Встраиваемые системы на основе Linux

Крис Симмондс

PACKT PUBLISHING

ДМК Пресс
издательство

Встраиваемые системы на основе Linux

Chris Simmonds

Mastering Embedded Linux Programming

Harness the power of Linux to create versatile and robust embedded solutions

Крис Симмондс

Встраиваемые системы на основе Linux

Москва, 2017

УДК 004.453/.453: 004.451.9Linux
ББК 32.972.1
С37

Симмондс К.

С37 Встраиваемые системы на основе Linux / пер. с англ. А. А. Слинкина. – М.: ДМК Пресс, 2017. – 360 с.: ил.

ISBN 978-5-97060-483-0

В книге подробно рассказано о том, как сконструировать встраиваемую Linux-систему из свободных программ с открытым исходным кодом, получив в результате надежный и эффективный продукт. Рассмотрены наборы инструментов, начальные загрузчики, ядро Linux и конфигурирование корневой файловой системы. Показано, как работать с системами сборки Buildroot и Yocto Project. Описаны процессы, потоки и управление памятью. Не обделены вниманием вопросы отладки и оптимизации платформы, а также выполнение приложений реального времени.

Издание рассчитано на разработчиков программного обеспечения на платформе Linux и системных программистов, уже знакомых со встраиваемыми системами. Предполагаются знание основ языка С и опыт системного программирования.

УДК 004.453/.453: 004.451.9Linux
ББК 32.972.1

Copyright © Packt Publishing 2016. First published in the English language under the title 'Mastering Embedded Linux Programming (9781784392536)'.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78439-253-6 (анг.)
ISBN 978-5-97060-483-0 (рус.)

Copyright © 2015 Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2017

Содержание

Предисловие 15
Об авторе 16
О рецензентах 17
Вступление 19

Глава 1. Приступая к работе 25
Выбор правильной операционной системы 26
Игроки 27
Жизненный цикл проекта 28
  Четыре составные части встраиваемой Linux-системы 29
Программное обеспечение с открытым исходным кодом 30
  Лицензии 30
Оборудование для встраиваемых Linux-систем 31
Используемое оборудование 33
  Плата BeagleBone Black 33
  QEMU 34
Используемое программное обеспечение 35
Резюме 35

Глава 2. О наборах инструментов 36
Что такое набор инструментов? 36
Типы наборов инструментов – платформенные и перекрестные 38
  Архитектура процессора 39
Выбор библиотеки C 40
Получение набора инструментов 42
  Сборка набора инструментов с помощью crosstool-NG 43
  Установка crosstool-NG 43
  Выбор набора инструментов 44
Анатомия набора инструментов 46
  Получение информации о кросс-компиляторе 46
  sysroot, библиотека и файлы-заголовки 48
Другие элементы набора инструментов 48
  Компоненты библиотеки C 49
Статическая и динамическая компоновка с библиотеками 50
  Статические библиотеки 50
  Разделяемые библиотеки 51

Искусство кросс-компиляции ... 53
Простые make-файлы ... 54
Autotools ... 54
Конфигурирование пакета ... 57
Проблемы кросс-компиляции ... 58
Резюме ... 58

Глава 3. Все о начальных загрузчиках ... 60

Что делает начальный загрузчик? ... 60
Последовательность начальной загрузки ... 61
Этап 1: код в ПЗУ ... 62
Этап 2: SPL ... 63
Этап 3: TPL ... 63
Загрузка из UEFI-прошивки ... 64
Переход от начального загрузчика к ядру ... 65
Введение в деревья устройств ... 66
Основные сведения о деревьях устройств ... 66
Свойство reg ... 67
Указатели на описатели и прерывания ... 68
Включаемые файлы деревьев устройств ... 69
Компиляция дерева устройств ... 71
Выбор начального загрузчика ... 71
U-Boot ... 72
Сборка U-Boot ... 72
Установка U-Boot ... 73
Работа с U-Boot ... 74
Загрузка Linux ... 78
Kconfig и U-Boot ... 79
Сборка и тестирование ... 82
Режим Сапсан ... 82
Barebox ... 82
Получение Barebox ... 83
Сборка Barebox ... 83
Резюме ... 84

Глава 4. Портирование и конфигурирование ядра ... 86

Что делает ядро? ... 86
Выбор ядра ... 88
Цикл разработки ядра ... 88
Стабильные и долгосрочные версии ... 89
Поддержка со стороны производителя ... 90
Сборка ядра ... 90
Получение исходного кода ... 90

О конфигурировании ядра ... 91
Использование переменной LOCALVERSION для идентификации ядра ... 95
Модули ядра ... 96
Компиляция ... 96
Компиляция образа ядра ... 97
Компиляция деревьев устройств ... 99
Компиляция модулей ... 99
Удаление артефактов сборки ... 99
Загрузка ядра ... 100
BeagleBone Black ... 100
QEMU ... 100
Паника ядра ... 101
Подготовка пользовательского пространства ... 102
Сообщения ядра ... 102
Командная строка ядра ... 103
Портирование Linux на новую плату ... 104
С деревом устройств ... 104
Без дерева устройств ... 105
Дополнительная литература ... 107
Резюме ... 107

Глава 5. Построение корневой файловой системы ... 109

Что должно быть в корневой файловой системе? ... 109
Структура каталогов ... 111
Каталог технологической подготовки ... 111
Программы в корневой файловой системе ... 114
Программа init ... 114
Оболочка ... 115
Утилиты ... 115
BusyBox спешит на помощь! ... 115
ToyBox – альтернатива BusyBox ... 117
Библиотеки для корневой файловой системы ... 118
Уменьшение размера путем удаления таблицы символов ... 119
Узлы устройств ... 119
Файловые системы proc и sysfs ... 120
Монтирование файловых систем ... 121
Модули ядра ... 122
Перенос корневой файловой системы на целевое устройство ... 122
Создание загрузочного ram-диска ... 123
Автономный ram-диск ... 123
Загрузка ram-диска ... 124
Встраивание ram-диска в формате cpio в образ ядра ... 125
Старый формат initrd ... 126

Программа init....126
Конфигурирование учетных записей пользователей....127
Добавление учетных записей пользователей в корневую файловую систему....129
Запуск процесса-демона....129
Улучшенный способ управления узлами устройств....129
Пример использования devtmpfs....130
Пример использования mdev....130
А так ли плохи статические узлы устройств?....131
Конфигурирование сети....131
Сетевые компоненты для glibc....132
Создание образов файловой системы с помощью таблиц устройств....133
Копирование корневой файловой системы на карту SD....134
Монтирование корневой файловой системы по NFS....134
Тестирование в эмуляторе QEMU....135
Тестирование с платой BeagleBone Black....136
Проблемы с правами доступа к файлам....136
Загрузка ядра по протоколу TFTP....137
Дополнительная литература....138
Резюме....138

Глава 6. Выбор системы сборки....139
Довольно самопальных встраиваемых систем....139
Системы сборки....139
Форматы пакетов и менеджеры пакетов....141
Buildroot....141
История....142
Стабильные версии и поддержка....142
Установка....142
Конфигурирование....143
Выполнение....144
Создание специального BSP-пакета....145
Добавление своего кода....146
Соответствие лицензионным требованиям....148
Yocto Project....149
История....149
Стабильные версии и поддержка....150
Установка Yocto Project....151
Конфигурирование....151
Сборка....152
Выполнение....153
Слои....154
Настройка образов с помощью local.conf....159
Рецепт создания образа....159

Создание SDK ... 160
Контроль лицензий ... 162
Дополнительная литература ... 162
Резюме ... 162

Глава 7. Выбор стратегии хранения ... 164

Типы запоминающих устройств ... 164
Флэш-память типа NOR ... 165
NAND-память ... 166
Управляемая флэш-память ... 168
Доступ к флэш-памяти из начального загрузчика ... 169
U-Boot и флэш-память типа NOR ... 170
U-Boot и флэш-память типа NAND ... 170
U-Boot и карты MMC, SD и eMMC ... 170
Доступ к флэш-памяти из Linux ... 170
Устройства на основе технологии памяти ... 171
Драйвер блочного устройства MMC ... 176
Файловые системы для флэш-памяти ... 177
Уровень флэш-преобразования ... 177
Файловые системы для флэш-памяти типа NOR и NAND ... 178
JFFS2 ... 178
YAFFS2 ... 181
UBI и UBIFS ... 182
Файловые системы для управляемой флэш-памяти ... 186
Flashbench ... 187
Discard и TRIM ... 188
Ext4 ... 189
F2FS ... 190
FAT16/32 ... 190
Сжатые неизменяемые файловые системы ... 191
squashfs ... 191
Временные файловые системы ... 192
Превращение корневой файловой системы в неизменяемую ... 193
Варианты выбора файловой системы ... 194
Обновление в месте эксплуатации ... 194
Степень детализации: файл, пакет или образ? ... 195
Атомарное обновление образа ... 196
Дополнительная литература ... 197
Резюме ... 198

Глава 8. Введение в драйверы устройств ... 199

Роль драйверов устройств ... 199
Символьные устройства ... 200

Блочные устройства 202
Сетевые устройства 203
Получение информации о драйверах на этапе выполнения 205
Получение информации из sysfs 207
Устройства: /sys/devices 207
Драйверы: /sys/class 208
Блочные устройства: /sys/block 209
Поиск подходящего драйвера устройства 209
Драйверы устройств в пользовательском пространстве 210
GPIO 210
Светодиоды 213
Шина I2C 214
Шина SPI 216
Написание драйвера устройства 216
Проектирование интерфейса символьного устройства 216
Анатомия драйвера устройства 218
Загрузка модулей ядра 222
Определение конфигурации оборудования 222
Деревья устройств 223
Платформенные данные 223
Связывание оборудования с драйверами 224
Дополнительная литература 226
Резюме 226

Глава 9. Инициализация системы – программа init 228

После того как ядро загрузилось 228
Введение в программы init 229
BusyBox init 229
Скрипты инициализации в Buildroot 231
System V init 231
inittab 233
Скрипты init.d 235
Добавление нового демона 235
Запуск и остановка служб 236
systemd 237
Сборка systemd в Yocto Project и Buildroot 237
Как systemd загружает систему 239
Добавление своей службы 240
Добавление сторожевого таймера 241
Применение во встраиваемых Linux-системах 242
Дополнительная литература 243
Резюме 243

Глава 10. Процессы и потоки 244

Процесс или поток? 244

Процессы 246

Создание нового процесса 246

Завершение процесса 247

Выполнение другой программы 249

Демоны 251

Межпроцессное взаимодействие 251

Потоки 256

Создание нового потока 256

Завершение потока 258

Компиляция многопоточной программы 258

Межпроцессное взаимодействие 258

Мьютексы 259

Изменение условий 259

Разбиение проблемы на части 260

Планирование 262

Справедливость и детерминированность 262

Политики с разделением времени 263

Политики реального времени 264

Выбор политики 265

Выбор приоритета реального времени 266

Дополнительная литература 266

Резюме 267

Глава 11. Управление памятью 268

Основы виртуальной памяти 268

Структура памяти ядра 269

Сколько памяти потребляет ядро? 270

Структура памяти в пользовательском пространстве 272

Карта памяти процесса 274

Подкачка 275

Выгрузка страниц в сжатую память (zram) 275

Отображение памяти с помощью mmap 276

Использование mmap для выделения частной памяти 276

Использование mmap для разделения памяти 276

Использование mmap для доступа к памяти устройства 277

Сколько памяти потребляет мое приложение? 277

Потребление памяти на уровне процесса 278

Использование top и ps 278

Использование smem 279

Другие инструменты 281

Обнаружение утечек памяти ... 281
mtrace ... 281
Valgrind ... 282
Нехватка памяти ... 283
Дополнительная литература ... 285
Резюме ... 285

Глава 12. Отладка в GDB ... 287

Отладчик GNU ... 287
Подготовка к отладке ... 287
Отладка приложений в GDB ... 288
Удаленная отладка с помощью gdbserver ... 289
Настройка Yocto Project ... 290
Настройка Buildroot ... 290
Начало отладки ... 290
Подключение GDB к gdbserver ... 290
Задание sysroot ... 291
Командные файлы GDB ... 292
Обзор команд GDB ... 292
Выполнение до точки прерывания ... 293
Отладка разделяемых библиотек ... 294
Yocto Project ... 294
Buildroot ... 295
Другие библиотеки ... 295
Своевременная отладка ... 295
Отладка разветвлений и потоков ... 296
Core-файлы ... 296
Использование GDB для анализа core-файлов ... 298
Пользовательские интерфейсы к GDB ... 298
Терминальный пользовательский интерфейс ... 299
Отладчик DDD ... 299
Eclipse ... 300
Отладка кода ядра ... 301
Отладка кода ядра в kgdb ... 301
Пример сеанса отладки ... 302
Отладка на ранних стадиях ... 304
Отладка модулей ... 304
Отладка кода ядра в kdb ... 305
Сообщения об ошибках ядра ... 306
Сохранение сообщения об ошибке ядра ... 307
Дополнительная литература ... 308
Резюме ... 309

Глава 13. Профилирование и трассировка 310

Эффект наблюдателя 310
Таблицы символов и флаги компиляции 311
Приступая к профилированию 311
Профилирование с помощью top 312
Профилировщик для бедных 313
Применение perf 314
Конфигурирование ядра для работы с perf 314
Сборка perf в Yocto Project 315
Сборка perf в Buildroot 315
Профилирование с помощью perf 315
Графы вызовов 317
perf annotate 318
Другие профилировщики: OProfile и gprof 319
Трассировка событий 321
Введение в Ftrace 321
Подготовка к работе с Ftrace 322
Динамический режим Ftrace и фильтры трассировки 324
События трассировки 325
Использование LTTng 326
LTTng и Yocto Project 327
LTTng и Buildroot 327
Применение LTTng для трассировки ядра 327
Использование Valgrind для профилирования приложений 330
Callgrind 330
Helgrind 330
Использование strace для показа системных вызовов 331
Резюме 333

Глава 14. Программирование в режиме реального времени 335

Что такое реальное время? 335
Определение источников недетерминированности 337
Задержки планирования 339
Вытеснение в ядре 340
Ядро Linux реального времени (PREEMPT_RT) 340
Потоковые обработчики прерываний 341
Вытесняемые блокировки ядра 343
Получение заплат PREEMPT_RT 344
Yocto Project и PREEMPT_RT 344
Таймеры высокого разрешения 345
Предотвращение страничных отказов в приложении реального времени 345

Экранирование прерываний 346
Измерение задержек планирования 347
  cyclictest 347
  Ftrace 350
  Комбинирование cyclictest и Ftrace 352
Дополнительная литература 352
Резюме 353

Предметный указатель 354

Предисловие

Linux – исключительно гибкая и мощная операционная система, и мне кажется, что мы еще не в полной мере осознали, какие преимущества сулит ее встраивание. Одна из возможных причин – многогранность этой темы и сложность ее изучения.

Конечно, можно самостоятельно пролагать себе путь в мире встраиваемых Linux-систем, как я и поступал в течение десяти лет, но отрадно видеть, что такие люди, как Крис, пишут книги, помогающие другим познакомиться со многими полезными вещами. Уж я-то точно нашел бы подобной книжке применение, если бы она существовала, когда я только начинал!

Понятно, что у меня есть личный интерес к проекту Yocto Project, поскольку это мое основное занятие и попытка внести свой вклад в мир встраиваемых Linux-систем. Одна из его основных целей – облегчить жизнь тем, кто создает такие системы. И мы добились кое-каких успехов, но есть области, требующие дальнейшей работы. Мы неустанно стремимся снять барьеры на пути начинающих и расширить круг посвященных, сделать технологию более доступной и распространенной.

В своей книге Крис преследует те же цели. Надеюсь, что вам понравится и книга, и сама система Linux и что в конечном итоге вы вольетесь в одно из активных сообществ, сформировавшихся вокруг проектов с открытым исходным кодом, стоящих за многими компонентами, о которых пойдет речь.

Ричард Пэрдай,
архитектор Yocto Project, член фонда Linux Foundation

Об авторе

Крис Симмондс – консультант по программному обеспечению и преподаватель, проживает в южной части Англии. С конца 1990-х годов занимается использованием Linux для создания встраиваемых систем и за это время успел поработать над многими интересными проектами, например: стереоскопическая камера, «умные» весы, различные абонентские приставки и домашние маршрутизаторы и даже большой шагающий робот.

Он часто выступает на конференциях по программному обеспечению с открытым исходным кодом и по встраиваемым системам, в том числе Embedded Linux Conference, Embedded World и Android Builders' Summit. С 2002 года читает курсы и проводит семинары по встраиваемым Linux-системам, а с 2010 – по встраиванию Android. Провел сотни презентаций во многих хорошо известных компаниях. Познакомиться с его работами можно в блоге «Inner Penguin» по адресу www.2net.co.uk.

Я благодарю своего редактора Саманту Гонсалвес, которая без устали следила за ходом работы и не давала мне сбиться с пути. Также выражаю благодарность всем, кто тратил время на чтение черновых вариантов, продираясь сквозь мои путаные словеса к сути. Спасибо Бихану Уэбстеру (Behan Webster), Клаасу ван Генду (Klaas van Gend), Тиму Бэрду (Tim Bird), Роберту Бергеру (Robert Berger), Матье Дешампу (Mathieu Deschamps) и Марку Фурману (Mark Furman). И конечно, я благодарен своей супруге Ширли Симмондс, которая всегда поддерживала меня и понимала, что я никак не мог помочь ей с ремонтом дома, потому что писал книгу.

О рецензентах

Роберт Бергер с 1993 года занимается проектированием, разработкой и управлением проектами в области встраиваемых систем как с жесткими требованиями к работе в режиме реального времени, так и без оных. Начиная с начала XXI века, он использует GNU/Linux на персональных компьютерах и серверах, но в основном для целей встраивания (в автомобильные системы, средства управления производственными процессами, робототехнику, телекоммуникационное оборудование, бытовую электронику и т. д.). Он регулярно присутствует на таких международных форумах, как Embedded World, Embedded Software Engineering Congress, Embedded Systems Conference и Embedded Linux Conference, в качестве эксперта и докладчика. Специализируется главным образом на преподавании, но также оказывает консультационные услуги (на английском и немецком языках) в различных странах. В сферу профессиональной компетенции Роберта входят как совсем небольшие системы реального времени (FreeRTOS), так и комплексы с несколькими процессорами или ядрами со встроенной системой GNU/Linux (уровень пространства ядра или пользователя, драйверы устройств, интерфейсы с оборудованием, отладка, многоядерная разработка, проект Yocto Project) с упором на свободное ПО и ПО с открытым исходным кодом. Он много путешествует по свету. Является исполнительным директором и специалистом по встраиваемому ПО в компании Reliable Embedded Systems со штаб-квартирой в городе Санкт-Барбата, Австрия. Проживает с семьей в Афинах. Связаться с Робертом можно через его сайт по адресу http://www.ReliableEmbeddedSystems.com.

Рецензировал книгу Rudolf J. Streif «Embedded Linux Systems with the Yocto Project» (Prentice Hall Open Source Software Development Series).

Тим Бэрд работает старшим программистом в компании Sony Mobile Communications, где отвечает за модификацию ядра Linux для использования в продуктах Sony. Также является председателем архитектурной группы рабочей группы по бытовой электронике (CE Working Group) в фонде Linux Foundation. Занимается Linux свыше 20 лет. Стоял у истоков двух отраслевых объединений, связанных со встраиваемыми Linux-системами, и является основателем и бессменным руководителем конференции Embedded Linux Conference. В прошлом написал вместе с Тимом книгу «Using Caldera OpenLinux».

Матье Дешамп – основатель компании ScourGE (www.scourge.fr), предлагающей клиентам инновационные услуги в области оборудования и ПО с открытым исходным кодом. Компания занимает лидирующие позиции в сфере телекоммуникаций, мобильной связи, управления промышленными процессами и систем поддержки принятия решений.

Оказывает консультационные услуги в области исследований и разработок, преподает. С 2003 года в качестве технического руководителя принимал участие

во многих мелких и крупных проектах, так или иначе связанных с GNU/Linux, Android, разработкой встраиваемых систем и безопасностью.

Марк Фурман, автор книги «OpenVZ Essentials», в настоящее время занимает должность инженера-системотехника в компании Info-Link Technologies. В сфере ИТ работает свыше 10 лет, специализируется на Linux и других продуктах с открытым исходным кодом, а также балуется с Arduino, Python и Raspberry Pi в Knox Labs, клубе компьютерщиков-энтузиастов в округе Нокс, штат Огайо.

Клаас ван Генд окончил факультет системной инженерии и управления Эйндховенского технического университета в Нидерландах. Работал в различных компаниях, включая Philips, Siemens и Bosch, писал программное обеспечение для экспериментальных образцов принтеров, шифрования видео, автомобильных информационно-развлекательных систем, медицинского оборудования, средств автоматизации производства и навигационных систем. В 2004 году перешел в компанию MontaVista Software, являющуюся идером на рынке встраиваемых Linux-систем. В роли системного архитектора и консультанта оказывал услуги многим европейским компаниям, помогая встраивать Linux в разработанные ими продукты.

Последние несколько лет работает преподавателем и консультантом в компании Vector Fabrics, небольшом стартапе, специализирующемся на программировании многоядерных компьютеров и динамическом анализе ПО. Преподает многоядерное программирование на C и C++, помогает заказчикам улучшить программы за счет правильного использования аппаратных ресурсов. Созданный компанией Vector Fabrics комплект инструментов Pareon помогает автоматически находить трудно воспроизводимые ошибки, в том числе гонку за данные, переполнение буфера, использование уже освобожденных областей кучи и стека, утечки памяти.

Клаас – автор более сотни журнальных статей на темы Linux (в том числе встраиваемых систем), программирования, производительности, проектирования систем и компьютерных игр. Он является соучредителем конференции Embedded Linux Conference Europe и выступал в роли ведущего разработчика в нескольких проектах с открытым исходным кодом, включая UMTSmon для сотовых сетей и игру-головоломку на темы физики The Butterfly Effect.

На досуге Класс читает городское фэнтези или отрывается в аэроклубе Нистелроде, где летает на планерах.

Бехан Уэбстер двадцать лет работал в различных технических отраслях, включая телекоммуникации, передачу данных, оптику, встраиваемые системы и автомобильную промышленность. Писал код для самых разных устройств – от крохотных до сверхбольших. Обладает опытом программирования ядра Linux, разработки встраиваемых систем и подготовки печатных плат к производству. В настоящее время занимает должность ведущего консультанта в компании Converse in Code Inc, где занимается встраиваемыми Linux-системами, а также руководит проектом LLVMLinux и преподает в интересах фонда Linux Foundation.

Вступление

Встраиваемая система – это устройство, содержащее внутри себя компьютер, но не выглядящее как компьютер. Стиральные машины, телевизоры, принтеры, автомобили, роботы – все они управляются каким-то компьютером, а иногда и не одним. Устройства становятся все сложнее, мы ожидаем от них все большего, а значит, растут требования к управляющей ими операционной системе. И все чаще такой системой становится Linux.

Богатство возможностей Linux проистекает из модели открытых исходных текстов, поощряющей совместное владение кодом. Это означает, что инженеры с разным образованием и опытом работы, зачастую работающие в конкурирующих компаниях, могут объединить усилия для создания ядра операционной системы, поддержания его актуальности и соответствия новейшим разработкам в области оборудования. Общая кодовая база позволяет поддержать самые разные устройства – от мощнейших суперкомпьютеров до наручных часов. ОС Linux – лишь один компонент, а для создания работающей системы нужно еще много чего: от базовых инструментов типа командной оболочки до графического интерфейса пользователя, который позволяет взаимодействовать с вебом и облачными службами. Ядро Linux в сочетании с обширным набором других открытых компонентов становится основой для системы, способной исполнять различные роли.

Однако гибкость – палка о двух концах. Да, у проектировщика системы появляется широкий спектр возможных решений задачи, но вместе с ним и проблема выбора оптимального решения. Цель этой книги – подробно рассказать о том, как сконструировать встраиваемую Linux-систему из свободных программ с открытым исходным кодом, получив в результате надежный и эффективный продукт. В основу книги положен многолетний опыт работы автора в качестве консультанта и преподавателя.

Структура книги

Организационно книга устроена так же, как жизненный цикл типичного проекта встраиваемой Linux-системы. В первых шести главах изложено все, что нужно знать о подготовке проекта и устройстве системы на базе Linux, а завершается эта часть соображениями о выборе подходящей системы сборки Linux. Затем настает пора принятия ключевых решений об архитектуре и составных частях системы, в том числе флэш-памяти, драйверах устройств и системе инициализации. Следующий этап – написание приложений для работы на собранной встраиваемой платформе, этой теме посвящены две главы о процессах, потоках и управлении памятью. И наконец, в главах 12 и 13 обсуждаются отладка и оптимизация платформы. А в последней главе описана конфигурация Linux для выполнения приложений реального времени.

В главе 1 «Приступая к работе» описываются альтернативы, имеющиеся у проектировщика системы в самом начале проекта.

В главе 2 «О наборах инструментов» описаны компоненты набора инструментальных средств с упором на кросс-компиляцию. Рассказывается, где взять набор инструментов и как собрать его из исходного кода.

В главе 3 «Все о начальных загрузчиках» объяснена роль загрузчика ОС в инициализации оборудования, а в качестве примеров взяты загрузчики U-Boot и Bareboot. Описывается также дерево устройств – способ представления конфигурации оборудования, используемый во многих встраиваемых системах.

Глава 4 «Портирование и конфигурирование ядра» содержит информацию о том, как выбрать ядро Linux для встраиваемой системы и настроить его для работы с оборудованием, находящимся внутри устройства. Здесь же рассказывается о портировании Linux на новое оборудование.

В главе 5 «Построение корневой файловой системы» мы познакомимся с пользовательским пространством встраиваемой Linux-системы и представим пошаговую инструкцию по конфигурированию корневой файловой системы.

В главе 6 «Выбор системы сборки» описаны две системы сборки встраиваемых Linux-систем, цель которых – автоматизация описанных ранее шагов. На этом завершается первая часть книги.

Глава 7 «Выбор стратегии хранения» посвящена обсуждению проблем, связанных с управлением флэш-памятью, в том числе неуправляемых микросхем флэш-памяти и карт типа MMC и eMMC для встраиваемых систем. Описаны файловые системы для каждой технологии. Рассматривается также вопрос об обновлении прошивки устройства пользователем.

В главе 8 «Введение в драйверы устройств» описывается, как находящийся в ядре драйвер устройства взаимодействует с оборудованием. Приведены примеры простых драйверов. Описаны также различные способы вызова драйвера из пользовательского адресного пространства.

В главе 9 «Инициализация системы – программа init» описано, как запускается первая программа, работающая в пользовательском пространстве, – `init`, и как она запускает все остальные программы в системе. Мы рассматриваем три варианта программы `init`, подходящие для встраиваемых систем разного типа: от самой простой, BusyBox `init`, до systemd.

Глава 10 «Процессы и потоки» посвящена рассмотрению встраиваемых систем с точки зрения прикладного программиста. В ней речь пойдет о процессах, потоках, межпроцессном взаимодействии и политиках планирования.

В главе 11 «Управление памятью» обсуждаются идеи, лежащие в основе механизма виртуальной памяти и отображения адресного пространства. Рассказано также о том, как узнать, какие части памяти используются, и обнаружить утечки памяти.

Глав 12 «Отладка в GDB» посвящена использованию отладчика GDB для интерактивной отладки кода, работающего в ядре и в пользовательском пространстве. Описан также отладчик ядра `kdb`.

В главе 13 «Профилирование и трассировка» рассматриваются методы измерения производительности системы, начиная с получения профиля работы системы в целом и заканчивая выявлением узких мест, приводящих к снижению производительности. Здесь же описана программа Valgrind, которая проверяет, правильно ли приложение синхронизирует потоки и управляет памятью.

В главе 14 «Программирование в режиме реального времени» вы найдете подробное руководство по созданию программ реального времени в Linux, включая настройку ядра и наложение на него заплат реального времени, а также описание инструментов для измерения задержек. Здесь же приводятся сведения о том, как уменьшить число страничных прерываний путем блокировки памяти.

Что необходимо для чтения этой книги

В книге используется только программное обеспечение с открытым исходным кодом, в большинстве случаев – последние версии, доступные на момент написания книги. Я старался описывать основные возможности, не привязываясь к конкретной версии, но некоторые команды могут не работать с более поздними версиями – от этого никуда не денешься. Надеюсь, впрочем, что сопроводительные описания достаточно информативны, так что вам не составит труда применить те же принципы к последующим версиям пакета.

В создании любой встраиваемой системы участвуют две системы: исходная – в которой кросс-компилируется программный код, и целевая – в которой этот код будет работать. В качестве исходной системы я использовал Ubuntu 14.04, но годится (возможно, с минимальной модификацией) почти любой дистрибутив Linux. Я вынужден был выбрать какую-то целевую систему для представления встраиваемой. В книге рассматриваются два варианта: BeagleBone Black и QEMU – эмулятор системы с процессором ARM. Второй вариант означает, что примеры можно выполнять, не тратясь на оборудование для реального целевого устройства. При этом ничто не мешает попробовать примеры на различных целевых платформах, изменив такие детали, как имена устройств и распределение памяти.

Версии основных пакетов для целевой системы: U-Boot 2015.07, Linux 4.1, Yocto Project 1.8 «Fido» и Buildroot 2015.08.

Предполагаемая аудитория

Эта книга рассчитана в первую очередь на разработчиков Linux и системных программистов, которые уже знакомы со встраиваемыми системами и хотят узнать, как создавать лучшие в своем классе устройства. Требуются понимание основ программирования на языке C и опыт системного программирования.

Графические выделения

В этой книге для выделения семантически различной информации применяются различные стили. Ниже приведены примеры стилей с пояснениями.

Фрагменты, имена функций, таблиц базы данных, папок и файлов, URL-адреса, адреса в Twitter и данные, вводимые пользователем, выглядят следующим образом: «Мы могли бы воспользоваться функциями потокового ввода-вывода `fopen(3)`, `fread(3)` и `fclose(3)`».

Отдельно стоящие фрагменты кода набраны так:

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
  /* Размеры разделов задаются в терминах размера блока NAND-памяти */
  {
    .name = "X-Loader",
    .offset = 0,
    .size = 4 * NAND_BLOCK_SIZE,
    .mask_flags = MTD_WRITEABLE, /* только для чтения */
  }
}
```

Чтобы привлечь внимание к части кода, строки или отдельные слова выделяются полужирным шрифтом:

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
  /* Размеры разделов задаются в терминах размера блока NAND-памяти */
  {
    .name = "X-Loader",
    .offset = 0,
    .size = 4 * NAND_BLOCK_SIZE,
    .mask_flags = MTD_WRITEABLE, /* только для чтения */
  }
}
```

Текст, который вводится на консоли или выводится на консоль, напечатан следующим образом:

```
# flash_erase -j /dev/mtd6 0 0
# nandwrite /dev/mtd6 rootfs-sum.jffs2
```

Новые термины и важные слова набраны полужирным шрифтом. Также выделяются элементы интерфейса, например пункты меню и поля в диалоговых окнах: «Во второй строке на консоль выводится сообщение **Please press Enter to activate this console**».

Предупреждения и важные замечания оформлены так.

Советы и рекомендации выглядят так.

Отзывы

Мы всегда рады отзывам читателей. Расскажите нам, что вы думаете об этой книге – что вам понравилось или, быть может, не понравилось. Читательские отзывы

важны для нас, так как помогают выпускать книги, из которых вы черпаете максимум полезного для себя.

Чтобы отправить обычный отзыв, просто пошлите письмо на адрес feedback@packtpub.com, указав название книги в качестве темы.

Если вы являетесь специалистом в некоторой области и хотели бы стать автором или соавтором книги, познакомьтесь с инструкциями для авторов по адресу www.packtpub.com/authors.

Поддержка клиентов

Счастливым обладателям книг Packt мы можем предложить ряд услуг, которые позволят извлечь из приобретения максимум пользы.

Загрузка кода примеров

Вы можете скачать код примеров ко всем приобретенным вами книгам издательства Packt в своем личном кабинете на сайте http://www.PacktPub.com. Если книга была куплена в другом месте, зайдите на страницу http://www.PacktPub.com/support, зарегистрируйтесь, и мы отправим файлы по электронной почте.

Опечатки

Мы проверяли содержимое книги очень тщательно, но какие-то ошибки все же могли проскользнуть. Если вы найдете в нашей книге ошибку, в тексте или в коде, пожалуйста, сообщите нам о ней. Так вы избавите других читателей от разочарования и поможете нам сделать следующие издания книги лучше. При обнаружении опечатки просьба зайти на страницу http://www.packtpub.com/submit-errata, выбрать книгу, щелкнуть по ссылке **Errata Submission Form** и ввести информацию об опечатке. Проверив ваше сообщение, мы поместим информацию об опечатке на нашем сайте или добавим ее в список замеченных опечаток в разделе **Errata** для данной книги.

Список подтвержденных опечаток можно просмотреть, введя название книги в поле поиска на странице https://www.packtpub.com/books/content/support. Информация появится в разделе **Errata**.

Нарушение авторских прав

Незаконное размещение защищенного авторским правом материала в Интернете – проблема для всех носителей информации. В издательстве Packt мы относимся к защите прав интеллектуальной собственности и лицензированию очень серьезно. Если вы обнаружите незаконные копии наших изданий в любой форме в Интернете, пожалуйста, незамедлительно сообщите нам адрес или название веб-сайта, чтобы мы могли предпринять соответствующие меры.

Просим отправить ссылку на вызывающий подозрение в пиратстве материал по адресу copyright@packtpub.com.

Мы будем признательны за помощь в защите прав наших авторов и содействие в наших стараниях предоставлять читателям полезные сведения.

Вопросы

Если вас смущает что-то в этой книге, вы можете связаться с нами по адресу questions@packtpub.com, и мы сделаем все возможное для решения проблемы.

Глава 1

Приступая к работе

Итак, вы приступаете к работе над очередным проектом, и на этот раз он будет основан на Linux. О чем следует подумать, прежде чем бежать к клавиатуре? Начнем с общего обзора встраиваемых Linux-систем, объясним, почему они так популярны, что подразумевают лицензии на ПО с открытым исходным кодом и какого рода оборудование необходимо для работы Linux.

Linux стала рассматриваться в качестве кандидата на роль управляющей ОС для встраиваемых устройств где-то в 1999 году. Именно тогда компания Axis (www.axis.com) выпустила свою первую сетевую камеру под управлением Linux, а компания TiVo (www.tivo.com) – первый видеомагнитофон. С тех пор популярность Linux постоянно росла, и сегодня это основная операционная система для многих классов изделий. На момент написания этой книги, в 2015 году, под управлением Linux работают примерно два миллиарда устройств. Сюда входит неисчислимая рать смартфонов на платформе Android с ядром Linux, сотни миллионов абонентских приставок, «умных» телевизоров (Smart TV) и маршрутизаторов Wi-Fi, а также множество других устройств, производимых меньшими тиражами, например: приборы автомобильной диагностики, весы, промышленные устройства и медицинские мониторы.

Так почему же в вашем телевизоре работает Linux? На первый взгляд, от телевизора требуется всего лишь отобразить на экране поток видеоданных – не так уж сложно. Зачем для этого навороченная операционная система типа Linux?

Простой ответ дает закон Мура: Гордон Мур, один из основателей компании Intel, в 1965 г. заметил, что плотность компонентов на кристалле микросхемы удваивается примерно каждые два года. Это относится не только к настольным компьютерам, ноутбукам и серверам, но и к устройствам, которыми мы пользуемся в повседневной жизни. Сердцем большинства встраиваемых устройств является интегральная микросхема, содержащая одно или несколько процессорных ядер и интерфейсы с оперативной памятью, массовым запоминающим устройством и разнообразными периферийными устройствами. Все это называется «системой на кристалле» (System on Chip, или SoC), и сложность таких систем растет в соответствии с законом Мура. К типичной SoC-системе прилагается справочное техническое руководство, насчитывающее тысячи страниц. Ваш телевизор не просто отображает видеосигнал, как старый добрый аналоговый приемник.

Поток данных представлен в цифровой форме и иногда зашифрован, его необходимо обработать, чтобы получить изображение. Ваш телевизор подключен к Интернету (или будет подключен в ближайшем будущем). Он может получать контент от смартфонов, планшетов и домашних медиа-серверов. На нем можно играть в игры. И так далее и тому подобное. Для управления всеми этими сложными функциями нужна полноценная операционная система.

Ниже перечислено несколько причин для выбора на эту роль Linux.

- В Linux уже имеется необходимая функциональность: хороший планировщик задач, хороший сетевой стек, поддержка USB, Wi-Fi, Bluetooth, многих запоминающих устройств, мультимедийных устройств и т. д. В общем, против любого пункта стоит галочка.
- Linux портирована на многие процессорные архитектуры, в том числе часто применяемые в системах на кристалле: ARM, MIPS, x86 и PowerPC.
- Исходный код Linux открыт, так что в него можно внести необходимые вам изменения. Вы или кто-то, работающий по вашему поручению, может создать пакет программ для поддержки конкретной SoC-платы или устройства. Можно добавить протоколы, функции и технологии, которых нет в базовом исходном коде. Или исключить ненужные возможности, чтобы уменьшить требования к оперативной и внешней памяти. Linux – гибкая система.
- Вокруг Linux сложилось активное сообщество, а если говорить о ядре, то даже очень активное. Новая версия ядра выходит раз в 10–12 недель, причем авторами кода являются порядка 1000 разработчиков. Такая активность означает, что Linux всегда актуальна и поддерживает все современное оборудование, протоколы и стандарты.
- Лицензия на ПО с открытым исходным кодом гарантирует доступ к исходному коду. Вы не привязаны к конкретному поставщику.

В силу этих причин Linux – идеальный выбор для создания сложных систем. Но есть и несколько подводных камней, о которых нельзя не упомянуть. В придачу к быстрому процессу разработки и децентрализованной структуре управления полагается нагрузка: вы должны приложить усилия, чтобы разобраться в том, как все это устроено, и продолжать учиться по мере дальнейшего развития. Надеюсь, эта книга поможет вам справиться.

