Таблица 19.1. Поддерживаемые аппаратные платформы

Аппаратная платформа	Описание	Размер слова, биты
alpha	Digital Alpha	64
arm	ARM и StrongARM	32
avr	AVR	32
blackfin	Blackfin	32
cris	CRIS	32
frv	FR-V	32
h8300	H8/300	32
ia64	IA-64	64
m32r	M32xxx	32
m68k	Motorola 68k	32
m86knommu	Motorola 68k без устройства MMU	32
mips	MIPS	32 и 64
parisc	HP PA-RISC	32 и 64
powerpc	PowerPC	32 и 64
s390	IBM S/390	32 и 64
Sh	Hitachi SH	32
Sparc	SPARC	32 и 64
Um	Usermode Linux	32 и 64
x86	x86-32 и x86-64	32 и 64
xtensa	Xtensa	32

Традиционно 32- и 64-разрядные варианты ОС Linux, работающие на одной и той же аппаратной платформе, были реализованы независимо друг от друга. Например, в ранних выпусках ядер серии 2.6 была предусмотрена поддержка обеих архитектур, `i386` и `x86-64`, `mips` и `mips64`, `ppc` и `ppc64`. Благодаря проведенной работе, которая была успешно завершена, удалось унифицировать большинство аппаратных платформ и поместить их код в один подкаталог каталога `arch/`. В результате стало возможным поддерживать 32- и 64-разрядные архитектуры с помощью одного дерева исходных кодов.

В стандарте языка С явно указывается, что размер памяти, которую занимают переменные стандартных типов данных, зависит от используемой реализации компилятора[3]. Неопределенность размеров стандартных типов языка С для различных реализаций и аппаратных платформ имеет свои положительные и отрицательные стороны. К плюсам можно отнести то, что для стандартных типов языка С можно воспользоваться преимуществами, связанными с размером машинного слова текущей аппаратной платформы, а также то, что не нужно явно указывать размер типа. При этом размер типа `long` языка С гарантированно будет совпадать с размером машинного слова. Как недостаток можно

[3] За исключением размера типа `char`, который всегда равен 8 бит.

отметить, что при разработке кода нельзя рассчитывать на то, что данные определенного типа будут занимать в памяти определенный размер. Более того, нельзя гарантировать, что переменные типа `int` будут занимать столько же памяти, сколько и переменные типа `long`[4].

Ситуация еще больше усложняется и тем, что одни и те же типы данных в пространстве пользователя и в пространстве ядра не обязательно должны соответствовать друг другу. Так, на аппаратной платформе `sparc64` реализовано 32-разрядное пространство пользователя. Поэтому указатели, типы `int` и `long` в нем имеют размер 32 бита. Однако в пространстве ядра для этой же аппаратной платформы размер типа `int` равен 32 битам, а размер указателей и типа `long` — 64 битам. Хотя стоит отметить, что такая ситуация не является нормой.

Всегда необходимо помнить о следующем.

- Как того требует стандарт языка C, размер типа `char` всегда равен 1 байту (8 бит).
- Нет никакой гарантии, что размер типа `int` для всех поддерживаемых аппаратных платформ будет равен 32 битам, хотя сейчас в ОС Linux для всех платформ он равен именно этому числу.
- То же касается и типа `short`, который для всех поддерживаемых аппаратных платформ сейчас составляет 16 бит.
- Никогда нельзя надеяться, что тип `long` или указатель имеет некоторый заданный размер. Этот размер для поддерживаемых аппаратных платформ может быть равен как 32, так и 64 битам.
- Так как размер типа `long` разный для различных аппаратных платформ, никогда нельзя предполагать, что `sizeof(int)` равно `sizeof(long)`.
- Точно так же нельзя предполагать, что размер типа `int` и размер указателя совпадают.

Для описания размеров типов, которые используются в конкретной версии операционной системы, используются простые мнемонические обозначения, соответствующие принятой модели данных компилятора. Например, 64-разрядные версии Windows обозначаются как *LLP64*. Это означает, что типы `int` и `long` имеют размер 32 бита, а тип `long long` и указатели — 64 бита. Для 64-разрядных систем Linux используется обозначение *LP64*, означающее, что тип `long` и указатели имеют размер 64 бита. Для 32-разрядных систем Linux используется обозначение *ILP32*, означающее, что типы `int` и `long`, а также указатели имеют размер 32 бита. Эти мнемонические обозначения пригодятся для краткой характеристики операционной системы, когда нужно подчеркнуть, какой размер машинного слова в ней используется. Дело в том, что данная характеристика влияет на коммерческий успех системы.

Рассмотрим подробнее обозначения ILP64, LP64 и LLP64. В системах ILP64 типы данных `int`, `long` и указатели имеют размер 64 бита. Этот факт существенно облегчает написание программ, поскольку все основные типы языка C имеют одинаковый размер. Дело в том, что из-за несоответствия размеров указателей и целых чисел типа `int` часто возникают ошибки в программе. Однако недостаток такого подхода заключается в том,

[4] На самом деле для 64-разрядных аппаратных платформ, которые поддерживаются в ОС Linux, размеры типов `int` и `long` не совпадают. Размер типа `int` равен 32 битам, а размер типа `long` — 64. Для знакомых 32-разрядных аппаратных платформ оба типа данных имеют размер 32 бита.

что размер чисел целого типа часто оказывается значительно бо́льшим, чем требуется. В системах LP64 программисты могут использовать числа целого типа, имеющие разный размер. Однако при этом следует помнить, что размер типа `int` будет всегда меньше, чем размер указателя. При использовании систем LLP64 программисты сталкиваются с тем, что типы `int` и `long` имеют одинаковый размер. При этом они также должны учитывать различия в размерах между целым числом и указателем. По этой причине большинство программистов предпочитают использовать модель LP64, которая и принята в Linux.

Скрытые типы данных

Скрытые (opaque) типы данных — это те типы, для которых не раскрывается их внутренняя структура, или формат. Они очень похожи на *черный ящик*, насколько это можно реализовать в языке программирования С. В данном языке программирования нет какой-либо особенной поддержки для этих типов. Поэтому разработчики определяют новый тип данных с помощью оператора `typedef`, называют его скрытым и надеются на то, что никто не будет преобразовывать этот тип в стандартный тип данных языка С. Любое использование этих типов данных возможно только через специальные интерфейсы, которые также создаются разработчиком. В качестве примера можно привести тип данных `pid_t`, в котором хранится информация об идентификаторе процесса. Размер этого типа данных не раскрывается, хотя каждый может исхитриться, используя максимальный размер, и работать с этим типом, как с типом `int`. Если в коде нигде явно не используется размер скрытого типа данных, то его всегда можно изменить, 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 `short`.

Еще один пример скрытого типа данных — `atomic_t`. Как уже обсуждалось в главе 10, “Средства синхронизации ядра“, в этом типе хранятся целочисленные значения, с которыми можно выполнять неделимые операции. Хотя этот тип и соответствует типу `int`, использование скрытого типа данных позволяет гарантировать, что данные этого типа будут использоваться только в специальных функциях, в которых выполняются неделимые операции. Скрытые типы позволяют замаскировать реально используемый размер данных, который не всегда равен полным 32 разрядам, как в случае 32-разрядной 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. На некоторых аппаратных платформах он может занимать 32-бита, а на других — 64 бита. Более того, разработчики ядра могут в любой момент изменить этот размер.
- Нельзя преобразовывать скрытый тип обратно в стандартный тип данных языка С.
- Всегда будьте скептиком. Разрабатывайте код с учетом того, что размер и внутреннее представление скрытого типа данных могут изменяться.

Специальные типы данных

Часть данных в ядре, кроме того, что они представляются с помощью скрытых типов, требуют еще и специальных типов данных. Одним из примеров является параметр `flags`, задействованный при обработке прерываний, и для представления которого всегда должен использоваться тип `unsigned long`.

При хранении и использовании специфических данных всегда необходимо обращать особое внимание на тот тип данных, который представляет эти данные, и использовать именно его. Часто встречающейся ошибкой является использование другого типа, например `unsigned int`. Хотя для 32-разрядных аппаратных платформ это не приведет к ошибке, на 64-разрядных системах возникнут проблемы.

Типы с явным указанием размера

При написании программ часто приходится иметь дело с данными, размер которых задается явно. Обычно это необходимо для удовлетворения некоторых внешних требований, связанных с аппаратным обеспечением, сетью или совместимостью на уровне бинарных файлов. Например, звуковой адаптер может иметь 32-разрядный регистр, пакет сетевого протокола — 16-разрядное поле данных, а исполняемый файл — 8-битовый идентификатор объекта. В этих случаях тип, с помощью которого представляются 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	Тип	Описание
`s8`	Байт со знаком	`s32`	32-разрядное целое число со знаком
`u8`	Байт без знака	`u32`	32-разрядное целое число без знака
`s16`	16-разрядное целое число со знаком	`s64`	64-разрядное целое число со знаком
`u16`	16-разрядное целое число без знака	`u64`	64-разрядное целое число без знака

Типы данных со знаком используются редко.

Описанные выше типы данных с явно заданным размером определены с помощью оператора `typedef` через стандартные типы данных языка C. 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 так, 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;
```

Приведенные выше типы данных можно использовать только внутри ядра, т.е. в коде, который никогда не будет задействован в пользовательских приложениях (например, в струк-

туре данных, находящейся в заголовочном файле, недоступном для пользователя). Так сделано с целью сохранения пространств имен. В ядре также определены варианты этих типов данных, доступных для пользовательских приложений. Они имеют аналогичные названия, перед которыми находятся два символа подчеркивания. Например, 32-разрядное беззнаковое целое число, которое можно безопасно использовать в пользовательских приложениях, имеет тип `__u32`. Этот тип совпадает с типом `u32`, разница заключается только в названии. Внутри ядра можно использовать любое из названий. Однако, если некоторый тип должен применяться в пользовательских приложениях, следует использовать вариант имени с двумя символами подчеркивания, чтобы не вносить путаницу в пользовательское пространство имен.

Знаковые и беззнаковые типы `char`

В стандарте языка С сказано, что тип данных `char` может быть как со знаком, так и без знака. Какой вариант типа данных `char` будет использоваться по умолчанию, зависит от реализации компилятора, типа процессора или от обоих сразу.

Для большинства аппаратных платформ тип `char` является знаковым, а диапазон значений для данных этого типа составляет от –128 до +127. Однако на некоторых платформах, например на ARM, тип `char` по умолчанию является беззнаковым, а диапазон значений для данных этого типа составляет от 0 до 255.

Например, в системах, где тип `char` по умолчанию является беззнаковым, выполнение приведенного ниже кода приведет к записи в переменную `i` числа 255 вместо –1.

```
char i = -1;
```

На других компьютерах, где тип `char` по умолчанию является знаковым, этот код выполнится правильно и в переменную `i` запишется значение –1. Поэтому если программисту действительно нужно, чтобы в переменную в любом случае было записано значение –1, то предыдущий код должен выглядеть следующим образом:

```
signed char i = -1;
```

А если вы хотите записать в переменную число 255, то код должен выглядеть так:

```
unsigned char = 255;
```

При использовании в коде типа `char` не забывайте, что он может быть как со знаком (`signed char`), так и без знака (`unsigned char`). Если же необходим строго определенный вариант, то это нужно явно декларировать.

Выравнивание данных

Под *выравниванием* (aligned) данных будем понимать их способ размещения в оперативной памяти компьютера. Говорят, что переменная имеет *естественное выравнивание* (*naturally aligned*), если она находится в памяти по адресу, значение которого кратно размеру этой переменной. Например, 32-разрядная переменная имеет естественное выравнивание, если она находится в памяти по адресу, кратному 4 байтам (т.е. два младших бита адреса равны нулю). Таким образом, при естественном выравнивании структура данных размером 2^n байтов должна храниться в памяти по адресу, младшие n битов которого равны нулю.

На некоторых аппаратных платформах существуют строгие требования относительно выравнивания данных. На некоторых системах, обычно RISC, загрузка неправильно вы-

ровненных данных приводит к генерации системного прерывания (trap), ошибки, которую можно обработать. На других системах допускается работа с невыровненными данными, но это приводит к уменьшению производительности. При написании переносимого кода необходимо предотвращать проблемы, связанные с выравниванием, а данные всех типов должны иметь естественное выравнивание.

Как избежать проблем с выравниванием

Обычно все проблемы, связанные с выравниванием, предотвращаются еще на этапе компиляции за счет естественного выравнивания всех типов данных. По сути, разработчики ядра не должны заниматься проблемами, связанными с выравниванием, поскольку об этом должны позаботиться разработчики компилятора `gcc`. Однако такие проблемы все же могут возникать, когда разработчику приходится выполнять операции с указателями и осуществлять доступ к данным, которые не были распределены компилятором.

Доступ к адресу памяти, для которого выполнено выравнивание, через преобразованный указатель на тип данных большего размера может привести к проблемам выравнивания (на разных аппаратных платформах это может проявляться по-разному). Данная проблема проиллюстрирована в приведенном ниже коде.

```
char wolf[] = "Like a wolf";
char *p = &wolf[1];
unsigned long l = *(unsigned long *)p;
```

В этом примере указатель на данные типа `char` используется в качестве указателя на тип `unsigned long`. Это может привести к тому, что 32- или 64-разрядное значение типа `unsigned long` будет считываться из памяти по адресу, не кратному четырем или восьми соответственно.

Приведенный выше пример запутанного доступа к памяти может показаться вам непонятным, собственно, так оно и есть на самом деле. Тем не менее такое иногда происходит, так что будьте внимательны! К сожалению, в реальных программах не всегда все так очевидно и понятно.

Выравнивание нестандартных типов данных

Как уже указывалось, при выравнивании адрес стандартного типа данных должен быть кратным размеру этого типа. Нестандартные (сложные) типы данных языка C подчиняются правилам выравнивания, приведенным ниже.

- Выравнивание массива выполняется так же, как и выравнивание типа данных его 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 должно выполняться так, чтобы каждый ее элемент был правильно выровнен.

Для целей выравнивания в структурах также могут использоваться пустые поля (padding), которые часто вызывают споры у разработчиков.

Пустые поля структур

Чтобы каждый элемент структуры имел естественное выравнивание, в нее вводятся пустые поля. В результате можно гарантировать, что при доступе процессора к любому элементу этой структуры он будет правильно выровнен. В качестве примера рассмотрим приведенную ниже структуру на 32-разрядной машине.

```
struct animal_struct {
   char           dog; /* 1 байт  */
   unsigned long  cat; /* 4 байта */
   unsigned short pig; /* 2 байта */
   char           fox; /* 1 байт  */
};
```

При расположении в памяти эта структура будет выглядеть немного не так, поскольку для некоторых ее членов нарушено правило естественного выравнивания. Поэтому компилятор создает в памяти похожую структуру, которая будет выглядеть так, как показано ниже.

```
struct animal_struct {
   char            dog;     /* 1 байт  */
   u8            __pad0[3]; /* 3 байта */
   unsigned long   cat;     /* 4 байта */
   unsigned short  pig;     /* 2 байта */
   char            fox;     /* 1 байт  */
   u8            __pad1;    /* 1 байт  */
};
```

Для того чтобы обеспечить естественное выравнивание полей структуры, определенных пользователем, компилятор добавил в структуру пустые поля. Первое из них состоит из трех байтов и позволяет разместить элемент `cat` на границе 4-байтового адреса. В результате остальные элементы этой структуры будут выровнены автоматически, поскольку их размер меньше, чем у элемента `cat`. Второе пустое поле, расположенное в конце структуры, позволяет выровнять ее длину. Один лишний байт в конце гарантирует, что длина структуры будет кратна 4 байтам. В результате при использовании массива из этих структур каждый из его элементов будет корректно выровнен.

Обратите внимание на то, что выражение `sizeof(animal_struct)` возвращает значение 12 для *обоих* вариантов этой структуры на большинстве 32-разрядных аппаратных платформ. Компилятор C автоматически добавляет в структуру пустые поля для обеспечения корректного выравнивания.

В большинстве случаев порядок следования элементов структуры можно изменить так, чтобы избежать добавления пустых полей. Это позволяет получить правильно выровненные данные без введения дополнительных пустых полей 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 */
};
```

Полученная структура данных имеет размер всего 8 байт. Однако не всегда существует возможность перестановки элементов структуры местами и изменения ее определения. Например, если структура определяется как часть стандарта или уже используется в существующем коде, то порядок следования полей менять нельзя. Следует сказать, что

описанные выше требования в меньшей степени относятся к ядру, чем к пользовательским приложениям. Дело в том, что для ядра отсутствуют какие-либо формальные соглашения по поводу двоичного интерфейса прикладных программ (ABI). Иногда по некоторым причинам может потребоваться специальный порядок следования полей структуры, например специальное выравнивание переменных для оптимизации попадания в кеш. Обратите внимание: согласно стандарту ANSI C, компилятору запрещено изменять порядок следования полей в структурах[5] данных — этим правом обладает только программист. Однако компилятор может помочь программисту. Для этой цели в компиляторе `gcc` введен параметр командной строки `-Wpadded`, который предписывает компилятору вывести предупредительное сообщение в случае добавления в структуру пустых полей.

Разработчикам ядра при массовом применении структур следует учитывать особенности выравнивания их полей для разных аппаратных платформ. Дело в том, что при передаче по сети или при сохранении структуры данных на диске процесс выравнивания их полей может отличаться. Это одна из причин, по которой в языке программирования C отсутствует оператор бинарного сравнения структур. В пустых полях структуры может содержаться случайный набор битов, поэтому невозможно выполнить побайтовое сравнение двух структур. Разработчики языка программирования C правильно сделали, что оставили решение задачи сравнения структур на усмотрение программиста, который может создавать собственные функции сравнения в каждом конкретном случае, чтобы использовать особенности размещения конкретных структур.

Порядок следования байтов

Порядок следования байтов (*byte ordering*) — это тот порядок, согласно которому байты расположены в машинном слове. В разных типах процессоров порядок нумерации байтов в машинном слове может отличаться. Так, самый **младший бит** машинного слова может располагаться в памяти как слева, так и справа, т.е. в машинном слове он может быть либо первым, либо последним. Порядок байтов называется *обратным* (*big-endian*), если **самый старший** байт машинного слова хранится в памяти первым (т.е. расположен по младшему адресу), а за ним идут байты в порядке убывания значимости. Порядок байтов называется *прямым* (*little-endian*), если **самый младший** байт слова хранится в памяти первым (т.е. расположен по младшему адресу), а за ним следуют байты в порядке возрастания значимости.

При написании кода ядра не стоит делать какие-либо предположения о порядке следования байтов (конечно, если код не предназначен для какой-либо конкретной аппаратной платформы). Операционная система Linux поддерживает аппаратные платформы с обоими порядками следования байтов, включая и те машины, на которых используемый порядок байтов можно сконфигурировать на этапе загрузки системы. При этом общий код должен быть совместим с любым порядком следования байтов.

Примеры обратного и прямого порядка следования байтов приведены на рис. 19.1 и 19.2 соответственно.

[5] Если бы компилятор мог изменять порядок следования полей структуры данных, то ранее откомпилированный код, в котором уже использовалась эта структура, работал бы некорректно. В языке программирования C положения полей структуры данных определяются путем введения смещения относительно начального адреса структуры в памяти.

Рис. 19.1. Обратный порядок следования байтов

Рис. 19.2. Прямой порядок следования байтов

На аппаратной платформе x86, как в 32-, так и в 64-разрядном варианте, используется прямой порядок следования байтов. На большинстве других аппаратных платформ обычно используется обратный порядок следования байтов.

Рассмотрим на практике, на что влияют эти типы кодирования. Для примера возьмем четырехбайтовое двоичное представление числа 1027.

```
00000000 00000000 00000100 00000011
```

Внутренние представления этого числа в памяти при использовании прямого и обратного порядка следования байтов отличаются, как показано в табл. 19.3.

Таблица 19.3. Расположение данных в памяти для разного порядка следования байтов

Адрес	Обратный порядок	Прямой порядок
0	00000000	00000011
1	00000000	00000100
2	00000100	00000000
3	00000011	00000000

Для аппаратной платформы с обратным порядком следования байтов самый **старший байт** расположен в памяти по **младшему адресу**. При использовании же прямого порядка следования байтов по **младшему адресу** в памяти располагается самый **младший байт**.

И наконец, еще один пример — фрагмент кода, который позволяет определить порядок следования байтов для той аппаратной платформы, на которой он выполняется.

```
int x = 1;

if (*(char *)&x == 1)
   /* Прямой порядок */
else
   /* Обратный порядок */
```

Этот код будет работать как в ядре, так и в пользовательских приложениях.

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 романе самой 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, — "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[6].

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 следования байтов.

В этот заголовочный файл также включаются макросы из каталога `include/linux/byteorder/`, которые помогают конвертировать один порядок байтов в другой. Ниже показаны часто используемые макросы.

```
u32 __cpu_to_be32(u32) ; /* Преобразовать текущий порядок в обратный */
u32 __cpu_to_le32(u32) ; /* Преобразовать текущий порядок в прямой   */
u32 __be32_to_cpu(u32) ; /* Преобразовать обратный порядок в текущий */
u32 __le32_to_cpus(u32); /* Преобразовать прямой порядок в текущий   */
```

Эти макросы позволяют преобразовать один порядок следования байтов в другой. В случае если порядок байтов, между которыми выполняется преобразование, совпадает (например, если выполняется преобразование в обратный порядок байтов и в процессоре также используется обратный порядок), то эти макросы не выполняют никаких действий. В противном случае возвращается преобразованное значение.

Учет времени

Учет времени — это еще одна абстракция ядра, реализация которой может отличаться как на разных аппаратных платформах, так и даже в разных выпусках ядра. По этой причине никогда нельзя привязываться к какой-либо конкретной частоте генерации прерывания системного таймера и, соответственно, к тому, сколько раз в секунду изменяется переменная `jiffies`. Для корректного определения временны*х* интервалов всегда необходимо использовать константу `HZ`. 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, для аппаратной платформы x86 стандартное значение константы `HZ` сейчас равно 100. Это означает, что прерывание от таймера возникает 100 раз в секунду, или каждые 10 миллисекунд. Однако в ранних выпусках ядер серии 2.6 для аппаратной платформы x86 значение константы `HZ` было равно 100. Для разных аппаратных платформ

---

[6] Исторически обратный порядок следования байтов стал использоваться раньше. Все дело в том, что на заре развития вычислительной техники в качестве устройства вывода использовались так называемые алфавитно-цифровые печатающие устройства (АЦПУ) — некий прообраз современных принтеров. Мониторы на электронно-лучевой трубке и интерактивные отладчики появились значительно позже! АЦПУ печатали построчно на длинной рулонной бумаге и могли выводить только буквы и цифры, правда, с очень высокой скоростью (от 400 до 1000 строк в минуту). "Древние" программисты при отладке программ часто печатали "километровые" рулоны (благо бумага была дешевая и ее было много!) шестнадцатеричных дампов, в которых содержимое памяти ЭВМ отображалось группами по 2 или 4 байта (в зависимости от разрядности процессора). Так вот, благодаря обратному порядку следования байтов упрощался анализ этих дампов. Поскольку дампы выводились слева направо, от младшего адреса к старшему, программистам не нужно было в уме "переворачивать байты" для анализа значения машинного слова. — *Примеч. ред.*

эти значения отличаются. Например, для аппаратной платформы Alpha константа HZ равна 1024, а для платформы ARM — 100.

Никогда не следует ради упрощения сравнивать значение переменной jiffies с некоторым числом, скажем, с 100, и думать, что оно всегда будет означать одно и то же. Для получения корректных временны*х* интервалов необходимо всегда умножать или делить их на константу HZ, как показано в следующем примере:

```
HZ          /* Одна секунда */
(2*HZ)      /* Две секунды  */
(HZ/2)      /* Полсекунды   */
(HZ/100)    /* 10 мс        */
(2*HZ/100) /* 20 мс         */
```

Константа HZ определена в файле <asm/param.h>. Об этом подробно рассказано в главе 11, “Таймеры и управление временем”.

Размер страницы памяти

При работе со страницами памяти никогда нельзя привязываться к конкретному размеру страницы. Программисты, которые разрабатывают код для аппаратной платформы x86-32, часто делают ошибку, считая, что размер страницы всегда равен 4 Кбайт. Хотя это справедливо для платформы x86-32, для других аппаратных платформ размер станицы может быть иным. По сути, на некоторых аппаратных платформах могут одновременно поддерживаться разные размеры страниц! В табл. 19.4 приведены размеры 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, 16, 32 Кбайт
avr	12	4 Кбайт
blackfin	12	4 Кбайт
cris	13	8 Кбайт
frv	14	16 Кбайт
h8300	12	4 Кбайт
ia64	12, 13, 14, 16	4, 8, 32, 64 Кбайт
m32r	12	4 Кбайт
m68k	12, 13	4, 8 Кбайт
m86knommu	12	4 Кбайт
mips	12	4 Кбайт
mn10300	12	4 Кбайт
parisc	12	4 Кбайт
powerpc	12	4 Кбайт
s390	12	4 Кбайт
sh	12	4 Кбайт

Окончание табл. 19.4

Аппаратная платформа	Значение PAGE_SHIFT	Значение PAGE_SIZE
sparc	12, 13	4, 8 Кбайт
um	12	4 Кбайт
x86	12	4 Кбайт
xtensa	12	4 Кбайт

При работе со страницами памяти для определения их размера в байтах используйте константу PAGE_SIZE. Константа PAGE_SHIFT определяет количество битов, на которое нужно сдвинуть значение адреса вправо, чтобы определить номер страницы. Например, для аппаратной платформы x86-32 размер страницы равен 4 Кбайт, поэтому константа PAGE_SIZE равна 4096, а PAGE_SHIFT — 12. Эти значения определены в файле <asm/page.h>.

Порядок выполнения операций процессором

В главе 9, "Общие сведения о синхронизации кода ядра", и главе 10, "Средства синхронизации ядра", мы уже говорили о том, что на разных аппаратных платформах процессоры могут в той или иной степени изменять порядок выполнения машинных команд. Для некоторых типов процессоров порядок выполнения команд неукоснительно соблюдается, запись данных в память и считывание данных из памяти выполняются в строго указанном в программе порядке. Для других процессоров порядок выполнения машинных команд не является жестким. В них с целью оптимизации может изменяться порядок выполнения команд считывания из памяти и записи данных в память.

Если код зависит от порядка выполнения операций чтения-записи данных, то необходимо гарантировать, что даже процессор с самыми слабыми ограничениями на порядок выполнения команд 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 10, "Средства синхронизации ядра".

Многопроцессорность, мультипрограммирование и верхняя память

Вам может показаться странным, что мы включили вопросы поддержки симметричной многопроцессорности (SMP), мультипрограммного режима работы ядра и обращения к верхней памяти в тему о переносимости. В конце концов, ведь это не особенности аппаратной платформы, которые влияют на операционную систему, а функции ядра Linux, которые во многом не зависят от аппаратной платформы. Тем не менее для этих функций существуют важные параметры конфигурации, которые необходимо учитывать при разработке кода. Поэтому, чтобы ваш код всегда корректно работал при любых конфигурациях параметров, его необходимо создавать с учетом этих функций. Кроме описанных в предыдущих разделах особенностей переносимости, необходимо также учитывать приведенные ниже рекомендации.

- Всегда необходимо учитывать, что код может выполняться на многопроцессорной системе и в нем могут использоваться соответствующие блокировки.

- Всегда необходимо учитывать, что код может выполняться при включенном мультипрограммном режиме работы ядра, поэтому необходимо всегда использовать соответствующие блокировки и команды для управления этим режимом работы.
- Всегда необходимо учитывать, что код может выполняться на системе с поддержкой верхней памяти (непостоянно отображаемая память), поэтому там где нужно используйте функцию `kmap()`.

Резюме

Если говорить коротко, то написание переносимого, ясного и красивого кода для ядра Linux подразумевает 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 наименьший общий знаменатель: нельзя полагаться на то, что будут доступны все возможности ядра, следует опираться только на минимум возможностей, которые доступны на всех аппаратных платформах.

Написание переносимого кода требует строгого учета многих факторов, в частности, размера машинного слова, размеров типов данных, выравнивания в памяти и добавления пустых полей, порядка следования байтов, знаковых и беззнаковых целых чисел, размера страницы, а также изменения порядка выполнения команд чтения/записи процессором. В большинстве случаев при создании кода ядра следует убедиться, что все типы данных используются правильно. Тем не менее время от времени все равно всплывают проблемы, связанные с особенностями той или иной аппаратной платформы. Поэтому важно понимать эти проблемы и всегда писать четкий и переносимый код ядра.

20
Заплаты, хакерство и сообщество

Одним из самых больших преимуществ операционной системы Linux является связанное с ней большое сообщество пользователей и разработчиков. Благодаря сообществу множество глаз может проверить корректность вашего кода, эксперты могут дать вам ценный совет, а множество пользователей протестируют ваш код и отправят сообщения об ошибках. Самым важным является то, что только сообщество может решить, какой именно код следует включить в официальное дерево исходных кодов ядра. Поэтому важно понимать, как это все происходит.