Выбор правильной операционной системы

Подходит ли Linux для вашего проекта? Linux стоит рассматривать, если решаемая задача оправдывает сложность системы. Особенно это относится к задачам, где важны способность к выходу в сеть, надежность и наличие развитых интерфейсов с пользователем. Однако не всякую проблему можно решить с помощью Linux, и ниже приведены вопросы, которые вы должны задать себе, прежде чем принимать решение.

- Подходит ли ваше оборудование для работы с Linux? По сравнению с традиционными операционными системами реального времени (ОСРВ) типа

VxWorks, Linux значительно требовательнее к ресурсам. Ей нужны, по меньшей мере, 32-разрядный процессор и гораздо больше памяти. Я еще вернусь к этому вопросу в разделе о типичных требованиях к оборудованию.

- Обладаете ли вы необходимыми знаниями и умениями? На ранних стадиях проекта, когда идет подготовка печатной платы к производству, потребуются детальное знание Linux и ее взаимодействие с вашим оборудованием. А на этапе отладки и оптимизации приложения вам придется интерпретировать результаты, и нужно знать, как это делается. Если внутри компании нет специалистов нужной квалификации, то, возможно, имеет смысл поручить часть работы внешним организациям. Но, конечно, чтение этой книги выручит!
- Должна ли ваша система работать в режиме реального времени? Linux способна справиться со многими задачами реального времени, если уделять внимание некоторым деталям, о которых я расскажу в главе 14.

Подойдите к этим вопросам со всей серьезностью. Быть может, стоит поискать похожие продукты, в которых используется Linux, и поинтересоваться, как они сделаны. Следуйте передовым образцам.

Игроки

Откуда берутся программы с открытым исходным кодом? Кто их пишет? И как они соотносятся с основными компонентами разработки встраиваемых систем – набором инструментов, начальным загрузчиком, ядром и основными утилитами, находящимися в корневой файловой системе?

Перечислим основных игроков.

- Сообщество, сложившееся вокруг ПО с открытым исходным кодом. Это и есть движущая сила, стоящая за тем программным обеспечением, которое вы собираетесь использовать. Сообщество – это неформальное объединение разработчиков, многие из которых тем или иным способом получают вознаграждение за свой труд, например, от некоммерческой организации, академического учреждения или коммерческой компании. Они совместно работают во имя достижения целей различных проектов. Сообществ много – больших и маленьких. В этой книге упоминаются сообщества вокруг самой ОС Linux, проектов U-Boot, BusyBox, Buildroot, Yocto Project и многих проектов под эгидой GNU.
- Архитекторы процессоров. Это организации, занимающиеся проектированием используемых нами процессоров. Основные из них: ARM/Linaro (SoC-системы на базе процессоров ARM), Intel (платформы x86 и x86_64), Imagination Technologies (MIPS) и Freescale/IBM (PowerPC). Они реализуют основные архитектуры процессоров или, по крайней мере, оказывают влияние на поддержку.
- Поставщики систем на кристалле (Atmel, Broadcom, Freescale, Intel, Qualcomm, TI и многие другие). Они получают ядро и набор инструментов от архи-

текторов процессоров и модифицируют их для поддержки своих микросхем. Они же разрабатывают эталонные платы: конструкции, которые на следующем уровне используются для создания макетных плат и готовых изделий.

- Поставщики плат и производители комплектного оборудования (OEM) – организации, которые получают типовой вариант конструкции от поставщиков систем на кристалле и встраивают его в конкретное изделие, например абонентскую приставку или камеру, либо создают макетные платы более общего назначения, как, например, компании Avantech или Kontron. Важным классом подобных изделий являются дешевые макетные платы, например BeagleBoard/BeagleBone и Raspberry Pi, вокруг которых сложились свои экосистемы программных и аппаратных дополнений.

Ваш проект обычно оказывается конечным звеном такой цепочки, т. е. свободы выбора компонентов вы лишены. Не получится просто взять последнюю версию ядра с сайта `kernel.org` (разве что в редчайших случаях), потому что она не поддерживает используемую вами микросхему или плату.

Это общая проблема, присущая разработке встраиваемых систем. В идеале разработчики, занятые в каждом звене, должны передавать сделанные ими изменения на следующий уровень, но они этого не делают. Не редкость встретить ядро с тысячами заплат, которые не пошли дальше. К тому же поставщики SoC-систем склонны активно разрабатывать открытый код только для последних версий своих микросхем, а значит, поддержка любой микросхемы после примерно двух лет замораживается, и выпуск обновлений прекращается.

В результате большинство встраиваемых систем основано на устаревших версиях ПО. Для них не выходят обновления, повышающие безопасность и производительность, им недоступны функции, имеющиеся в более поздних версиях. Уязвимости типа Heartbleed (из-за ошибки в библиотеках OpenSSL) или Shellshock (из-за ошибки в оболочке bash) так и остаются незакрытыми. Я еще вернусь к этому вопросу при обсуждении темы безопасности в этой главе.

Что вы можете с этим поделать? Во-первых, задать вопросы поставщикам: какова их стратегия обновления, как часто они пересматривают версии ядра, какую версию ядра используют сейчас и какую использовали перед этим? Некоторые поставщики добились заметного прогресса в этом отношении, поэтому стоит предпочесть их продукцию.

Во-вторых, вы можете предпринять некоторые шаги по обеспечению собственной автономности. В этой книге будут подробно рассмотрены зависимости и показано, что можно сделать в этом отношении. Не стоит слепо использовать пакет, предложенный производителем SoC-системы или платы, даже не посмотрев, какие есть альтернативы.

Жизненный цикл проекта

Эта книга состоит из четырех частей, соответствующих этапам работы над проектом. Этапы не обязательно следуют один за другим. Обычно они перекрываются,

иногда приходится возвращаться назад и пересматривать ранее принятые решения. Тем не менее они дают представление о типичной деятельности разработчика проекта.

- Главы 1-6, посвященные элементам встраиваемых Linux-систем, помогут организовать среду разработки и создать платформу для работы на следующих этапах. Часто этот этап называют «подготовкой платы к производству» (board bring-up).
- Главы 7–9 о выборе архитектуры и конструкции системы позволят взглянуть на некоторые проектные решения, относящиеся к хранению программ и данных, к разделению работы между драйверами устройств в ядре и приложениями и к инициализации системы.
- Главы 10 и 11 посвящены встраиваемым приложениям, в них рассказано, как эффективно использовать модель процессов и потоков в Linux и как управлять памятью в условиях ограниченности ресурсов.
- В главах 12 и 13, посвященных отладке и оптимизации, описаны методы трассировки, профилирования и отладки кода приложения и ядра.

Пятая часть, состоящая из главы 14, стоит особняком и посвящена системам реального времени – небольшому, но важному классу встраиваемых систем. Учет поведения в реальном времени оказывает влияние на все четыре основных этапа.

Четыре составные части встраиваемой Linux-системы

Любой проект начинается с получения, модификации под свои нужды и развертывания четырех элементов: набора инструментов, начального загрузчика, ядра и корневой файловой системы. Этой теме посвящена первая часть книги.

- **Набор инструментов**: состоит из компилятора и прочих инструментальных средств, необходимых для создания программы, управляющей целевым устройством. От набора инструментов зависит все остальное.
- **Начальный загрузчик** необходим для инициализации платы, загрузки и инициализации ядра Linux.
- **Ядро**: это сердце системы, оно управляет всеми ресурсами и осуществляет взаимодействие с оборудованием.
- **Корневая файловая система** содержит библиотеки и программы, исполняемые после завершения инициализации ядра.

Есть еще и пятый, не упомянутый здесь элемент. Это набор программ, составляющих ваше встраиваемое приложение, благодаря которому устройство выполняет свои функции, будь то взвешивание бакалейных товаров, показ фильмов, управление роботом или полет дрона.

Как правило, все или некоторые из этих элементов предлагаются в виде пакета при покупке нужной вам SoC-системы или платы. Но по вышеупомянутым причинам это не всегда оптимальный выбор. Материал первых шести глав поможет принять правильное решение. Мы также познакомимся с двумя инструментами автоматизации: Buildroot и Yocto Project.

Программное обеспечение с открытым исходным кодом

Компоненты встраиваемой Linux-системы – программы с открытым исходным кодом, поэтому самое время понять, что это означает, почему такой подход вообще работает и как отражается на создаваемом с помощью таких программ устройстве, которое зачастую защищено правом собственности.

Лицензии

Говоря об открытом исходном коде, часто употребляют слово «free»[1]. Люди, не знакомые с предметом, обычно предполагают, что речь идет о бесплатности, и лицензия действительно гарантирует, что ПО можно использовать для разработки и внедрения систем и никому ничего не платить. Однако более важно другое значение слова «free» – свободный, поскольку вы свободны в своем праве получить исходный код, модифицировать его, как считаете нужным, и распространять в составе других систем. Эти лицензии дают такое право. Сравните с условно-бесплатными лицензиями, которые разрешают бесплатно копировать двоичный код, но не дают доступа к исходному, или с лицензиями, разрешающими бесплатно использовать ПО только при определенных условиях, например для личного пользования, но запрещают использование в коммерческих целях. Ни то, ни другое нельзя назвать «открытым исходным кодом».

Чтобы вам было проще понять последствия работы с лицензиями на ПО с открытым исходным кодом, я дам несколько пояснений, но имейте в виду, что я инженер, а не юрист. Ниже изложено мое личное понимание лицензий.

Лицензии на ПО с открытым исходным кодом можно отнести к одному из двух классов: **GPL (General Public License** – генеральная общедоступная лицензия), предоставляемая Фондом свободного ПО (Free Software Foundation), и либеральные лицензии, производные от лицензий **BSD (Berkeley Software Distribution)**, фонда Apache Foundation и др.

Либеральная лицензия по существу означает, что вы вправе модифицировать исходный код и использовать его в своих системах при условии неизменности условий самой лицензии. Иными словами, при этом единственном ограничении вы можете делать с кодом все, что угодно, в том числе встраивать его в системы, защищенные правом собственности.

Лицензии GPL похожи, но включают оговорку, в соответствии с которой вы обязаны передать право на получение и модификацию ПО конечным пользователям вашего продукта. Иными словами, вы передаете свой исходный код в общее пользование. Один из вариантов – сделать его доступным для всех желающих, выложив на публичный сервер. Другой – предоставлять только вашим конечным пользователям, включив письменное обязательство предоставить код по запросу. Лицензия GPL идет еще дальше и запрещает включать распространяемый на ее

[1] В английском языке free означает как «бесплатный», так и «свободный». – *Прим. перев.*

условиях код в закрытые программы. Любое включение такого кода автоматически означает, что ко всей программе применяются условия GPL. Иначе говоря, в одной программе не может встречаться закрытый код и код, распространяемый на условиях GPL.

А как насчет библиотек? Если библиотека распространяется на условиях GPL, то любая скомпонованная с ней программа должна распространяться на тех же условиях. Однако большинство библиотек распространяется на условиях лицензии **LGPL (Lesser General Public License)**, которая разрешает компоновать библиотеку с закрытой программой.

Все вышесказанное относится к лицензиям GPL v2 и LGPL v2.1. Следует сказать также о последних версиях GPL v3 и LGPL v3. Они противоречивы, и признаюсь, что не вполне понимаю, какие последствия они влекут. Но идея была в том, чтобы гарантировать следующее положение: любой компонент системы, распространяемый по лицензии GPL v3 и LGPL v3, должен допускать замену конечным пользователем. И это, конечно, согласуется с духом ПО с открытым исходным кодом. Но есть и проблемы. Некоторые устройства, управляемые Linux, применяются для получения доступа к информации согласно с уровнем подписки или иным ограничением, а замена критической части программы может снять ограничение. Типичный пример – абонентские приставки. Есть также проблемы, связанные с безопасностью. Если владелец устройства имеет доступ к системному коду, то его может получить и незваный гость. Для защиты часто используют образы ядра, подписанные уполномоченным органом или поставщиком, так что несанкционированное изменение невозможно. Это следует считать покушением на мое право модифицировать собственное устройство? Есть разные мнения.

В этом споре часто упоминается абонентская приставка TiVo. В ней используется ядро Linux, распространяемое по лицензии GPL v2. Компания TiVo раскрывает код своей версии ядра – в полном соответствии с лицензией. Но у TiVo имеется также начальный загрузчик, который соглашается загружать только подписанный ей же двоичный код ядра. Следовательно, мы можем собрать модифицированное ядро для приставки TiVo, но не сумеем загрузить его в оборудование. Позиция Фонда открытого ПО заключается в том, что это не в духе ПО с открытым исходным кодом, а для самой процедуры используется термин «тивоизация». Версии GPL v3 и LGPL v3 специально разработаны, чтобы запретить такую ситуацию. Разработчики некоторых проектов и, в частности, ядра Linux не торопятся принимать лицензии третьей версии из-за ограничений на производителей оборудования.

Оборудование для встраиваемых Linux-систем

На что обращать внимание при проектировании или выборе оборудования для проекта встраиваемой Linux-системы?

Прежде всего на архитектуру процессора, поддерживаемую ядром, – если, конечно, вы не собираетесь самостоятельно разработать новую архитектуру! В исходном коде Linux 4.1 есть 30 архитектур, представленных отдельными подката-

логами каталога `arch/`. Среди них имеются 32- и 64-разрядные архитектуры, по большей части с блоком управления памятью (**MMU**), хотя это необязательно. Во встраиваемых устройствах чаще всего встречаются архитектуры ARM, MIPS, PowerPC и X86, каждая из которых существует в 32- и 64-разрядных вариантах, и все они включают блок управления памятью.

В этой книге в основном подразумеваются процессоры такого типа. Есть также группа устройств, не оснащенных MMU и управляемых подмножеством Linux, называемым «Linux для микроконтроллеров», или uClinux. В эту группу входят архитектуры процессоров ARC, Blackfin, Microblaze и Nios. В нескольких местах я упоминаю uClinux, но в детали не вхожу, потому что это слишком специальная тема.

Во-вторых, нужно иметь достаточный объем оперативной памяти (ОЗУ). 16 МиБ – неплохой минимум, хотя, в принципе, для работы Linux достаточно и половины. Можно запустить Linux даже в 4 МиБ, если вы готовы потратить усилия на оптимизацию каждой части системы. Можно пойти и дальше, но рано или поздно наступает момент, когда систему уже нельзя назвать Linux.

В-третьих, встает вопрос об энергонезависимой памяти, обычно флэш-памяти. 8 МиБ достаточно для простого устройства типа веб-камеры или простенького маршрутизатора. Как и в случае ОЗУ, при желании можно построить работоспособную Linux-систему с меньшим объемом внешней памяти, но чем он меньше, тем труднее становится задача. Linux прекрасно поддерживает флэш-память, в том числе приборы, выполненные по технологиям NOR и NAND, и управляемую флэш-память в форме SD-карт, микросхем eMMC, флэш-памяти с интерфейсом USB и т. д.

В-четвертых, очень полезен порт для отладки, обычно это порт последовательного интерфейса RS-232. Оснащать им готовое изделие не обязательно, но на этапе подготовки платы к производству он заметно ускоряет разработку и отладку.

В-пятых, если вы начинаете с чистого листа, то понадобятся какие-то средства загрузки программы. Несколько лет назад для этой цели платы оснащались интерфейсом JTAG, но современные системы на кристалле умеют читать код начальной загрузки прямо со съемного запоминающего устройства, в частности карты SD или microSD, или из последовательного интерфейса типа RS-232 или USB.

Помимо этих базовых вещей, существуют интерфейсы к специальному оборудованию, необходимому для работы устройства. В стандартную комплектацию Linux входят открытые драйверы для тысяч разных устройств, и существуют также драйверы (разного качества), написанные производителями SoC-систем и сторонних микросхем, которые тоже можно включить в проект. Однако не забывайте о моих комментариях по поводу добросовестности и возможностей производителей. Любой разработчик встраиваемых устройств тратит уйму времени на оценку и адаптацию стороннего кода, если он доступен, и на консультации с производителем в противном случае. Наконец, вам предстоит написать драйверы для уникальных интерфейсов разрабатываемого устройства или подрядить кого-то для выполнения этой работы.

Используемое оборудование

Приведенные в книге примеры носят общий характер, но чтобы их можно было выполнить, я был вынужден выбрать какое-то конкретное устройство. В качестве таковых я использую BeagleBone Black и QEMU. Первое – широко распространенная дешевая макетная плата, которую можно использовать для разработки серьезных встраиваемых систем. Второе – эмулятор, применяемый для создания широкого спектра типичного встраиваемого оборудования. Был соблазн ограничиться только QEMU, но, как и любой эмулятор, он все же не полностью имитирует реальное устройство. Работая с BeagleBone, вы получаете удовольствие от взаимодействия с настоящим оборудованием и видите, как мигают лампочки. Был также соблазн взять что-то поновее платы BeagleBone Black, которая была выпущена несколько лет назад, но я полагаю, что заслуженная популярность обеспечила ей долгую жизнь, так что еще сколько-то лет она никуда не денется.

Как бы то ни было, советую выполнить как можно больше примеров на одной из этих платформ или на любой другой, которая окажется под рукой.

Плата BeagleBone Black

BeagleBone и более поздняя версия BeagleBone Black – это макетные платы размером с кредитную карту производства компании Circuitco LLC, проходящие по разряду открытого аппаратного обеспечения. Авторитетный источник информации о них – сайт www.beagleboard.org. Вот их основные технические характеристики:

- TI AM335x 1 ГГц ARM® Cortex-A8 Sitara SoC;
- ОЗУ 512 МиБ DDR3;
- флэш-память на плате: 8-разрядная eMMC объемом 2 или 4 ГиБ;
- последовательный порт для разработки и отладки;
- разъем для карты microSD, которую можно использовать как загрузочное устройство;
- порт хоста/устройства mini-USB OTG, который можно использовать для подачи питания плате;
- полноразмерный порт хоста USB 2.0;
- порт Ethernet 10/100;
- порт HDMI для аудио- и видеовыхода.

Кроме того, имеются два 46-контактных разъема для подключения карт расширения, каковых существует великое множество, – они позволяют приспособить плату для выполнения различных функций. Впрочем, в этой книге карты расширения не используются.

Помимо самой платы, нам понадобятся:

- переходник между mini-USB и полноразмерным USB (прилагается к плате) для подачи питания, если только у вас нет аксессуара, указанного в конце этого списка;
- кабель RS-232, который можно подключить к 6-контактному элементу TTL с напряжением питания 3,3 В, имеющемуся на плате. На сайте BeagleBoard приведены ссылки на совместимые кабели;

- карта microSD и средства записи на нее с настольного ПК или ноутбука, на котором ведется разработка, она будет нужна для загрузки программы в память платы;
- Ethernet-кабель, поскольку для некоторых примеров нужен доступ к сети;
- необязательно, но рекомендуется 5-вольтовый источник питания с силой тока не менее 1 А.

QEMU

QEMU – это эмулятор оборудования. Он поставляется в разных вариантах, предназначенных для эмуляции той или иной архитектуры процессора и ряда плат на основе этой архитектуры, например:

- qemu-system-arm: ARM;
- qemu-system-mips: MIPS;
- qemu-system-ppc: PowerPC;
- qemu-system-x86: x86 and x86_64.

Для каждой архитектуры QEMU эмулирует различное оборудование, перечень которого можно увидеть, запустив эмулятор с параметром `-machine help`. Каждая «машина» эмулирует большую часть оборудования, обычно присутствующего на данной плате. Существует возможность связать оборудование с локальными ресурсами, скажем, использовать локальный файл в качестве эмулируемого дискового накопителя. Вот конкретный пример:

```
$ qemu-system-arm -machine vexpress-a9 -m 256M -drive file=rootfs.ext4,sd
-net nic -net use -kernel zImage -dtb vexpress-v2p-ca9.dtb
-append "console=ttyAMA0,115200 root=/dev/mmcblk0" - serial stdio
-net nic,model=lan9118 -net tap,ifname=tap0
```

Опишем параметры этой команды:

- `-machine vexpress-a9`: эмулировать макетную плату ARM Versatile Express с процессором Cortex A-9;
- `-m 256M`: наделить ее ОЗУ объемом 256 МиБ;
- `-drive file=rootfs.ext4,sd`: связать интерфейс `sd` с локальным файлом `rootfs.ext4` (который содержит образ файловой системы);
- `-kernel zImage`: загрузить ядро Linux из локального файла с именем `zImage`;
- `-dtb vexpress-v2p-ca9.dtb`: загрузить дерево устройств из локального файла `vexpress-v2p-ca9.dtb`;
- `-append "..."`: использовать эту строку как строку для инициализации ядра;
- `-serial stdio`: присоединить последовательный порт к терминалу, с которого запускался QEMU, обычно это делается для того, чтобы можно было войти в эмулируемую машину с последовательной консоли;
- `-net nic,model=lan9118`: создать сетевой интерфейс;
- `-net tap,ifname=tap0`: присоединить сетевой интерфейс к интерфейсу виртуальной сети `tap0`.

Чтобы сконфигурировать часть сети, относящуюся к хосту, понадобится утилита `tunctl` из проекта **User Mode Linux (UML)**; в дистрибутивах Debian и Ubuntu

пакет называется `uml-utilities`. С помощью этой утилиты виртуальная сеть создается следующим образом:

```
$ sudo tunctl -u $(whoami) -t tap0
```

Эта команда создает сетевой интерфейс `tap0`, который подключается к сетевому контроллеру эмулируемой QEMU машины. Затем `tap0` настраивается точно так же, как любой интерфейс.

Все параметры будут подробно описаны в последующих главах. В большинстве примеров используется плата Versatile Express, но взять другую машину или архитектуру столь же просто.

Используемое программное обеспечение

Все используемые инструменты разработки, целевая операционная система и приложения – программы с открытым исходным кодом. Предполагается, что разработка ведется в Linux. Все команды, выполняемые хостом, тестировались в системе Ubuntu 14.04, но должен подойти любой современный дистрибутив Linux.

Резюме

Встраиваемое оборудование со временем будет становиться все сложнее – в полном соответствии с законом Мура. Linux обладает достаточными возможностями и гибкостью для эффективной работы с оборудованием.

Linux – лишь один компонент ПО с открытым исходным кодом. Помимо него, для создания готового изделия нужно еще много чего. Поскольку весь код свободно доступен, свой вклад могут вносить многие люди и организации. Однако из-за многообразия встраиваемых платформ и высокой скорости разработки образуются изолированные островки ПО, для которых совместное владение организовано не настолько эффективно, как могло бы быть. Во многих случаях вы попадаете в зависимость от этого ПО, особенно если ядро Linux поставляет изготовитель SoC-системы или печатной платы. К набору инструментов это относится в меньшей степени. Некоторые изготовители SoC-систем более ответственно относятся к публикации внесенных изменений, тогда их сопровождение упрощается.

По счастью, имеются эффективные инструменты, позволяющие разрабатывать и сопровождать программное обеспечение устройства. Так, Buildroot идеален для небольших систем, а Yocto Project – для более крупных.

Прежде чем переходить к описанию этих средств сборки, я рассмотрю четыре элемента встраиваемой Linux-системы, которые встречаются во всех проектах, как бы они ни создавались. Следующая глава посвящена первому элементу – набору инструментов, необходимому, чтобы откомпилировать код для целевой платформы.

Глава 2

О наборах инструментов

Набор инструментов – это первый элемент встраиваемой Linux-системы и отправная точка проекта. Решения, принятые на этом раннем этапе, оказывают решающее влияние на конечный результат. Набор инструментов должен эффективно использовать имеющееся оборудование: выбирать оптимальный набор команд процессора, задействовать блок вычислений с плавающей точкой, если он присутствует, и т. д. Он должен поддерживать необходимые вам языки программирования и достаточно полно реализовывать стандарт POSIX и другие системные интерфейсы. При этом должны быть доступны обновления в случае обнаружения ошибок или уязвимостей. Наконец, выбранный набор инструментов не должен меняться на протяжении проекта. Произвольная замена компиляторов или библиотек может стать причиной тонких ошибок.

Получить набор инструментов просто – нужно лишь скачать и установить пакет. Однако сам он весьма сложен, что я и продемонстрирую в этой главе.

Что такое набор инструментов?

Набор инструментов предназначен для компиляции исходного кода в исполняемый файл, который можно выполнить на целевом устройстве. Он включает компилятор, компоновщик и библиотеки времени выполнения. Прежде всего набор инструментов нужен, чтобы собрать остальные три элемента встраиваемой Linux-системы: начальный загрузчик, ядро и корневую файловую систему. Инструменты должны уметь компилировать код, написанный на языке ассемблера, C и C++, поскольку именно эти языки используются в основных пакетах с открытым исходным кодом.

Обычно наборы инструментов для Linux основаны на компонентах из проекта GNU (http://www.gnu.org), и на момент написания книги это остается справедливым в большинстве случаев. Однако в последние несколько лет компилятор Clang и связанный с ним проект LLVM (http://llvm.org) стали вполне зрелыми продуктами и могут считаться альтернативой инструментам от GNU. Одно из основных различий между LLVM и GNU заключается в политике лицензирования: инструменты LLVM распространяются на условиях лицензии BSD, тогда как проект GNU основан на лицензии GPL. У Clang есть и некоторые технические преимущества,

в частности он быстрее компилирует и выдает более качественную диагностику, зато GNU GCC совместим с существующей кодовой базой и поддерживает широчайший спектр архитектур и операционных систем. До сих пор существуют области, в которых Clang не может заменить компилятор GNU C, особенно в части компиляции стандартного ядра Linux. Но очень вероятно, что через год-два Clang будет способен откомпилировать все компоненты встраиваемых Linux-систем и, следовательно, составить полноценную конкуренцию GNU. На странице http://clang.llvm.org/docs/CrossCompilation.html подробно описано, как использовать Clang для кросс-компиляции. Если вы хотите включить его в состав системы сборки встраиваемой Linux-системы, то обратите внимание на проект EmbToolkit (https://www.embtoolkit.org), который поддерживает наборы инструментов GNU и LLVM/Clang. Кроме того, разные люди работают над включением Clang в проекты Buildroot и Yocto Project. Системы сборки рассматриваются в главе 6, а пока сосредоточимся на наборе инструментов GNU, поскольку в настоящее время это единственный вариант, содержащий все необходимое.

Стандартный набор инструментов GNU состоит из трех основных компонентов.

- **Binutils**: набор двоичных утилит, включающий ассемблер и компоновщик ld. Размещен по адресу http://www.gnu.org/software/binutils/.
- **Набор компиляторов GNU (GCC)**: включает компиляторы C и других языков: C++, Objective-C, Objective-C++, Java, Fortran, Ada и Go (в зависимости от версии). Все они пользуются общим кодогенератором, который порождает ассемблерный код, поступающий на вход ассемблера GNU. Размещен по адресу http://gcc.gnu.org/.
- **Библиотека C**: стандартизованный API на основе спецификации POSIX, описывающей интерфейс между ядром операционной системы и приложениями. Есть несколько заслуживающих внимания библиотек на C, они будут описаны в следующем разделе.

Кроме того, понадобятся заголовочные файлы ядра Linux, содержащие определения и константы, необходимые для прямого доступа к ядру. Прямо сейчас они потребуются для компиляции библиотеки C, а впоследствии – при разработке программ и компиляции библиотек, обращающихся к конкретным устройствам Linux, например для вывода графики с помощью драйвера буфера кадра. Но нельзя просто скопировать заголовки из каталога `include` в исходном коде ядра, поскольку они предназначены только для использования ядром, и содержащиеся в них определения приведут к конфликтам при попытке включить их в код обычного приложения.

Вместо этого нужно сгенерировать набор «подчищенных» заголовков ядра, как будет показано в главе 5.

Обычно не так важно, сгенерированы ли заголовки ядра по коду именно той версии Linux, с которой вы собираетесь работать, или какой-то другой. Поскольку интерфейсы ядра всегда сохраняют обратную совместимость, нужно лишь, чтобы заголовки были взяты из версии не младше той, что будет использоваться в целевом устройстве.

Многие считают отладчик GNU, GDB, частью набора инструментов, поэтому он обычно тоже собирается на этом этапе. Об отладчике GDB речь пойдет в главе 12.

Типы наборов инструментов – платформенные и перекрестные

Нас будут интересовать два типа наборов инструментов.

- **Платформенный.** Работает в системе того же типа (иногда просто той же самой), что и система, для которой генерируются программы. Это типично для настольных ПК и серверов, но приобретает популярность и для некоторых классов встраиваемых устройств. Например, в системе Raspberry Pi, работающей под управлением Debian для ARM, есть собственные платформенные компиляторы.
- **Перекрестный.** Работает в системе, отличной от целевой платформы, т. е. позволяет, например, вести разработку на быстром настольном ПК и загружать сгенерированный код во встраиваемое устройство для тестирования.

Почти всегда встраиваемые Linux-системы разрабатываются с помощью перекрестных наборов инструментов, отчасти потому, что встраиваемые устройства не обладают достаточными ресурсами: быстродействием, ОЗУ и внешней памятью, но также для того, чтобы разделять исходное и целевое окружения. Последнее особенно важно, когда исходная и целевая платформы имеют общую архитектуру, например X86_64. В таком случае возникает искушение откомпилировать код в исходной системе и просто скопировать двоичные файлы в целевую. До какого-то момента это будет работать, но весьма вероятно, что исходная система обновляется чаще, чем целевая, поэтому может случиться, что разные программисты собирают систему, используя слегка различающиеся версии библиотек, а значит, нарушен принцип постоянства набора инструментов на всем протяжении проекта. Чтобы этот подход работал корректно, окружение сборки в исходной и целевой системах необходимо изменять синхронно, но гораздо проще разделить обе системы, это как раз и позволяет сделать перекрестный набор инструментов.

Однако есть и контраргумент в пользу платформенной разработки. При перекрестной разработке приходится кросс-компилировать все библиотеки и инструменты, необходимые для целевой платформы. Ниже в этой главе мы увидим, что кросс-компиляция – не всегда простое дело, потому что большинство пакетов с открытым исходным кодом не рассчитано на такую сборку. Интегрированные инструменты сборки, в том числе Buildroot и Yocto Project, решают эту проблему, инкапсулируя правила кросс-компиляции ряда пакетов, требуемых в типичной встраиваемой системе, но если нужно откомпилировать много дополнительных пакетов, то лучше делать это на той же платформе, где они будут исполняться. Например, невозможно собрать дистрибутив Debian для Raspberry Pi и BeagleBone с помощью кросс-компилятора, приходится прибегать к платформенным инструментам. Создать платформенное окружение сборки с нуля – непростая задача, для ее решения нужно сначала собрать кросс-компилятор, с его помощью построить

платформенное окружение сборки, в котором уже можно будет собирать пакеты. Для сборки потребуется целая ферма, состоящая из хорошо подготовленных целевых плат, хотя, если повезет, можно будет использовать QEMU для эмуляции целевой платформы. Если вас заинтересовала эта тема, можете познакомиться с проектом Scratchbox, который переживает свое второе рождение в облике Scratchbox2 (https://maemo.gitorious.org/scratchbox2). Проект был начат компанией Nokia для создания операционной системы Maemo Linux, а теперь используется в проектах Mer, Tizen и др.

Ну а в этой главе мы продолжим знакомство с более распространенным окружением для кросс-компиляции, которое сравнительно просто настроить и администрировать.

Архитектура процессора

Набор инструментов должен быть создан с учетом возможностей целевого процессора, в том числе:

- **Архитектура процессора**: arm, mips, x86_64 и т. д.
- **Порядок байтов в слове**: некоторые процессоры могут работать в обоих режимах (тупоконечном и остроконечном), но машинный код будет различаться.
- **Поддержка арифметики с плавающей точкой**: не во всех встраиваемых процессорах имеется аппаратный блок для операций с плавающей точкой, и в таком случае набор инструментов нужно сконфигурировать для вызова программной библиотеки.
- **Двоичный интерфейс приложений (ABI)**: соглашение о передаче параметров при вызове функций.

Для многих архитектур ABI один и тот же для всего семейства процессоров. Заметное исключение составляет архитектура ARM, для которой в конце 2000-х годов произошел переход на расширенный двоичный интерфейс приложений (**EABI**), а прежний ABI стал называться старым двоичным интерфейсом приложений (**OABI**). Хотя OABI теперь считается устаревшим, до сих пор можно встретить упоминания EABI. С той поры EABI разделился на два интерфейса в зависимости от метода передачи параметров с плавающей точкой. В оригинальном EABI используются регистры общего назначения (целочисленные), а в более позднем EABIHF – регистры с плавающей точкой. Интерфейс EABIHF показывает гораздо более высокое быстродействие на операциях с плавающей точкой, поскольку устраняет необходимость копирования между целыми и плавающими регистрами, однако он несовместим с процессорами, не оснащенными блоком операций с плавающей точкой. Следовательно, приходится выбирать между двумя несовместимыми ABI: использовать в одной программе оба нельзя, и решение нужно принимать на этом этапе.

В проекте GNU в названиях инструментов используется префикс, описывающий допустимые комбинации генерируемого кода. Префикс состоит из трех или четырех частей, разделенных дефисами.

- **Процессор**: архитектура процессора, например arm, mips или x86_64. Если процессор поддерживает оба порядка байтов, то для их различения можно добавить суффикс el (остроконечный – сначала младший байт) или eb (тупоконечный – сначала старший байт). Примерами могут служить остроконечный MIPS (mipsel) и тупоконечный ARM (armeb).
- **Поставщик**: производитель набора инструментов, например: buildroot, poky или просто unknown. Иногда эта часть вообще опускается.
- **Ядро**: для наших целей всегда 'linux'.
- **Операционная система**: имя компонента в пользовательском адресном пространстве: `gnu` или `uclibcgnu`. В конце может быть также указан ABI, так что наборы инструментов для ARM могут включать строки `gnueabi`, `gnueabihf`, `uclibcgnueabi` или `uclibcgnueabihf`.

Какая четверка была использована при сборке набора инструментов, можно узнать, выполнив команду с флагом `-dumpmachine`. Так, в исходной системе можно увидеть вот что:

```
$ gcc -dumpmachine
x86_64-linux-gnu
```

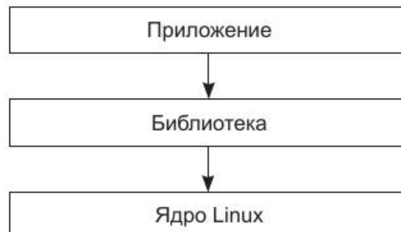
При установке на машину платформенного компилятора обычно создаются ссылки на все входящие в набор инструменты без префиксов, чтобы, к примеру, компилятор можно было запускать, набрав просто команду `gcc`.

Ниже приведен пример запуска кросс-компилятора:

```
$ mipsel-unknown-linux-gnu-gcc -dumpmachine
mipsel-unknown-linux-gnu
```

Выбор библиотеки C

Программный интерфейс обращения к операционной системе Unix определен на языке C и ныне закреплен в стандартах POSIX. Библиотека C представляет собой реализацию этого интерфейса – шлюз к ядру Linux из программ. Даже если программа написана на другом языке, скажем Java или Python, сопровождающие его библиотеки времени выполнения в конечном итоге должны будут обратиться к библиотеке C.



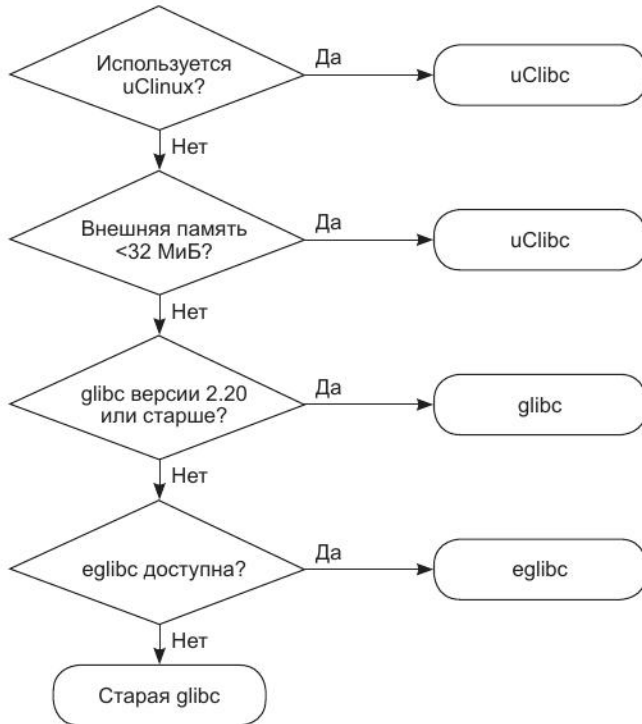
Библиотека C – шлюз к ядру из приложений

Когда библиотеке C необходимо получить доступ к сервису ядра, она производит системный вызов для перехода из пользовательского пространства в пространство ядра. Есть возможность вызвать ядро напрямую в обход библиотеки C, но это весьма трудоемко и почти никогда не нужно.

Существует несколько библиотек C на выбор. Перечислим основные.

- **glibc**. Размещена по адресу http://www.gnu.org/software/libc. Это стандартная библиотека GNU C. Она очень большая и до недавнего времени не допускала особой настройки, но содержит самую полную реализацию POSIX API.
- **eglibc**. Размещена по адресу http://www.eglibc.org/home. Это встраиваемая версия GLIBC. Раньше представляла собой набор заплат к glibc, которые добавляли возможности настройки и поддержку архитектур, не охваченных glibc (конкретно PowerPC e500). Разделение между eglibc и glibc всегда было несколько искусственным, но, к счастью, начиная с версии 2.20 код eglibc снова объединен с glibc, так что мы получили одну улучшенную библиотеку. Сопровождение eglibc прекращено.
- **uClibc**. Размещена по адресу http://www.uclibc.org. Буква 'u' на самом деле замещает греческую μ и означает, что речь идет о библиотеке C для микроконтроллеров. Изначально разрабатывалась для работы в uClinux (Linux для процессоров без блоков управления памятью), но затем была адаптирована для работы с полной версией Linux. Существует утилита конфигурирования, позволяющая точно настроить функциональность библиотеки для конкретных нужд. Даже в полной конфигурации эта библиотека меньше glibc, но уступает в полноте реализации стандартов POSIX.
- **musl libc**. Размещена по адресу http://www.musl-libc.org. Новая библиотека C, предназначенная для встраиваемых систем.