Сообщество

Когда речь заходит о том, где физически расположены члены сообщества, то в этой связи уместно упомянуть *список рассылки разработчиков ядра Linux* (Linux Kernel Mail List, или, сокращенно, *lkml*). Список рассылки — это то место, где размещаются важные сообщения и происходит большинство дискуссий, дебатов и флеймов вокруг ядра Linux. Здесь обсуждаются новые возможности и размещается бо́льшая часть кода прежде, чем он начнет для чего-нибудь использоваться. В списке рассылки насчитывается более 300 сообщений в день — количество не для слабонервных. Подписаться на этот список (или, по 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 по электронной почте приведенное ниже сообщение в виде простого текста по адресу `majordomo@vger.kernel.org`.

```
subscribe linux-kernel <your@email.address>
```

За более подробной информацией обращайтесь на сайт `http://vger.kernel.org/`. Список часто задаваемых вопросов (FAQ) находится здесь: `http://www.tux.org/lkml/`.

В Интернете можно найти огромное количество других веб-сайтов и списков рассылки, посвященных тематике Linux вообще и его ядру в частности.

Отличный ресурс для начинающих хакеров — `http://www.kernelnewbies.org/` — сайт, который сможет удовлетворить желания всех, кто, стачивая зубы, грызет гранит основ разработки ядра. Два других отличных источника информации — это сайт `http://www.lwn.net/`, Linux Weekly News, на котором есть большой раздел новостей ядра, и сайт `http://www.kerneltrap.org`, Kernel Trap, который содержит сводку сообщений из списка рассылки разработчиков ядра Linux с комментариями.

Стиль написания исходного кода

Как и для любого другого большого программного проекта, в ядре Linux определены правила написания исходного кода, которые определяют форматирование, стиль и размещение исходного кода. Это сделано не потому, что стиль написания, который принят для Linux, лучше других (хотя и очень может быть), и не потому, что все программисты пишут трудные для восприятия программы (хотя такое тоже бывает), а потому, что *одинаковость* стиля является важным моментом для обеспечения высокой *производительности* труда разработчиков.

Часто можно услышать, что стиль написания исходного кода не важен, потому что он не влияет на скомпилированный объектный код. Однако для большого программного проекта, такого как ядро Linux, в котором задействовано огромное количество разработчиков, важна слаженность стиля. Слаженность подразумевает одинаковость восприятия, что ведет к упрощению чтения кода, 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 прочитать 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 использовался при написании большей части кода и которого сейчас должны придерживаться разработчики. Подробное описание стиля со свойственным Линусу юмором приведено в файле `Documentation/CodingStyle`, расположенном в дереве исходных кодов ядра.

Отступы

Для выравнивания текста и введения отступов должны использоваться символы табуляции. Размер одного символа табуляции при отображении соответствует восьми позициям. Однако это не означает, что для структурирования вместо символа табуляции можно использовать восемь пробелов. Это означает только, что каждый последующий уровень отступа отстоит на один символ табуляции от предыдущего и что при отображении длина символа табуляции равна восьми символам, как показано в следующем примере:

```
static void get_new_ship(const char *name)
{
        if (!name)
                name = DEFAULT_SHIP_NAME;
        get_new_ship_with_name(name);
}
```

Однако по непонятным причинам данное правило почти всегда нарушается, несмотря на то, что оно очень сильно влияет на читабельность кода. Восьмисимвольная табуляция позволяет очень легко визуально различать отдельные блоки кода даже после нескольких

часов работы. Разумеется, при таком подходе существует и обратная сторона медали. Использование восьми символов для отступа приводит к тому, что после нескольких уровней вложенности строки кода смещаются резко вправо за границы экрана. Это справедливо, если используется стандартная ширина для окна кода, равная 80 символам (подробнее об этом речь пойдет в следующем разделе). Однако Линус парирует, что код не должен быть настолько сложен и запутан, чтобы вам не хватило для его оформления двух-трех уровней вложенности. Если же вы столкнулись с такой глубиной кода, продолжает Линус, вам следует переработать код (выполнить его рефакторинг) и вынести из него отдельные уровни сложности (и, как следствие, уровни отступа) в отдельные функции.

Оператор `switch`

Для уменьшения проблемы, связанной с восьмисимвольными отступами, подчиненные метки `case` оператора `switch` следует располагать на том же уровне, что и само ключевое слово `switch`, как показано в приведенном ниже примере.

```
switch (animal) {
case ANIMAL_CAT:
        handle_cats();
        break;
case ANIMAL_WOLF:
        handle_wolves();
        /* Проходим дальше */
case ANIMAL_DOG:
        handle_dogs();
        break;
default:
        printk(KERN_WARNING "Unknown animal %d!\n", animal);
}
```

Обратите внимание на то, что в приведенном выше примере в случае, когда управление переходит из одного блока `case` в другой, мы оставили явный комментарий. Это хорошая практика, и мы настоятельно рекомендуем придерживаться ее.

Пробелы

В этом разделе мы обсудим пробелы, окружающие символы языка и ключевые слова, а не те пробелы, которые используются для формирования отступов, описанные в предыдущих двух разделах. В сущности, согласно стилю оформления исходного кода, принятого в Linux, вокруг большинства ключевых слов и символов должен располагаться один пробел, а между именем функции и круглой скобкой не должно быть пробелов, как показано ниже.

```
if (foo)
while (foo)
for (i = 0; i < NR_CPUS; i++)
switch (foo)
```

Опять же функции, макросы и ключевые слова, которые по внешнему виду напоминают функции, такие как `sizeof`, `typeof` и `alignof`, не должны отделяться пробелом от следующей за ними круглой скобки.

```
wake_up_process(task);
size_t nlongs = BITS_TO_LONG(nbits);
int len = sizeof(struct task_struct);
typeof(*p)
```

```
__alignof__(struct sockaddr *)
__attribute__((packed))
```

Внутри круглых скобок не должно быть пробелов до первого и после последнего аргумента, как показано выше. Например, не допускается использовать приведенный ниже код.

```
int prio = task_prio( task ); /* Плохой стиль! */
```

Вокруг большинства двух- и трехоперандных операторов следует поместить по одному пробелу спереди и сзади знака операции, как показано в примере ниже.

```
int sum = a + b;
int product = a * b;
int mod = a % b;
int ret = (bar) ? bar : 0;
return (ret ? 0 : size);
int nr = nr ? : 1; /* тоже, что и "nr ? nr : 1" */
if (x < y)
if (tsk->flags & PF_SUPERPRIV)
mask = POLLIN | POLLRDNORM;
```

Соответственно, вокруг большинства унарных операторов не должно быть пробелов между знаком операции и операндом.

```
if (!foo)
int len = foo.len;
struct work_struct *work = &dwork->work;
foo++;
--bar;
unsigned long inverted = ~mask;
```

Отсутствие пробелов справа от оператора разыменования (косвенного обращения к памяти) особенно важно. Ниже приведен правильный стиль оформления такого оператора.

```
char *strcpy(char *dest, const char *src)
```

Не допускается указывать пробелы с обеих сторон этого оператора.

```
char * strcpy(char * dest, const char * src) /* Плохой стиль! */
```

Также не допускается стиль, принятый в языке C++, когда оператор разыменования помещают сразу за объявлением типа, как показано ниже.

```
char* strcpy(char* dest, const char* src) /* Плохой стиль! */
```

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. При разработке кода ядра было принято соглашение размещать открывающую скобку в первой строке, сразу за соответствующим оператором. Закрывающая скобка помещается в первой позиции с новой строки, как показано в следующем примере:

```
if (strncmp(buf, "NO_", 3) == 0) {
        neg  = 1;
        cmp += 3;
}
```

Если за закрывающей скобкой продолжается тот же самый оператор, то продолжение выражения записывается в той же строке, что и закрывающая скобка, как показано ниже.

```
if (ret) {
        sysctl_sched_rt_period = old_period;
```

```
        sysctl_sched_rt_runtime = old_runtime;
} else {
        def_rt_bandwidth.rt_runtime = global_rt_runtime();
        def_rt_bandwidth.rt_period = ns_to_ktime(global_rt_period());
}
```

А вот еще один пример.

```
do {
        percpu_counter_add(&ca->cpustat[idx], val);
        ca = ca->parent;
} while (ca);
```

На функции это правило не распространяется, потому что внутри одной функции тело другой функции описывать нельзя.

```
unsigned long func(void)
{
   /* ... */
}
```

И наконец, в выражениях, в которых фигурные скобки *не обязательны*, их можно опустить. Например, в следующем выражении наличие скобок приветствуется, но не требуется.

```
if (cnt > 63)
        cnt = 63;
```

Логика всего этого описана в книге K&R[1]. Большинство правил оформления исходного кода, принятых в ядре Linux, соответствуют стилю оформления кода, описанному авторами этой знаменитой книги.

Длина строки исходного кода

При написании исходного кода ядра необходимо стараться, насколько это возможно, чтобы длина строки не превышала 80 символов. Это позволяет просматривать исходный код без искажений, связанных с переносом строк, в окне стандартного терминала системы Unix, вмещающего 24 строки по 80 символов.

Не существует какой-либо стандартной рекомендации на тот случай, если длина строки кода абсолютно точно должна превысить 80 символов. Некоторые разработчики просто вводят длинные строки, возлагая ответственность за удобочитаемое отображение строк на программу текстового редактора. Другие же разбивают такие строки на части и вручную вставляют символы конца строки в тех местах, которые кажутся им наиболее подходящими. При этом для продолжения строки можно сделать отступ на один символ табуляции относительно ее начала.

Некоторые разработчики помещают параметры функции один под другим, если они не помещаются в одной строке, и выравнивают их относительно открывающей круглой скобки, как показано в следующем примере:

```
static void get_new_parrot(const char *name,
                           unsigned long disposition,
                           unsigned long feather_quality)
```

[1] Брайан У. Керниган, Деннис М. Ритчи. *Язык программирования C, 2-е изд.* ISBN 978-5-8459-0891-9, пер. с англ., ИД "Вильямс", 2006 г. K&R означает аббревиатуру, составленную из первых букв английских фамилий авторов, которая прочно закрепилась за этой книгой, написанной создателем языка программирования C и его коллегой.

Часть разработчиков просто разбивают длинную строку на части, но не помещают параметры функций один под другим, а для продолжения строки делают отступ на два символа табуляции относительно 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, т.е. за вами. Большинство разработчиков ядра, включая автора этой книги, предпочитают первый вариант, т.е. все строки, длина которых превышает 80 символов, разбиваются вручную на несколько строк. При этом все продолжения строки должны быть красиво выровнены относительно друг друга и ее начала.

Соглашения о присвоении имен

В именах переменных и функций не допускается использовать символы разных регистров. Выбирая для переменной замечательное, на ваш взгляд, и короткое имя, такое как `idx` или `i`, вы должны позаботиться о том, чтобы из контекста программы было понятно назначение этой переменной. Слишком хитрые имена, такие как `theLoopIndex`, не допускаются. Так называемая венгерская форма записи (Hungarian notation), когда тип переменной кодируется в ее имени, в данном случае неуместна и никогда не должна использоваться. В конце концов, это же C, а не Java и Unix, а не Windows.

Тем не менее глобальные переменные и функции должны иметь наглядные имена, записанные в нижнем регистре и по мере необходимости дополненные символом подчеркивания. Если глобальной функции присвоить имя, скажем, `atty()`, то это может привести к путанице. Более подходящим будет имя `get_active_tty()`. Это все-таки Linux, а не BSD.

Функции

Существует хорошее эмпирическое правило: функции не должны по объему кода превышать двух экранов текста и иметь больше *десяти* локальных переменных. Каждая функция должна выполнять только одно действие, но делать это хорошо. Полезно разбить функцию на последовательность более мелких функций. Если при этом вы беспокоитесь о снижении эффективности программы за счет дополнительных вызовов функций, то можно использовать встроенные функции, объявленные с помощью ключевого слова `inline`.

Комментарии

При написании исходного кода, безусловно, важно использовать комментарии, однако делать это нужно правильно. Обычно необходимо описывать, *что* делает код и *для чего* это делается, а не то, *как* это делается, поскольку последнее должно быть и так ясно из кода. Если так сделать не получается, то, возможно, стоит пересмотреть то, что вы написали, и, соответственно, изменить код. Кроме того, в комментарии не должна включаться информация о том, кто написал функцию, когда это было сделано, время модификации и пр. Такую информацию логично размещать в самом начале файла исходного кода.

В ядре используются комментарии в стиле языка C, несмотря на то, что компилятор `gcc` поддерживает также и комментарии в стиле C++. Как правило, комментарии в исходном коде ядра выглядят так, как показано ниже (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 спин-блокировке.
 */
```

В комментариях важные замечания часто начинаются со строки `"XXX: "`, а информация о дефектах — со строки `"FIXME: "`, как показано 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 утилите GNOME-doc, но немного модифицирована и теперь называется Kernel-doc. Для создания документации в формате HTML запустите команду

```
make htmldocs
```

Для генерации документации в формате PostScript команда должна быть следующей.

```
make psdocs
```

Для того чтобы можно было воспользоваться системой автоматической генерации документации, при создании функций вы должны использовать специальный формат комментариев, как показано ниже.

```
/**
 * 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 map, struct timeval *time)
{
   /* ... */
}
```

Более подробно система документации ядра описана в файле `Documentation/kernel-doc-nano-HOWTO.txt`.

Использование директивы `typedef`

В сообществе разработчиков ядра Linux отмечается стойкое неприятие к определению новых типов с помощью оператора `typedef`. Ниже приведены логические объяснения этому.

- 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.

Поэтому, чтобы избежать насмешек, лучше не использовать оператор `typedef`.

Разумеется, существуют ситуации, в которых полезно использовать оператор `typedef`. Речь идет о сокрытии специфичных для аппаратной платформы деталей реализации или

обеспечении совместимости при предполагаемом изменении типа в будущем. Сначала нужно хорошенько подумать, действительно ли оператор `typedef` так уж необходим, или он используется только для того, чтобы уменьшить количество символов при наборе кода.

Использование того, что уже есть

Не нужно изобретать велосипед! В ядре уже реализованы функции для работы со строками, подпрограммы для сжатия и декомпрессии данных и интерфейс для работы со связанными списками — так используйте же их!

Не нужно инкапсулировать стандартные интерфейсы в другие реализации обобщенных интерфейсов. Вам часто придется сталкиваться с кодом, который, без сомнения, был портирован из других операционных систем в систему Linux. При этом на основе существующих интерфейсов ядра создается громоздкая функция, которая служит для связи нового кода с существующим. Такое не нравится никому, поэтому необходимо напрямую использовать предоставляемые интерфейсы.

Избегайте директив `ifdef` в исходном коде

Использование директив препроцессора `ifdef` непосредственно в исходных файлах языка C не одобряется. Вы не должны использовать в своих функциях код, аналогичный приведенному ниже.

```
        ...
#ifdef CONFIG_FOO
        foo();
#endif
        ...
```

В случае, если параметр `CONFIG_FOO` не задан, определите в заголовочном файле холостую функцию `foo()`, как показано ниже.

```
#ifdef CONFIG_FOO
static int foo(void)
{
        /* .. */
}
#else
static inline int foo(void) { }
#endif /* CONFIG_FOO */
```

После этого можно вызывать функцию `foo()` без всяких условий. Пусть компилятор поработает за вас!

Инициализация структур

При инициализации структур необходимо использовать метки полей. Это позволяет предотвратить некорректную инициализацию при изменении структуры. Это также позволяет выполнять инициализацию избранных полей структуры. К сожалению, в стандарте C99 принят довольно уродливый формат меток полей, а в компиляторе `gcc` ранее использовавшийся формат меток полей в стиле GNU признан устаревшим, хотя он был намного удобнее. Следовательно, в коде ядра необходимо использовать новый формат, согласно стандарту C99, каким бы уродливым он ни был.

```
struct foo my_foo = {
    .a = INITIAL_A,
    .b = INITIAL_B,
};
```

В этом коде элементам a и b структуры foo присваиваются значения INITIAL_A и INITIAL_B соответственно. Согласно стандарту ANSI C, если при инициализации структуры какое-то поле не указано, то ему присваивается некоторое стандартное значение (для указателей — NULL, для целых чисел — 0, для чисел с плавающей запятой — 0.0). Например, если бы в рассматриваемой нами структуре foo также присутствовало поле int c, то в результате приведенного выше оператора инициализации структуры ему было бы присвоено значение 0.

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 отчаиваться. Немного упорства при освоении утилиты indent принесут свои плоды. Программа indent — великолепная утилита GNU, которая включена во многие поставки ОС Linux и предназначена для форматирования исходного кода в соответствии с заданными правилами. Ее стандартные установки 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 -l80 -ss -bs -psl <file>
```

Эти параметры позволяют отформатировать исходный код в соответствии со стилем, принятым при разработке ядра Linux. Кроме того, можно также использовать сценарий scripts/Lindent, который вызывает утилиту indent с необходимыми параметрами.

Организация команды разработчиков

Разработчики ядра — это команда хакеров, занятых развитием ядра Linux. Часть из них работает за деньги, для других это хобби, но практически все они делают свою работу 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* (*maintainers*), которые официально выполняют их поддержку. Ответственные разработчики — это один человек или группа людей, которые полностью отвечают за свою часть ядра. Например, каждый отдельный драйвер ядра имеет связанного с ним ответственного разработчика. Каждой подсистеме ядра, например сетевой подсистеме, также назначен связанный с ней ответственный разработчик. Разработчики, которые отвечают за определенный драйвер или подсистему, обычно перечислены в файле MAINTAINERS. Этот файл также находится в корневом каталоге дерева исходных кодов ядра.

Среди ответственных разработчиков есть человек, который отвечает за все ядро в целом (kernel maintainer). Этот человек фактически осуществляет поддержку дерева исходных кодов ядра. Исторически так сложилось, что за разрабатываемую серию ядер (где сосредоточена вся интересная работа) и за первые несколько выпусков стабильной версии ядра отвечает Линус. Вскоре после того как ядро становится стабильным, Линус передает бразды правления одному из ведущих разработчиков. Он продолжает поддержку стабильной серии ядра, а Линус начинает работу над новой разрабатываемой серией. Принятие “нового мирового порядка”, при котором разработка ядра серии 2.6 будет длиться бесконечно долго, сделало Линуса ответственным за разработку ядра этой серии. Еще один человек поддерживает разработку ядра серии 2.4, которая теперь ведется исключительно в режиме исправления ошибок.

Отправка сообщений об ошибках

Если вы обнаружили ошибку, то наилучшим решением будет исправить ее, сгенерировать соответствующую заплату, протестировать и опубликовать ее (как это сделать, рассказывается в следующих разделах). Разумеется, можно и просто сообщить об ошибке, чтобы кто-нибудь исправил ее вместо вас.

Наиболее важная часть сообщения об ошибке — это полное описание проблемы. Необходимо описать признаки, указать связанные с проблемой системные сообщения и приложить 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.

На следующем этапе нужно определить 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 не сможете найти необходимого разработчика, то отправьте отчет об ошибке 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`.

Заплаты

Все изменения, вносимые в исходный код 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)`.

Генерация заплат

Проще всего сгенерировать заплату в случае, если у вас имеются два дерева исходных кодов ядра: одно — стандартное, другое — с вашими изменениями. Обычно каталогу, в котором находится стандартное ядро, присваивается имя `linux-x.y.z` (оно получается в результате разворачивания архива дерева исходного кода в формате `tar`), а каталогу, в котором хранится ваше модифицированное ядро, присваивается имя `linux`. Тогда для генерации заплаты на основе двух каталогов с исходным кодом необходимо запустить приведенную ниже команду из каталога, в котором находятся два рассмотренных дерева исходного кода.

```
diff -urN linux-x.y.z/ linux/ > my-patch
```

Обычно такая операция выполняется в вашем 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 довольно уродливым и 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, внесенные в одном файле, то запустите команду

```
diff -u linux-x.y.z/some/file linux/some/file > my-patch
```

Обратите внимание на то, что для генерации изменений вы должны перейти в тот каталог, в котором 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 подобным образом, выполните приведенную ниже 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 команду

```
diffstat -p1 my-patch
```

Обычно полезно включить результат выполнения этой команды при отправлении заплаты в список рассылки lkml. Поскольку программа `patch(1)` игнорирует все строки до того момента, пока не будет обнаружен формат `diff`, то вначале заплаты можно включить короткое описание.

Генерирование заплат с помощью программы Git

Если для поддержки дерева исходных кодов ядра используется программа Git, то для генерации заплат вы должны использовать эту программу. При этом вам не понадобится выполнять все описанные выше шаги вручную и *углубляться* в сложные параметры настройки Git. Генерирование заплат с помощью Git — несложный двухэтапный процесс. Прежде всего нужно внести сами изменения в исходный код и *зафиксировать* их в локальном хранилище. Внесение изменений в дерево исходных кодов Git аналогично внесению изменений в стандартное дерево исходных кодов ядра Linux. Кроме изменения самого файла, включенного в дерево Git, вам больше не придется выполнять каких-либо особенных действий. После внесения изменений в файл зафиксируйте изменения в хранилище Git, запустив приведенную ниже команду.

```
git commit -a
```

Флаг `-a` указывает на то, что нужно зафиксировать *все* внесенные изменения. Если нужно зафиксировать изменения, внесенные только в один файл, то это можно сделать с помощью команды

```
git commit some/file.c
```

Даже при указании флага `-a` программа Git не будет помещать в свое хранилище вновь созданные файлы. Для этого их нужно добавить в хранилище вручную. С помощью двух приведенных ниже команд новый файл добавляется в хранилище, а затем выполняется фиксация всех изменений.

```
git add some/other/file.c
git commit -a
```

После запуска команды `git commit` программа запросит у вас информацию для журнала изменений. Введите максимально полные и ясные сведения, которые объясняют причину внесения изменений. (Что именно нужно включить в эти сведения, вы узнаете в следующем разделе.) В вашем локальном хранилище можно фиксировать произвольное количество изменений. Программа Git устроена так, что она фиксирует все последовательные изменения, внесенные даже в одном и том же файле, и впоследствии может их воспроизвести. После фиксации изменений в дереве исходных кодов Git можно сгенерировать заплату для каждого из них. Эти заплаты ничем не будут отличаться от тех, которые были получены по методике, описанной в предыдущем разделе.

```
git format-patch origin
```

Эта команда сгенерирует заплаты для всех изменений, внесенных в ваше локальное хранилище, а не в оригинальное дерево исходных кодов ядра. Программа Git создаст файлы заплат и поместит их в корневой каталог вашего дерева исходных кодов ядра. Чтобы сгенерировать заплаты только для последних *N* фиксаций, запустите приведенную ниже команду.

```
git format-patch -N
```

Например, чтобы сгенерировать заплату только для самой последней фиксации изменений, используйте команду

```
git format-patch -1
```

Публикация заплат

Файлы заплат должны быть сгенерированы так, как описано в предыдущих разделах. Если заплата сделана для определенного драйвера или подсистемы ядра, то ее нужно отправить соответствующему ответственному разработчику, одному из тех, которые перечислены в файле `MAINTAINERS`. И не забудьте направить копию своего сообщения в список рассылки разработчиков ядра по адресу `linux-kernel@vger.kernel.org`. Заплату можно отправлять ответственному разработчику ядра (например, Линусу) только после всестороннего обсуждения в сообществе разработчиков, или если она очень проста и ни у кого не возникает сомнения в ее правильности.

Обычно тема (subject) сообщения, содержащего заплату, должна быть сформирована так: "`[PATCH]` *`короткое описание`*". В теле сообщения нужно описать 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 обычно просматривают заплату прямо в теле сообщения и в случае необходимости сохраняют все сообщение в отдельном файле. Поэтому лучше всего поместить заплату прямо в самый конец тела сообщения. Не стоит забывать о том, что в некоторых почтовых клиентах текст вашего сообщения может быть переформатирован, например, длинные строки будут перенесены на новую строку. В результате заплата будет испорчена, что невероятно раздражает разработчиков. Если ваш почтовый клиент именно такой, то поищите в нем специальные команды для вставки предварительно отформатированного текста в тело сообщения, которые могут называться поразному, например Insert Inline, Preformat или что-то в этом духе. Если найти эти команды не удается, поместите файл заплаты в сообщение в виде вложения. При этом файл вложения должен быть в формате обычного текста, а режим его кодирования отключен.

Если заплата очень большая или содержит *несколько* принципиальных изменений, то лучше разбить ее на логические части. При этом одно изменение должно находиться в одной логической 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 сообщениями. Если в каком-либо случае необходимо вначале применить предыдущую заплату, то это необходимо специально указать.

После отправки сообщения 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, попытайтесь разобраться, что было сделано не так, и решить проблему. Попросите прокомментировать ваше сообщение в списке рассылки и у ответственного разработчика. Если повезет, то ваши изменения будут включены в новую версию ядра! Примите наши поздравления!

Резюме

Наиболее важными качествами любого хакера являются желание и умение работать — вы должны сами отыскивать проблемы и решать их. В этой книге описаны основные части ядра, рассказано об интерфейсах, структурах данных, алгоритмах и принципах их работы. В книге ядро Linux описано в практической форме с точки зрения разработчика. Она предназначена для того, чтобы удовлетворить ваше любопытство и стать отправной точкой в разработке ядра.

Тем не менее, как уже было сказано ранее, не существует другого способа начать разработку кода ядра, не *ознакомившись* с существующим кодом и не *начав* писать собственный 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! Удачи в хакинге!

Список литературы

В этом списке литературы мы постарались привести источники информации, близкие к содержанию данной книги. Однако самым лучшим источником, дополняющим книгу, является исходный код ядра. При работе с ОС Linux у нас есть неограниченный доступ к полному исходному коду ядра современной операционной системы. Не принимайте это как должное! Ознакомьтесь с ним! Читайте код! Пишите код!

Книги по основам построения операционных систем

В приведенных ниже книгах рассмотрены принципы работы операционных систем в объеме учебных курсов. В них описываются основные понятия, алгоритмы и проблемы, связанные с построением высоко функциональных операционных систем, а также решения указанных проблем. Все эти книги могут быть рекомендованы, но если нужно выделить одну, то это, конечно же, книга Х.М. Дейтеля.

1. Deitel H., Deitel P. and D. Choffnes. *Operating Systems*. Prentice Hall, 2003. Прекрасная книга по теории операционных систем с отличными примерами из теории и практики.
2. Tanenbaum Andrew. *Modern Operating Systems*. Prentice Hall, 2007. Детальный обзор стандартных проблем разработки операционных систем, а также обсуждение многих концепций, которые используются в современных операционных системах, таких как UNIX и Windows.
3. Tanenbaum Andrew. *Operating Systems: Design and Implementation*. Prentice Hall, 2006. Хорошие начальные сведения об основах построения, принципах работы и реализации Unix-подобной операционной системы Minix.
4. Silberschatz A., Galvin P. and G. Gagne. *Operating System Concepts*. John Wiley and Sons, 2008. Также известна как “книга динозавра”, 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 ядре Unix

В приведенных ниже книгах описываются принципы работы и особенности реализации ядер Unix. В первых пяти рассмотрены конкретные варианты Unix, в двух последних — общие моменты всех вариантов Unix. Если собираетесь приобрести только две из перечисленных ниже книг, остановите свой выбор на двух последних.

1. Bach Maurice. *The Design of the Unix Operating System*. Prentice Hall, 1986. Обсуждение особенностей построения операционной системы Unix System V, Release 2.
2. McKusick M., Bostic K., Karels M. and J. Quarterman. *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.
3. McKusick M. and G. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004. Описаны основы построения и реализации операционной системы FreeBSD 5.2.
4. McDougall R. and J. Mauro. *Solaris Internals: Solaris and OpenSolaris Kernel Architecture*. Prentice Hall, 2006. Интересное обсуждение основных подсистем и алгоритмов работы ядра ОС Solaris.
5. Cooper C. and C. Moore. *HP-UX 11i Internals*. Prentice Hall, 2004. Обзор внутреннего устройства операционной системы HP-UX и аппаратной платформы PA-RISC.
6. Vahalia, Uresh. *Unix Internals: The New Frontiers*. Prentice Hall, 1995. Отличная книга о возможностях современных Unix-подобных операционных систем, включая управление потоками и мультипрограммным режимом работы ядра.
7. 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. Настоятельно рекомендуется к прочтению!

Книги о ядре Linux

В этих книгах, как и в текущей, рассказывается о ядрах Linux. В этом разделе пока что не так много книг. Тем не менее я настоятельно рекомендую прочитать две книги, которые указаны здесь.

1. Benvenuti, Christian. *Understanding Linux Network Internals*. O'Reilly and Associates, 2005. Подробное описание сетевой подсистемы Linux.
2. Corbet, J.,A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers.* O'Reilly and Associates, 2005. Великолепное описание способов создания драйверов устройств для ядра Linux серии 2.6. Особый акцент сделан на программных интерфейсах, поддерживающих различные типы устройств.

Книги о ядрах других операционных систем

Всегда полезно понимать логику работы конкурентов. В приведенных здесь книгах описаны основы работы и особенности реализации операционных систем, отличных от операционной системы Linux. Смотрите сами, что у них хорошо, а что — плохо.