Так какую выбрать? Я рекомендую использовать uClibc, только если вы работаете с uClinux или очень ограничены в объеме ОЗУ либо внешней памяти, так что небольшой размер является преимуществом. В противном случае я предпочитаю современную версию glibc или eglibc. У меня нет опыта работы с musl libc, но если вы сочтете, что glibc (eglibc) не годится, можете устремить взоры в этом направлении. Процесс принятия решения показан на следующей блок-схеме.



Выбор библиотеки C

Получение набора инструментов

Есть три варианта получить набор инструментов: найти уже собранный набор, отвечающий вашим потребностям; взять тот, что сгенерирован инструментом сборки встраиваемой системы (см. главу 6), или собрать самостоятельно, как описано ниже в этой главе.

Готовый набор инструментов привлекателен тем, что вам остается только скачать и установить его, но при этом вы ограничены заданной кем-то конфигурацией и зависите от поставщика. Скорее всего, в роли поставщика будет выступать:

- Производитель SoC-системы или платы. Большинство производителей предлагает набор инструментов для Linux.
- Консорциум, организованный для системной поддержки определенной архитектуры. Например, некоммерческая организация Linaro (https://www.linaro.org) распространяет готовые наборы инструментов для архитектуры ARM.
- Сторонние поставщики инструментов для Linux, например: Mentor Graphics, TimeSys или MontaVista.
- Пакеты перекрестных инструментов, входящие в дистрибутив Linux для настольных ПК. Например, в дистрибутивах, производных от Debian, есть

пакеты кросс-компиляции для платформ ARM, MIPS и PowerPC.

- Двоичный SDK, сгенерированный одним из интегрированных инструментов сборки встраиваемых систем. В проекте Yocto Project есть примеры на странице http://autobuilder.yoctoproject.org/pub/releases/CURRENT/toolchain, а по адресу ftp://ftp.denx.de/pub/eldk/ выложен комплект средств разработки Denx Embedded Linux Development Kit.
- Опубликованная на каком-то форуме ссылка, которую вы больше не можете найти.

В любом случае предстоит решить, отвечает ли готовый набор инструментов вашим требованиям. Используется ли в нем предпочтительная для вас библиотека C? Публикует ли поставщик обновления для исправления ошибок и устранения уязвимостей (учтите при этом замечания по поводу поддержки и обновления, приведенные в главе 1). Если ответ хотя бы на один вопрос отрицательный, то стоит подумать о самостоятельном построении набора инструментов.

Увы, собрать набор инструментов не так-то просто. Если вы твердо решили пройти этот путь до конца, зайдите на сайт *Cross Linux From Scratch* (http://trac.clfs.org). Там приведены пошаговые инструкции по созданию каждого компонента.

Есть и более простой способ – воспользоваться средством crosstool-NG, которое инкапсулирует весь процесс в набор скриптов, управляемый меню. Но для работы с ним все равно нужно много знать.

Проще прибегнуть к системе сборки типа Buildroot или Yocto Project, поскольку генерация набора инструментов в этом случае является составной частью процесса сборки. Лично я предпочитаю именно такое решение и опишу его в главе 6.

Сборка набора инструментов с помощью crosstool-NG

Я начну с использования crosstool-NG, потому что это позволит нам увидеть весь процесс создания набора инструментов и создать несколько таких наборов разного типа.

Несколько лет назад Дэн Кегел (Dan Kegel) написал ряд скриптов и make-файлов для генерации перекрестных наборов инструментов и назвал их crosstool (kegel.com/crosstool). В 2007 году Ианн Э. Морин (Yann E. Morin) взял их за основу для создания следующего поколения crosstool, получившего названия crosstool-NG (crosstool-ng.org). В настоящий момент это самый удобный способ сборки автономного перекрестного набора инструментов из исходного кода.

Установка crosstool-NG

Прежде всего понадобится работоспособный платформенный набор инструментов и средства сборки на вашем исходном ПК. Для работы с crosstool-NG в системе Ubuntu нужно установить следующие пакеты:

```
$ sudo apt-get install automake bison chrpath flex g++ git gperf gawk
libexpat1-dev libncurses5-dev libsdl1.2-dev libtool
python2.7-dev texinfo
```

Затем скачайте текущую версию croostool-NG со страницы http://crosstool-ng.org/download/crosstool-ng. В примерах ниже используется версия 1.20.0. Распакуйте архив и создайте интерфейсную программу на основе ct-ng:

```
$ tar xf crosstool-ng-1.20.0.tar.bz2
$ cd crosstool-ng-1.20.0
$ ./configure --enable-local
$ make
$ make install
```

Флаг `--enable-local` означает, что программа будет установлена в текущий каталог. Это позволяет обойтись без привилегий root, которые понадобились бы для установки в каталог по умолчанию `/usr/local/bin`. Для запуска меню наберите `./ct-ng`, находясь в текущем каталоге.

Выбор набора инструментов

Программа Crosstool-NG умеет строить различные комбинации наборов инструментов. Для упрощения начальной настройки в ее состав входит ряд примеров, покрывающих многие типичные случаи. Чтобы получить их список, выполните команду `./ct-ng list-samples`.

Предположим, что целевой платформой является плата BeagleBone Black с процессорным ядром ARM Cortex A8 и блоком операций с плавающей точкой VFPv3 и что мы хотим использовать текущую версию glibc. Ближе всего к этой комбинации подходит пример `arm-cortex_a8-linux-gnueabi`. Чтобы узнать его текущую конфигурацию, добавим в начало имени префикс `show-`:

```
$ ./ct-ng show-arm-cortex_a8-linux-gnueabi
[L..] arm-cortex_a8-linux-gnueabi
OS              : linux-3.15.4
Companion libs  : gmp-5.1.3 mpfr-3.1.2 cloog-ppl-0.18.1 mpc-1.0.2 libelf-0.8.13
binutils        : binutils-2.22
C compiler      : gcc-4.9.1 (C,C++)
C library       : glibc-2.19 (threads: nptl)
Tools           : dmalloc-5.5.2 duma-2_5_15 gdb-7.8 ltrace- 0.7.3 strace-4.8
```

Чтобы выбрать именно эту целевую конфигурацию, выполните команду:

```
$ ./ct-ng arm-cortex_a8-linux-gnueabi
```

В этот момент можно просмотреть конфигурацию и внести необходимые изменения, воспользовавшись системой меню:

```
$ ./ct-ng menuconfig
```

Система меню основана на программе конфигурирования ядра Linux `menuconfig`, поэтому навигация по интерфейсу покажется знакомой любому, кто когда-нибудь конфигурировал ядро. Если вы не из их числа, обратитесь к главе 4, где описана программа `menuconfig`.

На этой стадии я рекомендую внести несколько изменений.

- В разделе **Paths and misc options** (Пути и разные параметры) выключите флажок **Render the toolchain read-only** (Генерировать доступный только для чтения набор инструментов) (`CT_INSTALL_DIR_RO`).
- В разделе **Target options → Floating point** (Параметры целевой платформы → Плавающая точка) выберите **hardware (FPU)** (аппаратная) (`CT_ARCH_FLOAT_HW`).
- В разделе **C-library → extra config** добавьте флаг **--enable-obsolete-rpc** (`CT_LIBC_GLIBC_EXTRA_CONFIG_ARRAY`).

Первое необходимо, если вы захотите добавить библиотеки в состав набора инструментов после его установкеи, о чем я расскажу ниже в этой главе. Второе служит для выбора оптимальной реализации операции с плавающей точкой в случае, когда процессор оснащен соответствующим аппаратным блоком. И последнее заставляет программу сгенерировать набор инструментов с устаревшим заголовком `rpc.h`, который все еще используется в ряде пакетов (отметим, что эта проблема существует, только если вы выбрали библиотеку glibc). Имена в скобках соответствуют меткам в конфигурационном файле. Внеся изменения, сохраните их и выйдите из `menuconfig`.

Конфигурационные данные сохраняются в файле `.config`. В первой его строчке написано *Automatically generated make config: don't edit* (Автоматически сгенерированный файл. Не редактировать). Это хороший совет, но сейчас мы им пренебрежем. Помните, обсуждая ABI набора инструментов для архитектуры ARM, мы упомянули о двух вариантах: передача параметров с плавающей точкой в целочисленных регистрах и использование только регистров с плавающей точкой? Выбранная конфигурация относится ко второму типу, поэтому часть, относящаяся к ABI, должна иметь вид `eabihf`. Существует конфигурационный параметр, который делает ровно то, что нам надо, но по умолчанию он выключен, а в меню отсутствует, по крайней мере в этой версии crosstool. Поэтому придется отредактировать файл `.config`, добавив в него строки, выделенные полужирным шрифтом:

```
[...]
#
# arm other options
#
CT_ARCH_ARM_MODE="arm"
CT_ARCH_ARM_MODE_ARM=y
# CT_ARCH_ARM_MODE_THUMB is not set
# CT_ARCH_ARM_INTERWORKING is not set
CT_ARCH_ARM_EABI_FORCE=y
CT_ARCH_ARM_EABI=y
CT_ARCH_ARM_TUPLE_USE_EABIHF=y
[...]
```

Теперь можно использовать crosstool-NG, чтобы получить, сконфигурировать и собрать компоненты в соответствии с заданной спецификацией:

```
$ ./ct-ng build
```

Сборка занимает примерно полчаса, построенный набор инструментов помещается в каталог ~/x-tools/arm-cortex_a8-linux-gnueabihf/.

Анатомия набора инструментов

Чтобы вы могли составить представление о типичном наборе инструментов, я рассмотрю только что созданный набор.

Набор инструментов находится в каталоге ~/x-tools/arm-cortex_a8-linux-gnueabihf/bin. Там вы найдете кросс-компилятор arm-cortex_a8-linux-gnueabihf-gcc. Чтобы им воспользоваться, нужно будет добавить каталог в переменную окружения PATH:

```
$ PATH=~/x-tools/arm-cortex_a8-linux-gnueabihf/bin:$PATH
```

Теперь напишем простейшую программу hello world:

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char *argv[])
{
printf ("Hello, world!\n");
return 0;
}
```

И откомпилируем ее:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -o helloworld
```

Выполнив команду file, которая печатает тип файла, убеждаемся, что она действительно кросс-компилирована:

```
$ file helloworld
helloworld: ELF 32-bit LSB executable, ARM, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 3.15.4, not stripped
```

Получение информации о кросс-компиляторе

Допустим, вы только что получили набор инструментов и хотели бы узнать, как он был сконфигурирован. Многое может сообщить сам gcc. Например, версию можно получить, задав флаг --version:

```
$ arm-cortex_a8-linux-gnueabi-gcc --version
arm-cortex_a8-linux-gnueabi-gcc (crosstool-NG 1.20.0) 4.9.1
Copyright (C) 2014 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Чтобы узнать, как был сконфигурирован кросс-компилятор, зададим флаг -v:

```
$ arm-cortex_a8-linux-gnueabi-gcc -v
Using built-in specs.
COLLECT_GCC=arm-cortex_a8-linux-gnueabihf-gcc
```

```
COLLECT_LTO_WRAPPER=/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/libexec/gcc/arm-cor-
tex_a8-linux-gnueabihf/4.9.1/lto-wrapper
Target: arm-cortex_a8-linux-gnueabihf
Configured with: /home/chris/hd/home/chris/build/MELP/build/crosstool-ng- 1.20.0/.build/
src/gcc-4.9.1/configure --build=x86_64-build_unknown-linux-gnu --host=x86_64-build_unknown-
linux-gnu --target=arm-cortex_a8-linux-gnueabihf --prefix=/home/chris/x-tools/arm-cortex_
a8- linux-gnueabihf --with-sysroot=/home/chris/x-tools/arm-cortex_a8- linux-gnueabihf/
arm-cortex_a8-linux-gnueabihf/sysroot --enable-languages=c,c++ --with-arch=armv7-a --with-
cpu=cortex-a8 --with-tune=cortex-a8 --with-float=hard --with-pkgversion='crosstool-
NG 1.20.0' --enable-__cxa_atexit --disable-libmudflap --disable-libgomp --disable-
libssp --disable-libquadmath --disable-libquadmath-support --disable-libsanitiz-
er --with-gmp=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng- 1.20.0/.build/
arm-cortex_a8-linux-gnueabihf/buildtools --with-mpfr=/home/chris/hd/home/chris/build/MELP/
build/crosstool-ng- 1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-mpc=/
home/chris/hd/home/chris/build/MELP/build/crosstool-ng- 1.20.0/.build/arm-cortex_a8-linux-
gnueabihf/buildtools --with-isl=/home/chris/hd/home/chris/build/MELP/build/crosstool-
ng- 1.20.0/.build/arm-cortex_a8-linux-gnueabihf/buildtools --with-cloog=/home/chris/hd/
home/chris/build/MELP/build/crosstool-ng- 1.20.0/.build/arm-cortex_a8-linux-gnueabihf/
buildtools --with-libelf=/home/chris/hd/home/chris/build/MELP/build/crosstool-ng- 1.20.0/.
build/arm-cortex_a8-linux-gnueabihf/buildtools --with-host-libstdcxx='-static-libgcc -Wl,-
Bstatic,-lstdc++,-Bdynamic -lm' -- enable-threads=posix --enable-target-optspace --en-
able-plugin -- enable-gold --disable-nls --disable-multilib --with-local-prefix=/home/
chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot --enable-
c99 --enable-long-long
Thread model: posix
gcc version 4.9.1 (crosstool-NG 1.20.0)
```

Напечатано много, но отметим самые интересные строки:

- `--with-sysroot=/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-linux-gnueabihf/sysroot`: это путь к каталогу sysroot по умолчанию, пояснения приведены ниже;
- `--enable-languages=c,c++`: это означает, что включена поддержка языков C и C++;
- `--with-arch=armv7-a`: кодогенератор использует набор команд ARM v7a;
- `--with-cpu=cortex-a8 and --with-tune=cortex-a8`: код оптимизирован для процессорного ядра Cortex A8;
- `--with-float=hard`: генерируются коды команд блока операций с плавающей точкой, параметры передаются в регистрах с плавающей точкой (VFP);
- `--enable-threads=posix`: включена поддержка потоков POSIX.

Это все параметры по умолчанию, используемые компилятором. Большинство из них можно переопределить в командной строке gcc. Например, чтобы откомпилировать программу для другого процессора, нужно переопределить сконфигурированный параметр `--with-cpu`, добавив в командную строку флаг `mcpu`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -mcpu=cortex-a5 helloworld.c -o helloworld
```

Для вывода списка архитектурно-зависимых параметров зададим флаг `target-help`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc --target-help
```

Возникает вопрос: так ли важно задавать точную конфигурацию в момент генерации набора инструментов, если впоследствии ее можно изменить. Ответ зависит от того, как вы планируете использовать инструменты. Если вы собираетесь создавать новый набор инструментов для каждой целевой платформы, то имеет смысл задать все с самого начала, чтобы уменьшить риски ошибки впоследствии. Забегая вперед, скажу, что я называю такой подход «философией Buildroot». Если же вы хотите собрать обобщенный набор инструментов и готовы подставлять правильные параметры при сборке для конкретной целевой платформы, то требования к точности конфигурации не так важны. Такой точки зрения придерживается проект Yocto Project. В примерах выше мы следовали философии Buildroot.

sysroot, библиотека и файлы-заголовки

В терминологии набора инструментов sysroot – это каталог, содержащий подкаталоги для библиотек, файлов-заголовков и других конфигурационных файлов. Его можно задать при конфигурировании набора с помощью флага `with-sysroot=` или в командной строке с помощью флага `sysroot=`. Чтобы узнать, куда указывает sysroot по умолчанию, запустите компилятор с флагом `-print-sysroot`:

```
$ arm-cortex_a8-linux-gnueabi-gcc -print-sysroot
/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8-
linux-gnueabihf/sysroot
```

В каталоге sysroot находятся такие подкаталоги:

- `lib`: содержит разделяемые объекты библиотеки C и динамического компоновщика/загрузчика `ld-linux`;
- `usr/lib`: статическая библиотека C и другие библиотеки, установленные позже;
- `usr/include`: заголовки для всех библиотек;
- `usr/bin`: утилиты, исполняемые на целевой платформе, например `ldd`;
- `/usr/share`: файлы, необходимые для локализации и интернационализации;
- `sbin`: утилита `ldconfig` для оптимизации загрузки динамических библиотек.

Одни файлы нужны на машине разработки для компиляции программ, другие, например разделяемые библиотеки и `ld-linux`, – используются на целевой платформе во время выполнения.

Другие элементы набора инструментов

В таблице ниже перечислены другие команды, входящие в набор инструментов:

Команда	Описание
`addr2line`	Преобразует адреса программных объектов в имена файлов и номера строк, читая таблицы отладочных символов в исполняемом файле. Очень полезно для интерпретации адресов, встречающихся в отчете о крахе системы

Команда	Описание
ar	Утилита архивации, применяемая для создания статических библиотек
as	Ассемблер GNU
c++filt	Декодирование декорированных имен в C++ и Java
cpp	Препроцессор C, применяется для обработки директив #define, #include и им подобных. Вам вряд ли придется использовать его напрямую
elfedit	Применяется для изменения заголовков ELF-файлов
g++	Фаза анализа компилятора GNU C++, предполагающая, что исходный файл написан на C++
gcc	Фаза анализа компилятора GNU C, предполагающая, что исходный файл написан на C
gcov	Средство исследования покрытия кода
gdb	Отладчик GNU
gprof	Профилировщик
ld	Компоновщик GNU
nm	Выводит список символов в объектном файле
objcopy	Средство трансляции и копирования одних объектных файлов в другие
objdump	Применяется для вывода информации, содержащейся в объектном файле
ranlib	Создает или модифицирует индекс статической библиотеки, что уменьшает время компоновки
readelf	Выводит информацию об объектных файлах в формате ELF
size	Выводит размеры каждой секции и общий размер
strings	Выводит строки печатных символов, встречающиеся в файле
strip	Удаляет из объектного файла таблицы отладочных символов с целью уменьшения его размера. Обычно этой операции подвергаются все исполняемые файлы, копируемые на целевую платформу

Компоненты библиотеки C

Библиотека C – это не один файл. Она состоит из четырех основных частей, которые в совокупности реализуют API, описанный в стандартах POSIX.

- libc: главная библиотека C, содержащая такие хорошо известные функции, как printf, open, close, read, write и т. д.
- libm: математические функции: cos, exp, log и т. д.
- libpthread: все функции POSIX для работы с потоками, имена которых начинаются с pthread_.
- librt: расширения реального времени, добавленные к POSIX, включая функции для работы с разделяемой памятью и асинхронным вводом-выводом.

С первой библиотекой, libc, компонуется любая программа, но остальные нужно задавать явно с помощью флага -l. В качестве значения флага задается имя библиотеки без префикса lib. Так, программа, в которой вызывается функция вычисления синуса sin(), должна быть скомпонована с библиотекой libm путем указания флага -lm:

```
arm-cortex_a8-linux-gnueabihf-gcc myprog.c -o myprog -lm
```

Чтобы проверить, с какими библиотеками скомпонована эта или любая другая программа, воспользуемся командой readelf:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog | grep "Shared library"
0x00000001 (NEEDED) Shared library: [libm.so.6]
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

Для разделяемых библиотек необходим компоновщик времени выполнения, о чем свидетельствует такая команда:

```
$ arm-cortex_a8-linux-gnueabihf-readelf -a myprog |
grep "program interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
```

Это настолько полезно, что я написал скрипт оболочки, вызывающий показанные выше команды:

```
#!/bin/sh
${CROSS_COMPILE}readelf -a $1 | grep "program interpreter"
${CROSS_COMPILE}readelf -a $1 | grep "Shared library"
```

Статическая и динамическая компоновка с библиотеками

Любое приложение для Linux, написанное на C или на C++, компонуется с библиотекой C, libc. Это требование настолько непреложно, что компиляторы gcc и g++ даже не нужно просить – они всегда прикомпоновывают libc. Все остальные библиотеки нужно указывать явно с помощью флага -l.

Библиотечный код можно компоновать двумя способами: статически, когда все функции, которые вызывает приложение, а также их зависимости извлекаются из архивного файла библиотеки и включаются в исполняемый файл, и динамически, когда в исполняемый файл включаются только ссылки на библиотечные файлы и функции в них, но сама компоновка производится во время выполнения.

Статические библиотеки

Статическая компоновка полезна в нескольких случаях. Например, при создании небольшой системы, которая включает только BusyBox и несколько скриптов, проще скомпоновать BusyBox статически и не копировать библиотеки времени выполнения и компоновщик. Так и общий объем будет меньше, потому что мы прикомпоновываем только код, который приложение действительно вызывает, не копируя всю библиотеку C целиком. Статическая компоновка полезна и тогда, когда нужно выполнить некоторую программу еще до того, как станет доступна файловая система, содержащая динамически загружаемые библиотеки.

Чтобы gcc компоновал все библиотеки статически, нужно задать флаг -static:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -static helloworld.c -o helloworld-static
```

Обратите внимание, как сильно увеличился размер исполняемого файла:

```
$ ls -l
-rwxrwxr-x 1 chris chris   5323 Oct  9 09:01 helloworld
-rwxrwxr-x 1 chris chris 625704 Oct  9 09:01 helloworld-static
```

При статической компоновке код извлекается из архивного файла, который обычно имеет имя вида lib[name].a. В примере выше это файл libc.a, находящийся в каталоге [sysroot]/usr/lib:

```
$ ls -l $(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)/usr/lib/libc.a
```

```
-r--r--r-- 1 chris chris 3434778 Oct 8 14:00 /home/chris/x-tools/arm-cortex_a8-linux-gnue-
abihf/arm-cortex_a8-linux-gnueabihf/sysroot/usr/lib/libc.a
```

Отметим, что конструкция $(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot) помещает вывод программы в командную строку. Я использую ее как обобщенный способ сослаться на файлы в каталоге sysroot.

Для создания статической библиотеки нужно просто создать архив объектных файлов с помощью команды ar. Если бы я хотел создать статическую библиотеку libtest.a, содержащую файлы test1.c и test2.c, то поступил бы следующим образом:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -c test1.c
$ arm-cortex_a8-linux-gnueabihf-gcc -c test2.c
$ arm-cortex_a8-linux-gnueabihf-ar rc libtest.a test1.o test2.o
$ ls -l
total 24
-rw-rw-r-- 1 chris chris 2392 Oct  9 09:28 libtest.a
-rw-rw-r-- 1 chris chris  116 Oct  9 09:26 test1.c
-rw-rw-r-- 1 chris chris 1080 Oct  9 09:27 test1.o
-rw-rw-r-- 1 chris chris  121 Oct  9 09:26 test2.c
-rw-rw-r-- 1 chris chris 1088 Oct  9 09:27 test2.o
```

А затем я мог бы скомпоновать libtest с программой helloworld:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest -L../libs -I../libs -o helloworld
```

Разделяемые библиотеки

Чаще библиотеки развертывают по-другому: в виде разделяемых объектов, которые компонуются во время выполнения. Это позволяет более эффективно использовать оперативную и внешнюю память, поскольку существует лишь одна копия кода, загружаемая в разные программы. Упрощается и обновление библиотечных файлов, так как не требуется перекомпоновывать все программы, в которых библиотеки используются.

Объектный код разделяемой библиотеки должен быть позиционно независимым, чтобы динамический компоновщик мог разместить его в памяти со следующего свободного адреса. Для генерации такого кода нужно запускать gcc с флагом -fPIC, а компоновать с флагом -shared:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test1.c
$ arm-cortex_a8-linux-gnueabihf-gcc -fPIC -c test2.c
$ arm-cortex_a8-linux-gnueabihf-gcc -shared -o libtest.so test1.o test2.o
```

Чтобы скомпоновать приложение с этой библиотекой, мы добавили флаг -ltest, как и в статическом случае, но теперь в исполняемый файл включается не код библиотеки, а ссылка на нее, которую должен будет разрешить динамический компоновщик:

```
$ arm-cortex_a8-linux-gnueabihf-gcc helloworld.c -ltest  L../libs
I../libs -o helloworld
$ list-libs helloworld
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
0x00000001 (NEEDED)            Shared library: [libtest.so]
0x00000001 (NEEDED)            Shared library: [libc.so.6]
```

Для этой программы динамическим компоновщиком будет файл /lib/ld-linux-armhf.so.3, и он должен находиться в целевой файловой системе. Компоновщик будет искать файл libtest.so в стандартных каталогах: /lib и /usr/lib. Если вы хотите, чтобы он просматривал и другие каталоги, поместите список путей к ним, разделенных двоеточиями, в переменную окружения LD_LIBRARY_PATH:

```
# export LD_LIBRARY_PATH=/opt/lib:/opt/usr/lib
```

О нумерации версий разделяемых библиотек

Одно из преимуществ разделяемых библиотек состоит в том, что их можно обновлять независимо от программ, в которых они используются. Есть два типа обновлений: одни служат для исправления ошибок или добавления новых функций с сохранением обратной совместимости, а другие нарушают совместимость с существующими приложениями. В GNU/Linux принята схема нумерации версий, учитывающая оба случая.

У каждой библиотеки есть номер версии и номер интерфейса. Номер версии – это просто строка, добавляемая в конец имени библиотеки. Например, текущая версия библиотеки для работы с графическими файлами в формате JPEG, libjpeg, имеет номер 8.0.2, поэтому файл библиотеки называется libjpeg.so.8.0.2. Символическая ссылка libjpeg.so указывает на libjpeg.so.8.0.2, поэтому при компиляции программы с флагом -ljpeg мы фактически компонуем ее с текущей версией библиотеки. При установке версии 8.0.3 ссылка обновляется, так что компоновка будет произведена с новой версией.

Предположим теперь, что вышла версия 9.0.0, нарушающая обратную совместимость. Теперь ссылка libjpeg.so указывает на libjpeg.so.9.0.0, поэтому все новые программы компонуются с новой версией, и попытка использовать старый интерфейс, скорее всего, приведет к ошибкам компиляции, которые разработчик сможет исправить. Но программы в целевой системе, которые не были перекомпилированы, будут «падать», потому что по-прежнему используют старый интерфейс. На помощь приходит soname. В soname закодирован номер интерфейса, существовавший на момент сборки библиотеки и используемый динамическим

компоновщиком на этапе загрузки библиотеки. Он имеет вид <имя библиотеки>.so.<имя интерфейса>. Для библиотеки libjpeg.so.8.0.2 строка soname имеет вид libjpeg.so.8:

```
$ readelf -a /usr/lib/libjpeg.so.8.0.2 | grep SONAME
0x000000000000000e (SONAME) Library soname: [libjpeg.so.8]
```

Любая программа, скомпилированная с такой библиотекой, будет запрашивать файл libjpeg.so.8 во время выполнения, а на целевой платформе это будет символическая ссылка, указывающая на файл libjpeg.so.8.0.2. Для версии 9.0.0 библиотеки libjpeg soname имеет вид libjpeg.so.9, поэтому в одной системе могут одновременно находиться две несовместимые версии библиотеки. Программы, скомпонованные с libjpeg.so.8.*.*, будут загружать libjpeg.so.8, а скомпонованные с libjpeg.so.9.*.* будут загружать libjpeg.so.9.

Вот почему в каталоге <sysroot>/usr/lib/libjpeg* мы находим четыре файла:

- libjpeg.a: это архивный файл, нужный для статической компоновки;
- libjpeg.so -> libjpeg.so.8.0.2: это символическая ссылка, нужная для динамической компоновки;
- libjpeg.so.8 -> libjpeg.so.8.0.2: это символическая ссылка, нужная для загрузки библиотеки во время выполнения;
- libjpeg.so.8.0.2: это сама разделяемая библиотека, которая используется на этапах компиляции и выполнения.

Первые два файла нужны только на исходном компьютере и используются во время сборки, а последние два должны присутствовать в целевой системе во время выполнения.

Искусство кросс-компиляции

Получение перекрестного набора инструментов – не конец, а только начало пути. В какой-то момент вы захотите произвести кросс-компиляцию различных инструментов, приложений и библиотек, которые должны быть помещены в целевую систему. Многие из них – пакеты с открытым исходным кодом, и у каждого свой собственный метод компиляции и свои особенности. Существует несколько общеупотребительных систем сборки, в том числе:

- чистые make-файлы, в которых работа набора инструментов управляется переменной make CROSS_COMPILE;
- система сборки GNU Autotools;
- CMake (https://cmake.org).

Я расскажу только о первых двух, поскольку они нужны для сборки даже самой простой встраиваемой Linux-системы. Что касается CMake, то на вышеупомянутом сайте проекта имеются отличные учебные ресурсы.

Простые make-файлы

Некоторые важные пакеты, в том числе ядро Linux, начальный загрузчик U-Boot и набор утилит Busybox, очень легко поддаются кросс-компиляции. Нужно лишь

записать префикс набора инструментов, например `arm-cortex_a8-linux-gnueabi-`, в переменную make `CROSS_COMPILE`. Обратите внимание на завершающий дефис `-`.

Таким образом, для компиляции Busybox нужно набрать:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Можно также записать префикс в переменную оболочки:

```
$ export CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
$ make
```

В случае U-Boot и Linux нужно еще записать в переменную make `ARCH` одну из поддерживаемых машинных архитектур; подробнее я расскажу об этом в главе 3 «Все о начальных загрузчиках» и в главе 4 «Портирование и конфигурирование ядра».

Autotools

За общим названием Autotools скрывается комплект инструментов, применяемых для сборки во многих проектах с открытым исходным кодом. Ниже перечислены отдельные инструменты вместе с адресами проектов:

- GNU Autoconf (http://www.gnu.org/software/autoconf/autoconf.html);
- GNU Automake (http://www.gnu.org/savannah-checkouts/gnu/automake);
- GNU Libtool (http://www.gnu.org/software/libtool/libtool.html);
- Gnulib (https://www.gnu.org/software/gnulib).

Идея Autotools – сгладить различия между системами разных типов, для которых может компилироваться пакет, учтя при этом различные версии компиляторов и библиотек, разные места нахождения файлов-заголовков и зависимости от других пакетов. Все пакеты, собираемые с помощью Autotools, содержат скрипт `configure`, который проверяет зависимости и генерирует make-файлы в соответствии с тем, что обнаруживает в системе. Кроме того, скрипт `configure` часто дает возможность включать или выключать отдельные функциональные возможности. Для получения списка параметров скрипта запустите его командой `./configure --help`.

Для конфигурирования, сборки и установки пакета обычно нужно выполнить такую последовательность команд:

```
$ ./configure
$ make
$ sudo make install
```

Autotools пригоден также для перекрестной разработки. На поведение скрипта `configure` влияют следующие переменные окружения:

- `CC`: команда запуска компилятора C;
- `CFLAGS`: дополнительные флаги компилятора C;
- `LDFLAGS`: дополнительные флаги компоновщика. Например, если библиотеки находятся в нестандартном каталоге `<lib dir>`, то его нужно добавить в список путей поиска библиотек, задав флаг `-L<lib dir>`;

- LIBS: список дополнительных библиотек, передаваемый компоновщику, например, -lm подключает библиотеку математических функций.
- CPPFLAGS: флаги препроцессора C/C++. Например, для поиска заголовков в нестандартном каталоге <include dir> нужно добавить флаг -I<include dir>;
- CPP: команда вызова препроцессора C.

Иногда достаточно задать только переменную CC:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc ./configure
```

Но бывает, что при этом печатается такое сообщение об ошибке:

```
[...]
checking whether we are cross compiling... configure: error: in '/home/ chris/MELP/build/
sqlite-autoconf-3081101':
configure: error: cannot run C compiled programs.
If you meant to cross compile, use '--host'.
See 'config.log' for more details
```

Причина в том, что configure часто пытается определить возможности набора инструментов, компилируя небольшую программу, запуская ее и анализируя результат. Но такой способ не работает, если программа была подвергнута кросс-компиляции. Тем не менее в сообщении об ошибке есть ключ к решению проблемы. Autotools выделяет три типа машин, потенциально участвующих в компиляции пакета.

- **Сборочная (build)**: это компьютер, на котором собирается пакет, – по умолчанию текущая машина.
- **Исходная (host)**: это компьютер, на котором будет выполняться собранная программа. Для платформенной компиляции не задается и по умолчанию совпадает со сборочной машиной. В случае кросс-компиляции это составное имя, определяемое используемым набором инструментов.
- **Целевая (target)**: это компьютер, для которого собранная программа будет генерировать код; его нужно задавать, например, при сборке кросс-компилятора.

Таким образом, для кросс-компиляции нужно только переопределить хост:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf
```

И последнее: по умолчанию пакет устанавливается в каталоги <sysroot>/usr/ local/*. Но обычно желательно установить его в <sysroot>/usr/*, чтобы файлы-заголовки и библиотеки оказались там, где их ожидают найти. Полная команда конфигурирования типичного пакета, собираемого с помощью Autotools, выглядит так:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

Пример: SQLite

Библиотека SQLite, реализующая простую реляционную базу данных, очень популярна во встраиваемых системах. Для начала скачаем SQLite:

```
$ wget http://www.sqlite.org/2015/sqlite-autoconf-3081101.tar.gz
$ tar xf sqlite-autoconf-3081101.tar.gz
$ cd sqlite-autoconf-3081101
```

Затем запустим скрипт configure:

```
$ CC=arm-cortex_a8-linux-gnueabihf-gcc \
./configure --host=arm-cortex_a8-linux-gnueabihf --prefix=/usr
```

Похоже, работает! Иначе на терминал и в файл config.log были бы выведены сообщения об ошибках. В результате работы было создано несколько make-файлов, поэтому можно приступать к сборке:

```
$ make
```

Наконец, устанавливаем собранный пакет в каталог набора инструментов, задав переменную make DESTDIR. Если этого не сделать, то make попытается установить пакет в каталог /usr исходного компьютера, а это не то, что нам нужно.

```
$ make DESTDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot) install
```

Последняя команда может не выполниться из-за отсутствия прав доступа. Набор инструментов crosstool-NG по умолчанию создает файлы, доступные только для чтения, поэтому полезно при его сборке устанавливать переменную CT_INSTALL_DIR_RO равной y. Другая типичная проблема связана с тем, что набор инструментов устанавливается в системный каталог, например /opt или /usr/local, и тогда для установки собранного пакета потребуются права пользователя root.

После успешной установки в набор инструментов будет добавлено несколько файлов:

- <sysroot>/usr/bin: sqlite3. Это командный интерфейс к SQLite, который можно установить и выполнить в целевой системе.
- <sysroot>/usr/lib: libsqlite3.so.0.8.6, libsqlite3.so.0, libsqlite3.so, libsqlite3.la, libsqlite3.a. Это разделяемые и статические библиотеки.
- <sysroot>/usr/lib/pkgconfig: sqlite3.pc. Это конфигурационный файл пакета, описанный в следующем разделе.
- <sysroot>/usr/lib/include: sqlite3.h, sqlite3ext.h. Это файлы-заголовки.
- <sysroot>/usr/share/man/man1: sqlite3.1. Это страница руководства.

Теперь можно компилировать программы, в которых используется sqlite3. Для этого нужно добавить флаг -lsqlite3 на этапе компоновки:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -lsqlite3 sqlite-test.c -o sqlite-test
```

Здесь sqlite-test.c – гипотетическая программа, вызывающая функции из библиотеки SQLite. Поскольку sqlite3 установлен в sysroot, компилятор легко найдет заголовок и библиотеки. Если бы пакет устанавливался в другой каталог, то нужно было бы добавить флаги -L<lib dir> и -I<include dir>.

Естественно, имеются также зависимости, необходимые во время выполнения; соответствующие файлы нужно будет установить в целевой каталог, как описано в главе 5 «Построение корневой файловой системы».

Конфигурирование пакета

Отслеживание зависимостей пакета – дело непростое. Утилита конфигурирования пакетов pkg-config (http://www.freedesktop.org/wiki/Software/pkg-config) поможет узнать, какие пакеты установлены и какие флаги компиляции нужны каждому пакету. Для этого она хранит базу данных Autotools-пакетов в каталоге [sysroot]/usr/lib/pkgconfig. Например, пакету SQLite3 соответствует файл sqlite3.pc, который содержит важную информацию, необходимую другим пакетам, использующим SQLite3:

```
$ cat $(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)\
/usr/lib/pkgconfig/sqlite3.pc
# Package Information for pkg-config
prefix=/usr
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include
Name: SQLite
Description: SQL database engine
Version: 3.8.11.1
Libs: -L${libdir} -lsqlite3
Libs.private: -ldl -lpthread
Cflags: -I${includedir}
```

Мы можем воспользоваться pkg-config, чтобы получить информацию в виде, пригодном для gcc. В случае библиотеки, например libsqlite3, нам нужно знать имя библиотеки (--libs) и дополнительные флаги компилятора С (--cflags):

```
$ pkg-config sqlite3 --libs --cflags
Package sqlite3 was not found in the pkg-config search path.
Perhaps you should add the directory containing `sqlite3.pc'
to the PKG_CONFIG_PATH environment variable
No package 'sqlite3' found
```

Вот те на! Ошибочка вышла... А все потому, что утилита искала в каталоге sysroot исходной системы, а пакет разработки для libsqlite3 установлен не туда. Мы должны указать на sysroot целевого набора инструментов, задав переменную оболочки PKG_CONFIG_LIBDIR:

```
$ PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)\
/usr/lib/pkgconfig pkg-config sqlite3 --libs -cflags
-lsqlite3
```

Теперь выводится -lsqlite3. В данном случае мы это знали заранее, но, вообще говоря, это не так, поэтому описанная техника представляет ценность. И наконец, команда компиляции:

```
$ PKG_CONFIG_LIBDIR=$(arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot)\
/usr/lib/pkgconfig arm-cortex_a8-linux-gnueabihf-gcc \
$(pkg-config sqlite3 --cflags -- libs) sqlite-test.c -o sqlite-
```

Проблемы кросс-компиляции

Пакет Sqlite3 устроен правильно и кросс-компилируется без ошибок, но не все пакеты ведут себя столь безупречно. Перечислим типичные проблемы.

- Нестандартные системы сборки. Например, в пакете zlib есть скрипт configure, но ведет он себя совсем не так, как одноименный скрипт из комплекта Autotools, описанный в предыдущем разделе.
- Скрипты configure, которые читают информацию от `pkg-config`, заголовки и прочие файлы прямо из исходной системы, не обращая внимания на флаг `host`.
- Скрипты, которые настойчиво пытаются выполнить кросс-компилированный код.

Каждый случай требует тщательного анализа ошибок и задания дополнительных параметров скрипта `configure`, сообщающих правильную информацию, либо внесения в код изменений, устраняющих саму возможность ошибки. Имейте в виду, что у пакета может быть много зависимостей, особенно это относится к программам с графическим интерфейсом, которым нужна библиотека GTK или Qt, и к программам обработки мультимедийных данных. Например, mplayer, популярная программа воспроизведения мультимедийных файлов, зависит от 100 с лишним библиотек. На то, чтобы все их собрать, уйдет несколько недель.

Поэтому я не рекомендую вручную заниматься кросс-компиляцией компонентов для целевой системы – разве что в безвыходной ситуации или когда собрать нужно совсем немного пакетов. Гораздо лучше воспользоваться каким-нибудь средством сборки, например Buildroot или Yocto Project, или вообще обойти проблему, подготовив систему сборки с точно такой же архитектурой, как у целевой системы. Теперь вы понимаете, почему дистрибутивы типа Debian всегда компилируются платформенным компилятором.

Резюме

Любой проект начинается с набора инструментов, от него зависит все остальное.

Большинство сред построения встраиваемых систем основано на наборе инструментов для перекрестной разработки, поскольку при этом имеется четкая граница между мощным компьютером, на котором собирается система, и компьютером, на котором она будет работать. Сам набор инструментов состоит из двоичных утилит GNU (пакет binutils), компилятора C (и, скорее всего, компилятора C++) из набора компиляторов GNU и одной из описанных выше библиотек C. Обычно на этом этапе собирается еще отладчик GNU, который будет описан в главе 12. Рекомендую также поглядывать на компилятор Clang, поскольку он будет активно развиваться в ближайшие несколько лет.

Можно начать, не имея ничего, кроме набора инструментов, быть может, собранного с помощью программы crosstool-NG или скачанного с сайта Linaro, и с его помощью откомпилировать все пакеты, которые должны присутствовать

в целевой системе, – понимая, что это потребует значительных усилий. А можно получить набор инструментов в составе дистрибутива, который включает также разнообразные пакеты. Дистрибутив может быть сгенерирован из исходного кода с помощью систем типа Buildroot или Yocto Project или получен в двоичном виде от сторонней компании, например коммерческой организации типа Mentor Graphics или проекта ПО с открытым исходным кодом типа Denx ELDK. Остерегайтесь наборов инструментов или дистрибутивов, предлагаемых бесплатно вместе с оборудованием: часто они плохо сконфигурированы и не сопровождаются. В общем, принимайте решение по ситуации, но, раз приняв, придерживайтесь его на протяжении всей работы над проектом.

Располагая набором инструментов, вы можете собрать другие компоненты встраиваемой Linux-системы. В следующей главе мы поговорим о начальном загрузчике, который пробуждает устройство к жизни и запускает процесс загрузки программного обеспечения.

Глава 3

Все о начальных загрузчиках

Начальный загрузчик – это второй элемент встраиваемой Linux-системы. Он запускает устройство и загружает ядро операционной системы. В этой главе я расскажу о роли начального загрузчика и, в частности, о том, как он передает управление ядру с помощью структуры данных, называемой деревом устройств, или линеаризованным деревом устройств (**flattened device tree** – **FDT**). Мы рассмотрим основы деревьев устройств, чтобы вы знали, как интерпретировать соединения, описанные в дереве, и как сопоставить дерево с реальным оборудованием.

Мы познакомимся с популярным начальным загрузчиком U-Boot с открытым исходным кодом и узнаем, как модифицировать его для поддержки нового устройства. Наконец, я расскажу о начальном загрузчике Barebox, который когда-то отпочковался от U-Boot, но, с моей точки зрения, имеет более элегантный дизайн.

Что делает начальный загрузчик?

Во встраиваемой Linux-системе начальный загрузчик решает две задачи: базовая инициализация и загрузка ядра. На самом деле первая задача подчинена второй в том смысле, что в работоспособное состояние нужно привести лишь те части системы, которые необходимы для загрузки ядра.

Когда выполняются первые строчки кода начального загрузчика – сразу после включения питания или сброса, – система еще почти ничего не может. Контроллер динамической памяти (DRAM) еще не инициализирован, так что оперативная память недоступна. То же относится ко всем остальным интерфейсам, поэтому недоступна и внешняя память, управляемая контроллерами NAND- или MMC-памяти, а также прочее оборудование. Обычно в самом начале работают только одно процессорное ядро и статическая память на кристалле очень небольшого объема. Поэтому загрузка системы состоит из нескольких этапов, на каждом из которых оживают все новые и новые ее части.

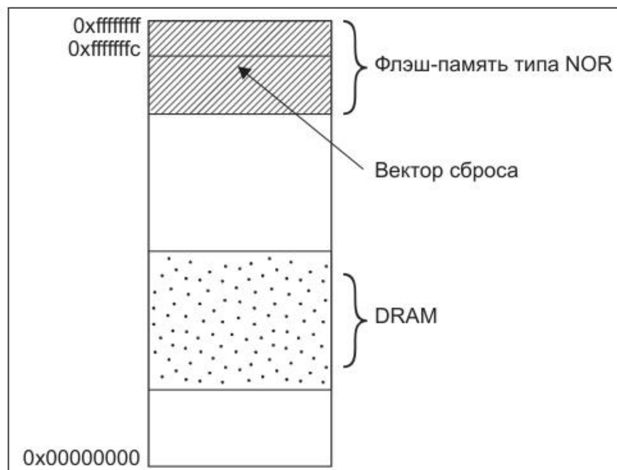
Первый этап загрузки завершается, когда приведены в рабочее состояние интерфейсы, необходимые для загрузки ядра. К ним относятся оперативная память

и периферийные устройства, используемые для доступа к ядру и другим образам, где бы они ни находились – в массовой памяти или в сети. Последнее действие начального загрузчика – загрузка ядра в память и создание окружения, в котором оно может выполняться. Детали интерфейса между начальным загрузчиком и ядром зависят от архитектуры, но в любом случае загрузчик должен передать указатель на информацию об известном ему оборудовании и командную ASCII-строку ядра, содержащую важную для Linux информацию. После того как началось выполнение ядра, надобность в начальном загрузчике отпадает, и занятую им память можно освободить.

У начального загрузчика есть еще и вспомогательная задача: организовать режим обслуживания для обновления конфигурации ядра, загрузки в память новых загрузочных образов и, быть может, прогона диагностических тестов. Обычно для этого используется простой интерфейс командной строки, часто через последовательный порт.

Последовательность начальной загрузки

Несколько лет назад, когда жизнь была простой и понятной, достаточно было поместить начальный загрузчик в область энергонезависимой памяти, расположенную по адресу вектора сброса процессора. Тогда доминировала флэш-память типа NOR и, поскольку ее можно было отобразить непосредственно на адресное пространство, такой метод хранения был идеален. На рисунке ниже показано, как выглядит такая конфигурация, где вектор сброса находится по адресу 0xfffffffc в самом конце адресного пространства, занятом флэш-памятью. По этому адресу находится команда перехода на начало кода начального загрузчика.



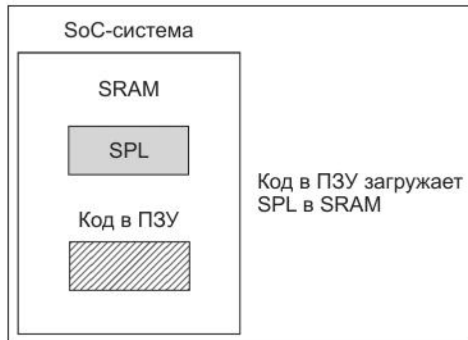
Начальная загрузка в старые добрые дни

Получив управление, начальный загрузчик может инициализировать контроллер памяти, сделав тем самым доступной оперативную DRAM-память, и скопировать себя в DRAM. Затем загрузчик может загрузить ядро из флэш-памяти в DRAM и передать ему управление.

Но с уходом от такого простого линейно адресуемого запоминающего устройства, как флэш-память типа NOR, последовательность начальной загрузки становится сложной многоступенчатой процедурой. Детали сильно зависят от конкретной SoC-системы, но, вообще говоря, последовательность состоит из перечисленных ниже этапов.

Этап 1: код в ПЗУ

В отсутствие надежной внешней памяти код, исполняемый сразу после сброса или включения питания, должен храниться на кристалле SoC-системы; он называется кодом в ПЗУ (ROM code). Этот код записывается на кристалл в процессе изготовления, поэтому является собственностью компании и не может быть заменен эквивалентным открытым кодом. Код в ПЗУ не может делать почти никаких предположений об оборудовании, внешнем по отношению к кристаллу, поскольку оно зависит от конструкции. Это относится даже к микросхемам DRAM, играющим роль оперативной памяти системы. Следовательно, коду в ПЗУ доступна только небольшая область статической памяти с произвольной выборкой (SRAM), присутствующая в большинстве SoC-систем. Размер области SRAM варьируется от 4 КиБ до нескольких сотен КиБ.



Этап 1 начального загрузчика

Код в ПЗУ может загрузить в SRAM небольшой фрагмент кода из нескольких предопределенных мест. Например, микросхемы TI OMAP и Sitara будут пытаться загрузить код из нескольких первых страниц флэш-памяти типа NAND, или из флэш-памяти, подключенной по интерфейсу **SPI (Serial Peripheral Interface** – последовательный периферийный интерфейс), или первых секторов MMC-устройства (это может быть микросхема eMMC или карта SD), или из файла с именем `MLO` в первом разделе MMC-устройства. Если загрузка из всех указанных

мест завершается неудачно, то код в ПЗУ пытается читать поток байтов из сети Ethernet, из порта USB или UART; последний служит в основном для загрузки кода во флэш-память на стадии производства, а не для использования в обычном режиме. В большинстве SoC-систем имеется код в ПЗУ, работающий примерно так, как описано выше. Для тех SoC-систем, где объем SRAM-памяти недостаточен для размещения всего начального загрузчика, например U-Boot, приходится создавать промежуточный загрузчик, который называется вторичным загрузчиком программ, или SPL (secondary program loader).

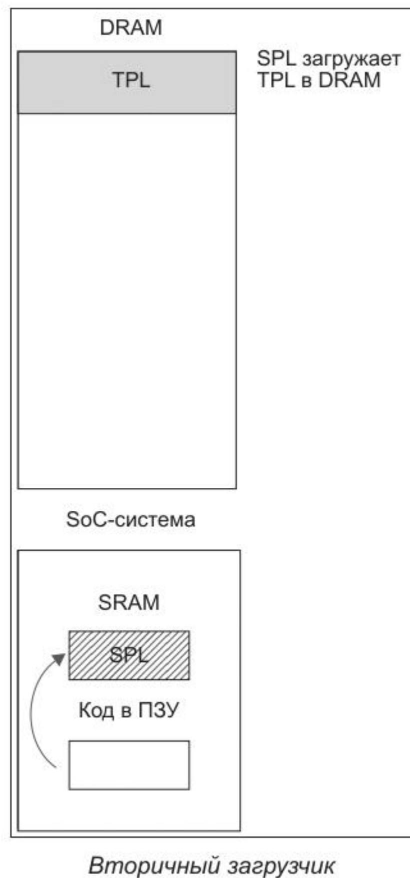
В конце этого этапа в памяти на кристалле находится вторичный начальный загрузчик, и код в ПЗУ переходит на его начало.

Этап 2: SPL

SPL должен настроить контроллер памяти и другие необходимые части системы в преддверии загрузки третичного загрузчика (**third stage program loader – TPL**) в динамическую оперативную память. Функциональность SPL ограничена его размером. Он может прочитать программу с одного из нескольких запоминающих устройств, как и код в ПЗУ, причем код может находиться, начиная с предопределенного смещения от начала флэш-памяти, или в известном файле, например `u-boot.bin`. Обычно SPL не взаимодействует с пользователем, но может выводить на консоль номер версии и сообщения о ходе работы. На рисунке показана архитектура второго этапа.

Код SPL может быть открыт, как обстоит дело в случае загрузчиков TI x-loader и Atmel AT91Bootstrap, но гораздо чаще это закрытый код, поставляемый производителем в виде двоичного файла.

В конце второго этапа в DRAM-памяти находится третичный загрузчик, и SPL может перейти на его начало.

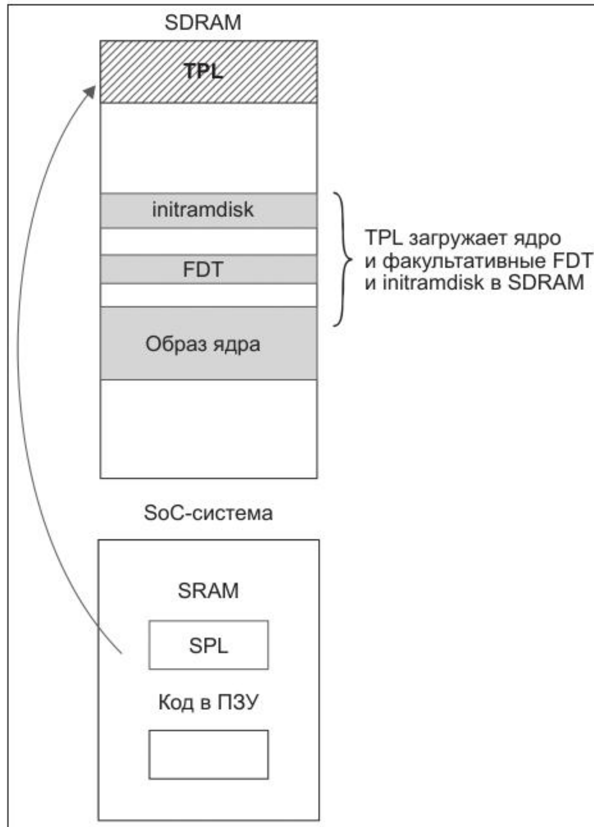


Вторичный загрузчик

Этап 3: TPL

И наконец-то начинает работать полный начальный загрузчик, например U-Boot или Barebox. Обычно для него существует простой интерфейс командной строки, который позволяет выполнять задачи обслуживания: загрузку нового загрузочного образа и образа ядра во флэш-память, считывание и загрузку ядра. Имеется также

способ загрузить ядро автоматически без вмешательства пользователя. На рисунке ниже показана архитектура третьего этапа.



Третичный загрузчик

В конце третьего этапа ядро находится в памяти, готовое к работе. Обычно загрузчики встраиваемых систем стираются из памяти после начала исполнения ядра и не принимают никакого участия в дальнейшей работе системы.

Загрузка из UEFI-прошивки

Большинство настольных ПК и некоторые системы на основе процессоров ARM имеют прошивки на основе стандарта **Universal Extensible Firmware Interface (UEFI)**, дополнительные сведения о котором можно найти на официальном сайте по адресу http://www.uefi.org. Принципиальная последовательность загрузки такая же, как описано в предыдущем разделе.

Этап 1. Процессор загружает диспетчер загрузки UEFI из флэш-памяти. Иногда он загружается непосредственно из флэш-памяти типа NOR, а иногда имеется код в ПЗУ на кристалле, который загружает диспетчер загрузки из флэш-памяти, подключенной по интерфейсу SPI. Функционально диспетчер загрузки примерно эквивалентен SPL, но может допускать взаимодействие с пользователем с помощью текстового или графического интерфейса.

Этап 2. Диспетчер загрузки загружает прошивку из системного раздела **EFI (EFI System Partition – ESP)**, или с твердого диска, или с SSD-диска, или с сетевого PXE-сервера. Если загрузка производится с локального диска, то ESP идентифицируется предопределенным GUID-ом C12A7328-F81F-11D2-BA4B-00A0C93EC93B. Раздел должен иметь формат FAT32. Третичный загрузчик должен находиться в файле с именем `<efi_system_partition>/boot/boot<machine_type_short_name>.efi`.

Например, в системе с архитектурой x86_64 путь к загрузчику выглядит так: `/efi/boot/bootx64.efi`

Этап 3. В этом случае TPL должен уметь загружать ядро Linux и факультативный RAM-диск в память. Часто встречаются следующие варианты:

- **GRUB 2.** Это загрузчик GNU Grand Unified Bootloader версии 2, который чаще всего используется для загрузки Linux на ПК. Однако он распространяется по лицензии GPL v3, из-за чего может оказаться несовместим с безопасной начальной загрузкой, так как лицензия требует, чтобы загрузочные ключи поставлялись вместе с кодом. Официальный сайт этого загрузчика – https://www.gnu.org/ software/grub/.
- **gummiboot.** Это простой совместимый с UEFI начальный загрузчик, интегрированный с systemd и распространяемый по лицензии LGPL v2.1 Официальный сайт – https://wiki.archlinux.org/index.php/Systemd-boot.

Переход от начального загрузчика к ядру

Передавая управление ядру, начальный загрузчик должен сообщить кое-какую информацию, например:

- для архитектур PowerPC и ARM: число, уникальное для данного типа SoC-систем;
- основные сведения об обнаруженном к этому моменту оборудовании, по крайней мере размер и адрес начала физической оперативной памяти и тактовую частоту процессора;
- командную строку ядра;
- факультативно размер и местоположение двоичного дерева устройств;
- факультативно размер и местоположение начального RAM-диска.

Командная строка ядра – это строка ASCII-символов, определяющая поведение Linux, например какое устройство содержит корневую файловую систему. Подробнее об этом я расскажу в следующей главе. Часто корневая файловая система

предоставляется в виде RAM-диска, и тогда начальный загрузчик должен загрузить этот диск в память. Как создать начальный RAM-диск, описано в главе 5.

Способ передачи этой информации зависит от архитектуры и в последние годы изменился. Например, раньше в случае PowerPC начальный загрузчик просто передавал указатель на структуру, содержащую информацию о плате, а в случае ARM – указатель на список «А-тегов». Хорошее описание формата исходного кода ядра имеется в разделе документации `Documentation/arm/Booting`.

В обоих случаях передавалось очень немного информации, а основную работу приходилось производить на этапе выполнения ядра или зашивать в ядро в форме «платформенных данных». Широкое использование платформенных данных означало, что для каждого устройства необходимо было специально конфигурировать и модифицировать ядро. Нужен был способ лучше, и таковым стало дерево устройств. В мире ARM расставание с А-тегами началось в феврале 2013 года, когда вышла версия Linux 3.8, но все еще эксплуатируется и даже разрабатывается много устройств, в которых используются А-теги.

Введение в деревья устройств

Почти наверняка вы рано или поздно столкнетесь с деревьями устройств. В этом разделе я кратко расскажу, что это такое и как они работают, опуская многие детали.

Дерево устройств – это гибкий способ описать аппаратные компоненты вычислительной системы. Обычно дерево устройств загружается начальным загрузчиком и передается ядру, хотя можно также поставлять дерево устройств вместе с самим образом ядра, компенсируя недостатки загрузчиков, которые не умеют работать с ним отдельно.

Формат заимствован из начального загрузчика Sun Microsystems OpenBoot и впоследствии был формализован в виде спецификации Open Firmware, которой соответствует стандарт IEEE 1275-1994. Этот формат использовался в компьютерах Macintosh на платформе PowerPC, поэтому было логично сохранить его при портировании Linux на PowerPC. С тех пор спецификация была поддержана во многих реализациях Linux на платформе ARM и в меньшей степени на платформах MIPS, MicroBlaze, ARC и др.

Дополнительные сведения можно найти на сайте http://devicetree.org.

Основные сведения о деревьях устройств

Ядро Linux содержит немало исходных файлов с деревьями устройств в каталоге `arch/$ARCH/boot/dts`, и это отличная отправная точка для их изучения. Такие файлы, правда, в меньшем количестве, есть и в исходном коде загрузчика U-boot в каталоге `arch/$ARCH/dts`. Если вы приобретали оборудование у сторонней компании, то dts-файл входит в пакет, поставляемый вместе с платой, наряду с другими исходными файлами.

Дерево устройств представляет вычислительную систему в виде набора компонентов, организованных в форме древовидной иерархии. Дерево начинается с корня, представленного символом косой черты /, и расположенных под ним узлов, описывающих имеющееся в системе оборудование. У каждого узла есть имя и ряд свойств вида `имя = "значение"`, например:

```
/dts-v1/;
/{
  model = "TI AM335x BeagleBone";
  compatible = "ti,am33xx";
  #address-cells = <1>;
  #size-cells = <1>;
  cpus {
    #address-cells = <1>;
    #size-cells = <0>;
    cpu@0 {
      compatible = "arm,cortex-a8";
      device_type = "cpu";
      reg = <0>;
    };
  };
  memory@0x80000000 {
    device_type = "memory";
    reg = <0x80000000 0x20000000>; /* 512 MB */
  };
};
```

Здесь корневой узел содержит узлы `cpus` и `memory`. Узел `cpus` содержит один узел процессора, называющийся `cpu@0`. Принято присваивать узлам имена, содержащие символ `@`, за которым следует адрес, отличающий этот узел от других.

У корневого узла и узлов с описанием процессоров имеется свойство `compatible`. Ядро Linux сравнивает указанное в нем имя со строками, которые драйверы экспортируют в структуре `struct of_device_id` (подробнее о ней будет рассказано в главе 8). По соглашению значение состоит из названия производителя и названия компонента, чтобы не путать одноименные устройства разных производителей, например: `ti,am33xx` и `arm,cortex-a8`. Кроме того, в свойстве `compatible` часто можно встретить несколько значений, если для устройства существует несколько драйверов. Предпочтительное значение указывается первым.

В узлах, относящихся к процессору и памяти, имеется свойство `device_type`, описывающее класс устройства. Имя узла часто производится из значения `device_type`.

Свойство reg

В узлах памяти и процессора имеется свойство `reg`, относящееся к диапазону в пространстве регистров устройств. Оно содержит два значения, представляющих начальный адрес и размер (длину) диапазона. Оба записываются в виде нуля или

более 32-разрядных целых чисел, которые называются ячейками (cell). Так, узел memory ссылается на один банк памяти с начальным адресом 0x80000000 и длиной 0x20000000 байтов.

Структура свойства reg усложняется, если адрес или длину нельзя представить 32 битами. Например, для устройства с 64-разрядной адресацией для каждого значения понадобятся две ячейки:

```
/ {
  #address-cells = <2>;
  #size-cells = <2>;
  memory@80000000 {
    device_type = "memory";
    reg = <0x00000000 0x80000000 0 0x80000000>;
  };
}
```

Информация о количестве ячеек хранится в объявлениях #address-cells и #size_cells в каком-то узле-предке. Иными словами, для интерпретации свойства reg нужно подняться вверх по иерархии, пока не встретятся объявления #address-cells и #size_cells. Если таковых не обнаружится, то принимается значение по умолчанию 1, но это дурной стиль – авторы деревьев устройств не должны полагаться на умолчания.

Но вернемся к узлам cpu и cpus. У процессоров тоже есть адреса: ядра четырехъядерного процессора можно адресовать как 0, 1, 2, 3. Можно считать, что это одномерный массив без «глубины», так что размер равен 0. Поэтому мы видим, что в узле cpus #address-cells = <1>, а #size-cells = <0>, а в его дочернем узле cpu@0 свойству reg присвоено всего одно значение: reg = <0>.

Указатели на описатели и прерывания

До сих пор мы предполагали, что иерархия компонентов всего одна, но на самом деле их несколько. Помимо очевидной связи между компонентом и другими частями системы, могут существовать связи с контроллером прерываний, с генератором тактовых импульсов и с регулятором напряжения. Для выражения таких связей применяются указатели на описатели (phandle).

Рассмотрим пример системы, содержащей последовательный порт, который может генерировать прерывания, и контроллер прерываний.

```
/dts-v1/;
{
  intc: interrupt-controller@48200000 {
    compatible = "ti,am33xx-intc";
    interrupt-controller;
    #interrupt-cells = <1>;
    reg = <0x48200000 0x1000>;
  };
  serial@44e09000 {
```

```
    compatible = "ti,omap3-uart";
    ti,hwmods = "uart1";
    clock-frequency = <48000000>;
    reg = <0x44e09000 0x2000>;
    interrupt-parent = <&intc>;
    interrupts = <72>;
  };
};
```

Мы видим узел `interrupt-controller`, обладающий специальным свойством `#interrupt-cells`, которое говорит, сколько 4-байтовых значений необходимо для представления линии прерывания. В данном случае значение равно 1, т. е. задается только номер прерывания, но часто можно встретить и дополнительные характеристики прерывания, например: `1 = low-to-high edge triggered` (с возбуждением по переходу от низкого к высокому уровню сигнала), `2 = high-to-low edge triggered` (с возбуждением по переходу от высокого к низкому уровню сигнала) и т. д.

В узле `serial` мы видим свойство `interrupt-parent`, которое ссылается на подключенный к последовательному порту контроллер прерываний по его метке. Это и есть указатель на описатель. Сама же линия прерывания задается свойством `interrupts`, которое в данном случае равно `72`.

В узле `serial` есть и другие свойства, которые нам еще не встречались: `clock-frequency` и `ti,hwmods`. Это часть привязок для данного типа устройств, т. е. драйвер будет читать эти свойства для управления устройством. Описание привязок можно найти в исходном коде ядра в каталоге `Documentation/devicetree/bindings/`.

Включаемые файлы деревьев устройств

В SoC-системах одного семейства и в платах, где используется одна и та же SoC-система, много одинакового оборудования. Чтобы представить эту ситуацию, в дереве устройств выделяются секции, описываемые включаемыми файлами, обычно с расширением `.dtsi`. В стандарте Open Firmware определен механизм `/include/`, демонстрируемый на примере, взятом из файла `vexpress-v2p-ca9.dts`:

```
/include/ "vexpress-v2m.dtsi"
```

Но, заглянув в `dts`-файла, в ядре, вы увидите другой синтаксис директивы `include`, заимствованный из языка C. Например, в файле `am335x-boneblack.dts` есть такие строки:

```
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
```

Вот еще пример из файла `am33xx.dtsi`:

```
#include <dt-bindings/gpio/gpio.h>
#include <dt-bindings/pinctrl/am33xx.h>
```

И наконец, файл `include/dt-bindings/pinctrl/am33xx.h` содержит обычные макросы C:

```
#define PULL_DISABLE (1 << 3)
#define INPUT_EN (1 << 5)
#define SLEWCTRL_SLOW (1 << 6)
#define SLEWCTRL_FAST 0
```

Все эти конструкции разрешаются, когда исходные файлы деревьев устройств обрабатываются системой kbuild, которая сначала подает их на вход препроцессору C (`cpp`), который преобразует директивы `#include` и `#define` в обычный текст, понятный компилятору деревьев устройств. Почему так сделано, ясно из показанного выше примера: чтобы в исходных файлах деревьев устройств можно было использовать те же определения констант, что и в коде ядра.

Когда файлы включаются таким образом, узлы накладываются один на другой, образуя составное дерево, в котором внешние уровни расширяют или модифицируют внутренние. Например, в файле `am33xx.dtsi`, общем для всех SoC-систем семейства am33xx, интерфейс с первым контроллером MMC определен так:

```
mmc1: mmc@48060000 {
  compatible = "ti,omap4-hsmmc";
  ti,hwmods = "mmc1";
  ti,dual-volt;
  ti,needs-special-reset;
  ti,needs-special-hs-handling;
  dmas = <&edma 24 &edma 25>;
  dma-names = "tx", "rx";
  interrupts = <64>;
  interrupt-parent = <&intc>;
  reg = <0x48060000 0x1000>;
  status = "disabled";
};
```

Обратите внимание, что свойство `status` равно `disabled`, т. е. к нему не следует привязывать никакой драйвер, а также что метка равна `mmc1`.

В файле `am335x-bone-common.dtsi`, который включается в деревья устройств BeagleBone и BeagleBone Black, этот узел модифицируется с помощью указателя на его описатель:

```
&mmc1 {
  status = "okay";
  bus-width = <0x4>;
  pinctrl-names = "default";
  pinctrl-0 = <&mmc1_pins>;
  cd-gpios = <&gpio0 6 GPIO_ACTIVE_HIGH>;
  cd-inverted;
};
```

Здесь `mmc1` включено (`status="okay"`), поскольку в обеих платах имеется физическое устройство MMC1, и конфигурируется `pinctrl`. Затем в файле `am335x-bone-`

black.dts мы встречаем еще одну ссылку на mmc1, в которой с устройством ассоциируется регулятор напряжения:

```
&mmc1 {
  vmmc-supply = <&vmmcsd_fixed>;
};
```

Такое расслоение исходных файлов обеспечивает необходимую гибкость и уменьшает дублирование кода.

Компиляция дерева устройств

Начальному загрузчику и ядру необходимо двоичное представление дерева устройств, поэтому исходный файл необходимо обработать компилятором dtc. В результате получается файл с расширением .dtb, который называется двоичным деревом устройством (иногда можно встретить выражение «device tree blob»).

Компилятор dtc находится в каталоге scripts/dtc/dtc исходного кода Linux, а также включен в виде пакета во многие дистрибутивы Linux. Компиляция простого дерева устройств (без #include) производится так:

```
$ dtc simpledts-1.dts -o simpledts-1.dtb
DTC: dts->dts on file "simpledts-1.dts"
```

Имейте в виду, что dtc не выдает содержательных сообщений об ошибках и не производит никаких проверок, кроме соблюдения базового синтаксиса языка, поэтому отладка опечатки в исходном файле может затянуться надолго.

В более сложных случаях нужно использовать систему kbuild для сборки ядра. Как это делается, я объясню в следующей главе.

Выбор начального загрузчика

Есть много разных начальных загрузчиков. Нас интересуют следующие характеристики: простота, возможность настройки и наличие примеров конфигураций для типичных макетных плат и устройств. В следующей таблице перечислено несколько общеупотребительных загрузчиков.

Название	Архитектуры
Das U-Boot	ARM, Blackfin, MIPS, PowerPC, SH
Barebox	ARM, Blackfin, MIPS, PowerPC
GRUB 2	X86, X86_64
RedBoot	ARM, MIPS, PowerPC, SH
CFE	Broadcom MIPS
YAMON	MIPS

Мы сосредоточимся на U-Boot, потому что он поддерживает разные процессорные архитектуры и много плат и устройств. Он существует уже довольно давно и располагает хорошим сообществом, способным оказать поддержку.

Возможно, начальный загрузчик входит в комплект поставки SoC-системы или платы. Как всегда, разберитесь, что вы получили, и спросите, где можно достать исходный код, какова политика обновления, как осуществляется техническая поддержка в случае, если вам понадобится внести изменения, и т. д. Быть может, вы решите отказаться от загрузчика, поставляемого изготовителем, и воспользоваться текущей версией открытого загрузчика.

U-Boot

U-Boot, или полностью Das U-Boot, поначалу представлял собой начальный загрузчик с открытым исходным кодом для встраиваемых плат на основе PowerPC. Затем он был портирован на платы на основе ARM, а потом и на другие архитектуры, в том числе MIPS, SH и x86. Сопровождение осуществляет компания Denx Software Engineering. В сети можно найти подробную информацию о программе, и начать стоит со страницы www.denx.de/wiki/U-Boot. Имеется также список рассылки u-boot@lists.denx.de.

Сборка U-Boot

Для начала нужно получить исходный код. Как и в большинстве проектов, рекомендуется клонировать архив git и выгрузить версию с интересующей вас меткой. В данном случае это версия, которая была текущей на момент написания книги:

```
$ git clone git://git.denx.de/u-boot.git
$ cd u-boot
$ git checkout v2015.07
```

Можно поступить и по-другому: скачать архивный файл по адресу ftp://ftp.denx.de/pub/u-boot/.

В каталоге `configs/` хранится свыше 1000 конфигурационных файлов для распространенных макетных плат и устройств. В большинстве случаев о том, какой использовать, можно догадаться по имени файла, а более подробные сведения имеются в файлах `README` для конкретных плат, которые находятся в каталоге `board/`. Можно также поискать информацию в подходящем онлайновом руководстве или на форуме. Однако имейте в виду, что начиная с версии 2014.10 порядок конфигурирования U-Boot существенно изменился. Убедитесь, что вы следуете актуальным инструкциям.

Взяв в качестве примера плату BeagleBone Black, мы найдем в `configs/` предполагаемый конфигурационный файл `am335x_boneblack_defconfig`, а в файле `README` для микросхемы am335x (`board/ti/am335x/README`) обнаружим фразу «**The binary produced by this board supports ... Beaglebone Black**» (Двоичный файл, порождаемый для этой платы, поддерживает ... Beaglebone Black). Это главное, а уж собрать U-Boot для BeagleBone Black легко. Нужно сообщить U-Boot префикс вашего кросс-компилятора, задав переменную `make CROSS_COMPILE`, а затем выбрать конфигурационный файл в команде в виде `make [board]_defconfig`:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-am335x_boneblack
_defconfig
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

В результате компиляции образуются следующие файлы:

- u-boot: это сам загрузчик U-Boot в объектном формате ELF, пригодном для работы с отладчиком;
- u-boot.map: таблица символов;
- u-boot.bin: загрузчик U-Boot в простом двоичном формате, пригодном для исполнения на устройстве;
- u-boot.img: это u-boot.bin с добавленным заголовком U-Boot, пригодный для загрузки в работающий экземпляр U-Boot;
- u-boot.srec: U-Boot в формате Motorola srec, пригодном для передачи через последовательный порт.

Для платы BeagleBone Black нужен также вторичный загрузчик программ (SPL), описанный выше. Он собирается вместе с U-Boot и называется MLO:

```
$ ls -l MLO u-boot*
-rw-rw-r-- 1 chris chris 76100 Dec 20 11:22 MLO
-rwxrwxr-x 1 chris chris 2548778 Dec 20 11:22 u-boot
-rw-rw-r-- 1 chris chris 449104 Dec 20 11:22 u-boot.bin
-rw-rw-r-- 1 chris chris 449168 Dec 20 11:22 u-boot.img
-rw-rw-r-- 1 chris chris 434276 Dec 20 11:22 u-boot.map
-rw-rw-r-- 1 chris chris 1347442 Dec 20 11:22 u-boot.srec
```

Для других целевых устройств процедура аналогична.

Установка U-Boot

Для первой установки начального загрузчика на плату требуется помощь извне. Если плата оснащена интерфейсом для отладки оборудования, например JTAG, то обычно можно загрузить U-Boot прямо в ОЗУ и запустить его. После этого можно воспользоваться командами U-Boot для копирования его во флэш-память. Детали сильно зависят от конкретной платы и выходят за рамки этой книги.

В некоторых SoC-системах имеется встроенное ПЗУ начальной загрузки, которое можно использовать для считывания кода начальной загрузки из различных внешних источников, включая SD-карту, последовательный интерфейс и USB-порт. Примером может служить микросхема AM335x на плате BeagleBone Black. Опишем, как загрузить U-Boot с карты micro-SD.

Прежде всего нужно отформатировать microSD, так чтобы первый раздел имел формат FAT16, и сделать этот раздел загрузочным. Если имеется разъем для карт SD, то карта будет видна как /dev/mmcblk0. Если же разъема нет, но есть устройства чтения карт SD, то они будут видны как /dev/sdb, /dev/sdc и т. д. В предположении, что карта видна как /dev/mmcblk0, введите следующую команду для разбиения карты microSD на разделы:

```
$ sudo sfdisk -D -H 255 -S 63 /dev/mmcblk0 << EOF
,9,0x0C,*
```

```
,,,-
EOF
```

Отформатируйте первый раздел под FAT16:

```
$ sudo mkfs.vfat -F 16 -n boot /dev/mmcblk0p1
```

Теперь смонтируйте только что отформатированный раздел. В некоторых системах достаточно просто вытащить карту microSD и снова вставить ее, в других нужно будет щелкнуть по значку на рабочем столе. В современных версиях Ubuntu карта монтируется на каталог /media/[user]/boot, поэтому для копирования UBoot и SPL я использовал такую команду:

```
cp MLO u-boot.img /media/chris/boot
```

И наконец, размонтируйте карту.

Не подавая питания на плату BeagleBone, вставьте карту microSD.

Подключите последовательный кабель. Последовательный порт должен быть виден на ПК как /dev/ttyUSB0 или что-то в этом роде.

Запустите какую-нибудь программу терминала, например gtkterm, minicom или picocom, и присоедините ее к порту на скорости 115 200 бит/с без управления потоком данных:

```
$ gtkterm -p /dev/ttyUSB0 -s 115200
```

Нажмите и не отпускайте кнопку **Boot Switch** на плате BeagleBone, запитайте плату от внешнего источника напряжением 5 В и отпустите кнопку примерно через 5 секунд. На последовательной консоли должно появиться приглашение U-Boot:

```
U-Boot#
```

Работа с U-Boot

В этом разделе я опишу некоторые типичные задачи, для которых можно использовать U-Boot.

Обычно U-Boot предлагает интерфейс командной строки для работы через последовательный порт. Приглашение можно настроить для конкретной платы. В примерах показано приглашение U-Boot#. Если ввести слово help, то будет выведен полный список команд, сконфигурированных для данной версии U-Boot; help <command> выводит дополнительную информацию о конкретной команде.

По умолчанию интерпретатор команд чрезвычайно прост. Нет никаких средств редактирования команды с помощью клавиш со стрелками вправо или влево; нет механизма автоматического завершения команды по нажатии клавиши **Tab**; нет истории команд по клавише со стрелкой вверх. Нажатие любой из упомянутых клавиш портит вводимую команду, поэтому нужно будет нажать **Ctrl**+**C** и начать набирать ее заново. Единственная доступная клавиша редактирования – забой. Но можно сконфигурировать альтернативный командный интерпретатор Hush, обладающий более развитыми средствами интерактивной работы.

По умолчанию предполагается шестнадцатеричный формат чисел. Например, команда

```
nand read 82000000 400000 200000
```

читает 0x200000 байтов, начиная со смещения 0x400000 от начала флэш-памяти типа NAND, в ОЗУ, начиная с адреса 0x82000000.

Переменные окружения

В U-Boot активно используются переменные окружения для хранения и передачи информации между функциями и даже для создания скриптов. Переменные окружения – это просто пары вида `имя=значение`, которые хранятся в некоторой области памяти. Начальный набор переменных можно записать в конфигурационный заголовочный файл платы, например:

```
#define CONFIG_EXTRA_ENV_SETTINGS \
"myvar1=value1\0" \
"myvar2=value2\0"
```

Для создания и модификации переменных из командной строки U-Boot служит команда `setenv`. Так, команда `setenv foo bar` создает переменную `foo` со значением `bar`. Отметим, что между именем и значением переменной не должно быть знака `=`. Чтобы удалить переменную, нужно присвоить ей пустую строку: `setenv foo`. Команда `printenv` выводит на консоль все переменные, а команда `printenv foo` – только одну переменную `foo`.