1. Kogan M. and H. Deitel. *The Design of OS/2*. Addison-Wesley, 1996. Интересный обзор операционной системы OS/2 2.0.
2. Singh, Amit. *Mac OS X Internals: A Systems Approach*. Addison-Wesley Professional, 2006. Глубокий и всеобъемлющий трактат обо всех особенностях реализации Mac OS X.
3. Solomon D. and M. Russinovich. *Windows Internals: Covering Windows Server 2008 and Windows Vista*. Microsoft Press, 2009. Интересный взгляд на операционные системы, которые отличаются от Unix.

Книги по API Unix

Детальное описание системы Unix и API этой операционной системы важно не только для того, чтобы писать многофункциональные прикладные программы, но и для понимания того, какие функции выполняет ядро.

1. Love, Robert. *Linux System Programming*. O'Reilly and Associates, 2007. Книга вашего покорного слуги, в которой рассматривается создание программ для ОС Linux на системном уровне, а также описываются системные функции и API библиотеки `libc`. Особое внимание уделяется специфичным для Linux особенностям программирования.
2. Stevens, W.R. and S. Rago. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 2008. Отличное, если не самое полное, описание интерфейса системных функций Unix.
3. Stevens W. Richard. *UNIX Network Programming, Volume 1*. Prentice Hall, 2004. Классический учебник по API сокетов операционной системы Unix.

Книги по программированию на языке C

Как и большинство других программ для ОС Unix, ядро Linux написано на языке C. В двух перечисленных ниже книгах язык C описан очень подробно.

1. Брайан У. Керниган, Деннис М. Ритчи. *Язык программирования C, 2-е издание*, ISBN 978-5-8459-0891-9. Пер. с англ., ИД "Вильямс", 2012. Самая авторитетная книга по языку C, написанная самим автором этого языка и его коллегой.
2. van der Linden, Peter. *Expert C Programming*. Prentice Hall, 1994. Очень подробное описание некоторых непонятных моментов программирования на языке C. У автора этой книги отличное чувство юмора.

Другие работы

Ниже приведена подборка других книг, прямо не связанных с технологией разработки операционных систем, однако в них обсуждаются темы, которые, без сомнения, будут вам интересны.

1. Hofstadter, Douglas. *Gцdel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1999. Глубокий и беспристрастный взгляд на проблемы человечества сквозь призму исследований различных предметов, включая кибернетику.

2. Дональд Э. Кнут. *Искусство программирования, том 1. Основные алгоритмы, 3-е издание*. ISBN 978-5-8459-0080-7. Пер. с англ. ИД "Вильямс", 2000. Бесценный том, посвященный основным алгоритмам информатики, включая алгоритмы управления памятью, используемые в ядре.

Веб-сайты

1. Kernel.org. Официальное хранилище исходного кода ядра Linux. Кроме того, здесь же хранится большое количество заплат ядра, сделанных разработчиками. `http://www.kernel.org`.
2. Linux Weekly News. Великолепный новостной сайт, посвященный толковым и подробным комментариям, посвященным недельным новостям, произошедшим в мире Linux, включая ядро. Настоятельно рекомендуется! `http://www.lwn.net`.
3. OS News. Новости в мире операционных систем, включая свежие статьи, интервью и обзоры. `http://www.osnews.com`.

Предметный указатель

A

ABI, 115; 123; 421; 454
ACPI, 415
Andrew Morton, 35
API, 25; 104
Application
    Binary Interface, 115; 421
    Programming Interface, 104
Application Binary Interface, 123
Associative array, 134
ATM, 202
Atomic, 433
    operations, 217
Atomic64, 222
Automated Teller Machine, 202

B

Backing store, 380
blkdevs, 395
BogoMIPS, 275
Bottom half, 150
BSD process accounting, 67
BST, 138
bzip2, 38

C

Cache
    coloring, 53
    hit, 380
    miss, 380
cdevs, 396
CFS, 75
CMOS, 264
Concurrent programming, 63
Copy-on-write, 60
COW, 60
cpuset, 288
CVS, 37

D

Deadlock, 210
Dennis Ritchie, 25
Dentry, 325
Dequeue, 131
Designated initializers, 45
Device node, 117
Digital Alpha, 445
Dirty, 381
    list, 381
DMA, 281

E

EDD, 415
Edsger Wybe Dijkstra, 235
EFI, 415
Enqueue, 131
Exceptions, 148

F

FIFO, 130
Filesystem, 314
Free list, 294

G

gcc, 45
    встраиваемые функции, 45
    встроенный ассемблер, 46
    комментирование ветвлений, 46
Git, 37
GNOME, 29
GNU, 45
    C, 45
    Compiler Collection, 45
GPL, 28
gzip, 38

H

Hard real-time, 100

I

Idle process, 31
Incremental patch, 38
Initscripts, 59
Inline functions, 45
Interactive processes, 75
Interprocess communication, 27; 32
Interrupt, 30; 147
    context, 149
    controller, 148
    handler, 147; 149
    safe, 209
    service routine, 149
    synchronous, 148
Interrupt context, 31
Interrupt handler, 31
IPC, 27; 32
IRQ, 148
ISO
    C90, 45
    C99, 45

J

jiffies, 244
Jiffy, 259

K

Ken Thompson, 25
Kernel
    thread, 65
Kernel preemption, 32
Key, 88
kfifo, 130

L

Laptop mode, 390
Least recently used, 381
LIFO, 129
Lightweight process, 63
Linus Elevator, 351
Linus Torvalds, 17; 27
Linux, 25
Linux Kernel Mail List, 461
Lkml, 35; 441; 461
Load balancer, 102
Lock, 206
LRU, 381; 382
LRU/2, 382
LRU/n, 382

M

Map, 134
MATLAB, 76
McCreight, Edward M., 384
Memory
    Management Unit, 279
Memory barrier, 58
Microsoft Windows, 63
Migration threads, 102
Minimum granularity, 84
miscdevs, 396
MMU, 31; 279
Morton, Andrew, 35
Multitasking, 73
Mutex, 238

N

Namespace, 314
Nice-значение, 77

O

Opaque type, 55

P

PAE, 303
Page
    fault, 149
    global directory, 376
    writeback, 379
Patch, 38
PGD, 376
Physical Address Extension, 303
PID, 55
Pidhash, 68
PIT, 265
Polling, 147
POSIX, 104
PowerPC, 56
Priority flag, 44
Processor
    affinity, 100
Processor slice, 78

Q

Quantum, 78

R

Race
    condition, 93; 202
Radix trees, 143
RAID, 348
Rate limiting, 438
Rbtree, 87; 140
Read memory barrier, 248
Ritchie, Dennis, 25
RSDL, 75
Runnable, 56
    process, 73
runqueue, 56

S

Scatter-gather I/O, 346
Scheduler, 48
    entity structure, 85
Segmentation Fault, 360
Seq lock, 243
Sequential lock, 243
Slab, 296
    allocator, 53; 294
    layer, 294
SMP, 33; 40; 458
SMP-safe, 209
Soft real-time, 100
Softirq, 172
Spin lock, 226
STREAMS, 33
Sun Solaris, 63
Supervisor, 29
Synchronization, 202
Syscalls, 105
System call, 29

T

Tarball, 37
Targeted latency, 83
Task, 53
    array, 53
    queue, 173
Task list, 53
Tasklet, 172
Temporal locality, 379
Text section, 51
Thompson, Ken, 25
Thread, 33; 63; 201; 363
    local storage, 65
Threaded tree, 362
Timeslice, 74; 78
TLB, 293; 377
TLS, 65
Top half, 150
Torvalds, Linus, 17; 27
Translation Lookaside Buffer,
    293; 377
Trap, 452
Trie, 143
TSC, 265

U

Unix, 25
User preemption, 97
Usermode Linux, 445

V

VFS, 311
Virtual
    memory area, 364
    runtime, 86
Virtual memory, 26
VM, 26
VMA, 364

W

Wait queue, 93
Work queue, 173
Write-through cache, 380

X

X Windows, 28

Y

Yielding, 74

Z

Zombie, 52

A

Адресное пространство
  линейное, 359
  процесса, 359; 361
  структура
    address_space, 346
Алгоритм, 143
  масштабируемость, 143
  наименьшего использования, 381
  сложность, 143
Алгоритмическая сложность, 143
Аппаратная платформа, 445
Ассоциативный массив, 134
Атрибуты
  volatile, 274
АЦПУ, 456

Б

Банкомат, 202
Барьеры, 247
  памяти, 58
ББП, 293; 377
Библиотека C, 104
Бинарное дерево
  базисное, 387
Блок, 341; 342
Блокировки, 206
  dcache_lock, 329
  dentry->d_lock, 329
  mmlist_lock, 363
  page_table_lock, 376
  s_lock, 319; 320
  self-deadlock, 211
  xtime_lock, 265; 268
  большая блокировка ядра (BKL), 242
    lock_kernel(), 242
    unlock_kernel(), 242
  в обработчиках нижних половин, 196
  взаимоблокировка типа ABBA, 211
  защита данных, 228
  порядок захвата, 212
  последовательные, 243; 268
    захват на запись, 244
    проверка чтения, 244
  рекурсивность, 227
  рекурсивные, 211
  самоблокировка, 211
  семафоры, 227; 233
    down(), 235; 236
    down_interruptible(), 236
    up(), 235
    бинарные, 235
    захват, 237
    инициализация, 235; 237
    инкремент, 235
    использование, 241
    мьютекс, 235
    освобождение, 237
    проверка, 235
    счетчик, 234
    чтения-записи, 237
  спин-блокировки
    вытесняемость, 245
    использование, 233; 240
    отладка, 229
    по записи, 230
    по чтению, 230
  структуры
    completion, 241
    rw_semaphore, 237
  условные переменные, 241
Блочное устройство, 395
  очередь запросов, 349
  сегмент, 346
  структура
    bio, 346
    buffer_head, 344; 349
    request, 349
    request_queue, 349
Большое “O”, 144
Буфер, 343
  быстрого преобразования адреса (TLB), 377
  быстрой переадресации, 293
  заголовок, 344
  кольцевой, 429
  сообщений ядра, 429
  состояние, 344

B

Ввод-вывод
  распределенный, 346
Верхняя половина, 150
Взаимоблокировка, 210
Виртуальная память, 26
Виртуальный ресурс
  память, 52; 359
  процессор, 52
Внешняя память, 380
Возможности
  CAP_SYS_, 112
  CAP_SYS_NICE, 112
Время
  абсолютное (wall time), 253; 267
  начало эпохи (epoch), 268
  операции
    gettimeofday(), 269
    settimeofday(), 269
  относительное, 253
  часовой пояс (time zone), 269
Выделенные инициализаторы, 45
Выравнивание, 451
  естественное, 451
  массивов, 452
  нестандартных типов данных, 452
  объединений, 452
  переменных, 451
  полей структур, 453
  проблемы, 452
  структур, 452
Вытеснение, 74
Вытесняемость
  запрещение, 246

Г

Головка, 343
Гранулярность
  cource, 213
  fine, 213
  минимальная, 84
  уровень
    крупных структурных единиц, 213

мелких структурных единиц, 213
Граф, 138

Д

Датацентры, 28
Двоичное дерево, 138; 362
красно-черное, 368
поиска, 138
Двоичный интерфейс приложений, 115; 123
Дейкстра, Эдсгер Вибе, 235
Демоны
klogd, 429
ksoftirqd, 184
pdflush, 346
syslogd, 429
Деннис Ритчи, 25
Дерево
высота, 139
глубина узла, 139
двоичное, 138
поиска, 138
красно-черное, 87; 139
реализация, 140
листья, 139
нагруженное, 143
остатков, 143
переплетенное, 362
префиксное, 143
самобалансирующееся двоичное поиска, 87; 135; 139
сбалансированное двоичное поиска, 139
Дескриптор
процесса, 53
Директивы
bool, 403
default, 404
depends, 403
help, 403
if, 404
ifdef, 468
likely(), 46
select, 403
tristate, 403
unlikely(), 46
препроцессора
HZ, 255
Драйвер, 149

Ж

Жесткий
диск, 341
режим реального времени, 100

З

Завершение процесса, 66
Загрузчики
grub, 43
LILO, 43
Задачи, 53
flush, 65
ksoftirqd, 65
kthreadd, 65
Задержка выполнения, 273
Закон Мура, 393
Заплата, 470
инкрементальная, 38
применение, 39; 471
утилиты
diff, 470
diffstat, 471
patch, 470
Запросы
на прерывание, 148
Значение
-EFAULT, 111
-ENOSYS, 107; 108

И

Идеальная многозадачность, 83
Идентификатор процесса, 55
Инициализаторы
выделенные, 45
Инициализированные переменные, 360
Инсталляция
модулей, 43; 401
ядра, 42
Интерактивные процессы, 75
Интерфейс
netlink, 422
прикладных программ, 25
Исключительная ситуация, 148

К

Каталоги, 314
/proc, 41
arch, 39
arch/x86, 445
block, 39
crypto, 39
Documentation, 39
drivers, 39
firmware, 39
fs, 39
fs/proc, 163
include, 39
init, 39
ipc, 39
kernel, 39
lib, 39
mm, 39
net, 39
samples, 39
scripts, 39
security, 39
sound, 39
tools, 39
usr, 39
virt, 39
Квант, 78
времени, 74; 78
процессора, 78
Кен Томпсон, 25
Кеш
L1, 379
L2, 379
буферный, 388
вытеснение данных, 381
дисковый, 379
страничный, 379
Кеширование
при записи, 380
стратегии
без записи, 380
отложенная, 380
со сквозной записью, 380
Кластер, 343
Ключ, 88
Ключевые слова
inline, 45
static, 45
Код
устойчивый

к мультипрограммированию, 209
к прерывания, 209
к симметричной обработке, 209
ядра
дерево, 39
заплаты, 38
конфигурация, 40
make defconfig, 41
make gconfig, 41
make oldconfig, 41
файл .config, 41
получение, 37
сборка, 40
параллельная, 42
Код ядра
инсталляция, 38
конфигурация
make config, 40
Команды
cli, 164
CMPXCHG, 207
csh, 26
int, 107
LTR, 56
ps, 57
ps -ef, 65
ps -el, 77
sti, 164
sysenter, 108
vi, 26
Компании
Digital, 26
Hewlett Packard, 26
IBM, 26
Sequent, 26
SGI, 26
Sun, 26
Компиляторы
GNU C, 45
Intel C, 45
Компьютеры
PDP-7, 25
Константы
__BIG_ENDIAN, 456
__LITTLE_ENDIAN, 456
BITS_PER_LONG, 446
HZ, 456
PAGE_SHIFT, 458
PAGE_SIZE, 458
USER_HZ, 263
Контейнеры
std
map, 135
Контекст
неделимый, 149
переключение, 96
Контекст прерывания, 31
Контроллер
прерываний, 148
Контроль версий, 37
Конфигурация
CONFIG_PREEMPT, 210
CONFIG_SMP, 209
Конфигурация ядра
CONFIG_DEBUG_LOCK_ALLOC, 229
CONFIG_DEBUG_SPINLOCK, 229
Конфликт, 93
из-за доступа к ресурсу, 202
при блокировках, 213
Копирование при записи, 60