Обычно доступна команда `saveenv`, которая сохраняет все окружение в какой-то постоянной памяти. Если имеется неуправляемая флэш-память типа NAND или NOR, то для этой цели зарезервирован стираемый блок, а часто даже два – чтобы иметь резервную копию на случай повреждения памяти. При наличии карты памяти eMMC или SD окружение может быть сохранено в каком-то файле на диске. Возможно также сохранение в последовательном электрически стираемом ПЗУ (EEPROM), подключенном через интерфейс I2C или SPI, или в энергонезависимом ОЗУ.

Формат загрузочного образа

В U-Boot нет файловой системы. Вместо этого он помечает блоки информации 64-байтовыми заголовками, позволяющими идентифицировать содержимое. Для подготовки файлов для U-Boot служит команда `mkimage`. Ниже приведена краткая справка по работе с ней.

```
$ mkimage
Usage: mkimage -l image
-l ==> list image header information
mkimage [-x] -A arch -O os -T type -C comp -a addr -e ep -n name -d data_file[:data_
file...] image
-A ==> set architecture to 'arch'
-O ==> set operating system to 'os'
```

```
-T ==> set image type to 'type'
-C ==> set compression type 'comp'
-a ==> set load address to 'addr' (hex)
-e ==> set entry point to 'ep' (hex)
-n ==> set image name to 'name'
-d ==> use image data from 'datafile'
-x ==> set XIP (execute in place)
mkimage [-D dtc_options] -f fit-image.its fit-image
mkimage -V ==> print version information and exit
```

Например, чтобы подготовить образ ядра для процессора ARM, нужно выполнить такую команду:

```
$ mkimage -A arm -O linux -T kernel -C gzip -a 0x80008000 \
-e 0x80008000 -n 'Linux' -d zImage uImage
```

Загрузка образа

Обычно образы загружаются со съемного запоминающего устройства типа карты SD, или из сети. Для работы с картами SD в U-Boot применяется драйвер `mmc`. Типичная последовательность загрузки образа в память такова:

```
U-Boot# mmc rescan
U-Boot# fatload mmc 0:1 82000000 uimage
reading uimage
4605000 bytes read in 254 ms (17.3 MiB/s)
U-Boot# iminfo 82000000

## Checking Image at 82000000 ...
Legacy image found
Image Name: Linux-3.18.0
Created: 2014-12-23 21:08:07 UTC
Image Type: ARM Linux Kernel Image (uncompressed)
Data Size: 4604936 Bytes = 4.4 MiB
Load Address: 80008000
Entry Point: 80008000
Verifying Checksum ... OK
```

Команда `mmc rescan` повторно инициализирует драйвер `mmc`, например для того, чтобы найти недавно вставленную карту SD. Затем по команде `fatload` читается файл из раздела в формате FAT на карте SD. Формат команды такой:

```
fatload <interface> [<dev[:part]> [<addr> [<filename> [bytes [pos]]]]]
```

Если `<interface>` равно `mmc`, как в нашем случае, то `<dev:part>` – это номер устройства `mmc`, начинающийся с 0, и номер раздела, начинающийся с 1. Следовательно, `<0:1>` обозначает первый раздел на первом устройстве. Предполагается, что область ОЗУ, начинающаяся с адреса `0x82000000`, еще не используется. Если мы собираемся загружать ядро, то должны быть уверены, что эта область не затрется, когда ядро будет распаковано и перемещено в область выполнения с начальным адресом `0x80008000`.

Для загрузки образов по сети используется протокол **Trivial File Transfer Protocol (TFTP)**. Для его работы в системе разработки должен быть поднят демон TFTP, tftpd. Необходимо также настроить брандмауэры на пути между ПК и макетной платой, чтобы они пропускали трафик протокола TFTP в порт UDP 69. По умолчанию tftpd разрешает доступ только к каталогу `/var/lib/tftpboot`. Следующий шаг – скопировать в этот каталог файлы, подлежащие передаче в целевую систему. Затем в предположении, что используются два статических IP-адреса, так что дополнительные действия по администрированию сети не нужны, нужно выполнить следующую последовательность команд, загружающих файлы образа ядра:

```
U-Boot# setenv ipaddr 192.168.159.42
U-Boot# setenv serverip 192.168.159.99
U-Boot# tftp 82000000 uImage
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.159.99; our IP address is 192.168.159.42
Filename 'uImage'.
Load address: 0x82000000
Loading:
#################################################################
#################################################################
#################################################################
######################################################
3 MiB/s
done
Bytes transferred = 4605000 (464448 hex)
```

Наконец, посмотрим, как записать образ программы во флэш-память типа NAND и прочитать его обратно. Для этого служит команда `nand`. В примере ниже образ ядра читается по протоколу TFTP и записывается во флэш-память:

```
U-Boot# fatload mmc 0:1 82000000 uimage
reading uimage
4605000 bytes read in 254 ms (17.3 MiB/s)

U-Boot# nandecc hw
U-Boot# nand erase 280000 400000

NAND erase: device 0 offset 0x280000, size 0x400000
Erasing at 0x660000 -- 100% complete.
OK
U-Boot# nand write 82000000 280000 400000

NAND write: device 0 offset 0x280000, size 0x400000
4194304 bytes written: OK
```

Теперь можно загрузить ядро из флэш-памяти:

```
U-Boot# nand read 82000000 280000 400000
```

Загрузка Linux

Команда bootm начинает исполнение образа ядра. Вот ее синтаксис:

```
bootm [адрес ядра] [адрес ram-диска] [адрес dtb].
```

Адрес образа ядра обязателен, а адреса ram-диска и двоичного дерева устройств (dtb) можно опустить, если ядро сконфигурировано так, что они не используются. Если dtb есть, а ram-диска нет, то вместо второго адреса следует задать дефис (), например:

```
U-Boot# bootm 82000000 - 83000000
```

Автоматизация начальной загрузки с помощью скриптов U-Boot

Понятно, что вводить длинную последовательность команд для загрузки платы всякий раз, как она включается, никто не станет. Для автоматизации процесса U-Boot сохраняет последовательность команд в переменных окружения. Если специальная переменная bootcmd содержит скрипт, то он начинает выполняться спустя bootdelay секунд после подачи питания. Если вы наблюдаете за процессом загрузки на последовательной консоли, то увидите, как ведется обратный отсчет до нуля. Нажатие любой клавиши останавливает обратный отсчет и приводит к переходу в режим интерактивной работы с U-Boot.

Создать скрипт просто, хотя читать его нелегко. Нужно лишь записать команды подряд через точку с запятой, которой должен предшествовать знак обратной косой черты. Например, команда считывания образа ядра с заданного смещения от начала флэш-памяти и его последующей загрузки выглядит так:

```
setenv bootcmd nand read 82000000 400000 200000\;bootm 82000000
```

Портирование U-Boot на новую плату

Предположим, что отдел разработок создал новую плату под названием «Nova», основанную на BeagleBone Black, а вам надо портировать на нее U-Boot. Для этого нужно понимать структуру кода U-Boot и работу механизма конфигурирования платы. В версии U-Boot 2014.10 был принят такой же механизм конфигурирования, как в ядре Linux, – Kconfig. По мере выхода следующих версий существующие конфигурационные параметры будут перемещены из файлов-заголовков в каталоге include/configs в файлы Kconfig. В версии 2014.10 для каждой платы имелся файл Kconfig, содержащий минимальную информацию из старого файла boards.cfg.

Перечислим основные интересующие нас каталоги.

- arch. Содержит архитектурно-зависимый код для каждой из поддерживаемых архитектур в каталогах arm, mips, powerpc и т. д. Внутри каждого архитектурного каталога имеются подкаталоги для семейств, например в каталоге arch/arm/cpu есть подкаталоги для вариантов процессора amt926ejs, armv7 и armv8.

- board. Содержит код, зависящий от платы. Если имеется несколько плат одного производителя, то они будут собраны в один подкаталог, поэтому код для платы am335x evm, на которой основана BeagleBone, находится в каталоге board/ti/am335x.
- common. Содержит общие функции, в том числе командные оболочки и команды, которые из них вызываются. Код команды находится в файле cmd_[command name].c.
- doc. Содержит несколько файлов README с описанием различных аспектов UBoot. Если вас интересует, как портировать U-Boot, то начать стоит отсюда.
- include. Помимо множества общих файлов-заголовков, здесь находится весьма важный подкаталог include/configs, где вы найдете большинство конфигурационных параметров плат. По мере перехода на Kconfig данные будут перемещаться отсюда в файлы Kconfig, но на момент написания книги этот процесс только начался.

Kconfig и U-Boot

О том, как Kconfig извлекает информацию из файлов и сохраняет полную конфигурацию системы в файле .config, более подробно рассказано в главе 4. В UBoot системы kconfig и kbuild реализованы с одним изменением. Процедура сборки U-Boot может порождать до трех двоичных файлов: обычный u-boot.bin, вторичный загрузчик (SPL) и третичный загрузчик (TPL), и у каждого могут быть свои конфигурационные параметры. Поэтому строки в файле .config и конфигурационных файлах, подразумеваемых по умолчанию, могут содержать префиксы, описывающие, к какому артефакту они относятся. Эти префиксы приведены в таблице ниже.

Нет префикса	Только обычный образ
S:	Только образ SPL
T:	Только образ TPL
ST:	Образы SPL и TPL
+S:	Обычный образ и образ SPL
+T:	Обычный образ и образ TPL
+ST:	Обычный образ и образы SPL и TPL

Для каждой платы существует конфигурация по умолчанию, хранящаяся в файле configs/[board name}_defconfig. Для своей платы Nova вы должны будете создать файл с именем nova_defonfig и поместить в него, к примеру, такие строки:

```
CONFIG_SPL=y
CONFIG_SYS_EXTRA_OPTIONS="SERIAL1,CONS_INDEX=1,EMMC_BOOT"
+S:CONFIG_ARM=y
+S:CONFIG_TARGET_NOVA=y
```

Первая строка CONFIG_SPL=y означает, что нужно сгенерировать двоичный образ SPL – MLO. Строка CONFIG_ARM=y означает, что нужно включить в содержимое

файлы arch/arm/Kconfig. Четвертая строка CONFIG_TARGET_NOVA=y служит для выбора вашей платы. Отметим, что префикс +S: в третьей и четвертой строках означает, что обе относятся к обычному образу и образу SPL.

Кроме того, нужно добавить пункт меню в файл Kconfig для архитектуры ARM, чтобы пользователь мог выбрать плату Nova в качестве целевой:

```
CONFIG_SPL=y
config TARGET_NOVA
bool "Support Nova!"
```

Файлы, специфичные для платы

Для каждой платы имеется подкаталог с именем board/[название платы] или board/[производитель]/[название платы], в котором должны находиться следующие файлы:

- Kconfig: содержит конфигурационные параметры платы;
- MAINTAINERS: содержит сведения о том, осуществляется ли сопровождение платы, и если да, то кем;
- Makefile: нужен для сборки специфичного для платы кода;
- README: содержит полезную информацию об этом портировании U-Boot, например какие варианты оборудования поддерживаются.

Помимо этих, могут существовать файлы с исходным кодом специфичных для платы функций.

Ваша плата Nova основана на BeagleBone, которая, в свою очередь, основана на TI AM335x EVM, поэтому можно начать с копирования файлов для платы am335x:

```
$ mkdir board/nova
$ cp -a board/ti/am335x board/nova
```

Затем изменим файл Kconfig, внеся в него изменения для платы Nova:

```
if TARGET_NOVA

config SYS_CPU
default "armv7"

config SYS_BOARD
default "nova"

config SYS_SOC
default "am33xx"

config SYS_CONFIG_NAME
default "nova"
endif
```

Поскольку SYS_CPU равно armv7, то будет откомпилирован и скомпонован код в каталоге arch/arm/cpu/armv7. Так как SYS_SOC равно am33xx, то включается код из каталога arch/arm/cpu/armv7/am33xx, запись значения nova в переменную SYS_BOARD подключает файлы из каталога board/nova, а значения nova в переменную SYS_CON-

FIG_NAME означает, что в последующих конфигурационных параметрах используется файл-заголовок include/configs/nova.h.

В каталоге board/nova есть еще один файл, который нужно будет изменить: скрипт компоновщика board/nova/u-boot.lds, в котором зашита ссылка на файл board/ti/am335x/built-in.o. Измените его, так чтобы упоминалась локальная копия в каталоге nova:

```
diff --git a/board/nova/u-boot.lds b/board/nova/u-boot.lds
index 78f294a..6689b3d 100644
--- a/board/nova/u-boot.lds
+++ b/board/nova/u-boot.lds
@@ -36,7 +36,7 @@ SECTIONS
*(.__image_copy_start)
*(.vectors)
CPUDIR/start.o (.text*)
- board/ti/am335x/built-in.o (.text*)
+ board/nova/built-in.o (.text*)
*(.text*)
}
```

Конфигурационные файлы-заголовки

Для каждой платы в каталоге include/configs имеется файл-заголовок, содержащий большинство конфигурационных параметров. Имя этого файла задано в переменной SYS_CONFIG_NAME в файле Kconfig данной платы. Формат файла подробно описан в файле README на верхнем уровне дерева исходных кодов U-Boot.

Для своей платы Nova просто скопируйте файл am335x_evm.h в nova.h и внесите небольшие изменения:

```
diff --git a/include/configs/nova.h b/include/configs/nova.h
index a3d8a25..8ea1410 100644
--- a/include/configs/nova.h
+++ b/include/configs/nova.h
@@ -1,5 +1,5 @@
/*
- * am335x_evm.h
+ * nova.h, based on am335x_evm.h
*
* Copyright (C) 2011 Texas Instruments Incorporated - http://www.ti.com/
*
@@ -13,8 +13,8 @@
* GNU General Public License for more details.
*/
-#ifndef __CONFIG_AM335X_EVM_H
-#define __CONFIG_AM335X_EVM_H
+#ifndef __CONFIG_NOVA
+#define __CONFIG_NOVA
#include <configs/ti_am335x_common.h>
```

```
@@ -39,7 +39,7 @@
#define V_SCLK (V_OSCK)
/* Custom script for NOR */
-#define CONFIG_SYS_LDSCRIPT "board/ti/am335x/u-boot.lds"
+#define CONFIG_SYS_LDSCRIPT "board/nova/u-boot.lds"
/* Always 128 KiB env size */
#define CONFIG_ENV_SIZE (128 << 10)
@@ -50,6 +50,9 @@
#define CONFIG_PARTITION_UUIDS
#define CONFIG_CMD_PART
+#undef CONFIG_SYS_PROMPT
+#define CONFIG_SYS_PROMPT "nova!> "
+
#ifdef CONFIG_NAND
#define NANDARGS \
"mtdids=" MTDIDS_DEFAULT "\0" \
```

Сборка и тестирование

Чтобы собрать загрузчик для платы Nova, выберите только что созданную конфигурацию:

```
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi- nova_defconfig
$ make CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Скопируйте файлы `MLO` и `u-boot.img` в созданный ранее FAT-раздел на карте microSD и загрузите плату.

Режим Сапсан

Мы исходили из того, что загрузка современного встраиваемого процессора состоит из трех этапов: сначала код в ПЗУ на кристалле загружает вторичный загрузчик SPL, затем тот загружает файл `u-boot.bin`, который, в свою очередь, загружает ядро Linux. А нельзя ли уменьшить число шагов и тем самым упростить и ускорить процесс начальной загрузки? Для этого предназначен режим Сапсан (Falcon) загрузчика U-Boot, названный в честь сокола-сапсана, считающегося самой быстрой птицей на Земле.

Идея проста: сделать так, чтобы SPL загружал ядро непосредственно, минуя загрузку `u-boot.bin`. Не нужно ни взаимодействия с пользователем, ни скриптов. Ядро просто загружается с известного адреса во флэш-памяти или на карте eMMC в оперативную память, ему передается заранее подготовленный блок параметров, и начинается выполнение. Детали конфигурирования режима Сапсан выходят за рамки этой книги. Дополнительные сведения имеются в файле `doc/README.falcon`.

Barebox

В завершение этой главы я познакомлю вас еще с одним начальным загрузчиком, который имеет те же корни, что и U-Boot, но знаменует новый подход к началь-

ной загрузке. Он отпочковался от U-Boot и поначалу даже назывался U-Boot v2. Разработчики Barebox ставили целью объединить лучшие черты U-Boot и Linux, включая API, организованный по образцу POSIX, и монтируемые файловые системы.

Сайт проекта Barebox находится по адресу www.barebox.org, а список рассылки для разработчиков – по адресу barebox@lists.infradead.org.

Получение Barebox

Чтобы получить Barebox, склонируйте репозиторий git и выгрузите интересующую вас версию:

```
$ git clone git://git.pengutronix.de/git/barebox.git
$ cd barebox
$ git checkout v2014.12.0
```

Структурно код устроен примерно так же, как код U-Boot.

- `arch`: содержит архитектурно-зависимый код для всех основных встраиваемых архитектур. Код поддержки SoC-систем находится в каталогах `arch/[архитектура]/mach-[SoC]`, а код поддержки отдельных плат – в каталогах `arch/[архитектура]/boards`.
- `common`: содержит общие функции, в том числе командную оболочку.
- `commands`: содержит код команд, вызываемых из оболочки.
- `Documentation`: содержит шаблоны файлов с документацией. Для создания этих файлов нужно выполнить команду `make docs`. Результаты помещаются в каталог `Documentation/html`.
- `drivers`: код драйверов устройств.
- `include`: файлы-заголовки.

Сборка Barebox

В проекте Barebox уже давно используется система `kconfig/kbuild`. Конфигурационные файлы по умолчанию находятся в каталогах `arch/[архитектура]/configs`. Предположим, к примеру, что нужно собрать Barebox для платы BeagleBoard C4. Понадобятся две конфигурации: одна — для SPL, другая — для основного двоичного загрузчика. Сначала соберем `MLO`:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-omap3530_beagle_xload_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

В результате будет создан файл вторичного загрузчика, `MLO`.

Затем соберем Barebox:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-omap3530_beagle_defconfig
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-
```

Скопируем оба файла на карту SD:

```
$ cp MLO /media/boot/
$ cp barebox-flash-image /media/boot/barebox.bin
```

Теперь загрузим плату. На консоль будут выведены такие сообщения:

```
barebox 2014.12.0 #1 Wed Dec 31 11:04:39 GMT 2014

Board: Texas Instruments beagle
nand: Trying ONFI probe in 16 bits mode, aborting !
nand: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xba (Micron ), 256MiB, page
size: 2048, OOB size: 64
omap-hsmmc omap3-hsmmc0: registered as omap3-hsmmc0
mci0: detected SD card version 2.0
mci0: registered disk0
malloc space: 0x87bff400 -> 0x87fff3ff (size 4 MiB)
booting from MMC

barebox 2014.12.0 #2 Wed Dec 31 11:08:59 GMT 2014

Board: Texas Instruments beagle
netconsole: registered as netconsole-1
i2c-omap i2c-omap30: bus 0 rev3.3 at 100 kHz
ehci ehci0: USB EHCI 1.00
nand: Trying ONFI probe in 16 bits mode, aborting !
nand: NAND device: Manufacturer ID: 0x2c, Chip ID: 0xba (Micron NAND 256MiB 1,8V
16-bit), 256MiB, page size: 2048, OOB size: 64
omap-hsmmc omap3-hsmmc0: registered as omap3-hsmmc0
mci0: detected SD card version 2.0
mci0: registered disk0
malloc space: 0x85e00000 -> 0x87dfffff (size 32 MiB)
environment load /boot/barebox.env: No such file or directory
Maybe you have to create the partition.
no valid environment found on /boot/barebox.env. Using default environment
running /env/bin/init...

Hit any key to stop autoboot: 0
```

Barebox продолжает развиваться. На момент написания книги ему не хватало широты охвата оборудования, присущей U-Boot, но в новых проектах имеет смысл рассмотреть его в качестве альтернативы.

Резюме

В любой системе необходим начальный загрузчик, который приведет оборудование в работоспособное состояние и загрузит ядро. Загрузчик U-Boot снискал любовь многих разработчиков, поскольку поддерживает широкий спектр оборудования и относительно легко портируется на новые устройства. В последние годы увеличение сложности и разнообразия встраиваемого оборудования привело к появлению нового способа описания оборудования: деревьям устройств. Дерево устройств – это просто текстовое представление системы, которое компилируется в двоичное дерево устройств **(dtb)**, передаваемое ядру на этапе загрузки. Видя дерево, ядро загружает и инициализирует драйверы описанных в нем устройств.

U-Boot очень гибок в использовании, он допускает загрузку образов из массовой памяти, флэш-памяти или из сети. Barebox умеет все то же самое, но поддерживает меньше устройств. Несмотря на более элегантный дизайн и POSIX-подобный внутренний API, на момент написания этой книги он был принят на вооружение лишь небольшой группой преданных поклонников.

Рассмотрев некоторые тонкости начальной загрузки Linux, мы в следующей главе увидим, как вступает в игру третий элемент проекта встраиваемой системы – ядро.

Глава 4

Портирование и конфигурирование ядра

Ядро – третий элемент встраиваемой Linux-системы. Оно отвечает за управление ресурсами и организацию интерфейса с оборудованием, а следовательно, влияет практически на все аспекты окончательной программной системы. Обычно оно оптимизируется для конкретной конфигурации оборудования, хотя, как мы видели в главе 3, деревья устройств позволяют создать обобщенное ядро, которое адаптируется к оборудованию, перечисленному в дереве.

В этой главе мы узнаем, как получить ядро для платы, сконфигурировать и откомпилировать его. Мы еще раз вернемся к начальной загрузке, но на этот раз нас будет интересовать роль ядра. Мы также рассмотрим драйверы устройств и то, как они получают информацию из дерева устройств.

Что делает ядро?

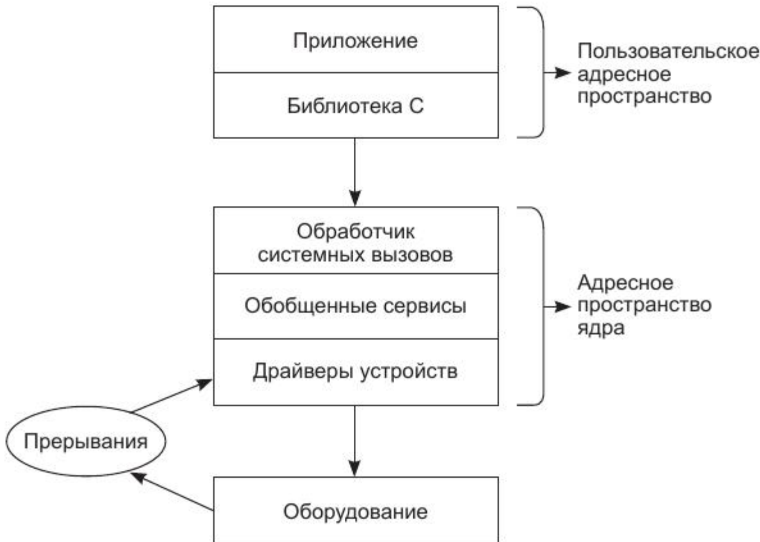
Жизнь Linux началась в 1991-м, когда Линус Торвальдс приступил к написанию операционной системы для персональных компьютеров на базе процессоров Intel 386 и 486. За образец он взял операционную систему Minix, которую Эндрю С. Таненбаум написал четырьмя годами ранее. Linux отличалась от Minix во многих отношениях, но прежде всего тем, что в ядре использовалась 32-разрядная виртуальная память, а исходный код был открыт и впоследствии выпущен на условиях лицензии GPL 2.

25 августа 1991 года Линус разместил в группе новостей comp.os.minix знаменитое объявление, начинавшееся словами: «Привет всем пользователям minix! Я пишу (бесплатную) операционную систему (это просто хобби, ничего большого и профессионального вроде gnu) для AT 386(486). Я вожусь с этим с апреля, и она, похоже, скоро будет готова. Напишите мне, кому что нравится/не нравится в minix, поскольку моя ОС на нее похожа (кроме всего прочего, у нее – по практическим соображениям – такая же физическая структура файловой системы)».

Если уж быть совсем точным, Линус написал не операционную систему, а только ядро – один из компонентов ОС. Для создания работоспособной системы он

воспользовался компонентами их проекта GNU и прежде всего набором инструментов, библиотекой C и основными командными утилитами. Так обстоит дело и по сей день, и это причина колоссальной гибкости в способах использования Linux. Ее можно комбинировать с программами GNU, работающими в пользовательском адресном пространстве, создавая дистрибутивы для работы на настольных компьютерах или на серверах, иногда такую комбинацию называют GNU/Linux. А можно объединить с программами для Android и получить хорошо известную мобильную операционную систему. Или с небольшим комплектом пользовательских утилит Busybox для создания компактной встраиваемой системы. Сравните с FreeBSD, OpenBSD и NetBSD, где ядро, набор инструментов и пользовательские приложения объединены в единую кодовую базу.

У ядра есть три основные задачи: управление ресурсами, интерфейс с оборудованием и предоставление API, реализующего уровень абстракции, полезный для пользовательских приложений. Все это показано на рисунке ниже.



Приложения, работающие в пользовательском пространстве, исполняются с низким уровнем привилегий. Они могут разве что обращаться к библиотеке. Основным интерфейсом между пользовательским пространством и пространством ядра является библиотека C, которая транслирует вызовы пользовательских функций, в частности определенные в стандарте POSIX, в системные вызовы ядра. Для реализации интерфейса системных вызовов применяется архитектурно-зависимый метод, например ловушка или программное прерывание, позволяющий переключить процессор из непривилегированного пользовательского режима в привилегированный режим ядра, в котором разрешен доступ ко всем адресам в памяти и регистрам процессора.

Обработчик системных вызовов направляет вызов подходящей подсистеме ядра: планировщику, файловой системе и т. д. Те вызовы, которым необходимы входные данные от оборудования, передаются драйверу устройства. В некоторых случаях оборудование само вызывает функцию ядра, генерируя прерывание. Прерывания могут быть обработаны только драйвером устройства и никогда – приложением, работающим в пользовательском пространстве.

Иными словами, вся полезная функциональность приложения опосредуется ядром, и, значит, ядро – один из важнейших элементов системы.

Выбор ядра

Следующий шаг – выбрать для своего проекта ядро, стремясь уравновесить желание использовать самую последнюю версию ПО и потребность в добавлениях, специфичных для поставщика оборудования.

Цикл разработки ядра

Темп разработки Linux довольно высок, новые версии выходят с промежутком от 8 до 12 недель. Схема нумерации версий в последние годы несколько изменилась. До июля 2011-го номер версии состоял из трех частей, например: 2.6.39. Средняя часть означала тип версии: нечетными числами (2.1.x, 2.3.x, 2.5.x) обозначались версии для разработчиков, четными – стабильные версии для конечных пользователей. Начиная с версии 2.6 идея долгосрочной разрабатываемой ветви (с нечетными номерами) была отброшена, поскольку замедляла предоставление новой функциональности пользователям. Переход от версии 2.6.39 к 3.0 в июле 2011-го произошел просто потому, что Линус решил, что номера стали слишком большими: между этими версиями не было гигантского прорыва в функциональности или архитектуре Linux. Заодно Линус воспользовался возможностью убрать среднюю часть номера. Затем, в апреле 2015-го, главный номер версии был изменен с 3 на 4, опять-таки из эстетических соображений, а не вследствие крупного изменения архитектуры.

Линус руководит разработкой ядра. Вы можете следить за его действиями, клонировав его репозиторий в git:

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

В результате будет создан подкаталог `linux`. Для получения актуальной версии время от времени выполняйте команду `git pull`, находясь в этом каталоге.

В настоящее время полный цикл разработки ядра начинается окном слияния продолжительностью две недели, в течение которого Линус принимает дополнения и исправления, содержащие новую функциональность. В конце этого периода начинается фаза стабилизации, в течение которой Линус генерирует предвыпускные версии с номерами, заканчивающимися на -rc1, -rc2 и т. д., обычно до -rc7 или -rc8. В течение этого периода пользователи тестируют систему и отправляют извещения об ошибках и исправлениях. Когда все серьезные ошибки исправлены, версия выпускается.

Код, включаемый на протяжении окна слияния, должен быть уже достаточно зрелым. Обычно он извлекается из репозиториев лиц, отвечающих за сопровождение различных подсистем и архитектурных вариантов ядра. Благодаря короткому циклу разработки новая функциональность может быть включена по мере готовности. Если разработчики ядра считают функцию недостаточно стабильной, ее включение можно просто отложить до следующей версии.

Отслеживать, что изменилось в каждой версии, нелегко. Можно почитать журнал фиксаций в git-репозитории Линуса, но для каждой версии в нем порядка 10 000 записей, так что составить их обзор затруднительно. К счастью, существует сайт *Linux Kernel Newbies*, где на странице http://kernelnewbies.org/LinuxVersions имеется краткий обзор каждой версии.

Стабильные и долгосрочные версии

Высокий темп изменения Linux – вещь хорошая, поскольку стержневая кодовая база пополняется новыми возможностями, но это плохо стыкуется с более длительным жизненным циклом встраиваемых систем. У разработчиков ядра двоякий подход к этой проблеме. Прежде всего признается, что любая версия может содержать ошибки, которые должны быть исправлены до выхода следующей версии. В этом заключается роль стабильного ядра Linux, которое сопровождает Грег Кроа-Хартман (Greg Kroah-Hartman). После выпуска ядро переходит из стержневого состояния (сопровождаемого Линусом) в стабильное (сопровождаемое Грегом). Версии стабильного ядра, содержащие исправления, получают третью часть номера: 3.18.1, 3.18.2 и т. д. До выхода версии 3 номера состояли из четырех частей: 2.6.29.1, 2.6.39.2 и т. д.

Для получения стабильного дерева служит такая команда:

```
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git
```

Можно получить и конкретную версию командой `git checkout`, например:

```
$ cd linux-stable
$ git checkout v4.1.10
```

Обычно стабильное ядро сопровождается только до выхода очередной стержневой версии – спустя 8–12 недель, так что на сайте `kernel.org` присутствует всего одна стабильная версия, реже две. Для учета интересов пользователей, которые хотели бы, чтобы обновления производились реже, но с гарантией, что все ошибки найдены и исправлены, некоторые версии ядра помечаются как долгосрочные (long term) и сопровождаются в течение двух и более лет. Каждый год выходит, по меньшей мере, одно долгосрочное ядро. На момент написания книги на сайте `kernel.org` было всего восемь долгосрочных ядер: 4.1, 3.18, 3.14, 3.12, 3.10, 3.4, 3.2, 2.6.32. Последнее сопровождается уже пять лет, его текущая версия имеет номер 2.6.32.68. Если вы собираетесь поддерживать свое изделие в течение столь длительного времени, рекомендую выбирать последнее долгосрочное ядро.

Поддержка со стороны производителя

В идеальном мире мы могли бы скачать ядро с сайта `kernel.org` и сконфигурировать его для любого устройства, поддерживающего Linux. Но это не всегда возможно: на самом деле в стержневой версии Linux гарантирована поддержка лишь небольшого подмножества многочисленных устройств, работающих под управлением Linux. Поддержку интересующей платы или SoC-системы можно найти на сайтах независимых проектов с открытым исходным кодом, например Linaro или Yocto Project, или получить у компаний, предлагающих стороннюю поддержку встраиваемых Linux-систем. Но во многих случаях мы вынуждены просить работающее ядро у производителя платы или SoC-системы. Как мы уже знаем, одни производители в этом отношении лучше, другие хуже.

Я могу лишь посоветовать выбирать производителей, предоставляющих качественную поддержку или, что еще лучше, взявших на себя труд поместить внесенные в ядро изменения в стержневую версию.

Однако в области лицензирования Linux имеется одна проблема, вызывающая сомнения и нескончаемые споры: модули ядра. Модуль ядра – это код, который динамически прикомпоновывается к ядру во время выполнения, расширяя его функциональность. В GPL нет никакого различия между статической и динамической компоновками, поэтому на первый взгляд кажется, что исходный код модулей ядра подпадает под действие GPL. Но на заре становления Linux велись дебаты по поводу исключений из этого правила, например в связи с файловой системой Andrew. Этот код был написан раньше Linux и потому (по мнению одной стороны) не является производным произведением, а значит, лицензия на него не распространяется. Подобные споры возникали и по поводу других фрагментов кода, и в результате возобладало мнение, что GPL не обязательно применима к модулям ядра. Это положение закреплено в макросе ядра `MODULE_LICENSE`, который может принимать в качестве аргумента значение `Proprietary`, означающее, что код не подпадает под действие GPL. Если вы собираетесь воспользоваться теми же аргументами, то почитайте часто цитируемую переписку «Linux GPL and binary module exception clause?» (http://yarchive.net/comp/ linux/gpl_modules.html).

Лицензию GPL следует рассматривать как благо, потому что она гарантирует, что, работая над проектом встраиваемой системы, и я, и вы сможем получить исходный код ядра. Иначе встраиваемые Linux-системы было бы гораздо труднее использовать, а поддержка оказалась бы фрагментарной.

Сборка ядра

Решив, какое ядро положить в основу, можно переходить к следующему шагу – сборке.

Получение исходного кода

Предположим, что ваша плата поддерживается стержневой версией. Исходный код можно получить из git или в виде архивного файла. Git лучше, потому что по-

зволяет ознакомиться с историей фиксаций, просматривать собственные изменения и переключаться между ветвями и версиями. В примере ниже мы клонируем стабильное дерево и выгружаем версию с меткой 4.1.10:

```
$ git clone \
git://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git linux
$ cd linux
$ git checkout v4.1.10
```

Можно было бы вместо этого скачать архив по адресу https://cdn.kernel.org/pub/linux/kernel/v4.x/linux-4.1.10.tar.xz.

Объем кода очень велик. В ядре 4.1 свыше 38 000 файлов с исходным кодом на C (включая заголовки) и ассемблере, что составляет более 12,5 миллиона строк кода (согласно утилите `cloc`). Тем не менее следует понимать общую структуру кода и знать, в каком приблизительно месте искать конкретную компоненту. Ниже перечислены основные каталоги.

- `arch`. Здесь находятся архитектурно-зависимые файлы. Каждой архитектуре отведен свой подкаталог.
- `Documentation`. Документация по ядру. Желая найти сведения о каком-то аспекте Linux, начинайте отсюда.
- `drivers`. Содержит тысячи драйверов устройств. Для каждого типа драйверов существует отдельный подкаталог.
- `fs`. Код файловой системы.
- `include`. Файлы-заголовки ядра, в том числе необходимые для сборки набора инструментов.
- `init`. Код инициализации ядра.
- `kernel`. Базовые функции, включая планирование, блокировку, таймеры, управление энергосбережением и код отладки и трассировки.
- `mm`. Код подсистемы управления памятью.
- `net`. Сетевые протоколы.
- `scripts`. Множество полезных скриптов, включая компилятор деревьев устройств dtc, описанный в главе 3.
- `tools`. Многочисленные полезные инструменты, в том числе система измерения производительности Linux perf, описанная в главе 13.

Со временем вы освоитесь с этой структурой и поймете, что код управления последовательным портом для конкретной SoC-системы нужно искать в каталоге `drivers/tty/serial`, а не в `arch/$ARCH/mach-foo`, потому что это драйвер устройств, а не что-то, необходимое для выполнения Linux в данной SoC-системе.

О конфигурировании ядра

Одна из сильных сторон Linux – широчайшие возможности конфигурирования ядра для решения различных задач, от небольшого специализированного устройства типа интеллектуального термостата до сложного мобильного телефона. В современных версиях число конфигурационных параметров исчисляется многими тысячами. Правильное задание конфигурации – сама по себе сложная задача, но прежде я хочу показать, как это работает, чтобы вы понимали, что происходит.

Механизм конфигурирования называется Kconfig, а интегрированная с ним система сборки – Kbuild. Документация по тому и другому находится в каталоге Documentation/kbuild/. Связка Kconfig/Kbuild используется не только в ядре, но и во многих других проектах, в том числе crosstool-NG, U-Boot, Barebox и BusyBox.

Конфигурационные параметры представлены в иерархии файлов с именем Kconfig с использованием синтаксиса, описанного в документе Documentation/kbuild/kconfig-language.txt. В Linux Kconfig верхнего уровня выглядит так:

```
mainmenu "Linux/$ARCH $KERNELVERSION Kernel Configuration"
config SRCARCH
  string
  option env="SRCARCH"
  source "arch/$SRCARCH/Kconfig"
```

В последней строке упоминается архитектурно-зависимый конфигурационный файл, который подтягивает другие файлы Kconfig в зависимости от выбранных параметров. Наличие ссылки на архитектуру влечет за собой два следствия. Во-первых, при конфигурировании Linux нужно обязательно задавать архитектуру в виде присваивания ARCH=[архитектура], иначе по умолчанию будет выбрана архитектура локальной машины. А во-вторых, структура меню верхнего уровня зависит от архитектуры.

В качестве значения переменной ARCH присваивается путь к одному из подкаталогов каталога arch. При этом значениям ARCH=i386 и ARCH=x86_64 соответствует один и тот же файл arch/x86/Kconfig.

Файлы Kconfig по большей части состоят из меню, которые разграничены ключевыми словами menu "заголовок меню" и endmenu, и пунктами меню, которые начинаются ключевым словом config. Вот пример, взятый из файла drivers/char/Kconfig:

```
menu "Character devices"
[...]
config DEVMEM
  bool "/dev/mem virtual device support"
  default y
    help
    Say Y here if you want to support the /dev/mem device.
    The /dev/mem device is used to access areas of physical
    memory.
    When in doubt, say "Y".
```

Следующий за словом config параметр – это имя переменной, в данном случае DEVMEM. Параметр имеет булев тип, и, значит, переменная может принимать одно из двух значений: если включен, то y, в противном случае она не определена вовсе. В качестве названия меню на экране отображается строка, следующая за ключевым словом bool.