Л

Линус Торвальдс, 17; 27
Листья, 139
Лицензия
GNU, 28
Локализация
временная, 379

М

Мак-Крейг, Эдвард М., 384
Макросы
_syscalln(), 116
alloc_percpu(), 307
allocate_mm(), 363
BUILD_BUG_ON(), 434
container_of(), 123
current, 55; 58; 158
DECLARE_TASKLET(), 182
DECLARE_TASKLET_DISABLED(), 182; 184
DECLARE_WAITQUEUE(), 93
DECLARE_WORK(), 192
DEFINE_WAIT(), 94
EXPORT_SYMBOL(), 406
EXPORT_SYMBOL_GPL(), 406
for_each_process(), 59
get_cpu_var(), 308
in_interrupt(), 166; 167
in_irq(), 166
INIT_WORK(), 192
irqs_disabled(), 166
list_entry(), 127
list_for_each(), 127
list_for_each_entry(), 128
list_for_each_entry_reverse(), 129
list_for_each_entry_safe(), 129
list_for_each_entry_safe_reverse(), 130
MODULE_AUTHOR(), 398
MODULE_DESCRIPTION(), 398
module_exit(), 398
module_init(), 397
MODULE_LICENSE(), 398; 407
module_param(), 405
module_param_array_named(), 406
module_param_named(), 405
module_param_string(), 405
MODULE_PARM_DESC(), 406
next_task(), 59
prev_task(), 59
put_cpu_ptr(), 308
time_after(), 262
time_after_eq(), 262
time_before(), 262
time_before_eq(), 262
Массив
dentry_hashtable, 327
ассоциативный, 134
задач, 53
Масштабируемость, 213
Машинное слово, 446
Машинный код, 360
Межпроцессное взаимодействие, 32
Метаданные, 315
Методы
show(), 419
store(), 419
Механизм порождения процессов, 60
Миграционные потоки, 102
Многозадачность
идеальная, 83
кооперативная, 74

приоритетная, 74
Многопоточность, 33
Множества
kset, 410
Модуль
загружаемый, 396
insmod, 401
Makefile, 399
modprobe, 402
module_init(), 397
MODULE_LICENSE(), 398; 407
module_param(), 405
module_param_array_named(), 406
MODULE_PARM_DESC(), 406
rmmod, 401
зависимости, 401
инсталляция, 401
конфигурация, 402
параметры, 404
разработка, 397
сборка, 398
экспорт символов, 406
Монтирование, 325
Мортон, Эндрю, 35
Мультипрограммные ОС, 73
Мультипрограммный режим работы, 208
Мьютекс, 238
определение, 239
Мягкий режим реального времени, 100

H

Неделимое выполнение кода, 202
Неделимость, 433
Неделимые операции, 217
битовые, 223
целочисленные, 218
Нижние половины, 150
bottom half, 172
local_bh_disable(), 197
softirq, 173
запрещение обработки, 197
интерфейс BH, 172
интерфейсы BH, 186
отложенное прерывание, 172; 208; 230
вытеснение, 175
генерация, 175
демон ksoftirqd, 176
демоны ksoftirqd, 185
индекс, 177
использование, 177
тасклет, 179
спин-блокировки, 230
структуры
tasklet_struct, 182
тасклет, 172; 208; 230
планирование, 180; 183
создание, 182

O

Область памяти, 360; 364
деревья, 368
диапазон адресов, 365
операции, 367
права доступа, 365; 373
списки, 368
структура vm_operations_struct, 367
флаги, 365
Обработчик прерывания, 31
Объекты
dentry_operations, 316
file_operations, 316
inode_operations, 316
kobject, 408
декларация, 412
захват, 412
инициализация, 412
освобождение, 412
счетчик ссылок, 412
super_operations, 316
Операторы
switch, 463
typedef, 449; 467
unlikely(), 434
Операции
неделимые, 217
Операционная система
AIX, 26
Darwin, 26
DYNIX/ptx, 26
FreeBSD, 26
HP-UX, 26
IRIX, 26
Linux, 27
Multics , 25
NetBSD, 26
OpenBSD, 26
Solaris, 26
SunOS, 26
Tru64, 26
Unix, 25
1BSD, 26
2BSD, 26
3BSD, 26
4.4BSD, 26
4BSD, 26
AT&T, 26
BSD, 26
System V6, 25
особенности, 26
определение, 29
с виртуальной памятью, 359
Опции
CONFIG_IKCONFIG_PROC, 41
CONFIG_SMP, 40
конфигурации, 40
Отладка, 17; 425
BUG(), 433
BUG_ON(), 433; 434
dump_stack(), 434
исследование и тестирование, 437
клавиша <SysRq>, 434
объявление об ошибках, 433
ограничение частоты событий, 438
параметры конфигурации, 433
утилита ksymoops, 431
функция kallsyms, 432
Отладчики, 435
gdb, 436
kgdb, 436
Отложенная запись, 379
Отложенное действие
work queue, 187
демон keventd, 194
использование, 191
ожидание завершения, 193
очереди заданий, 194
очереди задач, 172
рабочий поток, 188
реализация, 188
создание, 193
структура

work_struct, 189
Отложенные прерывания, 174
обработчик, 175
Отображение, 134
анонимное, 361; 373
области ввода-вывода, 367
совместно используемое, 366
страницы памяти, заполненной нулями, 360
файла, 370; 373
частное, 367
Очереди, 130
kfifo, 130
выборка, 132
задач, 173
ожидания, 93
add_wait_queue(), 94
операции, 131
отложенных действий, 173
постановка, 132
процессов, 56
размер, 133
сброс, 133
создание, 132
удаление, 133
Ошибки
-EAGAIN, 136
-EBUSY, 152
-ENOSPC, 136
oops, 47
использование после удаления, 130

П

Пакеты
procps, 369
Память
MMU, Memory Management Unit, 279
блочный распределитель, 296
NUMA, 295
верхняя
переносимость, 458
виртуальная, 280
выделение
gfp_mask, 287
kmalloc(), 286
vmalloc(), 292
блочный распределитель, 294
виртуально непрерывный участок, 293
временное отображение, 304
контекст процесса, 291
модификаторы зоны, 288
модификаторы операций, 287
низкоуровневое, 284
обработчик прерывания, 291
отложенное прерывание, 292
отображение верхней памяти, 303
постоянное отображение, 303
связанная с процессором (per-CPU), 305
списки свободных ресурсов, 294
стек
ядра, 303
тасклет, 292
физически непрерывный участок, 284; 286; 293
флаги типов, 289
дескриптор, 361
выделение, 363
счетчик использования, 362
удаление, 363
защита, 47
зоны, 281
ZONE_DMA, 281
ZONE_DMA32, 281
ZONE_HIGHMEM, 281
ZONE_NORMAL, 281
верхняя память, 280
верхняя память (high memory), 282
нижняя память (low memory), 282
кеширование, 308
область, 360
освобождение
kfree(), 292
прямой доступ (ПДП, DMA), 281
связанная с процессором
get_cpu_ptr(), 308
get_cpu_var(), 306
put_cpu_ptr(), 308
стек
процесса, 360
ядра, 48; 302
страница
вытеснение, 280
гигантская, 366
данные буфера, 345
размер, 457
флаги, 280
страничный кеш, 281
структура
kmem_cache, 297
mm_struct, 361
page, 347
vm_area_struct, 364; 367
физическая, 280
Параллелизм
pseudo-concurrency, 208
защита данных, 209
истинный, 208
конкурентный доступ, 201
критический участок, 202
многопоточность, 63
многопроцессорность, 458
псевдопараллелизм, 208
симметричная многопроцессорная обработка, 208
симметричная многопроцессорная обработка, 201
состояние конкуренции за ресурс, 48
Параметры
конфигурации
CONFIG_HZ, 259
CONFIG_DEBUG_KERNEL, 433
CONFIG_DEBUG_MUTEXES, 240
CONFIG_KALLSYMS, 432
CONFIG_KALLSYMS_EXTRA_PASS, 432
ПДП, 281
Переменные
dirty_background_ratio, 390
dirty_expire_interval, 390
dirty_ratio, 390
dirty_writeback_interval, 390
jiffies, 260; 456

переполнение, 261
jiffies_64, 245; 260
laptop_mode, 390
loops_per_jiffy, 275
need_resched, 98
preempt_count, 98
se, 86
USER_HZ, 263
vruntime, 86
xtime, 264; 268
Переносимость, 48; 443
Переплетенное дерево, 362
Планирование
SCHED_FIFO, 99
SCHED_NORMAL, 99
SCHED_RR, 99
в Unix, 81
выбор процесса, 87
на основе приоритетов, 77
справедливое, 83
алгоритм, 87
учет времени, 85
Планировщик, 48; 73
CFS, 75; 80; 83
RSDL, 75
ввода-вывода, 350
CFQ, 356
deadline, 353
Noop, 357
задержки обслуживания, 353
лифтовый алгоритм, 351
объединение, 350
прогнозирующий, 355
с лимитом по времени, 353
с отсутствием операций, 357
с полностью равноправными очередями, 356
слияние, 350
сортировка, 350
классы, 80
очередь выполнения, 213
реального времени, 99
стратегия, 76
точка входа, 91
функции управления, 100
Планируемая задержка, 83
Пользователи
root, 112
Порядок
выполнения
барьеры, 247; 458
barrier(), 250
smp_mb(), 249
smp_rmb(), 249
smp_wmb(), 249
компилятора, 250
переносимость, 458
следования
big-endian, 454
little-endian, 454
байтов, 454
определение, 455
обратный, 454
прямой, 454
Порядок выполнения
барьеры
mb(), 248
read_barrier_depends(), 248
rmb(), 248
wmb(), 248
для чтения памяти, 248
по записи, 248
Поток, 363
Потоки, 63; 201; 363
bdflush, 391
kupdated, 391
pdflush, 391
выполнения кода, 51
команд, 201
синхронизатора, 388; 389
создание, 64
ядра, 65; 364
Права использования
CAP_SYS_TIME, 269
Прерывания, 30; 147; 208
верхняя половина, 150; 170
запрещение, 164
из-за отсутствия страницы, 149
контекст, 149; 158
нижняя половина, 150; 170
обработчик, 149
RTC, 156
общих запросов, 155
обработчики, 147
описание обработчика, 154
освобождение обработчика, 153
отложенные, 174
реализация, 175
пример обработчика, 153
разрешение, 164
реализация обработки, 159
регистрация обработчика, 150
реентерабельность обработчика, 155
синхронные, 148
стек, 159
управление, 163
флаги обработчика, 151
Приоритетное прерывание обслуживания, 74
Приоритеты
реального времени, 77
Программы
sshkeygen, 76
syslog, 44
Пространство
задачи, 29
пользователя, 58
ядра, 29; 58
Пространство имен, 314
Процесс, 51; 53
I/O-bound, 76
init, 58; 69
parent, 59
processor-bound, 76
SCHED_NORMAL, 80
SCHED_OTHER, 80
адресное пространство, 58
активизация, 95
виртуальное время выполнения, 86
вытеснение пространства пользователя, 97
готовый к выполнению, 73
дескриптор, 53; 337
добавление в дерево, 88
завершение, 66
идентификатор, 55
иерархия, 58
интерактивный, 75
квант времени, 78
контекст, 58; 113
корневой каталог, 338
макрос current, 55
операции
wake_up(), 278
определение, 51
ориентированный
на ввод-вывод, 76
на вычисления, 76

передача управления, 74
переназначение
родительского процесса, 68
порожденный, 52
приоритет, 77
пространство имен, 338
родительский, 52
родственный, 59
с быстрым переключением контекста, 63
создание, 60
состояние, 56
__TASK_STOPPED, 57
__TASK_TRACED, 57
EXIT_ZOMBIE, 67
TASK_INTERRUPTIBLE, 56; 93; 276
TASK_RUNNING , 56
TASK_UNINTERRUPTIBLE, 56; 93; 276
зомби, 52
ожидания, 208
структура
task_struct, 53
текущий каталог, 338
удаление из дерева, 90
Процессор
Intel 80386, 27
Пул энтропии ядра, 151
Пустые поля структур, 453

P

Раскрашивание стека, 53
Распределение памяти блочного типа, 53
Реентерабельность, 155
Режим ноутбука, 390
Ритчи, Деннис, 25

C

Сборка
модулей, 398
Связанный список, 119; 369
головной элемент, 121
двунаправленный, 120
добавление узла, 124
обход
в обратном направлении, 129
с удалением, 129
обход узлов, 127
однонаправленный, 120
определение, 123
перемещение по элементам, 121
перемещение узла, 126
работа, 124
реализация, 122
соединение узлов, 126
удаление узла, 125
циклический, 120
Сегмент
bss, 360
данных, 51; 360
кода, 51; 360
Сектор, 342
размер, 342
Семафоры
реализация, 235
со счетчиком, 235
счетный, 235
Сетевые устройства, 396
Сигналы
SIGKILL, 57
SIGSEGV, 47
SIGTSTP, 58
SIGTTIN, 58
SIGTTOU, 58
Символьные устройства, 396
Симметричная многопроцессорная обработка, 33; 40
Синхронизация, 202
Система, 29
контроля версий, 37
Системные функции, 105
Системный вызов, 29
регистрация, 114
Скрытый тип, 55
Смешанные устройства, 396
Сокет, 26
Сообщение
Oops, 430
уровень важности, 428
Сообщество разработчиков, 35; 461
maintainers, 469
ответственные разработчики, 469
отправка сообщений об ошибках, 470
список рассылки, 35; 461
Состояние
ожидания, 56
паники, 430
Спин-блокировка, 226
Список
задач, 53
связанный, 119
Справедливое планирование задач, 83
Средняя загруженность системы (load average), 266
Средства
межпроцессного взаимодействия, 27
Стабилизатор нагрузки, 102
Стандарты
ISO
C90, 45
C99, 45
Стек
прерывания, 159
указатель, 54
Стиль написания исходного кода, 462
ifdef, 468
indent, 469
typedef, 467
длина строки, 465
инициализация структур, 468
комментарии, 466
отступы, 462
присвоение имен, 466
фигурные скобки, 464
функции, 466
Страничная память, 31
Страничный кеш, 379
структура
address_space, 383
Стратегия
с двумя списками, 382
Структуры
address_space, 346; 384
address_space_operations, 385
attribute, 409; 418
bio, 346; 347
bio_vec, 347
buffer_head, 344; 349
cdev, 409
completion, 241
cpu_workqueue_struct, 189; 190

dentry, 326
dentry_operations, 328
device, 155
file, 316; 330
file_operations, 331
file_system_type, 316; 335; 336
files_struct, 337; 338
fs_struct, 316; 337; 338
idr, 135
  аннулирование, 137
  выделение UID, 135
  инициализация, 135
  поиск UID, 137
  удаление UID, 137
inode, 295; 296; 383
inotify_watch, 128; 135
k_itimer, 135
kmem_cache, 297
kobj_type, 409; 411; 418
kobject, 408; 410
kref, 408; 413
kset, 408; 410
ktype, 408
list_head, 123; 124; 127
mm_struct, 361; 363
mutex, 239
nameidata, 324
namespace, 337; 338
page, 141; 280; 347
pt_regs, 160
rb_node, 140
rb_root, 140
request, 349
request_queue, 349
rtc_callback, 157
rw_semaphore, 237
sched_entity, 85
sched_param, 101
semaphore, 235
softirq_action, 175
statfs, 320
super_block, 317
super_operations, 318
sysfs_dirent, 408
sysfs_ops, 409; 418
task_struct, 53; 55; 61; 97; 295; 300; 363
  распределение памяти, 53
tasklet_hi_vec, 180
tasklet_struct, 180; 183; 184
tasklet_vec, 180
thread_info, 54; 55; 61; 97; 303
  определение, 54
timer_list, 270
timespec, 267
vfsmount, 316; 336
vm_area_struct, 364; 367; 370; 371; 384
vm_operations_struct, 367
work_struct, 189; 191
workqueue_struct, 189; 190
данных, 119
данных kset, 411
объекта планировщика, 85
структура
  slab, 297
Суперблок, 315; 317
Супервизор, 29
Счетный семафор, 235
Счетчик
  preempt_count, 98; 198
  команд, 51

T

Таблица
  страниц, 375
    PGD, 376
    PMD, 376
    PTE, 376
    глобальный каталог, 376
    каталог среднего уровня, 376
    управление, 377
    уровни, 376
Таблицы
  отображения, 134
Таймер
  APIC, 265
  декрементный счетчик, 264
  динамический, 254; 269
    обработчик, 272
  задержки
    BogoMIPS, 275
    mdelay(), 275
    udelay(), 275
    короткие, 274
    очередь ожидания, 278
    с помощью цикла, 273
  константа
    USER_HZ, 263
  операции
    del_timer_sync(), 271
    mod_timer(), 271
    schedule_timeout(), 276
  переменная
    jiffies, 259; 271
    jiffies_64, 261
    xtime, 265
  переполнение, 261
  прерывание, 254; 265
  программируемый интервальный (PIT), 265
  системный, 253; 264
    обработчик прерывания, 265
  структура
    timer_list, 270
    timespec, 268
  счетчик временных меток (TSC), 265
  точность работы, 256
  частота импульсов (tick rate), 255
  часы реального времени, 156
  часы реального времени (real-time clock, RTC), 264
  ядра, 173; 269
Таймеры
  использование, 270
Тасклеты, 179
  использование, 182
  объявление, 182
  реализация, 179
Типы
  atomic_t, 218
  atomic64_t, 222
  gfp_t, 284; 287
  ktype, 409
  slab, 297
  wait_queue_head_t, 93
Типы данных
  __u32, 451
  atomic_t, 449
  char, 451
  int, 448
  long, 448
  pid_t, 449
  s16, 450
  s32, 450
  s64, 450
  s8, 450
  u16, 450
  u32, 450

u64, 450
u8, 450
оператор typedef, 449
размер указателя, 448
с явным указанием размера, 450
скрытые, 449
Томпсон, Кен, 25
Торвальдс, Линус, 17; 27
Точка монтирования, 325
Трассировка, 57
вызовов функций, 430

У

Указатель стека, 54
Уменьшение числа выводимых сообщений, 42
Упреждающее чтение, 367
Уровень блочного ввода-вывода, 342
Устройства
/dev/null, 42
блочные, 341; 395
сетевые, 396
символьные, 341; 396
смешанные, 396
унифицированная модель представления, 407
Утилиты
ccache, 42
diff, 470
diffstat, 471
distcc, 42
gdb, 436
git, 440
indent, 469
insmod, 401
ksymoops, 432
modprobe, 402
patch, 470
pmap, 369
rmmod, 401

Ф

Файл, 314
Файловая система, 314
/proc, 369; 414
ext3, 385
procfs, 162
sysfs, 404; 405; 408; 414
HAL, 417
kobject_add(), 417
kobject_del(), 417
sysfs_create_file(), 419
добавление и удаление объектов, 417
добавление файлов, 418
операция show(), 419
операция store(), 419
соглашения, 420
файлы, 418
блок, 342
виртуальная (VFS), 311
диспетчер логических томов (LVM), 320
каталог (directory), 314
монтирование (mount), 336
флаги mnt_flags, 336
объектная ориентированность, 315
операции
файл
aio_fsync(), 333
aio_read(), 332
aio_write(), 332
check_flags(), 334
compat_ioctl(), 333
fasync(), 333
flock(), 334
flush(), 333
fsync(), 333
get_unmapped_area(), 334
ioctl(), 332
llseek(), 332
lock(), 333
mmap(), 333
open(), 333
poll(), 332
read(), 332
readdir(), 332
readv(), 334
release(), 333
sendfile(), 334
sendpage(), 334
unlocked_ioctl(), 332
write(), 332
writev(), 334
файловый индекс
create(), 323
follow_link(), 324
getattr(), 324
getxattr(), 325
link, 323
listxattr(), 325
lookup(), 323
mkdir(), 323
mknod(), 323
permission(), 324
put_link(), 324
readlink(), 324
removexattr(), 325
rename(), 324
rmdir(), 323
setattr(), 324
setxattr(), 325
symlink(), 323
truncate(), 324
unlink(), 323
элемент каталога
d_compare(), 329
d_delete(), 329
d_hash(), 329
d_iput(), 329
d_release(), 329
d_revalidate(), 329
особенности Unix, 314
пространство имен, 62; 338
пространство имен (namespace), 314
путь (path), 314
системные вызовы, 312
структура
dentry, 326
dentry_operations, 316; 328
file, 316; 330
file_operations, 316; 331
file_system_type, 316; 336
files_struct, 337
fs_struct, 316; 337
inode, 321
inode_operations, 316; 322
namespace, 337
super_block, 317
super_operations, 316; 318
vfsmount, 316; 336
суперблок (superblock), 315; 317
управляющий блок (control block), 317
файл (file), 314; 316; 329
файловый индекс (inode), 315; 320

кеш (icache), 328
элемент каталога (dentry), 314; 316; 325
LRU, 327
кеш (dcache), 327
состояния, 326
хеш-таблица, 327
Файлы
/dev/full, 396
/dev/mem, 396
/dev/null, 396
/dev/random, 396
/dev/urandom, 396
/dev/zero, 396
/etc/syslog.conf, 430
/proc/cpuinfo, 275
/proc/interrupts, 162
/proc/kmsg, 429
/proc/sys/kernel/pid_max, 55
/var/log/messages, 430
<asm/atomic.h>, 219
<asm/bitops.h>, 223
<asm/byteorder.h>, 456
<asm/irq.h>, 163
<asm/mman.h>, 373
<asm/page.h>, 377; 458
<asm/param.h>, 255; 457
<asm/percpu.h>, 306
<asm/semaphore.h>, 235
<asm/softirq.h>, 198
<asm/spinlock.h>, 227
<asm/system.h>, 163; 166
<asm/thread_info.h>, 54
<asm/types.h>, 450
<asm/unistd.h>, 114; 116
<asm-generic/kmap_types.h>, 304
<linux/bio.h>, 346; 347
<linux/blkdev.h>, 349
<linux/buffer_head.h>, 344
<linux/capability.h>, 113
<linux/cdev.h>, 409
<linux/completion.h>, 241
<linux/dcache.h>, 326; 328
<linux/fdtable.h>, 337
<linux/fs.h>, 317; 318; 322; 330; 335; 384
<linux/fs_struct.h>, 337
<linux/gfp.h>, 284; 287
<linux/hardirq.h>, 166
<linux/interrupt.h>, 151; 175; 177; 180; 182; 187
<linux/jiffies.h>, 260; 274
<linux/kernel.h>, 428
<linux/kfifo.h>, 130
<linux/kobject.h>, 408; 412; 413; 417; 420; 422
<linux/kref.h>, 413; 414
<linux/kthread.h>, 65
<linux/list.h>, 122; 124; 126; 130
<linux/mm.h>, 371; 372; 373; 375
<linux/mm_types.h>, 280; 361; 364
<linux/mmzone.h>, 282; 283
<linux/mnt_namespace.h>, 338
<linux/module.h>, 406
<linux/mount.h>, 336
<linux/page-flags.h>, 280
<linux/percpu.h>, 306; 307
<linux/radix-tree.h>, 387
<linux/rbtree.h>, 140
<linux/sched.h>, 53; 58; 64; 363
<linux/slab.h>, 286; 287; 292
<linux/smp_lock.h>, 243
<linux/spinlock.h>, 227
<linux/sysfs.h>, 418
<linux/threads.h>, 55
<linux/timer.h>, 270
<linux/types.h>, 219; 450
<linux/vmalloc.h>, 293
<linux/workqueue.h>, 189
arch/i386/kernel/syscall_64.c, 107
arch/x86/kernel/entry_32.S, 162
arch/x86/kernel/entry_64.S, 162
arch/x86/kernel/irq.c, 162
arch/x86/kernel/vmlinux.lds.S, 261
asm/mmu_context.h, 96
asm/unistd.h, 115
block/as-iosched.c, 356
block/cfq-iosched.c, 357
block/deadline-iosched.c, 355
block/elevator.c, 352
block/noop-iosched.c, 357
config.gz, 41
COPYING, 39
CREDITS, 39; 469
Documentation/CodingStyle, 462
Documentation/sysrq.txt, 435
drivers/char/rtc.c, 156
drivers/char/sysrq.c, 435
entry.S, 97; 114
entry_64.S, 107
fs/ext3/inode.c, 385
fs/fs-writeback.c, 390
fs/notify/inotify/inotify_user.c, 94
fs/super.c, 318
fs/sysfs/file.c, 420
grub.conf, 43
init/main.c, 275
Kconfig, 402
kernel/exit.c, 66; 363
kernel/fork.c, 241; 300; 363
kernel/irq/handler.c, 160
kernel/kfifo.c, 130
kernel/sched.c, 80; 91; 96; 115; 241; 444
kernel/sched_fair.c, 80; 85; 86; 88
kernel/sched_rt.c, 99
kernel/softirq.c, 174; 175
kernel/sys.c, 115
kernel/time.c, 263; 268
kernel/time/tick-common.c, 245
kernel/time/timekeeping.c, 267
kernel/timer.c, 245; 270
kernel/workqueue.c, 189
lib/kobject.c, 417
lib/kobject_uevent.c, 422
lib/kref.c, 414
lib/radix-tree.c, 387
lib/rbtree.c, 140
linux/sched.h, 85
MAINTAINERS, 39; 469
Makefile, 39; 42
mm/backing-dev.c, 390
mm/filemap.c, 386
mm/mmap.c, 371; 372; 375
mm/page_alloc.c, 284
mm/page-writeback.c, 390
mm/slab.c, 297; 306
mm/vmalloc.c, 293
net/core/dev.c, 178
scripts/Lindent, 469
string.c, 44
string.h, 44
sys.c, 114
System.map, 43; 432

заголовочные, 44
исполняемые, 58
сценариев начальной загрузки, 59
устройства, 117
Флаги
CLONE_VM, 363
IRQF_DISABLED, 151; 155; 161; 169; 171
IRQF_SAMPLE_RANDOM, 151; 161
IRQF_SHARED, 152; 155
IRQF_TIMER, 152
need_resched, 97; 162; 258; 274
PF_FORKNOEXEC, 61
PF_SUPERPRIV, 61
SA_INTERRUPT, 151
важности вывода, 44
Функции
__alloc_percpu(), 307
__clone(), 61
__dequeue_entity(), 91
__do_softirq(), 176
__enqueue_entity(), 89
__exit_signal(), 68
__ffs(), 225
__ffz(), 225
__free_pages(), 285
__get_free_page(), 284; 285
__get_free_pages(), 284; 285; 287; 289; 297
__pick_next_entity(), 92
__tasklet_hi_schedule(), 180
__tasklet_schedule(), 180
__test_bit(), 224
__unhash_process(), 68
__update_curr(), 87
access_process_vm(), 368
account_process_tick(), 267
acct_update_integrals(), 67
add_interrupt_randomness(), 161
add_wait_queue(), 94
aio_fsync(), 333
aio_read(), 332
aio_write(), 332
alloc_inode(), 319
alloc_page(), 284; 285
alloc_pages(), 284; 285; 287; 289; 303; 309
alloc_percpu(), 307
alloc_pid(), 61
alloc_super(), 318
atomic_add(), 219; 220
atomic_add_negative(), 220
atomic_add_return(), 220
atomic_dec(), 220
atomic_dec_and_test(), 220; 221
atomic_dec_return(), 220
atomic_inc(), 219; 220
atomic_inc_and_test(), 221
atomic_inc_return(), 220
ATOMIC_INIT(), 220
atomic_read(), 219; 220
atomic_set(), 219; 220
atomic_sub(), 220
atomic_sub_and_test(), 220
atomic_sub_return(), 220
atomic64_add(), 222
atomic64_add_negative(), 223
atomic64_add_return(), 223
atomic64_dec(), 222
atomic64_dec_and_test(), 223
atomic64_dec_return(), 223
atomic64_inc(), 222
atomic64_inc_and_test(), 223
atomic64_inc_return(), 223
ATOMIC64_INIT(), 222
atomic64_read(), 222
atomic64_set(), 222
atomic64_sub(), 222
atomic64_sub_and_test(), 222
atomic64_sub_return(), 223
barrier(), 250
bdi_writeback_all(), 389
bh_action(), 187
bio_get(), 348
bio_put(), 348
bread(), 388
BUG(), 433
BUG_ON(), 433
calc_global_load(), 266
cancel_delayed_work(), 193
capable(), 112
change_bit(), 224
check_flags(), 334
clear_bit(), 224
clear_tsk_need_resched(), 97
cli(), 165
clone(), 52; 61; 64; 338; 363
флаги, 64
close(), 26
compat_ioctl(), 333; 335
complete(), 241; 242
complete(), 241
cond_resched(), 274
context_switch(), 444
copy_flags(), 61
copy_from_user(), 111
copy_mm(), 363
copy_process(), 61; 62
copy_to_user(), 111
cpu_idle(), 431
create(), 316; 323
create_workqueue(), 193
ctime(), 269
current_thread_info(), 55
d_compare(), 316; 329
d_delete(), 316; 329
d_hash(), 327; 329
d_iput(), 329
d_lookup(), 328
d_release(), 329
d_revalidate(), 329
default_idle(), 431
del_timer(), 271; 272
del_timer_sync(), 67; 271; 272
delete_inode(), 319
destroy_inode(), 319
detach_pid(), 68
disable_irq(), 165; 166
disable_irq_nosync(), 165
do_exit(), 66; 67; 69
do_fork(), 61; 63; 301
do_gettimeofday(), 269
do_IRQ(), 160; 162
do_mmap(), 373
do_munmap(), 375
do_softirq(), 176; 179; 181; 186
do_timer(), 266
down(), 235; 236; 237; 238
down_interruptible(), 236; 237
down_read_trylock(), 238
down_trylock(), 236; 237
down_write_trylock(), 238
downgrade_writer(), 238
dump_stack(), 434
dup_task_struct(), 61; 301
early_printk(), 427
enable_irq(), 166
exec(), 52; 60; 61
execle(), 60
execlp(), 60
execv(), 60