Этот элемент конфигурации, как и все остальные, сохраняется в файле .config (начальная точка '.' означает, что это скрытый файл, который команда ls не по-

казывает, если только не запущена с флагом -a). Имена переменных в файле .config получают префикс CONFIG_, так что если переменная DEVMEM включена, то в файле будет присутствовать строка

```
CONFIG_DEVMEM=y
```

Помимо bool, существуют и другие типы данных. Ниже приведен полный перечень.

- bool: либо y, либо не определено.
- tristate: используется, когда некоторая функциональная возможность может быть собрана как модуль ядра или встроена в основной образ ядра. Если значение равно m, то функция собирается как модуль, если y – то встраивается, а если не определено – то не включается вовсе.
- int: целое число в десятичной записи.
- hex: целое число без знака в шестнадцатеричной записи.
- string: строковое значение.

Между элементами могут существовать зависимости, выражаемые фразой depends on, например:

```
config MTD_CMDLINE_PARTS
  tristate "Command line partition table parsing"
  depends on MTD
```

Если переменная CONFIG_MTD нигде не определена, то этот пункт меню не показывается и, стало быть, не может быть выбран.

Существуют также обратные зависимости: ключевое слово select активирует другие параметры, если активирован данный. В файле Kconfig в каталоге arch/$ARCH есть много предложений select, активирующих различные архитектурно-зависимые возможности. Вот пример для архитектуры arm:

```
config ARM
  bool
default y
  select ARCH_HAS_ATOMIC64_DEC_IF_POSITIVE
  select ARCH_HAS_ELF_RANDOMIZE
[...]
```

Существует несколько конфигурационных утилит, которые умеют читать файлы Kconfig и порождать файл .config. Некоторые отображают меню на экране и позволяют интерактивно выбирать параметры. Пожалуй, самая известная из них – Menuconfig, но есть также xconfig и gconfig.

Все утилиты запускаются через make, памятуя о том, что в случае ядра нужно указать архитектуру, например:

```
$ make ARCH=arm menuconfig
```

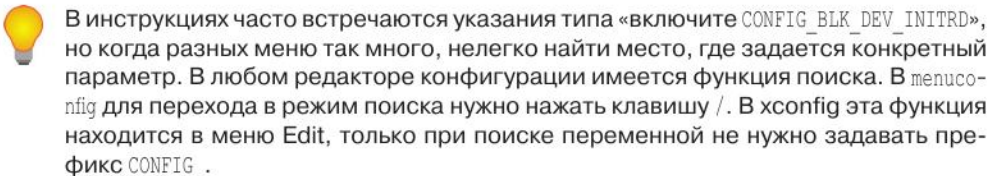
На рисунке ниже показан экран menuconfig с выделенным параметром DEVMEM, о котором мы говорили выше.

```
.config - Linux/arm 4.1.10 Kernel Configuration
> Device Drivers > Character devices ─────────────────────────────
                     Character devices
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>
  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to
  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]
 ┌────^(-)──────────────────────────────────────────────────────────
 │    < >   Trace data sink for MIPI P1149.7 cJTAG standard
 │    [*] /dev/mem virtual device support
 │    [*] /dev/kmem virtual device support
 │        Serial drivers  --->
 │    <M> TTY driver to output user messages via printk
 │    [ ] ARM JTAG DCC console
 │    < > IPMI top-level message handler  ----
 │    <M> Hardware Random Number Generator Core support  --->
 │    < > Siemens R3964 line discipline
 │    < > Applicom intelligent fieldbus card support
 └────┴(+)──────────────────────────────────────────────────────────

      <Select>    < Exit >    < Help >    < Save >    < Load >
```

Конфигурирование ядра с помощью menuconfig

Звездочка (*) слева от названия пункта меню означает, что он выбран (="y"), а буква M – что выбран и будет собран в виде модуля ядра.



В инструкциях часто встречаются указания типа «включите CONFIG_BLK_DEV_INITRD», но когда разных меню так много, нелегко найти место, где задается конкретный параметр. В любом редакторе конфигурации имеется функция поиска. В menuconfig для перехода в режим поиска нужно нажать клавишу /. В xconfig эта функция находится в меню Edit, только при поиске переменной не нужно задавать префикс CONFIG_.

При таком количестве конфигурационных параметров неразумно начинать с чистого листа всякий раз, как нужно собрать ядро. Поэтому в каталоге arch/$ARCH/configs находится много заведомо правильных конфигурационных файлов, каждый из которых содержит конфигурацию для одной SoC-системы или группы SoC-систем. Для выбора файла введите команду make [имя конфигурационного файла]. Например, чтобы сконфигурировать Linux для широкого спектра SoC-систем с архитектурой armv7-a, в том числе BeagleBone Black AM335x, нужно ввести:

```
$ make ARCH=arm multi_v7_defconfig
```

В результате будет получено обобщенное ядро, работающее на различных платах. В случае более специализированного приложения, например при использовании ядра, поставленного изготовителем, конфигурационный файл, подразумеваемый по умолчанию, входит в состав ПО, поставляемого вместе с платой; вам предстоит разобраться, какой файл взять для сборки ядра.

Существует еще одна полезная цель make-файла конфигурирования: oldconfig. Она принимает существующий файл .config и просит указать значения незаданных конфигурационных параметров. Эта цель используется, когда нужно перенести конфигурацию на новую версию ядра: мы копируем .config старого ядра в каталог с исходным кодом нового и выполняем make ARCH=arm oldconfig, чтобы актуализировать его. Цель oldconfig можно использовать и для проверки вручную измененного файла .config (если вы не обращаете внимания на фразу «Automatically generated file; DO NOT EDIT» в начале файла; впрочем, иногда игнорировать предупреждения полезно).

Если в конфигурацию внесены изменения, то модифицированный файл .config становится частью системы для вашего устройства, и его нужно поместить в систему управления версиями.

В ходе сборке ядра генерируется файл-заголовок include/generated/autoconf.h, содержащий директивы #define для всех конфигурационных параметров. Этот файл включается в исходный код ядра точно так же, как при сборке U-Boot.

Использование переменной LOCALVERSION для идентификации ядра

Узнать версию построенного ядра позволяет цель kernelversion:

```
$ make kernelversion
4.1.10
```

Эта строка печатается командой uname и используется также для именования каталога, в котором хранятся модули ядра.

Если вы собираетесь изменить подразумеваемые по умолчанию значения конфигурационных параметров, то рекомендуется дописать в конец номера версии свою информацию, задав ее с помощью переменной CONFIG_LOCALVERSION, которая находится в меню **General setup configuration**. Можно также (хотя и не рекомендуется) добиться того же эффекта, отредактировав make-файл верхнего уровня: дописав в конец строки, начинающейся словом EXTRAVERSION. Так, чтобы пометить собранное ядро идентификатором melp и версией 1.0, я должен был бы определить локальную версию в файле .config следующим образом:

```
CONFIG_LOCALVERSION="-melp-v1.0"
```

Команда make kernelversion порождает те же файлы, что и раньше, но если теперь выполнить make kernelrelease, то мы увидим:

```
$ make kernelrelease
4.1.10-melp-v1.0
```

Эта строка печатается и в начале журнала ядра:

```
Starting kernel ...
[ 0.000000] Booting Linux on physical CPU 0x0
[ 0.000000] Linux version 4.1.10-melp-v1.0 (chris@builder) (gcc version 4.9.1 (crosstool-
NG 1.20.0) ) #3 SMP Thu Oct 15 21:29:35 BST 2015
```

Теперь я вижу, что это мое специальное ядро.

Модули ядра

Я уже несколько раз упоминал модули ядра. В дистрибутивах Linux для настольных ПК они используются очень активно, что позволяет загружать нужные драйверы и функции ядра во время выполнения в зависимости от обнаруженного оборудования и требуемой функциональности. Не будь модулей, пришлось бы статически компоновать с ядром все вообще драйверы и возможности, что сделало бы его неприемлемо большим.

С другой стороны, для встраиваемых устройств конфигурация оборудования и ядра обычно известна на этапе сборки ядра, поэтому модули не так полезны. На самом деле они даже порождают проблемы, потому что создают зависимость между ядром и корневой файловой системой, а это может привести к ошибкам начальной загрузки, если одно обновлено, а другое нет. Поэтому встраиваемые ядра, как правило, собирают без модулей. Но есть несколько случаев, когда модули ядра имеют смысл:

- если имеются защищенные правом собственности модули – по описанным выше причинам, связанным с лицензированием;
- чтобы уменьшить время загрузки, отложив на потом загрузку драйверов, без которых можно обойтись на этом этапе;
- когда потенциально возможных драйверов чересчур много и для их статической компоновки потребовалось бы слишком много памяти. Например, если есть интерфейс USB, поддерживающий различные устройства. Это по существу та же причина, по которой модули используются в дистрибутивах для ПК.

Компиляция

Система сборки ядра `kbuild` представляет собой набор скриптов `make`, которые читают информацию из файла `.config`, определяют зависимости и компилируют все необходимое для создания образа ядра, содержащего все статически скомпонованные компоненты, возможно, двоичное дерево устройств и, возможно, один или несколько модулей ядра. Зависимости закодированы в make-файлах, которые находятся в каждом каталоге, содержащем компоненты, требующие сборки. Например, следующие две строки взяты из файла `drivers/char/Makefile`:

```
obj-y += mem.o random.o
obj-$(CONFIG_TTY_PRINTK) += ttyprintk.o
```

Правило `obj-y` безусловно компилирует файл для создания цели, поэтому `mem.c` и `random.c` всегда входят в ядро. Во второй строке обработка файла `ttyprintk.c` зависит от конфигурационного параметра. Если переменная `CONFIG_TTY_PRINTK` равна `y`, то файл компилируется как часть ядра, если `m` – то как модуль, а если не определена, то не компилируется вовсе.

Для большинства целей достаточно просто набрать команду make (с подходящими значениями переменных ARCH и CROSS_COMPILE), но рассмотреть отдельные шаги поучительно.

Компиляция образа ядра

Для сборки образа ядра нужно знать, чего ожидает начальный загрузчик. Приведем краткое руководство.

- **U-Boot**: традиционно U-Boot требовал наличия файла uImage, но последние версии могут загружать файл zImage с помощью команды bootz.
- **Цели x86**: требуется файл bzImage.
- Большинство прочих загрузчиков: требуется файл zImage.

Вот пример сборки файла zImage:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-zImage
```

Параметр -j 4 говорит, сколько задач запускать параллельно, чтобы уменьшить время сборки. Рекомендуется запускать примерно столько задач, сколько имеется процессорных ядер.

Сборка файлов bzImage и uImage аналогична.

Существует одна тонкость при сборке файла uImage для архитектуры ARM с многоплатформенной поддержкой, что является нормой для современного поколения ядер для SoC-систем с процессором ARM. Многоплатформенная поддержка для ARM появилась в версии Linux 3.7. Она позволяет исполнять одно двоичное ядро на нескольких платформах и является шагом в направлении уменьшения числа ядер для ARM-устройств. Ядро выбирает правильную платформу, читая номер машины или дерево устройств, переданные ему начальным загрузчиком. Проблема возникает из-за того, что местоположение физической памяти может зависеть от платформы, поэтому адрес перемещения ядра (обычно 0x8000 байтов от начала физической ОЗУ) также может различаться. Адрес перемещения записывается в заголовок uImage командой mkimage на этапе сборки ядра, но это решение не годится, если существует несколько возможных адресов. Иначе говоря, формат uImage несовместим с многоплатформенными образами. Однако создать двоичный образ uImage в процессе сборки многоплатформенной системы все же можно, если задать адрес LOADADDR для конкретной SoC-системы, на которую это ядро предположительно будет загружаться. Узнать адрес загрузки можно, поискав значение zreladdr-y в файле mach-[ваша SoC]/Makefile.boot.

В случае платы BeagleBone Black полная команда выглядит так:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- LOADADDR=0x80008000 uImage
```

Процедура сборки ядра порождает два файла в каталоге верхнего уровня: vmlinux и System.map. Первый содержит ядро в виде файла в формате ELF. Если ядро компилировалось в отладочном режиме (CONFIG_DEBUG_INFO=y), то файл будет содержать отладочные символы, полезные отладчикам, например kgdb. К нему можно применить также другие инструменты для работы с двоичными файлами в формате ELF, например size:

```
$ arm-cortex_a8-linux-gnueabihf-size vmlinux
text data bss dec hex filename
8812564 790692 8423536 18026792 1131128 vmlinux
```

Файл `System.map` содержит таблицу символов в понятном человеку виде.

Большинство начальных загрузчиков не может работать с кодом в формате ELF напрямую. На следующем этапе файл `vmlinux` обрабатывается, и результаты помещаются в каталог `arch/$ARCH/boot` в формате, пригодном для различных загрузчиков.

- `Image`: `vmlinux` преобразуется в чисто двоичный формат.
- `zImage`: в случае архитектуры PowerPC сжатый вариант `Image`. Следовательно, предполагается, что начальный загрузчик умеет выполнять распаковку. Для прочих архитектур к сжатому файлу `Image` присоединяется короткий код, который распаковывает и перемещает его в памяти.
- `uImage`: `zImage` плюс 64-байтовый заголовок U-Boot.

В ходе сборки печатаются исполняемые команды:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-zImage
CC    init/main.o
CHK    include/generated/compile.h
CC    init/version.o
CC    init/do_mounts.o
CC    init/do_mounts_rd.o
CC    init/do_mounts_initrd.o
LD    init/mounts.o
[...]
```

Если сборка завершается с ошибкой, бывает полезно узнать, какие точно команды выполнялись. Для этого добавьте в командную строку параметр `V=1`:

```
$ make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- V=1 zImage
[...]
arm-cortex_a8-linux-gnueabihf-gcc -Wp,- MD,init/.do_mounts_initrd.o.d -nostdinc -isys-
tem /home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/lib/gcc/arm-cortex_a8-linux-gnue-
abihf/4.9.1/include -I./arch/arm/include - Iarch/arm/include/generated/uapi -Iarch/
arm/include/generated - Iinclude -I./arch/arm/include/uapi - Iarch/arm/include/gen-
erated/uapi -I./include/uapi - Iinclude/generated/uapi -include ./include/linux/
kconfig.h - D__KERNEL__ -mlittle-endian -Wall -Wundef -Wstrict-prototypes - Wno-tri-
graphs -fno-strict-aliasing -fno-common -Werror-implicit-function-declaration -Wno-
format-security -std=gnu89 -fno-dwarf2- cfi-asm -mabi=aapcs-linux -mno-thumb-inter-
work -mfpu=vfp -funwind-tables -marm -D__LINUX_ARM_ARCH__=7 -march=armv7-a -msoft-
float - Uarm -fno-delete-null-pointer-checks -O2 --param=allow-store-data-races=0 -Wframe-
larger-than=1024 -fno-stack-protector -Wno-unused-but-set-variable -fomit-frame-point-
er -fno-var-tracking-assignments -Wdeclaration-after-statement -Wno-pointer-sign -fno-str-
ict-overflow -fconserve-stack -Werror=implicit-int - Werror=strict-prototypes -Werror=date-
time -DCC_HAVE_ASM_GOTO - D"KBUILD_STR(s)=#s" - D"KBUILD_BASENAME=KBUILD_STR(do_mounts_
initrd)" - D"KBUILD_MODNAME=KBUILD_STR(mounts)" -c -o init/do_mounts_initrd.o init/
do_mounts_initrd.c
```

```
[...]
```

Компиляция деревьев устройств

Следующий шаг – построить дерево устройств или несколько деревьев, если производится многоплатформенная сборка. Цель dtbs строит деревья устройств по правилам, записанным в файле arch/$ARCH/boot/dts/Makefile, из исходных файлов, находящихся в том же каталоге:

```
$ make ARCH=arm dtbs
...
DTC arch/arm/boot/dts/omap2420-h4.dtb
DTC arch/arm/boot/dts/omap2420-n800.dtb
DTC arch/arm/boot/dts/omap2420-n810.dtb
DTC arch/arm/boot/dts/omap2420-n810-wimax.dtb
DTC arch/arm/boot/dts/omap2430-sdp.dtb
...
```

Сгенерированные dtb-файлы помещаются в тот же каталог, что исходные.

Компиляция модулей

Если в процессе конфигурирования было указано, что некоторые части системы должны быть собраны в виде модулей, то для их построения используется отдельная цель modules:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-modules
```

Файлы откомпилированных модулей имеют расширение .ko и создаются в том же каталоге, что исходный код, т. е. оказываются разбросанными по всему дереву исходного кода ядра. Искать их затруднительно, но цель modules_install поместит их в нужное место. По умолчанию этим местом является каталог /lib/modules в системе, на которой ведется разработка, но это почти наверняка не то, что вам нужно. Чтобы установить модули в область технологической подготовки в корневой файловой системе (о которой мы будем говорить в следующей главе), задайте путь к ней в переменной INSTALL_MOD_PATH:

```
$ make -j4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- INSTALL_MOD_PATH=$HOME/
rootfs modules_install
```

Теперь модули ядра будут помещены в каталог /lib/modules/[версия ядра] относительно корня файловой системы.

Удаление артефактов сборки

Для очистки дерева исходного кода ядра предназначены три цели make:

- clean: удаляет объектные файлы и большинство промежуточных;
- mrproper: удаляет все промежуточные файлы, включая .config. Эта цель применяется для возврата дерева в состояние, в котором оно находилось сразу

после клонирования репозитория или распаковки исходного кода. Название выбрано, потому что под маркой Mr Proper в некоторых странах продается чистящее средство. Назначение команды `make mrproper` – как следует отчистить исходное дерево ядра;
- `distclean`: то же, что `mrproper`, но удаляет также резервные копии, созданные редакторами, файлы, оставшиеся после наложения заплат, и другие артефакты разработки.

Загрузка ядра

Загрузка сильно зависит от устройства. Ниже приведены примеры использования U-Boot для платы BeagleBone Black и эмулятора QEMU.

BeagleBone Black

Следующие команды U-Boot показывают, как загрузить Linux на плату BeagleBone Black:

```
U-Boot# fatload mmc 0:1 0x80200000 zImage
reading zImage
4606360 bytes read in 254 ms (17.3 MiB/s)
U-Boot# fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
reading am335x-boneblack.dtb
29478 bytes read in 9 ms (3.1 MiB/s)
U-Boot# setenv bootargs console=ttyO0,115200
U-Boot# bootz 0x80200000 - 0x80f00000
Kernel image @ 0x80200000 [ 0x000000 - 0x464998 ]
## Flattened Device Tree blob at 80f00000
  Booting using the fdt blob at 0x80f00000
  Loading Device Tree to 8fff5000, end 8ffff325 ... OK
Starting kernel ...
[ 0.000000] Booting Linux on physical CPU 0x0
...
```

Обратите внимание на переменную `bootargs`, заданную как `console=ttyO0,115200`. Она говорит Linux, какое устройство использовать в качестве консоли для вывода. В данном случае – первый контроллер UART на плате, устройство `ttyO0`, на скорости 115 200 бит в секунду. Не задав этого параметра, мы не увидели бы никаких сообщений после `Starting the kernel ...`, а значит, не знали бы, работает программа или нет.

QEMU

В предположении, что уже установлена программа `qemu-system-arm`, мы можем запустить ее с ядром `multi_v7 kernel` и `dtb`-файлом для устройства ARM Versatile Express следующим образом:

```
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage - dtb vexpress-v2p-ca9.
dtb -append "console=ttyAMA0"
```

Отметим, что задание QEMU_AUDIO_DRV=none нужно только для того, чтобы подавить сообщения об ошибках QEMU об отсутствующей конфигурации аудиодрайверов, которые мы не используем.

Для выхода из QEMU нажмите **Ctrl-A**, затем **x** (два отдельных нажатия).

Паника ядра

Все так хорошо начиналось и так печально закончилось:

```
[ 1.886379] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(0,0)
[ 1.895105] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-
block(0, 0)
```

Типичный пример паники ядра. Паника случается, когда в ядре возникает неисправимая ошибка. По умолчанию ядро выводит сообщение на консоль и останавливается. Параметр командной строки panic позволяет несколько секунд подождать, а затем начать перезагрузку.

В данном случае ошибка произошла из-за отсутствия корневой файловой системы, и это свидетельствует о том, что ядро бесполезно, если нет пользовательских программ для управления им. Пользовательские приложения можно предоставить в виде корневой файловой системы на ram-диске или на монтируемом массовом запоминающем устройстве. О том, как создать корневую файловую систему, мы поговорим в следующей главе, а сейчас, чтобы все-таки запустить систему, предположим, что имеется на ram-диске в файле uRamdisk, и тогда мы сможем загрузить командную оболочку, введя в U-Boot такие команды:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
fatload mmc 0:1 0x81000000 uRamdisk
setenv bootargs console=tty00,115200 rdinit=/bin/sh
bootz 0x80200000 0x81000000 0x80f00000
```

Я добавил в командную строку параметр rdinit=/bin/sh, чтобы ядро запустило оболочку, которая выдаст приглашение. Теперь на консоль будут выведены такие сообщения:

```
...
[ 1.930923] sr_init: No PMIC hook to init smartreflex
[ 1.936424] sr_init: platform driver register failed for SR
[ 1.964858] Freeing unused kernel memory: 408K (c0824000 - c088a000)
/ # uname -a
Linux (none) 3.18.3 #1 SMP Wed Jan 21 08:34:58 GMT 2015 armv7l GNU/Linux
/ #
```

Наконец-то мы получили приглашение и можем взаимодействовать с устройством.

Подготовка пользовательского пространства

Для перехода от этапа инициализации в пользовательское пространство ядро должно смонтировать корневую файловую систему и выполнить некоторую хранящуюся в ней программу. Корневая файловая система может находиться на ram-диске, как показано в предыдущем разделе, или на монтируемом блочном устройстве. Соответствующий код находится в файле init/main.c и начинается выполнением функции rest_init(), которая создает первый поток с идентификатором 1, выполняющий код функции kernel_init(). Если имеется ram-диск, то функция попытается выполнить программу /init, которая займется настройкой пользовательского пространства.

Если найти или запустить /init не удается, то функция монтирует файловую систему, обращаясь к функции prepare_namespace() из файла init/do_mounts.c. Для ее работы в командной строке должен присутствовать параметр root=, задающий имя монтируемого блочного устройства, обычно в виде:

```
root=/dev/<имя диска><номер раздела>
root=/dev/<имя диска>p<номер раздела>
```

Так, первый раздел на карте SD будет называться root=/dev/mmcblk0p1. Если монтирование прошло успешно, то функция попытается выполнить одну из программ /sbin/init, /etc/init, /bin/init, /bin/sh, перебирая их в указанном порядке и прекращая перебор, как только будет найдена первая работающая программа.

Программу init можно переопределить в командной строке. В случае ram-диска используется параметр rdinit= (выше я указал rdinit=/bin/sh для запуска оболочки), а в случае файловой системы – init=.

Сообщения ядра

Разработчики ядра обожают выводить полезную информацию, щедро разбрасывая по коду вызовы функции printk() и ей подобных. Сообщения классифицируются по важности, причем самым важным приписывается уровень 0.

Уровень	Значение	Пояснение
KERN_EMERG	0	Система неработоспособна
KERN_ALERT	1	Необходимо немедленно предпринять какое-то действие
KERN_CRIT	2	Критическая ошибка
KERN_ERR	3	Ошибка
KERN_WARNING	4	Предупреждение
KERN_NOTICE	5	Нормальная, но требующая внимания ситуация
KERN_INFO	6	Информационное сообщение
KERN_DEBUG	7	Отладочное сообщение

Сообщения сначала записываются в буфер __log_buf, размер которого равен 2 в степени CONFIG_LOG_BUF_SHIFT. Например, если эта переменная равна 16, то раз-

мер __log_buf составляет 64 КиБ. Буфер можно сбросить на диск командой dmesg.

Если уровень сообщения меньше уровня вывода на консоль, то сообщение не только помещается в __log_buf, но и выводится на консоль. По умолчанию этот порог равен 7, т. е. сообщения уровня 6 и ниже отображаются на экране, а сообщения уровня KERN_DEBUG – нет. Изменить уровень вывода на консоль можно несколькими способами, в том числе с помощью параметра ядра loglevel=<level> или командой dmesg -n <level>.

Командная строка ядра

Командная строка ядра – это строка, которую начальный загрузчик передает ядру – в случае U-Boot с помощью переменной bootargs. Ее можно также определить в дереве устройств или задать как часть конфигурации ядра в параметре CONFIG_CMDLINE.

Мы уже видели примеры командной строки ядра, но возможных параметров гораздо больше. Их полный перечень приведен в файле Documentation/kernel-parameters.txt. Ниже перечислены наиболее полезные.

Имя	Описание
debug	Устанавливает максимальное значение (8) уровня вывода на консоль, при котором на консоль выводятся все сообщения ядра
init=	Имя программы init в монтируемой корневой файловой системе. По умолчанию /sbin/init
lpj=	Задает значение параметра loops_per_jiffy (см. следующий абзац)
panic=	Поведение в случае паники ядра. Значение, большее 0, интерпретируется как число секунд до перезагрузки. Если значение равно 0, то система ждет бесконечно (это режим по умолчанию), а если меньше 0, то перезагружается немедленно
quiet	Устанавливает уровень вывода на консоль равным 1, при этом подавляются все сообщения, кроме аварийных. Поскольку большинство устройств оснащено последовательной консолью, для вывода сообщений требуется заметное время. Поэтому уменьшение их числа сокращает время загрузки
rdinit=	Имя программы init на ram-диске. По умолчанию /init
ro	Корневое устройство монтируется в режиме чтения. На ram-диск не оказывает влияния, он всегда допускает чтение и запись
root=	С какого устройства монтировать корневую файловую систему
rootdelay=	Сколько секунд ждать до попытки смонтировать корневое устройство, по умолчанию 0. Полезно, если устройству требуется время для получения ответа от оборудования. См. также rootwait
rootfstype=	Тип файловой системы на корневом устройстве. Во многих случаях определяется автоматически на стадии монтирования, но для файловых систем типа jffs2 должен быть задан
rootwait	Ждать обнаружения корневого устройства бесконечно. Обычно необходимо для устройств mmc
rw	Монтировать корневое устройство в режиме чтения-записи (режим по умолчанию)

Параметр lpj часто упоминается в контексте уменьшения времени загрузки

ядра. На этапе инициализации ядро крутится в цикле примерно 250 мс для калибровки цикла задержки. Полученное значение сохраняется в переменной loops_per_jiffy и выводится в составе следующего сообщения:

```
Calibrating delay loop... 996.14 BogoMIPS (lpj=4980736)
```

Если ядро всегда исполняется на одном и том же оборудовании, то вычисленное значение всегда будет одинаково. Поэтому можно уменьшить время загрузки на 250 мс, включив в командную строку параметр lpj=4980736.

Портирование Linux на новую плату

Сложность этой работы зависит от того, насколько новая плата похожа на какую-нибудь из существующих макетных плат. В главе 3 было описано, как портировать U-Boot на гипотетическую плату Nova, основанную на BeagleBone Black (говоря «основана», я имею в виду «совпадает»), и в этом случае изменений в ядре будет совсем немного. Но если производится портирование на совершенно новое оборудование, то задача усложняется. Я рассмотрю только простой случай.

Организация архитектурно-зависимого кода в каталоге arch/$ARCH различна для разных систем. С архитектурой x86 все сравнительно просто, потому что аппаратные детали определяются во время выполнения. Для архитектуры PowerPC файлы, специфичные для конкретных SoC-систем и плат, находятся в разных платформенных подкаталогах. Для архитектуры ARM количество файлов, зависящих от SoC-систем и плат, максимально, потому что устройств с такой архитектурой очень много. Платформенно-зависимый код находится в подкаталогах с именами вида mach-* каталога arch/arm, чаще всего по одному на SoC-систему. Есть также подкаталоги с именами вида plat-*, в которых находится код, общий для нескольких версий SoC-системы. В случае платы Nova нас будет интересовать каталог mach-omap2. Несмотря на имя, в нем находится код для поддержки микросхем OMAP2, 3 и 4.

В следующих разделах я покажу два способа портирования на плату Nova: сначала с деревом устройств, а потом без него, поскольку в мире существует очень много устройств, попадающих в эту категорию. Мы убедимся, что при наличии дерева устройств все существенно упрощается.

С деревом устройств

Прежде всего нужно создать дерево устройств для платы и модифицировать его, описав новое или измененное оборудование. В нашем простом случае достаточно скопировать файл am335x-boneblack.dts в nova.dts и изменить название платы:

```
/dts-v1/;
#include "am33xx.dtsi"
#include "am335x-bone-common.dtsi"
/ {
      model = "Nova";
```

```
        compatible = "ti,am335x-bone-black", "ti,am335x-bone", "ti,am33xx";
    };
...
```

Файл nova.dtb можно построить явно:

```
$ make ARCH=arm nova.dtb
```

А если мы хотим, чтобы файл nova.dtb по умолчанию генерировался для платформы OMAP2 командой make ARCH=arm dtbs, то добавим следующие строки в файл arch/arm/boot/dts/Makefile:

```
dtb-$(CONFIG_SOC_AM33XX) += \
[...]
nova.dtb \
[...]
```

Теперь можно загрузить тот же файл zImage, что раньше был построен на основе конфигурации multi_v7_ defconfig, но с деревом nova.dtb:

```
Starting kernel ...
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Initializing cgroup subsys cpuset
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.18.3-dirty (chris@builder) (gcc version 4.9.1 (crosstool-N
G 1.20.0) ) #1 SMP Wed Jan 28 07:50:50 GMT 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c5387d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine model: Nova
...
```

Можно было бы создать собственную конфигурацию, скопировав файл multi_v7_defconfig, добавив нужную нам функциональность и удалив все лишнее, чтобы уменьшить размер ядра.

Без дерева устройств

Прежде всего нужно придумать конфигурационное имя для платы, пусть это будет NOVABOARD. Его нужно включить в файл Kconfig в каталоге mach- для вашей SoC-системы, а затем добавить зависимость для поддержки самой SoC-системы – OMAPAM33XX.

Таким образом, в файл arch/arm/mach-omap2/Kconfig добавляем такие строки:

```
config MACH_NOVABOARD
bool "Nova board"
depends on SOC_OMAPAM33XX
default n
```

Для каждой платы существует исходный файл с именем вида board-*.c, он

содержит код и конфигурационные параметры, специфичные для этой платы. В нашем случае файл будет называться board-nova.c и содержать копию board-am335xevm.c. Должно существовать правило для его компиляции, обусловленное переменной CONFIG_MACH_NOVABOARD, поэтому добавим в файл arch/arm/mach-omap2/Makefile такую строку:

```
obj-$(CONFIG_MACH_NOVABOARD) += board-nova.o
```

Поскольку мы не используем дерево устройств для идентификации платы, то должны прибегнуть к старому механизму, основанному на номерах машин. Это число, уникально идентифицирующее плату, передается начальным загрузчиком в регистре r1, а код инициализации ARM с его помощью выберет нужный код поддержки платы. Официальный список номеров машин ARM опубликован по адресу www.arm.linux.org.uk/developer/machines/download.php. Заявку на получение нового номера машины можно заполнить на странице www.arm.linux.org.uk/developer/ machines/?action=new#.

Если нам выделили номер машины 4242, то нужно добавить его в файл arch/arm/tools/mach-types:

```
machine_is_xxx      CONFIG_xxxx         MACH_TYPE_xxx       number
...
nova_board          MACH_NOVABOARD      NOVABOARD           4242
```

Во время сборки ядра этот файл используется для генерации файла-заголовка mach-types.h в каталоге include/generated/.

Номер машины и код поддержки платы связываются вместе с помощью следующей структуры:

```
MACHINE_START(NOVABOARD, "nova_board")
/* Maintainer: Chris Simmonds */
.atag_offset      = 0x100,
.map_io           = am335x_evm_map_io,
.init_early       = am33xx_init_early,
.init_irq         = ti81xx_init_irq,
.handle_irq       = omap3_intc_handle_irq,
.timer            = &omap3_am33xx_timer,
.init_machine     = am335x_evm_init,
MACHINE_END
```

Отметим, что в файле платы может быть несколько структур с описанием машины, что позволяет создавать ядра, работающие на разных платах. Номер машины, переданный начальным загрузчиком, используется для выбора нужного кода поддержки.

Наконец, для нашей платы нужно создать конфигурацию по умолчанию, в которой будут заданы переменная CONFIG_MACH_NOVABOARD и другие конфигурационные параметры. В примере ниже файл называется arch/arm/configs/novaboard_defconfig. Теперь можно собирать образ ядра, как обычно:

```
$ make ARCH=arm novaboard_defconfig
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabi-zImage
```

Остался еще один шаг. Необходимо модифицировать начальный загрузчик, так чтобы он передавал правильный номер машины. В предположении, что используется U-Boot, нам понадобится скопировать номера машин, сгенерированные Linux, из файла `arch/arm/include/asm/mach-types.h` в файл U-Boot `arch/arm/include/asm/mach-types.h`. Затем нужно обновить конфигурационный файл-заголовок для Nova, `include/configs/nova.h`, добавив в него строку:

```
#define CONFIG_MACH_TYPE MACH_TYPE_NOVABOARD
```

И наконец, мы можем собрать U-Boot и воспользоваться им для загрузки нового ядра на плату Nova:

```
Starting kernel ...
[    0.000000] Linux version 3.2.0-00246-g0c74d7a-dirty (chris@builder) (gcc version 4.9.
1 (crosstool-NG 1.20.0) ) #3 Wed Jan 28 11:45:10 GMT 2015
[    0.000000] CPU: ARMv7 Processor [413fc082] revision 2 (ARMv7), cr=10c53c7d
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
[    0.000000] Machine: nova_board
```

Дополнительная литература

На указанных ниже сайтах имеется дополнительная информация о затронутых в этой главе темах:

- Linux Kernel Newbies, kernelnewbies.org.
- Linux Weekly News, www.lwn.net.

Резюме

Ядро Linux – чрезвычайно мощная и сложная система, которую можно объединить с различными пользовательскими приложениями: простыми встраиваемыми устройствами, мобильными устройствами различной сложности на платформе Android или полнофункциональным сервером. Одной из его сильных сторон является степень конфигурируемости. Официальным источником исходного кода является сайт www.kernel.org, но, вероятно, вам понадобится код для конкретной SoC-системы или платы, поставляемый изготовителем устройства или сторонней компанией. Модификация ядра для конкретного целевого устройства может состоять в изменении базового кода ядра, добавлении драйверов устройств, отсутствующих в стержневой версии Linux, изменении стандартного файла конфигурации ядра и исходного кода дерева устройств.

Обычно мы начинаем с конфигурации целевой платы по умолчанию, а затем вносим изменения с помощью какого-либо инструмента конфигурирования, например `menuconfig`. При этом надо подумать о том, как компилировать те или иные

функции ядра: как модули или встроенные части. Обычно модули ядра не дают преимущества во встраиваемых системах, где функциональность и оборудование определены заранее. Тем не менее модули часто используются для импорта в ядро закрытого кода, а также чтобы уменьшить время загрузки за счет откладывания на потом загрузки некритичных драйверов. Процедура сборки ядра порождает сжатый образ в виде файла `zImage`, `bzImage` или `uImage` в зависимости от используемого начального загрузчика и целевой архитектуры. Кроме того, порождаются сконфигурированные модули ядра (ko-файлы) и, если требуется, двоичные деревья устройств (dtb-файлы).

Портирование Linux на новую плату может оказаться совсем простой или весьма сложной задачей в зависимости от того, насколько оборудование отличается от всего, что имеется в стержневой версии или в ядре, поставленном изготовителем. Если в основе вашего оборудования лежит хорошо известная эталонная конструкция, то, возможно, понадобится внести лишь минимальные изменения в дерево устройств или в платформенные данные. Не исключено также, что понадобится добавить драйверы устройств, о чем пойдет речь в главе 8. Но если оборудование кардинально отличается от известного, то, возможно, придется модифицировать базовый код ядра, а это уже выходит за рамки книги.

Ядро – это сердце Linux-системы, но оно не может работать само по себе. Необходима корневая файловая система, содержащая пользовательские приложения. Она может находиться на ram-диске или на блочном устройстве. Это и будет темой следующей главы. Как мы увидим, попытка загрузить ядро без корневой файловой системы приводит к панике.

Глава 5

Построение корневой файловой системы

Корневая файловая система – четвертый и последний элемент встраиваемой Linux-системы. Прочитав эту главу, вы будете знать, как собрать, загрузить и работать с такой системой.

Фундаментальные концепции, стоящие за корневой файловой системой, мы изучим путем ее построения с нуля. Основная цель – рассказать достаточно для понимания и полноценного использования таких систем сборки, как Buildroot и Yocto Project, которым будет посвящена глава 6.

Описываемая техника широко известна под названием «самопальной» (**roll your own** – **RYO**[1]). Когда встраиваемые Linux-системы только начали появляться, это был единственный способ создать корневую файловую систему. И до сих пор встречаются ситуации, когда техника самопальных систем применима, например, если объем ОЗУ или внешней памяти сильно ограничен, для оперативного проведения демонстраций и вообще всегда, когда требования таковы, что стандартные средства сборки системы плохо подходят. Но такие случаи все же редки. Хочу специально подчеркнуть, что эта глава написана прежде всего в педагогических целях, не следует рассматривать ее как рецепт построения типичных встраиваемых систем – для этой цели есть инструменты, описанные в следущей главе.

Основная задача – создать минимальную корневую файловую систему, достаточную, чтобы получить приглашение оболочки. Затем, заложив этот фундамент, мы сможем добавить скрипты для запуска других программ и конфигурирования сетевого интерфейса и прав пользователей. Знать, как корневая файловая система строится с нуля, полезно, это поможет понять, что происходит в более сложных примерах, рассматриваемых в последующих главах.

Что должно быть в корневой файловой системе?

Ядро получает корневую файловую систему либо в виде ram-диска, указатель на который передает начальный загрузчик, либо путем монтирования блочного

---

[1] Буквально «самокрутка», так говорят о сигаретах. – *Прим. перев.*

устройства, заданного в командной строке ядра с помощью параметра root=. Имея корневую файловую систему, ядро исполняет первую программу, по умолчанию называемую init, как описано в разделе «Подготовка пользовательского пространства» главы 4. После этого задачу ядра можно считать выполненной. Именно init должна запускать скрипты инициализации и другие программы, вызывая для этого системные функции из библиотеки C, которая транслирует их в вызовы ядра.

Для создания полезной системы необходимы как минимум следующие компоненты:

- **init:** программа, которая запускает все остальное, обычно с помощью последовательности скриптов;
- **оболочка:** нужна для получения приглашения к вводу команд и, что еще важнее, для исполнения скриптов, вызываемых init и другими программами;
- **демоны:** различные серверные программы, которые запускает init;
- **библиотеки:** обычно все вышеупомянутые программы компонуются с разделяемыми библиотеками, которые должны находиться в корневой файловой системе;
- **конфигурационные файлы:** текстовые файлы в кодировке ASCII, содержащие конфигурационные параметры init и других демонов, обычно находятся в каталоге /etc;
- **узлы устройств:** специальные файлы, которые обеспечивают доступ к различным драйверам устройств;
- **/proc и /sys:** две псевдофайловые системы, представляющие структуры данных ядра в виде иерархии каталогов и файлов. Их читают многие программы и библиотечные функции;
- **модули ядра:** если на этапе конфигурирования вы указали, что некоторые части ядра должны быть собраны в виде модулей, то соответствующие файлы обычно помещаются в каталог /lib/modules/[версия ядра].

Помимо этого, существуют системные приложения, выполняющие то, для чего предназначено устройство, и пользовательские данные, собираемые этими приложениями.

Попутно отметим, что все вышеперечисленное можно объединить в единственную статически скомпонованную программу, которая выполняется вместо init и больше ничего не запускает. Правда, такая конфигурация встретилась мне всего один раз. Но если бы ваша программа называлась /myprog, то можно было бы включить такой параметр в командную строку ядра:

```
init=/myprog
```

Если же корневая файловая система загружена в виде ram-диска, то надо было включать такой параметр:

```
rdinit=/myprog
```

Недостаток этого подхода в том, что вы лишаете себя возможности воспользоваться многочисленными инструментами, которые обычно включаются в состав встраиваемой системы, и должны все делать сами.

Структура каталогов

Любопытно, что Linux безразлична структура файлов и каталогов, ей нужно только, чтобы существовала программа, заданная в параметре `init=` или `rdinit=`, а в остальном вы вольны помещать файлы куда хотите. Сравните, например, структуры каталогов на устройстве под управлением Android и в дистрибутиве Linux для настольных ПК: они абсолютно различны.

Тем не менее многие программы ожидают найти определенные файлы в определенных местах, и разработчикам удобнее, когда структуры файловых систем на разных устройствах похожи (Android оставляем в стороне). Эта базовая структура системы на основе Linux определена в стандарте иерархии файловой системы (Filesystem Hierarchy Standard – FHS), см. ссылку в конце главы. Стандарт FHS охватывает все реализации операционных систем Linux – от самых больших до самых маленьких. К встраиваемым системам по необходимости относится только подмножество стандарта, но обычно они содержат следующие каталоги:

- `/bin`: программы, необходимые всем пользователям;
- `/dev`: узлы устройств и другие специальные файлы;
- `/etc`: системные конфигурационные файлы;
- `/lib`: необходимые разделяемые библиотеки, в том числе входящие в состав библиотеки C;
- `/proc`: файловая система `proc`;
- `/sbin`: программы, необходимые системному администратору;
- `/sys`: файловая система `sysfs`;
- `/tmp`: место размещения временных и недолговечных файлов;
- `/usr`: как минимум должен содержать подкаталоги `/usr/bin`, `/usr/lib` и `/usr/sbin`, где находятся дополнительные программы, библиотеки и утилиты для системного администрирования;
- `/var`: иерархия файлов и каталогов, которые могут изменяться во время выполнения, например журналы, некоторые из которых должны сохраняться после перезагрузки.

Здесь есть несколько тонких моментов. Разделение на `/bin` и `/sbin` введено просто потому, что обычным пользователям не нужно включать `/sbin` в список путей поиска. Пользователям дистрибутивов, производных от Red Hat, это хорошо известно. Смысл каталога `/usr` в том, чтобы его можно было отделить от корневой файловой системы, поэтому там не должно быть ничего такого, что нужно системе для инициализации. Именно это означает слово «необходимые» выше: файлы, которые должны присутствовать на этапе загрузки и инициализации, и, стало быть, являющиеся частью корневой файловой системы.

На первый взгляд кажется, что четыре каталога для программ – перебор, но можно привести и контраргумент: никакого вреда в этом нет, а польза налицо – возможность разместить каталог `/usr` в другой файловой системе.

Каталог технологической подготовки

Начать следует с создания каталога технологической подготовки (staging directory) на исходном компьютере. В этом каталоге мы будем собирать файлы, кото-

рые в конечном итоге должны быть перемещены в целевую систему. В примерах ниже я использую для этой цели каталог ~/rootfs. В нем нужно создать заготовку структуры каталогов, например:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ mkdir bin dev etc home lib proc sbin sys tmp usr var
$ mkdir usr/bin usr/lib usr/sbin
$ mkdir var/log
```

Чтобы более наглядно представить иерархию, воспользуемся командой tree с флагом -d, означающим, что нужно показывать только каталоги:

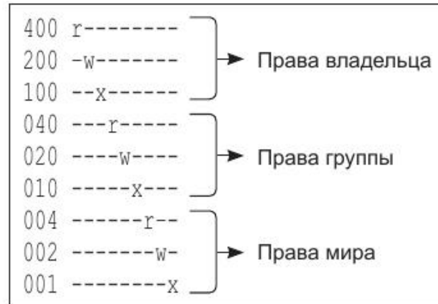
```
$ tree -d
├── bin
├── dev
├── etc
├── home
├── lib
├── proc
├── sbin
├── sys
├── tmp
├── usr
│ ├── bin
│ ├── lib
│ └── sbin
└── var
└── log
```

Права доступа к файлам в стандарте POSIX

Каждый процесс, каковым в контексте этого обсуждения считается любая исполняемая программа, принадлежит некоторому пользователю и одной или нескольким группам. Пользователь представлен 32-разрядным числом, которое называется идентификатором пользователя, или **UID**. Информация о пользователях, в том числе об именах, соответствующих идентификаторам, хранится в файле /etc/passwd. Аналогично группы представлены идентификаторами групп, или **GID**, а информация о них хранится в файле /etc/group. Всегда существуют пользователь root с идентификатором UID = 0 и группа root с идентификатором GID = 0. Пользователь root называется также суперпользователем, потому что в конфигурации по умолчанию для него пропускается большинство проверок прав и ему доступны все ресурсы системы. Когда говорят о безопасности в Linux-системах, имеют в виду прежде всего ограничение доступа пользователю root.

У любого файла и каталога имеется владелец, и он принадлежит ровно одной группе. Уровень доступа к файлу или каталогу со стороны процесса определяется совокупностью флагов разрешений, которая называется режимом файла. Существуют три набора по три бита: первый относится к владельцу файла, второй –

к членам той группы, которой принадлежит файл, третий – ко всем остальным, или к «миру». Биты одного набора интерпретируются как разрешение чтения (r), разрешение записи (w) и разрешение выполнения (x) данного файла. Поскольку три бита составляют одну восьмеричную цифру, то они обычно и представляются в восьмеричном виде, как показано на рисунке ниже:



Существует еще одна группа из трех битов со специальным назначением:

- **SUID (4):** для исполняемых файлов изменяет эффективный UID процесса, делая его равным идентификатору владельца файла;
- **SGID (2):** для исполняемых файлов изменяет эффективный GID процесса, делая его равным идентификатору группы файла;
- **бит закрепления (1):** для каталогов ограничивает возможность удаления, т. е. пользователь не вправе удалять файлы, принадлежащие другому пользователю. Обычно этот бит выставляется для каталогов `/tmp` и `/var/tmp`.

Чаще всего используется бит SUID. Он временно повышает привилегии пользователей, отличных от root, давая им возможность выполнить работу. Примером может служить программа `ping`: она открывает простой сокет, что считается привилегированной операцией. Чтобы обычный пользователь мог выполнить программу `ping`, которой владеет `root`, для нее выставляется бит SUID, так что любой пользователь, запустивший `ping`, выполняет ее от имени пользователя с UID 0, каким бы ни был его собственный UID.

Для задания этих битов команде `chmod` передается комбинация восьмеричных цифр 4, 2, 1. Например, чтобы установить бит SUID для файла `/bin/ping` в каталоге технологической подготовки, можно выполнить такие команды:

```
$ cd ~/rootfs
$ ls -l bin/ping
-rwxr-xr-x 1 root root 35712 Feb  6 09:15 bin/ping
$ sudo chmod 4755 bin/ping
$ ls -l bin/ping
-rwsr-xr-x 1 root root 35712 Feb  6 09:15 bin/ping
```

Обратите внимание на последнюю строчку: в ней видно, что бит SUID поднят.

Права доступа к файлам в каталоге технологической подготовки

В целях безопасности и стабильности чрезвычайно важно обращать внимание на владельцев и права доступа для тех файлов, которые будут скопированы на целевое устройство. Вообще говоря, критичные ресурсы должны быть доступны только пользователю root, а программы должны выполняться преимущественно от имени пользователей, отличных от root, чтобы в случае их компрометации в результате атаки извне противник получил в свое распоряжение как можно меньше системных ресурсов. Например, узел устройства `/dev/mem` дает доступ к памяти системы, что необходимо некоторым программам. Но если он доступен для чтения и записи всем, то ни о какой безопасности говорить не приходится, потому что любой пользователь может получить доступ ко всему. Поэтому `/dev/mem` должен принадлежать пользователю root, группе root и иметь режим 600, запрещающий доступ для чтения и записи всем, кроме владельца и группы.

Но с каталогом технологической подготовки связана одна проблема. Создаваемые в нем файлы принадлежат вам, а после установки на устройство должны принадлежать конкретным пользователям и группам, по большей части пользователю root. Очевидное решение – изменить владельца и группу на этой стадии следующей командой:

```
$ cd ~/rootfs
$ sudo chown -R root:root *
```

Беда в том, что для выполнения этой команды нужны привилегии root, и, начиная с этого момента, вы должны будете выполнять любые модификации файлов в каталоге технологической подготовки от имени root. И, стало быть, всю разработку приходится вести, зайдя в систему как root, а это нехорошо. Мы еще вернемся к этой проблеме позже.

Программы в корневой файловой системе

Настало время приступить к заполнению корневой файловой системы необходимыми программами и поддерживающими их библиотеками, конфигурационными файлами и файлами данных, нужными для работы программ. Начнем с обзора типов программ, которые нам потребуются.

Программа init

В предыдущей главе мы видели, что `init` – первая выполняемая программа, поэтому ее PID (идентификатор процесса) равен 1. Она работает от имени root, поэтому имеет полный доступ к системным ресурсам. Обычно она выполняет скрипты оболочки, которые запускают демонов – так называется программа, которая работает в фоновом режиме и не имеет связи с терминалом. В других контекстах употребляется термин «серверная программа».

Оболочка

Оболочка нужна для запуска скриптов, а кроме того, она дает командную строку, позволяющую нам взаимодействовать с системой. Интерактивная оболочка для готового устройства, пожалуй, не нужна, но на этапах разработки, отладки и сопровождения весьма полезна. Во встраиваемых системах можно встретить разные оболочки.

- `bash`: эту штуку, повсеместно используемую в Linux для настольных ПК, мы все знаем и любим. Она включает в себя оболочку Борна для Unix и дополнена многими расширениями, которые называют башизмами (bashism).
- `ash`: тоже основана на оболочке Борна и имеет долгую историю в вариантах Unix от BSD. В комплект утилит Busybox входит версия `ash`, расширенная в целях большей совместимости с `bash`. Она гораздо меньше `bash` и потому популярна во встраиваемых системах.
- `hush`: очень компактная оболочка, с которой мы встречались в главе о начальных загрузчиках. Полезна для устройств с минимальным объемом памяти. Присутствует в комплекте утилит BusyBox.

При использовании в целевой системе оболочки `ash` или `hush` не забудьте протестировать все скрипты. Ограничившись тестированием в исходной системе в оболочке `bash`, можно нарваться на сюрприз, когда окажется, что после копирования в целевую систему ничего не работает.

Утилиты

Оболочка – не более чем способ запуска других программ, а скрипт оболочки – всего лишь список запускаемых программ, дополненный средствами управления потоком выполнения и средствами передачи информации между программами. Чтобы оболочка была полезной, нам нужны программы-утилиты, от наличия которых зависит режим командной строки в Unix. Даже в простейшей корневой файловой системе имеется приблизительно 50 утилит, в связи с чем возникают две проблемы. Во-первых, получать их исходный код и кросс-компилировать его – большая работа. Во-вторых, результирующий набор программ занимает десятки мегабайтов, что никуда не годилось на заре развития встраиваемых Linux-систем, когда можно было рассчитывать только на несколько мегабайтов. Для решения этих проблем был создан BusyBox.

BusyBox спешит на помощь!

Происхождение BusyBox не имеет ничего общего со встраиванием Linux. В 1996 году проект инициировал Брюс Перенс (Bruce Perens), которого интересовала возможность загрузить дистрибутив Debian с дискеты емкостью 1,44 МБ. Так совпало, что именно таков был объем памяти в тогдашних устройствах, поэтому сообщество, занимающееся встраиваемыми Linux-системами, ухватилось за идею. С тех пор комплект утилит BusyBox лежит в основе подобных систем.

BusyBox был написан с чистого листа для выполнения абсолютно необходимых функций тех утилит Linux, без которых нельзя обойтись. Разработчики придерживались правила 80:20: наиболее полезные 80 % функциональности программы реализованы в 20% ее кода. Поэтому утилиты, вошедшие в BusyBox, реализуют лишь подмножество функциональности аналогов из системы для ПК, однако в большинстве случаев его достаточно.

Еще один трюк, примененный в BusyBox, – объединение всех утилит в один двоичный файл, что упрощает совместное использование кода. Работает это так: BusyBox представляет собой набор аплетов, каждый из которых экспортирует свою функцию main под именем `[аплет]_main`. Например, код команды `cat` находится в файле `coreutils/cat.c` и экспортирует функцию `cat_main`. Функция main самого комплекта BusyBox осуществляет диспетчеризацию, направляя обращение к нужному аплету в зависимости от аргументов в командной строке.

Итак, чтобы прочитать файл, мы запускаем `busybox`, указав имя соответствующего аплета и аргументы для него:

```
$ busybox cat my_file.txt
```

Чтобы получить список всех включенных в состав программы аплетов, нужно вызвать команду `busybox` без аргументов.

Работать с BusyBox таким образом неудобно. Для выполнения аплета `cat` лучше создать символическую ссылку `/bin/cat` на `/bin/busybox`:

```
$ ls -l bin/cat bin/busybox
-rwxr-xr-x 1 chris chris 892868 Feb 2 11:01 bin/busybox
lrwxrwxrwx 1 chris chris      7 Feb 2 11:01 bin/cat -> busybox
```

Теперь, когда мы вводим в командной строке `cat`, на самом деле запускается программа `busybox`, которой нужно лишь проверить, что передано в `argv[0]`. Увидев, что это `/bin/cat`, программа выделит имя приложения `cat` и по своей внутренней таблице найдет, что этой строке соответствует функция `cat_main`. Эта логика реализована в следующем (слегка упрощенном) фрагменте кода из файла `libbb/appletlib.c`:

```
applet_name = argv[0];
applet_name = bb_basename(applet_name);
run_applet_and_exit(applet_name, argv);
```

В BusyBox включено свыше трехсот аплетов, включая программу `init`, несколько оболочек разного уровня сложности и утилиты для большинства административных задач. Имеется даже упрощенная версия редактора `vi`, чтобы можно было изменять текстовые файлы прямо на устройстве.

Короче говоря, типичный дистрибутив BusyBox состоит из одной программы и символических ссылок на все содержащиеся в ней аплеты, но эта программа ведет себя так, как будто имеется набор отдельных приложений.

Сборка BusyBox

Для сборки BusyBox используется та же система `Kconfig` и `Kbuild`, что и для ядра, так что компиляция не вызывает затруднений. Для получения исходного кода

можно клонировать репозиторий git и выгрузить желаемую версию (на момент написания книги последней была 1_24_1):

```
$ git clone git://busybox.net/busybox.git
$ cd busybox
$ git checkout 1_24_1
```

Можно вместо этого загрузить архивный файл со страницы http://busybox.net/downloads. Затем сконфигурируйте BusyBox, начав с конфигурации по умолчанию, которая включает практически все, что есть в BusyBox:

```
$ make distclean
$ make defconfig
```

Теперь вы, наверное, захотите выполнить команду `make menuconfig` и точно настроить конфигурацию. Почти наверняка вы решите с помощью параметра **Busybox Settings | Installation Options** (`CONFIG_PREFIX`) задать путь к каталогу установки, направив его на каталог технологической подготовки. Затем можно запустить стандартную кросс-компиляцию:

```
$ make -j 4 ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf-
```

В результате будет создан исполняемый файл `busybox`. В случае цели `defconfig` для архитектуры ARM v7a его размер равен приблизительно 900 КиБ. Если для вас это слишком много, то можно на этапе конфигурирования вычеркнуть ненужные утилиты.

Для установки BusyBox выполните следующую команду:

```
$ make install
```

В результате двоичный файл будет скопирован в каталог, заданный параметром `CONFIG_PREFIX`, и созданы все символические ссылки на него.

ToyBox – альтернатива BusyBox

BusyBox – не единственная возможность. Например, для Android имеется эквивалентный комплект под названием Toolbox, но он оптимизирован под потребности Android и не годится в качестве универсального встраиваемого окружения. Полезнее проект ToyBox, который создал и сопровождает Роб Лэндли (Rob Landley), ранее сопровождавший BusyBox. У ToyBox та же цель, что у BusyBox, но больше внимания уделено соответствию стандартам, особенно POSIX-2008 и LSB 4.1, а меньше – совместимости с расширениями этих стандартов, реализованными в проекте GNU. ToyBox меньше BusyBox, отчасти потому, что реализовано меньше аплетов.

Однако основное различие заключается в лицензии: BSD вместо GPL v2, что делает ToyBox совместимым по политике лицензирования с операционными системами, в которых пользовательские приложения распространяются по лицензии BSD, в частности с Android.

Библиотеки для корневой файловой системы

Программы компонуются с библиотеками. Компоновка может быть статической, и тогда на целевом устройстве никаких библиотек не будет. Однако если программ больше двух-трех, то при этом без нужды расходуется место на запоминающем устройстве. Поэтому лучше скопировать разделяемые библиотеки из набора инструментов в каталог технологической подготовки. Но как узнать, какие именно библиотеки?

Один из вариантов – скопировать все, потому что для чего-то же они нужны, иначе зачем бы их включали! Это, конечно, логично, и если вы создаете платформу для работы различных приложений, то такой подход правилен. Однако имейте в виду, что полная библиотека `glibc` очень велика. Когда для сборки glibc 2.19 используется CrossTool-NG, файлы, созданные в каталогах `/lib` и `/usr/lib`, занимают 33 МиБ. Конечно, можно значительно сэкономить, воспользовавшись библиотеками uClibc или Musel `libc`.

Другой вариант – тщательно отобрать только нужные библиотеки, для чего понадобится средство определения зависимостей между библиотеками. Вспомнив, что было написано в главе 2, мы можем воспользоваться для этого программой `readelf`:

```
$ cd ~/rootfs
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "program interpreter"
[Requesting program interpreter: /lib/ld-linux-armhf.so.3]
$ arm-cortex_a8-linux-gnueabihf-readelf -a bin/busybox | grep "Shared library"
0x00000001 (NEEDED)                Shared library: [libm.so.6]
0x00000001 (NEEDED)                Shared library: [libc.so.6]
```

Теперь нужно найти эти файлы в наборе инструментов и скопировать их в каталог технологической подготовки. Напомним, как определить путь к `sysroot`:

```
$ arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot
/home/chris/x-tools/arm-cortex_a8-linux-gnueabihf/arm-cortex_a8- linux-gnueabihf/sysroot
```

Чтобы меньше набирать, я сохраню этот путь в переменной оболочки:

```
$ export SYSROOT=`arm-cortex_a8-linux-gnueabihf-gcc -print-sysroot`
```

Взглянув на файл `/lib/ld-linux-armhf.so.3` в `sysroot`, мы обнаружим, что это символическая ссылка:

```
$ ls -l $SYSROOT/lib/ld-linux-armhf.so.3
[...]/sysroot/lib/ld-linux-armhf.so.3 -> ld-2.19.so
```

Проделав то же упражнение для файлов `libc.so.6` и `libm.so.6`, мы получим список из трех файлов и трех символических ссылок. Скопируйте их командой `cp a`, которая сохраняет символические ссылки:

```
$ cd ~/rootfs
$ cp -a $SYSROOT/lib/ld-linux-armhf.so.3 lib
$ cp -a $SYSROOT/lib/ld-2.19.so lib
```

```
$ cp -a $SYSROOT/lib/libc.so.6 lib
$ cp -a $SYSROOT/lib/libc-2.19.so lib
$ cp -a $SYSROOT/lib/libm.so.6 lib
$ cp -a $SYSROOT/lib/libm-2.19.so lib
```

Повторите эту процедуру для каждой программы.

Это стоит делать только для того, чтобы получить минимально возможную встраиваемую систему. Существует опасность пропустить библиотеки, загружаемые с помощью вызовов функции `dlopen(3)`, – в основном это подключаемые модули. Пример – библиотеку NSS – мы рассмотрим, когда дойдем до конфигурирования сетевых интерфейсов ниже в этой главе.

Уменьшение размера путем удаления таблицы символов

При компиляции библиотек и программ в двоичный файл часто включается таблица символов, особенно если файл компилируется с флагом отладки `-g`. В целевой системе эта таблица редко бывает нужна. Поэтому для экономии место можно ее просто удалить. Ниже показан размер библиотеки `libc` до и после применения к ней команды `strip`:

```
$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamical-
ly linked (uses shared libs), for GNU/Linux 3.15.4, not stripped

$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1547371 Feb  5 10:18 rootfs/lib/libc-2.19.so

$ arm-cortex_a8-linux-gnueabi-strip rootfs/lib/libc-2.19.so

$ file rootfs/lib/libc-2.19.so
rootfs/lib/libc-2.19.so: ELF 32-bit LSB shared object, ARM, version 1 (SYSV), dynamical-
ly linked (uses shared libs), for GNU/Linux 3.15.4, stripped

$ ls -l rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 chris chris 1226024 Feb  5 10:19 rootfs/lib/libc-2.19.so

$ ls -og rootfs/lib/libc-2.19.so
-rwxrwxr-x 1 1226024 Feb  5 10:19 rootfs/lib/libc-2.19.so
```

В данном случае мы сэкономили 321 347 байтов, примерно 20%.

Чтобы проделать ту же операцию для модулей ядра, воспользуйтесь командой

```
strip --strip-unneeded <имя модуля>
```

Иначе будут удалены символы, необходимые для перемещения кода модуля в памяти, и он не загрузится.

Узлы устройств

Большинство устройств в Linux представлено узлами устройств в соответствии с философией Unix, согласно которой все является файлом (кроме сетевых интерфейсов, которые являются сокетами). Узел устройства может ссылаться на блоч-

ное или символьное устройство. Блочными являются массовые запоминающие устройства, например карты SD и жесткие диски. Символьным устройством может быть все, что угодно, опять-таки за исключением сетевых интерфейсов. Традиционно узлы устройств находятся в каталоге /dev. Например, последовательный порт может быть представлен узлом /dev/ttyS0.

Узлы устройств создаются программой mknod (сокращение от make node):

```
mknod <name> <type> <major> <minor>
```

Здесь name – имя создаваемого узла устройства, type принимает значение c (символьное устройство) или b (блочное устройство). Каждый узел имеет старший и младший номера, которые ядро использует для маршрутизации запросов к файлам соответствующему драйверу устройства. В исходном коде ядра список стандартных старших и младших номеров находится в файле Documentation/devices.txt.

Вы должны будете создать узлы всех устройств, к которым собираетесь обращаться. Это можно сделать вручную командой mknod, как будет показано ниже, или автоматически во время выполнения, воспользовавшись каким-нибудь менеджером устройств, о которых я еще скажу.

Для загрузки с помощью BusyBox нужны всего два узла: console и null. К консоли будет обращаться только root, владелец узла устройства, поэтому нужно установить права доступа 600. Устройство null должно быть доступно для чтения и записи всем, поэтому права доступа к нему 666. При создании узла командой mknod флаг -m позволяет сразу задать режим. Создавать узлы устройства может только пользователь root.

```
$ cd ~/rootfs
$ sudo mknod -m 666 dev/null c 1 3
$ sudo mknod -m 600 dev/console c 5 1
$ ls -l dev
total 0
crw------- 1 root root 5, 1 Oct 28 11:37 console
crw-rw-rw- 1 root root 1, 3 Oct 28 11:37 null
```

Удалить узел можно командой rm; специальная команда rmnod не нужна, потому что, будучи создан, узел устройства является файлом.

Файловые системы proc и sysfs

proc и sysfs – две псевдофайловые системы, открывающие окно для наблюдения за внутренней жизнью ядра. Та и другая представляют данные ядра в виде файлов, организованных в виде иерархии каталогов. При чтении таких файлов данные поступают не с диска, а форматируются на лету функцией, находящейся в ядре. Некоторые файлы допускают запись, в результате чего вызывается некоторая функция ядра, которой передаются записанные данные, и если формат данных правилен, а у пользователя достаточно прав, то будет модифицировано значение

в памяти ядра. Иными словами, proc и sysfs дают еще один способ взаимодействия с драйверами устройств и прочим кодом ядра.

Файловые системы proc и sysfs следует монтировать на каталоги /proc и /sys:

```
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

Хотя концептуально эти файловые системы очень похожи, функциональность их различна. proc была частью Linux с первых дней. Изначально она использовалась для раскрытия информации о процессах пользовательским приложениям, отсюда и название. Для этого каждому процессу соответствует каталог с именем /proc/<PID>, содержащий сведения о его состоянии. Команда вывода списка процессов ps читает эти файлы и таким образом формирует результат. Кроме того, имеются файлы, несущие информацию о других частях ядра, например в /proc/cpuinfo хранятся сведения о процессоре, в /proc/interrupts – о прерываниях и т. д. Наконец, в каталоге /proc/sys находятся файлы, позволяющие просматривать и управлять состоянием и поведением подсистем ядра, в особенности планирования, управления памятью и сетевой. Самую полную информацию об этих файлах можно найти на странице руководства proc(5).

С течением времени количество файлов в proc выросло, а их формат стал настолько разнообразен, что воцарился хаос. В Linux 2.6 была введена файловая система sysfs, в которую было перенесено подмножество данных.

В отличие от proc, система sysfs экспортирует весьма упорядоченную иерархию файлов, относящихся к устройствам и их связям между собой.

Монтирование файловых систем

Команда mount позволяет присоединить одну файловую систему к каталогу внутри другой, образовав тем самым иерархию файловых систем. Файловая система, находящаяся на верхнем уровне, монтируется ядром на этапе загрузки и называется корневой. Команда mount имеет такой порядок вызова:

```
mount [-t vfstype] [-o options] устройство каталог
```

Необходимо задать тип файловой системы vfstype, узел блочного устройства, на котором она располагается, и каталог, на который монтируется. После флага -o можно задавать различные параметры, о которых написано в руководстве. Например, чтобы смонтировать на каталог /mnt карту SD, содержащую в первом разделе файловую систему типа ext4, нужно выполнить такую команду:

```
mount -t ext4 /dev/mmcblk0p1 /mnt
```

Если монтирование завершилось успешно, то мы сможем увидеть в каталоге /mnt файлы, хранящиеся на карте SD. Иногда тип файловой системы можно опускать, позволив ядру самостоятельно определить, что находится на устройстве.

Взглянув на пример монтирования файловой системы proc, мы обнаружим одну странность: узел устройства /dev/proc не указывается, потому что это не настоящая, а псевдофайловая система. Однако же mount требует задания устройства.

Следовательно, мы должны подставить какую-нибудь строку вместо параметра, задающего устройство, но какую именно, совершенно не важно. Обе показанные ниже команды дают один и тот же результат:

```
mount -t proc proc /proc
mount -t proc nodevice /proc
```

Довольно часто при монтировании псевдофайловой системы указывают ее тип вместо устройства.

Модули ядра

Если имеются модули ядра, то их нужно установить в корневую файловую систему, воспользовавшись целью `make modules_install`, как было показано в предыдущей главе. В результате модули будут скопированы в каталог `/lib/modules/<версия ядра>` вместе с конфигурационными файлами, необходимыми команде `modprobe`.

Не забывайте, что тем самым вы создали зависимость между ядром и корневой файловой системой. Обновив одно, нужно будет обновить и другое.

Перенос корневой файловой системы на целевое устройство

Создав заготовку корневой файловой системы в каталоге технологической подготовки, мы можем перейти к следующему шагу – переносу ее на целевое устройство. Я опишу три возможности.

- **Ram-диск:** образ файловой системы, который загружается в ОЗУ начальным загрузчиком. Ram-диски легко создать, и при этом отсутствует зависимость от драйверов массовых запоминающих устройств. Их можно использовать в режиме обслуживания, когда требуется обновить главную корневую файловую систему. Они даже могут играть роль главной корневой файловой системы в небольших встраиваемых устройствах, и, конечно же, они используются для подготовки пользовательского пространства в популярных дистрибутивах Linux. Сжатый ram-диск занимает минимум места во внешней памяти, но все же потребляет ОЗУ. Его содержимое энергозависимо, поэтому для хранения постоянных данных, например конфигурационных параметров, понадобится запоминающее устройство другого типа.
- **Образ диска:** копия корневой файловой системы, отформатированная и готовая к загрузке на массовое запоминающее устройство в целевой системе. Например, это может быть образ в формате `ext4`, готовый к загрузке во флэш-память с помощью начального загрузчика. Создание образа диска – пожалуй, самый распространенный вариант. Дополнительные сведения о различных типах массовых запоминающих устройств приведены в главе 7.

- **Сетевая файловая система:** каталог технологической подготовки можно экспортировать в сеть с помощью NFS-сервера и смонтировать на целевом устройстве во время загрузки. Часто так поступают на этапе разработки, чтобы избежать многократного создания образа диска и загрузки его в массовую память, поскольку это весьма длительный процесс.

Я начну с ram-диска, чтобы проиллюстрировать некоторые дополнения к корневой файловой системе, в частности добавление имен пользователей и использование менеджера устройств для автоматического создания узлов устройств. Затем я покажу, как создать образ диска и, наконец, как воспользоваться NFS-сервером для монтирования корневой файловой системы по сети.

Создание загрузочного ram-диска

Загрузочный ram-диск Linux, а точнее, начальная файловая система в ОЗУ, или **initramfs**, – это сжатый архив в формате `cpio`. `cpio` – старый архивный формат Unix, аналогичный TAR и ZIP, но более простой для декодирования и потому требующий меньше кода в ядре. Для поддержки `initramfs` необходимо при конфигурировании ядра задать параметр `CONFIG_BLK_DEV_INITRD`.

Существуют три способа создать загрузочный ram-диск: в виде автономного архива `cpio`, в виде архива `cpio`, включенного в образ ядра, и в виде таблицы устройств, создаваемой в процессе сборки ядра. Первый способ самый гибкий, потому что позволяет как угодно сочетать ядро с ram-диском. Однако в этом случае нам придется иметь дело с двумя файлами вместо одного, а не все начальные загрузчики умеют загружать отдельный ram-диск. Как встроить ram-диск в ядро, я покажу позже.

Автономный ram-диск

Показанная ниже последовательность команд создает архив, сжимает его и добавляет заголовок U-Boot, после чего файл готов к загрузке в целевую систему:

```
$ cd ~/rootfs
$ find . | cpio -H newc -ov --owner root:root > ../initramfs.cpio
$ cd ..
$ gzip initramfs.cpio
$ mkimage -A arm -O linux -T ramdisk -d initramfs.cpio.gz uRamdisk
```

Обратите внимание на флаг `--owner root:root` при запуске `cpio`. Это простое решение вышеупомянутой проблемы владения файлами – все файлы, находящиеся в архиве `cpio`, после распаковки будут иметь нулевые UID и GID.

Размер файла `uRamdisk` без модулей ядра приблизительно равен 2,9 МиБ. Добавим к этому файл образа ядра `zImage` размером 4,4 МиБ и сам загрузчик U-Boot размером 440 КиБ – в результате получится 7,7 МиБ, именно столько памяти необходимо для загрузки платы. Это больше, чем размер дискеты, 1,44 МиБ, с которого все начиналось. Если бы размер действительно составлял проблему, то можно было бы пойти по одному из следующих путей:

- уменьшить ядро, исключив ненужные драйверы и функции;
- уменьшить BusyBox, исключив ненужные утилиты;
- использовать библиотеку uClibc или musl libc вместо glibc;
- откомпилировать BusyBox статически.

Загрузка ram-диска

Самое простое, что можно сделать, – запустить на консоли оболочку, получив возможность взаимодействовать с устройством. Для этого добавим параметр `rdinit=/bin/sh` в командную строку ядра. Теперь можно загружать устройство.

Загрузка в QEMU

У QEMU имеется флаг `-initrd`, который позволяет загрузить файл `initframfs` в память, так что полная команда имеет вид:

```
$ cd ~/rootfs
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage -ap-
pend "console=ttyAMA0 rdinit=/bin/sh" -dtb vexpress-v2p-ca9.dtb -initrd initramfs.cpio.gz
```

Загрузка платы BeagleBone Black

Для загрузки BeagleBone Black получите приглашение U-Boot и введите следующие команды:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
fatload mmc 0:1 0x81000000 uRamdisk
setenv bootargs console=ttyO0,115200 rdinit=/bin/sh
bootz 0x80200000 0x81000000 0x80f00000
```

Если не возникнет ошибок, то на консоли появится приглашение оболочки в режиме root.

Монтирование proc

Отметим, что команда `ps` не работает, потому что файловая система `proc` еще не смонтирована. Смонтируйте ее и попробуйте снова выполнить `ps`.

Описанную схему можно улучшить, написав скрипт оболочки, который содержит все, что нужно сделать на этапе начальной загрузки, и передать этот скрипт в параметре `rdinit=`. Скрипт должен выглядеть примерно так:

```
#!/bin/sh
/bin/mount -t proc proc /proc
/bin/sh
```

Такое использование оболочки в роли `init` очень удобно, когда нужно что-то быстро поправить, например спасти систему с поврежденной программой `init`. Но в большинстве случаев следует использовать саму программу `init`, о чем будет рассказано ниже.

Встраивание ram-диска в формате cpio в образ ядра

В некоторых случаях предпочтительнее встроить ram-диск в образ ядра, например если начальный загрузчик не умеет работать с файлом ram-диска. Для этого нужно на этапе конфигурирования ядра присвоить параметру CONFIG_INITRAMFS_SOURCE полный путь к созданному ранее архиву cpio. В программе menuconfig он находится на странице **General setup → Initramfs source file(s)**. Отметим, что необходимо указать несжатый архив cpio с расширением .cpio. Затем соберите ядро. Оно окажется больше, чем раньше.

Порядок загрузки такой же, с тем отличием, что файла ram-диска больше нет. Для QEMU команда выглядит так:

```
$ cd ~/rootfs
$ QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel zImage - ap-
pend "console=ttyAMA0 rdinit=/bin/sh" -dtb vexpress-v2p-ca9.dtb
```

Для платы BeagleBone Black введите следующие команды в ответ на приглашение U-Boot:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
setenv bootargs console=ttyO0,115200 rdinit=/bin/sh
bootz 0x80200000 - 0x80f00000
```

Разумеется, нужно не забывать пересобирать ядро всякий раз, как изменяется содержимое ram-диска и файл cpio генерируется заново.

Другой способ построить ядро с ram-диском

Интересный способ встроить ram-диск в образ ядра заключается в использовании таблицы устройств для генерации архива cpio. Таблица устройств – это текстовый файл, в котором перечислены файлы, каталоги, узлы устройств и ссылки, которые должны быть включены в архив. Чрезвычайно привлекательное достоинство этого способа – тот факт, что вы можете поместить в файл cpio объекты, принадлежащие пользователю root или любому другому, не имея привилегий root. Можно даже создавать узлы устройств. И все потому, что архив – это всего лишь файл данных. Его интерпретирует Linux на этапе загрузки, и именно тогда создаются реальные файлы и каталоги с теми атрибутами, которые вы задали. Ниже приведена таблица устройств для нашей простой корневой файловой системы rootfs, но большинство символических ссылок на busybox опущено, иначе таблица заняла бы слишком много места:

```
dir /proc 0755 0 0
dir /sys 0755 0 0
dir /dev 0755 0 0
nod /dev/console 0600 0 0 c 5 1
nod /dev/null 0666 0 0 c 1 3
nod /dev/ttyO0 0600 0 0 c 252 0
```

```
dir /bin 0755 0 0
file /bin/busybox /home/chris/rootfs/bin/busybox 0755 0 0
slink /bin/sh /bin/busybox 0777 0 0
dir /lib 0755 0 0
file /lib/ld-2.19.so /home/chris/rootfs/lib/ld-2.19.so 0755 0 0
slink /lib/ld-linux.so.3 /lib/ld-2.19.so 0777 0 0
file /lib/libc-2.19.so /home/chris/rootfs/lib/libc-2.19.so 0755 0 0
slink /lib/libc.so.6 /lib/libc-2.19.so 0777 0 0
file /lib/libm-2.19.so /home/chris/rootfs/lib/libm-2.19.so 0755 0 0
slink /lib/libm.so.6 /lib/libm-2.19.so 0777 0 0
```

Синтаксис понятен:

- dir <name> <mode> <uid> <gid>
- file <name> <location> <mode> <uid> <gid>
- nod <name> <mode> <uid> <gid> <dev_type> <maj> <min>
- slink <name> <target> <mode> <uid> <gid>

Ядро предоставляет средство для чтения этого файла и создания архива cpio. Его исходный код находится в файле usr/gen_init_cpio.c. Файл scripts/gen_initramfs_list.sh содержит удобный скрипт, который создает таблицу устройств по содержимому указанного каталога, что избавляет от необходимости набивать ее вручную.

И напоследок нужно задать конфигурационный параметр CONFIG_INITRAMFS_SOURCE, указывающий на файл с таблицей устройств, а затем собрать ядро. Все остальное – как и раньше.

Старый формат initrd

Существует также старый формат ram-диска для Linux – initrd. До версии Linux 2.6 он был единственным и до сих пор необходим, если используется вариант Linux без подсистемы управления памятью – uCLinux. Формат довольно запутанный, и здесь я его рассматривать не буду. Более полные сведения имеются в файле Documentation/initrd.txt в исходном тексте ядра.

Программа init

Запуск оболочки или даже скрипта оболочки на этапе загрузки – вещь хорошая в простых случаях, но вообще-то нужно что-то, более гибкое. Обычно Unix-системы выполняют программу init, а та уже запускает другие программы и наблюдает за ними. За прошедшие годы было написано много программ init, и некоторые из них описаны в главе 9. А сейчас я кратко опишу init из комплекта BusyBox.

Работа init начинается с чтения конфигурационного файла /etc/inittab. Вот простой пример, которого хватит для наших целей:

```
::sysinit:/etc/init.d/rcS
::askfirst:-/bin/ash
```

В первой строке запускается скрипт оболочки rcS. Во второй строке печатается сообщение **Please press Enter to activate this console** (Нажмите Enter для акти-

вации этой консоли), и после нажатия **Enter** запускается оболочка. Знак - перед /bin/ash означает, что это начальная оболочка, которая читает файлы /etc/profile и $HOME/.profile, перед тем как напечатать приглашение. Одно из преимуществ запуска оболочки таким образом заключается в том, что активируется механизм управления задачами. Самый заметный эффект – возможность прервать текущую программу нажатием **Ctrl+C**.

Программа init, входящая в BusyBox, включает таблицу inittab по умолчанию на случай, если ее не окажется в корневой файловой системе. Она чуть побольше, чем показано выше.

В скрипт /etc/init.d/rcS помещаются команды инициализации, которые нужно выполнить на этапе загрузки, например монтирование файловых систем proc и sysfs:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
```

Не забудьте сделать файл rcS исполняемым:

```
$ cd ~/rootfs
$ chmod +x etc/init.d/rcS
```

Можете протестировать его в QEMU, изменив параметр -append:

```
-append "console=ttyAMA0 rdinit=/sbin/init"
```

Для достижения того же результата в случае платы BeagelBone Black нужно будет изменить переменную bootargs в U-Boot:

```
setenv bootargs console=ttyO0,115200 rdinit=/sbin/init
```

Конфигурирование учетных записей пользователей

Как я уже говорил, не стоит выполнять все программы от имени пользователя root, потому что если какая-то программа будет скомпрометирована в результате атаки извне, то вся система окажется под угрозой, а некорректная программа может нанести больше вреда, если работает с привилегиями суперпользователя. Поэтому предпочтительно создавать непривилегированных пользователей и работать от их имени, если полный доступ не нужен.