execve(), 60
execvp(), 60
exit(), 52; 66; 67
exit_files(), 67
exit_fs(), 67
exit_mm(), 67; 363
exit_notify(), 67; 69
exit_sem(), 67
fasync(), 333
fcntl(), 334
find_first_bit(), 224
find_first_zero_bit(), 224
find_get_page(), 385; 387
find_vma(), 371
find_VMA_intersection(), 372
find_vma_prev(), 372
finish_wait(), 94
flock(), 334
flush(), 333
flush_scheduled_work(), 193; 194
flush_workqueue(), 194
follow_link(), 324
forget_original_parent(), 69
fork(), 27; 52; 60; 61; 62; 64; 88; 363; 377
free_irq(), 153
free_mm(), 364
free_page(), 285
free_pages(), 285; 298
free_percpu(), 307; 308
free_task_struct(), 301
fsync(), 333; 389
ftime(), 269
get_bh(), 345
get_block(), 344
get_cpu(), 246; 306
get_cpu_var(), 306; 308
get_jiffies_64(), 245; 261
get_sb(), 336
get_unmapped_area(), 334
get_user_pages(), 368
get_zeroed_page(), 284
getattr(), 324
getpid(), 105; 106
gettimeofday(), 268
getxattr(), 325
handle_IRQ_event(), 160
idr_destroy(), 137
idr_find(), 137
idr_get_new(), 136
idr_get_new_above(), 136
idr_init(), 135
idr_pre_get(), 136
idr_remove(), 137
idr_remove_all(), 138
init_completion(), 241
init_MUTEX(), 237
init_MUTEX_LOCKED(), 237
init_rwsem(), 237
init_waitqueue_head(), 93
inotify_read(), 94
ioctl(), 109; 117; 332; 334; 420
iput(), 329
jiffies_64_to_clock_t(), 263
jiffies_to_clock_t(), 263
kernel_locked(), 242; 243
kfifo_alloc(), 132
kfifo_avail(), 133
kfifo_in(), 132
kfifo_init(), 132; 133
kfifo_is_empty(), 133
kfifo_is_full(), 133
kfifo_len(), 133
kfifo_out(), 132; 133
kfifo_out_peek(), 133
kfifo_reset(), 133
kfifo_size(), 133
kfree(), 292
kmalloc(), 286; 287; 289; 292; 293; 307; 309; 310
kmap(), 303; 310; 459
kmap_atomic(), 304
kmem_cache(), 298
kmem_cache_alloc(), 300
kmem_cache_destroy(), 299; 300
kmem_cache_free(), 300; 364
kmem_freepages(), 298
kmem_getpages(), 298
kobject_add(), 417
kobject_create(), 412
kobject_create_and_add(), 417
kobject_del(), 417
kobject_get(), 413
kobject_init(), 412
kobject_uevent(), 422
kref_get(), 413
kref_init(), 413
kref_put(), 413; 414
kthread_run(), 66
kthread_stop(), 66
kunmap_atomic(), 305
link(), 316; 323
list_add(), 123; 124
list_add_tail(), 125
list_del(), 125; 126
list_del_init(), 126
list_empty(), 126
list_entry(), 123
list_move(), 126
list_move_tail(), 126
list_splice(), 126
list_splice_init(), 126
listxattr(), 325
llseek(), 332
local_bh_disable(), 197
local_bh_enable(), 197
local_irq_disable(), 164
local_irq_enable(), 164
lock(), 333
lock_kernel(), 242; 243
lookup(), 323
lseek(), 26; 332
madvice(), 367
main(), 66
malloc(), 286; 292
mark_bh(), 187
mask_and_ack_8295A(), 160
mb(), 248; 250
mdelay(), 275; 276
mkdir(), 323
mknod(), 323
mm_release(), 63
mmap(), 333; 367; 371; 373; 374; 375; 384
mmap2(), 374
mmdrop(), 363
mmput(), 363
mod_timer(), 271
munmap(), 375
mutex_init(), 239
mutex_is_locked(), 239
mutex_lock(), 239
mutex_trylock(), 239; 240
mutex_unlock(), 239
ndelay(), 275
need_resched(), 97
net_tx_action(), 179
nice(), 100; 101
notify_change(), 324
open(), 26; 30; 116; 312; 323; 327; 333
page_address(), 284
page_count(), 280
panic(), 301; 434
permission(), 324

perror(), 105
pick_next_entity(), 92
pick_next_task(), 91; 92
poll(), 257; 332
preempt_count(), 246
preempt_disable(), 246
preempt_enable(), 246
preempt_enable_no_resched(), 246
prepare_to_wait(), 94
printf(), 30; 44; 66; 417; 427
printk(), 44; 427
printk_ratelimit(), 439
proc_create(), 152
proc_mkdir(), 152
ptrace(), 57
ptrace_exit_finish(), 70
put_bh(), 345
put_cpu(), 306
put_cpu_ptr(), 308
put_cpu_var(), 306
put_link(), 324
put_task_struct(), 68
queue_delayed_work(), 194
queue_work(), 194
radix_tree_lookup(), 387
raise_softirq(), 179
raise_softirq_irqoff(), 179
rb_erase(), 91
rb_insert_color(), 90; 142
rb_insert_page_cache(), 142
rb_link_node(), 90; 142
rb_next(), 91
read(), 26; 117; 312; 316; 332; 380
read_barrier_depends(), 248; 249; 250
read_inode(), 316
read_lock(), 232
read_lock_irq(), 232
read_lock_irqsave(), 232
read_seqbegin(), 268
read_seqretry(), 268
read_unlock(), 232
read_unlock_irq(), 232
read_unlock_irqrestore(), 232
readdir(), 332
readlink(), 324
readpage(), 385
readv(), 334
reboot(), 112
release(), 333; 414
release_task(), 68
removexattr(), 325
rename(), 324
request_irq(), 151; 153; 154; 155; 163
ret_from_intr(), 162
rmb(), 248; 250; 458
rmdir(), 323
rtc_init(), 156
run_local_timers(), 267; 272
run_timer_softirq(), 272
run_workqueue(), 190
rw_lock_init(), 232
sched_get_priority_max(), 100; 101
sched_get_priority_min(), 100
sched_getaffinity(), 101
sched_getparam(), 100; 101
sched_getscheduler(), 100; 101
sched_rr_get_interval(), 101
sched_setaffinity(), 101
sched_setparam(), 100; 101
sched_setscheduler(), 100; 101
sched_yield(), 101; 102
schedule(), 67; 91; 93; 94; 98; 113; 162; 186; 190; 433
schedule_delayed_work(), 192; 193; 194
schedule_timeout(), 276; 278
  реализация, 276
schedule_work(), 192; 194
scheduler_tick(), 96; 266; 267
select(), 257
sema_init(), 236; 237
sendfile(), 334
sendpage(), 334
set_bit(), 224
set_current_state(), 58
set_task_state(), 58
set_tsk_need_resched(), 97
set_user_nice(), 101
setattr(), 324
SetPageDirty(), 386
settimeofday(), 269
setup_arch(), 427
setxattr(), 325
show_interrupts(), 163
smp_mb(), 250
smp_processor_id(), 306
smp_read_barrier_depends(), 250
smp_rmb(), 250
smp_wmb(), 250
softirq_pending(), 186
spin_is_locked(), 229; 230
spin_lock(), 227; 229
spin_lock_bh(), 230
spin_lock_init(), 229; 230
spin_lock_irq(), 228; 229
spin_lock_irqsave(), 228; 229
spin_trylock(), 229; 230
spin_unlock(), 227; 229
spin_unlock_bh(), 230
spin_unlock_irq(), 228; 230
spin_unlock_irqrestore(), 228; 230
sti(), 165
strcpy(), 30
suser(), 112
switch_mm(), 96; 444
switch_to(), 96; 444
symlink(), 323
sync(), 389
sync_fs(), 316
synchronize_irq(), 166
sys_getpid(), 106
sys_gettimeofday(), 268
sys_ni_syscall(), 107
sys_time(), 269
sys_write(), 313
sysfs_create_file(), 419
sysfs_create_link(), 419
sysfs_remove_file(), 420
sysfs_remove_link(), 420
syslog(), 429
system_call(), 108; 114
tasklet_action(), 181
tasklet_disable(), 183
tasklet_disable_nosync(), 184
tasklet_enable(), 184
tasklet_hi_action(), 181
tasklet_hi_schedule(), 180
tasklet_init, 183
tasklet_kill(), 184
tasklet_schedule(), 180; 183
test_and_change_bit(), 224
test_and_clear_bit(), 224
test_and_set_bit(), 224
test_bit(), 224
threadfn(), 65
tick_periodic(), 265; 267
time(), 269
truncate(), 324
try_to_wake_up(), 95; 96
udelay(), 275
unlink(), 323
unlock_kernel(), 242; 243

unlocked_ioctl(), 332; 334
up(), 235; 236; 237
update_curr(), 86
update_process_times(), 266; 272
update_wall_time(), 266
vfork(), 61; 62; 64; 241
vfree(), 294
vma_link(), 374
vmalloc(), 292
wait(), 52; 68; 70
wait_for_completion(), 241; 242
wait3(), 52
wait4(), 52; 68
waitpid(), 52
wake_up(), 94; 95; 278
wake_up_process(), 66
wakeup_bdflush(), 391
wakeup_flusher_threads(), 389
wb_writeback(), 389; 391
wmb(), 248; 250; 458
work_handler(), 192
worker_thread(), 189
write(), 26; 30; 117; 312; 313; 316; 332; 380
write_lock(), 232
write_lock_irq(), 232
write_lock_irqsave(), 232
write_trylock(), 232
write_unlock(), 232
write_unlock_irq(), 232
write_unlock_irqrestore(), 232
writepage(), 385
writev(), 334
yield(), 102
встраиваемые, 45
системные, 105
  быстродействие, 107
  макросы-оболочки, 116
  номера, 106
  параметры, 109
  проверка параметров, 110
  разработка, 109
  реализация, 109

Х

Хеш-таблица, 135
  страниц, 387
Холостой процесс, 31

Ц

Цилиндр, 343

Ш

Шаблоны
  Producer and consumer, 130; 142
  производитель и потребитель, 130; 142

Э

Эдсгер Вибе Дейкстра, 235
Элемент каталога, 325
Эндрю Мортон, 35

Я

Ядро
  Linux, 33
    версии, 34
    особенности, 33
    получение с помощью Git, 37
    разрабатываемое, 34
    стабильное, 34
    схема нумерации, 34
  Mach, 32
  tainted, 398
  Unix, 31
  вытесняемое, 98
  конфигурирование, 40
  микроядро, 32
  модульное, 396
  монолитное, 32; 396
  особенности, 43
  отладка, 425
  ошибки, 426
  приоритетный мультипрограммный режим, 98
  с приоритетным мультипрограммированием, 32
  уровень событий, 421
  экзоядро, 32

UNIX И LINUX
РУКОВОДСТВО СИСТЕМНОГО АДМИНИСТРАТОРА
4-Е ИЗДАНИЕ

Эви Немет
Гарт Снайдер
Трент Хейн
Бэн Уэйли

www.williamspublishing.com

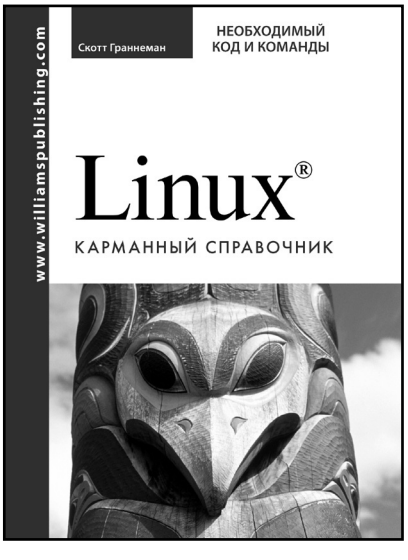
В новом издании всемирно известной книги описываются эффективные методы работы и все аспекты управления системами UNIX и Linux, включая управление памятью, проектирование и управление сетями, электронную почту, веб-хостинг, создание сценариев, управление конфигурациями программного обеспечения, анализ производительности, взаимодействие с системой Windows, виртуализацию, DNS, безопасность, управление провайдерами IT-услуг и многое другое. В справочнике описаны современные версии систем UNIX и Linux — Solaris, HP-UX, AIX, Ubuntu Linux, openSUSE и Red Hat Enterprise Linux. Книга будет чрезвычайно полезной всем системным администраторам, а также пользователям систем UNIX и Linux, студентам, преподавателям и специалистам по сетевым технологиям.

ISBN 978-5-8459-1740-9

в продаже

LINUX
КАРМАННЫЙ СПРАВОЧНИК

Скотт Граннеман



www.williamspublishing.com

Данная книга представляет собой краткое пособие по основным командам операционной системы Linux. Читатель найдет в ней описание большинства команд, необходимых ему в повседневной работе.
В первых главах представлены те средства, с которых любой новичок начинает знакомство с неизвестной ему операционной системой. В них рассматриваются вопросы вывода на экран информации о каталогах, перехода из одного каталога в другой, создания каталогов, отображения содержимого файлов и т.д. По мере чтения книги материал усложняется. Читатель получает представление о работе с принтерами, правах доступа к файлам, архивировании и сжатии данных, поиске информации и др.
В книгу включено большое количество примеров, иллюстрирующих использование каждой описанной в ней команды.

ISBN 978-5-8459-1118-6

в продаже