Имена пользователей хранятся в файле /etc/passwd. Для каждого пользователя в нем есть одна строка, содержащая семь полей, разделенных двоеточиями:

- имя для входа в систему;
- свертка пароля или – гораздо чаще – буква x, означающая, что пароль хранится в файле /etc/shadow;
- UID;
- GID;

- поле комментария, обычно пустое;
- домашний каталог пользователя;
- (факультативно) оболочка, в которую попадает пользователь после входа.

Например, в следующих строках описаны два пользователя: root с UID 0 и daemon с UID 1:

```
root:x:0:0:root:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/false
```

Поскольку для пользователя daemon в качестве оболочки указана программа /bin/false, то любая попытка войти в систему от его имени закончится неудачно.

Различные программы читают /etc/passwd, когда требуется искать пользователя по UID или по имени, поэтому он должен быть доступен всем для чтения. Если свертки паролей хранятся в этом файле, то возникает проблема, потому что вредоносная программа сможет скопировать его и определить сами пароли, применяя различные инструменты взлома. Поэтому, чтобы не раскрывать эту секретную информацию, свертки паролей хранятся в файле /etc/shadow, а в файл /etc/passwd вместо них помещается символ x, показывающий, что дело обстоит именно так. Файл /etc/shadow может читать только root, поэтому, коль скоро работа от имени root ограничена, пароли в безопасности.

В теневом файле паролей каждому пользователю соответствует строка, содержащая девять полей. Вот как выглядит этот файл для файла passwd, показанного выше:

```
root::10933:0:99999:7:::
daemon:*:10933:0:99999:7:::
```

В первых двух полях находятся имя пользователя и свертка пароля. Остальные семь относятся к политике устаревания пароля, для встраиваемых устройств они обычно не применяются. Полная информация имеется на странице руководства *shadow(5)*.

В нашем примере пароль пользователя root пустой, т. е. он может входить в систему без пароля. В среде разработки это еще куда ни шло, но в производственной системе такого быть не должно! Свертку пароля можно получить от команды mkpasswd или выполнив команду passwd в целевой системе и скопировав поле свертки из файла /etc/shadow в целевой системе. В любом случае свертку нужно записать в теневой файл паролей в каталоге технологической подготовки.

Для пользователя daemon задан пароль *, не совпадающий ни с каким паролем, введенным при входе. Это еще одна гарантия того, что учетная запись daemon не будет использоваться для нормального входа пользователя.

Имена групп хранятся в файле /etc/group, имеющем следующий формат:

- имя группы;
- пароль группы, обычно символ x, означающий, что у группы нет пароля;
- GID;
- факультативный список пользователей, принадлежащих группе, через запятую.

Приведем пример:

```
root:x:0:
daemon:x:1:
```

Добавление учетных записей пользователей в корневую файловую систему

Прежде всего нужно добавить в каталог технологической подготовки файлы etc/passwd, etc/shadow и etc/group, описанные в предыдущем разделе. Не забудьте задать для файла shadow права 0600.

Процедура входа запускается программой getty, включенной в BusyBox. Она прописывается в файле inittab с ключевым словом respawn, означающим, что getty следует перезапустить после завершения оболочки. Таким образом, inittab должен выглядеть следующим образом:

```
::sysinit:/etc/init.d/rcS
::respawn:/sbin/getty 115200 console
```

После этого перестройте ram-диск и протестируйте его с помощью QEMU или BeagleBone Black, как и раньше.

Запуск процесса-демона

Обычно на этапе инициализации системы запускаются определенные фоновые процессы. В качестве примера рассмотрим демон протоколирования syslogd. Его назначение – собирать сообщения, которые разные программы, чаще всего другие демоны, записывают в журнал. Разумеется, в BusyBox для него есть аплет!

Для запуска демона нужно всего лишь поместить в etc/inittab строку такого вида:

```
::respawn:syslogd -n
```

Слово respawn означает, что в случае завершения программа автоматически перезапускается, а флаг -n – что программа должна работать в приоритетном процессе. Журнал записывается в файл /var/log/messages.

Вероятно, вы захотите запустить также демон klogd, который отправляет сообщения ядра syslogd, чтобы они были сохранены и не пропали.
Попутно отмечу, что в типичной встраиваемой Linux-системе записывать файлы журналов во флэш-память не стоит, так как это приводит к ее быстрому износу. Варианты протоколирования будут рассмотрены в главе 7.

Улучшенный способ управления узлами устройств

Создавать узлы устройств статически с помощью программы mknod трудно, и этому способу не хватает гибкости. Существуют другие способы создания узлов – автоматически по запросу:

- devtmpfs: это псевдофайловая система, монтируемая на каталог /dev на этапе загрузки. Ядро помещает в нее узлы для всех устройств, о которых знает в данный момент, и создает новые по мере обнаружения узлов во время работы. Все узлы принадлежат пользователю root и по умолчанию имеют права 0600. Для некоторых стандартных узлов, например /dev/null и /dev/random, устанавливается режим 0666 (см. структуру struct memdev в файле drivers/char/mem.c).
- mdev: это аплет BusyBox, который помещает узлы устройств в определенный каталог и создает новые узлы по мере необходимости. В конфигурационном файле /etc/mdev.conf записаны правила задания владельца и режима узлов.
- udev: теперь это часть подсистемы systemd, применяемой в Linux для ПК и в некоторых встраиваемых устройствах. Решение очень гибкое и хорошо подходит для более сложных и дорогих устройств.

Хотя mdev и udev создают узлы устройств самостоятельно, чаще эту работу поручают devtmpfs, а mdev/udev используют как надстройку, реализующую политику задания владельцев и прав доступа.

Пример использования devtmpfs

Если вы загружались с одного из созданных ранее ram-дисков, то для тестирования devtmpfs достаточно ввести такую команду:

```
# mount -t devtmpfs devtmpfs /dev
```

В каталоге /dev должно появиться множество узлов устройств. Чтобы сделать это изменение постоянным, добавьте следующие строки в /etc/init.d/rcS:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev
```

На самом деле при инициализации ядра это делается автоматически, если только не предоставлен ram-диск initramfs, что мы как раз и сделали! Код находится в функции prepare_namespace() в файле init/do_mounts.c.

Пример использования mdev

Настроить mdev чуть сложнее, зато открывается возможность модифицировать права доступа к узлам по мере их создания. Прежде всего существует фаза инициализации, задаваемая флагом -s, во время которой mdev просматривает каталог /sys в поисках информации о текущих устройствах и помещает соответствующие им узлы в каталог /dev.

Чтобы оперативно отслеживать вновь появляющиеся устройства и создавать для них узлы, необходимо сделать программу mdev клиентом подсистемы горячего подключения, добавив ее в файл /proc/sys/kernel/hotplug. Ниже показано, как при этом должен выглядеть файл /etc/init.d/rcS:

```
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devtmpfs devtmpfs /dev
echo /sbin/mdev > /proc/sys/kernel/hotplug
mdev -s
```

По умолчанию устанавливаются режим 660 и владелец `root:root`. Чтобы изменить это, нужно добавить правила в файл `/etc/mdev.conf`. Например, чтобы установить правильный режим для устройств `null`, `random` и `urandom`, добавьте такие строки:

```
null    root:root 666
random  root:root 444
urandom root:root 444
```

Формат файла документирован в исходном коде BusyBox (файл `docs/mdev.txt`), а в каталоге `examples` есть дополнительные примеры.

А так ли плохи статические узлы устройств?

У статически созданных узлов устройств все же есть одно преимущество: в отличие от других методов, на их создание не тратится время в период загрузки. Если требуется свести время загрузки к минимуму, то экономия за счет статического создания узлов будет заметна.

Конфигурирование сети

Далее мы рассмотрим некоторые базовые конфигурации сети, позволяющие общаться с внешним миром. Будем предполагать, что имеется интерфейс Ethernet `eth0` и что нам достаточно простой конфигурации IP v4.

В примерах ниже используются сетевые утилиты, входящие в состав BusyBox, которых вполне достаточно в простых случаях: старые, но надежные программы `ifup` и `ifdown`. На страницах руководства они описаны во всех подробностях. Основная конфигурация сети хранится в файле `/etc/network/interfaces`. В каталоге технологической подготовки нужно будет создать следующие каталоги:

```
etc/network
etc/network/if-pre-up.d
etc/network/if-up.d
var/run
```

Для задания статического IP-адреса файл `etc/network/interfaces` должен выглядеть примерно так:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet static
```

```
  address 10.0.0.42
  netmask 255.255.255.0
  network 10.0.0.0
```

А в случае получения динамического IP-адреса по протоколу DHCP – так:

```
auto lo
iface lo inet loopback
auto eth0
iface eth0 inet dhcp
```

Необходимо будет также настроить клиентскую программу DHCP. В состав BusyBox входит программа udchpcd. Ей необходим скрипт оболочки /usr/share/udhcpc/default.script. Подходящий пример имеется в исходном коде BusyBox в файле examples/udhcp/simple.script.

Сетевые компоненты для glibc

В библиотеке glibc применяется диспетчер службы имен (name service switch – **NSS**), который управляет преобразованием символических имен сетевых объектов и пользователей в числовые эквиваленты. Так, имена пользователей преобразуются в UID посредством файла /etc/passwd; для преобразования имен сетевых служб, например HTTP, в номера портов используется файл /etc/services и т. д. Все это конфигурируется в файле /etc/nsswitch.conf, полное описание которого можно найти на странице руководства *nss(5)*. Примера ниже достаточно для большинства встраиваемых Linux-систем:

```
passwd:    files
group:     files
shadow:    files
hosts:     files dns
networks:  files
protocols: files
services:  files
```

Для всех преобразований используется файл с соответствующим именем в каталоге /etc, за исключением имен хостов, для разрешения которых может дополнительно использоваться поиск в системе DNS.

Чтобы все это заработало, нужно поместить необходимые файлы в каталог /etc. Сети, протоколы и службы одинаковы во всех системах на базе Linux, поэтому эти файлы можно скопировать из каталога /etc на машине разработки. Файл /etc/hosts должен содержать как минимум возвратный адрес:

```
127.0.0.1 localhost
```

К файлам passwd, group и shadow мы вернемся позже.

Последний фрагмент пазла – библиотеки, реализующие преобразование имен. Это подключаемые модули, которые загружаются по мере необходимости в зависимости от содержимого файла nsswitch.conf, и, значит, ни readelf, ни какая-ни-

будь другая программа такого рода не покажет их в качестве зависимостей. Просто скопируйте их из каталога sysroot набора инструментов:

```
$ cd ~/rootfs
$ cp -a $TOOLCHAIN_SYSROOT/lib/libnss* lib
$ cp -a $TOOLCHAIN_SYSROOT/lib/libresolv* lib
```

Создание образов файловой системы с помощью таблиц устройств

В исходном коде ядра имеется утилита gen_init_cpio, которая создает файл в формате cpio, следуя инструкциям в текстовом файле, который называется таблицей устройств. Это позволяет обычному пользователю создавать узлы устройств и назначать произвольные UID и GID любому файлу или каталогу.

Та же идея применима к инструментам, создающим образы файловой системы в других форматах:

- jffs2: mkfs.jffs2
- ubifs: mkfs.ubifs
- ext2: genext2fs

jffs2 и ubifs мы рассмотрим в главе 7, когда будем говорить о файловых системах для флэш-памяти. А ext2 – довольно старая файловая система для жестких дисков.

Любая программа принимает файл таблицы устройств, строки которого записаны в формате <name> <type> <mode> <uid> <gid> <major> <minor> <start> <inc> <count>, где:

- name: имя файла;
- type: может принимать следующие значения:
  - f: обычный файл;
  - d: каталог;
  - c: специальный файл символьного устройства;
  - b: специальный файл блочного устройства;
  - p: FIFO (именованный канал);
- uid: UID файла;
- gid: GID файла;
- major и minor: старший и младший номера устройства (только для узлов устройств);
- start, inc и count: (только для узлов устройств) позволяет создать группу из count узлов устройств с младшим номером, начинающимся с start и изменяющимся с шагом inc.

Нет необходимости описывать каждый файл, как в случае gen_init_cpio: нужно лишь указать на каталог, где эти файлы находятся, – каталог технологической подготовки – и перечислить те отличия, которые необходимо внести при построении образа конечной файловой системы.

Ниже приведен простой пример описания статических узлов устройств:

```
/dev         d 755 0   0   -  -  -  -  -
/dev/null    c 666 0   0   1  3  0  0  -
/dev/console c 600 0   0   5  1  0  0  -
/dev/tty00   c 600 0   0 252  0  0  0  -
```

Затем воспользуемся программой genext2fs для генерации образа файловой системы емкостью 4 МиБ (т. е. содержащей 4096 блоков стандартного размера 1024 байта):

```
$ genext2fs -b 4096 -d rootfs -D device-table.txt -U rootfs.ext2
```

Теперь можно скопировать получившийся образ rootfs.ext на карту SD или аналогичное устройство.

Копирование корневой файловой системы на карту SD

Здесь описан пример монтирования файловой системы с обычного блочного устройства, каковым является карта SD. Но те же принципы применимы и к другим типам файловых систем, и мы рассмотрим их подробнее в главе 7.

Предположим, что устройство оснащено картой SD и что первый раздел используется для хранения загрузочных файлов, MLO и u-boot.img, – как в случае платы BeagleBone Black. Предположим также, что с помощью программы genext2fs создан образ файловой системы. Чтобы скопировать его на карту SD, вставьте карту и выясните, какое блочное устройство ей сопоставлено; обычно это /dev/sd или /dev/mmcblk0. В последнем случае скопируйте образ файловой системы во второй раздел:

```
$ sudo dd if=rootfs.ext2 of=/dev/mmcblk0p2
```

Затем вставьте карту SD в устройство и задайте такую командную строку ядра: root=/dev/mmcblk0p2. Полная последовательность загрузки выглядит следующим образом:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
setenv bootargs console=ttyO0,115200 root=/dev/mmcblk0p2
bootz 0x80200000 - 0x80f00000
```

Монтирование корневой файловой системы по NFS

Если устройство оснащено сетевым интерфейсом, то во время разработки проще всего монтировать корневую файловую систему по сети. Тем самым вы получаете доступ к практически неограниченной внешней памяти, так что сможете добавить средства отладки и исполняемые файлы с большими таблицами символов. А в качестве дополнительного бонуса обновления корневой файловой системы на

машине разработки сразу же становятся доступны на целевом устройстве. Кроме того, у вас имеются копии журналов.

Для работы в таком режиме нужно при конфигурировании ядра задать параметр CONFIG_ROOT_NFS. Тогда можно будет сконфигурировать Linux, так чтобы монтирование производилось на этапе загрузки, добавив в командную строку ядра параметр

```
root=/dev/nfs
```

Укажите параметры экспорта NFS:

```
nfsroot=<host-ip>:<root-dir>
```

Сконфигурируйте сетевой интерфейс, подключенный к NFS-серверу, так чтобы он был доступен на этапе загрузки до запуска программы init. Для этого добавьте параметр

```
ip=<target-ip>
```

Дополнительные сведения о монтировании корневой файловой системы по NFS приведены в файле Documentation/filesystems/nfs/nfsroot.txt в исходном коде ядра.

Необходимо еще установить и настроить NFS-сервер на своей машине. В дистрибутиве Ubuntu это делается командой

```
$ sudo apt-get install nfs-kernel-server
```

NFS-серверу нужно сказать, какие каталоги следует экспортировать в сеть. Для этого служит файл /etc/exports. Добавьте в него строку вида:

```
/<path to staging> *(rw,sync,no_subtree_check,no_root_squash)
```

Затем перезапустите сервер, чтобы изменения вступили в силу. В Ubuntu это делается так:

```
$ sudo /etc/init.d/nfs-kernel-server restart
```

Тестирование в эмуляторе QEMU

Показанный ниже скрипт создает виртуальную сеть между сетевым устройством tap0 в исходной системе и интерфейсом eth0 в целевой системе в предположении, что обоим присвоены статические IPv4-адреса. Затем скрипт запускает QEMU с параметрами, при которых tap0 используется как эмулированный интерфейс. В качестве пути к корневой файловой системе ROOTDIR вы должны будете задать полный путь к своему каталогу технологической подготовки и, быть может, изменить IP-адреса, если они конфликтуют с конфигурацией вашей сети:

```
#!/bin/bash

KERNEL=zImage
DTB=vexpress-v2p-ca9.dtb
ROOTDIR=/home/chris/rootfs
```

```
HOST_IP=192.168.1.1
TARGET_IP=192.168.1.101
NET_NUMBER=192.168.1.0
NET_MASK=255.255.255.0

sudo tunctl -u $(whoami) -t tap0
sudo ifconfig tap0 ${HOST_IP}
sudo route add -net ${NET_NUMBER} netmask ${NET_MASK} dev tap0
sudo sh -c "echo 1 > /proc/sys/net/ipv4/ip_forward"

QEMU_AUDIO_DRV=none \
qemu-system-arm -m 256M -nographic -M vexpress-a9 -kernel $KERNEL -append \
"console=ttyAMA0 root=/dev/nfs rw nfsroot=${HOST_IP}:${ROOTDIR} \
ip=${TARGET_IP}" -dtb ${DTB} -net nic -net tap,ifname=tap0,script=no
```

Скрипт называется `run-qemu-nfs.sh`.

В результате система должна загружаться, как и раньше, но теперь каталог технологической подготовки непосредственно экспортируется по NFS. Файлы, создаваемые в этом каталоге, сразу становятся видны на целевом устройстве, и наоборот.

Тестирование с платой BeagleBone Black

При работе с платой BeagleBone Black нужно ввести следующие команды в ответ на приглашение U-Boot:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
setenv npath [path to staging directory]
setenv bootargs console=ttyO0,115200 root=/dev/nfs rw \
nfsroot=${serverip}:${npath} ip=${ipaddr}
```

Затем загрузите ядро и двоичное дерево устройств с карты SD, как и раньше:

```
fatload mmc 0:1 0x80200000 zImage
fatload mmc 0:1 0x80f00000 am335x-boneblack.dtb
bootz 0x80200000 - 0x80f00000
```

Проблемы с правами доступа к файлам

Файлы, уже находящиеся в каталоге технологической подготовки, принадлежат вам, и команда `ls -l` на целевом устройстве покажет ваш UID – как правило, 1000. Однако файлы, созданные самой целевой системой, будут принадлежать пользователю `root`. Получается неразбериха.

Увы, простого решения у этой проблемы не существует. Лучшее, что я могу посоветовать, – сделать копию каталога технологической подготовки и изменить владельца на `root:root` (командой `sudo chown -R 0:0 *`), а затем экспортировать эту копию по NFS. Это в какой-то мере уменьшает неудобства, связанные с наличием единственного экземпляра корневой файловой системы, общего для исходной и целевой систем.

Загрузка ядра по протоколу TFTP

При работе с реальным оборудованием, например платой BeagleBone Black, лучше всего загружать ядро по сети, особенно если корневая файловая система монтируется по NFS. Тогда локальная память устройства вообще не используется. Тем самым вы экономите время перезаписи флэш-памяти и к тому же можете работать, когда драйверы флэш-памяти еще только разрабатываются (такое тоже случается).

Загрузчик U-Boot уже много лет поддерживает протокол **Trivial File Transfer Protocol (TFTP)**. Сначала установите на машину разработки демон `tftp`. В Ubuntu для этого нужно установить пакет `tftpd-hpa`, который дает доступ для чтения к файлам в каталоге `/var/lib/tftpboot` TFTP-клиентам, в частности U-Boot.

В предположении, что файлы `zImage` и `am335x-boneblack.dtb` скопированы в каталог `/var/lib/tftpboot`, введите в ответ на приглашение U-Boot следующие команды:

```
setenv serverip 192.168.1.1
setenv ipaddr 192.168.1.101
tftpboot 0x80200000 zImage
tftpboot 0x80f00000 am335x-boneblack.dtb
setenv npath [path to staging]
setenv bootargs console=ttyO0,115200 root=/dev/nfs rw \
nfsroot=${serverip}:${npath} ip=${ipaddr}
bootz 0x80200000 - 0x80f00000
```

Ответ на команду `tftpboot` частенько имеет такой вид:

```
setenv ipaddr 192.168.1.101
nova!> setenv serverip 192.168.1.1
nova!> tftpboot 0x80200000 zImage
link up on port 0, speed 100, full duplex
Using cpsw device
TFTP from server 192.168.1.1; our IP address is 192.168.1.101
Filename 'zImage'.
Load address: 0x80200000
Loading: T T T T
```

Последовательность символов `T` в последней строке означает, что имеет место какая-то ошибка и на запросы TFTP-клиента не приходит ответ в отведенное время. Обычно это объясняется одной из следующих причин:

- неправильно задан IP-адрес сервера;
- на сервере не запущен демон TFTP;
- брандмауэр, работающий на сервере, блокирует протокол TFTP. И действительно, большинство брандмауэров по умолчанию блокирует порт TFTP, 69.

В данном случае демон tftp не был запущен, поэтому достаточно выполнить команду:

```
$ sudo service tftpd-hpa restart
```

Дополнительная литература

- *Filesystem Hierarchy Standard*. Текущая версия стандарта, 3.0, доступна по адресу http://refspecs.linuxfoundation.org/fhs.shtml.
- *ramfs, rootfs and initramfs* , *Rob Landley*. Эта статья, датированная 17 октября 2005 г., теперь включена в исходный код Linux (файл `Documentation/filesystems/ramfs-rootfs-initramfs.txt`).

Резюме

Одна из сильных сторон Linux – поддержка широкого круга корневых файловых систем, что позволяет адаптироваться к самым разным потребностям. Мы видели, как вручную построить простую корневую файловую систему с небольшим набором компонентов, и поняли, что BusyBox может оказать в этом деле существенную помощь. Проделав все шаги один за другим, мы разобрались в некоторых внутренних механизмах работы Linux, в том числе в том, как конфигурировать сеть и учетные записи пользователей. Однако с ростом сложности устройств задача быстро становится неподъемной. К тому же нужно всегда помнить о том, что в реализации может оказаться уязвимость, которую мы не заметили. В следующей главе мы узнаем, чем могут помочь средства сборки встраиваемых систем.

Глава 6

Выбор системы сборки

В предыдущих главах мы рассмотрели все четыре элемента встраиваемой Linux-системы и шаг за шагом проследили за тем, как собрать набор инструментов, начальный загрузчик, ядро и корневую файловую систему, а затем объединить их в простую встраиваемую Linux-систему. И шагов этих было много! Настало время узнать о том, как можно упростить этот процесс, автоматизировав все, что возможно. Я расскажу, как в этом могут помочь средства сборки встраиваемых систем и конкретно две из них: Buildroot и Yocto Project. То и другое – сложные и гибкие инструменты, и для полного их описания потребовалась бы целая книга. В этой главе я остановлюсь лишь на общих идеях, лежащих в основе систем сборки. Я покажу, как построить простой образ устройства, чтобы вы могли составить общее представление о системе, а затем – как внести полезные изменения на примере гипотетической платы Nova, упомянутой в предыдущих главах.

Довольно самопальных встраиваемых систем

В главе 5 был описан процесс ручной сборки самопальной файловой системы. Его преимущество в том, что вы сохраняете полный контроль над программным обеспечением и можете адаптировать его под свои нужды произвольным образом. Хотите сделать что-то странное, но инновационное? Пожалуйста. Уменьшить объем занимаемой памяти до абсолютного минимума? Нет проблем. Но в большинстве случаев ручная сборка – пустая трата времени, и в результате получаются системы низкого качества, неудобные для сопровождения.

Обычно они строятся постепенно на протяжении многих месяцев, часто не документируются и редко могут быть воссозданы с чистого листа, потому что никто не знает, откуда была взята какая часть.

Системы сборки

Идея системы сборки – автоматизировать все описанные выше шаги. Она должна уметь собирать из полученного извне исходного кода все или хотя бы некоторые из следующих компонентов:

- набор инструментов;
- начальный загрузчик;

- ядро;
- корневую файловую систему.

Наличие исходного кода важно по нескольким причинам. Во-первых, есть уверенность, что систему можно будет пересобрать в любое время, не оглядываясь на внешние зависимости. Во-вторых, исходный код доступен для отладки, и вы точно знаете, что сможете выполнить условия лицензии, передав его пользователям в случае необходимости.

Поэтому для достижения своих целей система сборки должна уметь следующее:

- скачать исходный код из источника – прямо из системы управления версиями или в виде архива – и сохранить его на локальной машине;
- применить заплаты, обеспечивающие возможность кросс-компиляции, исправить архитектурно-зависимые ошибки, применить локальные политики конфигурирования и т. д.;
- собрать различные компоненты;
- создать область технологической подготовки и построить корневую файловую систему;
- создать образы в различных форматах, готовые к загрузке на целевое устройство.

Полезны также следующие возможности:

- добавлять собственные пакеты, содержащие, к примеру, приложения или модификации ядра;
- выбирать различные профили корневой файловой системы: большая или маленькая, с поддержкой или без поддержки графики и т. д.;
- создавать автономный SDK, который можно распространять среди других разработчиков, чтобы им не приходилось устанавливать полную систему сборки;
- следить за тем, какие лицензии применяются в различных использованных пакетах с открытым исходным кодом;
- способность создавать обновления, применимые к уже установленным системам;
- наличие удобного интерфейса с пользователем.

В любом случае компоненты системы инкапсулированы в виде пакетов, одни из которых предназначены для системы разработки, другие – для целевой системы. Для каждого пакета определен набор правил получения исходного кода, его сборки и установки результата в нужное место. Между пакетами существуют зависимости, и механизм сборки должен уметь их разрешать и создавать полный набор требуемых пакетов.

За последние годы системы сборки с открытым исходным кодом достигли высокой степени зрелости. Их довольно много:

- **Buildroot**: простая система на основе GNU make и Kconfig (http://buildroot.org);
- **EmbToolkit**: простая система генерации корневых файловых систем, на момент написания книги единственная, изначально поддерживающая LLVM/Clang (https://www.embtoolkit.org);
- **OpenEmbedded**: мощная система, входящая в состав Yocto Project и других проектов (http://openembedded.org);

- **OpenWrt**: система, ориентированная на сборку прошивок для беспроводных маршрутизаторов (https://openwrt.org);
- **PTXdist**: система сборки с открытым исходным кодом, спонсируемая компанией Pengutronix (http://www.pengutronix.de/software/ptxdist/index_en.html);
- **Tizen**: всеохватывающая система с упором на мобильные, мультимедийные и автомобильные устройства (https://www.tizen.org);
- **The Yocto Project**: расширяет OpenEmbedded за счет средств конфигурирования, дополнительных уровней, инструментов и документации; пожалуй, самая популярная система (http://www.yoctoproject.org).

Я сосредоточу внимание на двух системах: Buildroot и Yocto Project. Они знаменуют два разных подхода к задаче и имеют разные цели.

Основная цель Buildroot – сборка образа корневой файловой системы, отсюда и ее название, хотя она умеет также собирать начальный загрузчик и образы ядра. Ее легко установить и настроить, а образы генерируются быстро.

С другой стороны, Yocto Project – более общая система в том смысле, что позволяет описать целевую систему и способна собирать встраиваемые системы для весьма сложных устройств. Каждый компонент генерируется в виде пакета в формате RPM, .dpkg или .ipk (см. следующий раздел), а затем все пакеты собираются в единый образ файловой системы. Можно даже включить в этот образ менеджер пакетов, который позволит обновлять их во время работы. Проще говоря, Yocto Project создает ваш собственный дистрибутив Linux.

Форматы пакетов и менеджеры пакетов

Основные дистрибутивы Linux строятся, как правило, из набора двоичных (откомпилированных) пакетов в формате RPM или deb. Менеджер пакетов **RPM (Red Hat Package Manager)** используется в дистрибутивах Red Hat, Suse, Fedora и производных от них. В дистрибутивах, производных от Debian, в том числе Ubuntu и Mint, используется менеджер пакетов в формате `deb`. Кроме того, существует упрощенный формат на основе `deb`, разработанный специально для встраиваемых систем: **Itsy PacKage**, или **ipk**.

Включение менеджера пакетов в программное обеспечение устройства – одна из важнейших отличительных особенностей системы сборки. Если на целевом устройстве есть менеджер пакетов, то открывается простая возможность устанавливать на него новые пакеты и обновлять существующие. О том, к каким последствиям это приводит, я расскажу в следующей главе.

Buildroot

Сайт проекта Buildroot расположен по адресу http://buildroot.org.

Текущая версия Buildroot умеет собирать набор инструментов, начальный загрузчик (U-Boot, Barebox, GRUB2 или Gummiboot), ядро и корневую файловую систему. В качестве основного средства сборки используется GNU `make`.

На странице http://buildroot.org/docs.html опубликована прекрасная документация, в том числе руководство пользователя «The Buildroot User Manual».

История

Buildroot – одна из первых систем сборки. Поначалу она была частью проектов uClinux и uClibc и применялась для генерации небольших корневых файловых систем, используемых для тестирования. Проект отделился в конце 2001 года и продолжал развиваться до 2006 года, после чего впал в спячку. Но в 2009 году бразды правления взял в свои руки Петер Корсгаард (Peter Korsgaard), и проект быстро двинулся вперед: были добавлены поддержка наборов инструментов на базе `glibc` и средства сборки начального загрузчика и ядра.

Buildroot также лежит в основе еще одной популярной системы сборки, OpenWrt (http://wiki.openwrt.org), которая отпочковалась от Buildroot в 2004 году. В OpenWrt акцент сделан на производстве ПО для беспроводных маршрутизаторов, поэтому состав пакетов ориентирован на сетевую инфраструктуру. Включен также менеджер пакетов в формате `.ipk`, поэтому устройство можно обновлять и модернизировать без полной перепрошивки образа.

Стабильные версии и поддержка

Разработчики Buildroot выпускают стабильные версии четыре раза в год: в феврале, мае, августе и ноябре. В репозитории git им соответствуют метки вида `<year>.02`, `<year>.05`, `<year>.08` и `<year>.11`. Обычно, начиная новый проект, имеет смысл взять последнюю стабильную версию. Однако стабильные версии редко обновляются после выпуска. Чтобы получить исправления уязвимостей и других ошибок, следует либо все время переходить на новые стабильные версии по мере их выхода, либо выполнять обратное портирование исправлений в используемую версию.

Установка

Как обычно, для установки Buildroot нужно либо клонировать репозиторий, либо скачать архив. Ниже приведен пример получения версии 2015.08.1 – последней стабильной версии на момент написания книги:

```
$ git clone git://git.buildroot.net/buildroot
$ cd buildroot
$ git checkout 2015.08.1
```

Эквивалентный архив в формате TAR находится по адресу http://buildroot.org/downloads.

Далее следует прочитать раздел «System Requirement» (Требования к системе) руководства пользователя Buildroot, опубликованного по адресу http://buildroot.org/downloads/manual/manual.html, и убедиться, что в вашей системе установлены все перечисленные там пакеты.

Конфигурирование

В Buildroot используются механизмы Kconfig и Kbuild, описанные в разделе «О конфигурировании ядра» главы 4. Ядро можно сконфигурировать с нуля, воспользовавшись программой menuconfig (или xconfig, или gconfig), либо выбрать какую-нибудь из примерно 90 конфигураций для различных плат и эмулятора QEMU, которые находятся в каталоге configs/. Команда make help выводит все цели, включая конфигурации по умолчанию.

Для начала соберем конфигурацию по умолчанию, которая может работать под управлением эмулятора ARM QEMU:

```
$ cd buildroot
$ make qemu_arm_versatile_defconfig
$ make
```

Обратите внимание, что мы не задали флаг -j, определяющий количество параллельных задач. Buildroot сама определяет, как оптимально задействовать имеющиеся процессоры. Если хотите ограничить число задач, запустите make menuconfig и загляните в раздел **Build**.

Сборка занимает от получаса до часа в зависимости от возможностей исходной машины и скорости доступа к Интернету. По завершении будут созданы два новых каталога:

- dl/: содержит архивные копии скачанных проектов, которые были собраны Buildroot;
- output/: промежуточные и окончательные откомпилированные ресурсы.

В каталоге output/ находятся следующие подкаталоги:

- build/: отдельные каталоги сборки для каждого компонента;
- host/: содержит различные необходимые Buildroot инструменты, которые работают в исходной системе, в том числе исполняемые файлы из набора инструментов (в каталоге output/host/usr/bin);
- images/: самый важный каталог – здесь находятся результаты сборки. В зависимости от сконфигурированных параметров это может быть начальный загрузчик, ядро и один или несколько образов корневой файловой системы;
- staging/: символическая ссылка на каталог sysroot набора инструментов. Имя ссылки может ввести в заблуждение, поскольку она указывает не на область технологической подготовки в том смысле, который был определен в главе 5;
- target/: это область технологической подготовки корневого каталога. Отметим, что использовать ее в качестве корневой файловой системы нельзя, так как владельцы файлов и права доступа к ним установлены неправильно. Buildroot использует таблицу устройств, описанную в предыдущей главе, для установки владельцев и прав на этапе создания образа файловой системы.

Выполнение

Для некоторых примеров конфигураций в каталоге boards/ имеется подкаталог, где хранятся конфигурационные файлы и сведения об установке результатов сборки на целевое устройство. К только что собранной нами системе относится файл board/qemu/arm-vexpress/readme.txt, где написано, как запустить ее под управлением QEMU.

В предположении, что программа qemu-system-arm уже установлена (см. главу 1), можно выполнить такую команду:

```
$ qemu-system-arm -M vexpress-a9 -m 256 \
-kernel output/images/zImage \
-dtb output/images/vexpress-v2p-ca9.dtb \
-drive file=output/images/rootfs.ext2,if=sd \
-append "console=ttyAMA0,115200 root=/dev/mmcblk0" \
-serial stdio -net nic,model=lan9118 -net user
```

В том окне терминала, где был запущен QEMU, должны появиться сообщения о загрузке ядра и в конце приглашение для входа в систему:

```
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset

Linux version 4.1.0 (chris@builder) (gcc version 4.9.3 (Build-
root 2015.08) ) #1 SMP Fri Oct 30 13:55:50 GMT 2015

CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c5387d

CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: V2P-CA9
[...]
VFS: Mounted root (ext2 filesystem) readonly on device 179:0.
devtmpfs: mounted
Freeing unused kernel memory: 264K (8061e000 - 80660000)
random: nonblocking pool is initialized
Starting logging: OK
Starting mdev...
Initializing random number generator... done.
Starting network...

Welcome to Buildroot
buildroot login:
```

Войдите как root без пароля.

Вы увидите, как QEMU открывает черное окно в дополнение к тому, где выведены сообщения ядра. Именно в нем отображается видеобуфер целевой системы. В данном случае целевая система ничего не пишет в видеобуфер, поэтому окно так и остается черным. Чтобы выйти из QEMU, введите команду poweroff или просто закройте окно видеобуфера. Это работает в версии QEMU 2.0 (установлена по умолчанию в Ubuntu 14.04), но приводит к ошибке в предыдущих версиях, включая QEMU 1.0.50 (установлена по умолчанию в Ubuntu 12.04), из-за проблем с эмуляцией SCSI.

Создание специального BSP-пакета

Теперь воспользуемся Buildroot, чтобы создать пакет поддержки платы (Board Support Package – BSP) для нашей платы Nova с теми же версиями U-Boot и Linux, что были описаны в предыдущих главах. Вот рекомендуемые места для хранения собственных изменений:

- `board/<организация>/<устройство>`: заплаты, двоичные объекты, дополнительные шаги сборки, конфигурационные файлы для Linux, U-Boot и других компонентов;
- `configs/<устройство>_defconfig`: конфигурация платы по умолчанию;
- `packages/<организация>/<имя_пакета>`: место для хранения дополнительных пакетов для платы.

Мы можем взять за основу конфигурационный файл платы BeagleBone, поскольку Nova очень близка к ней:

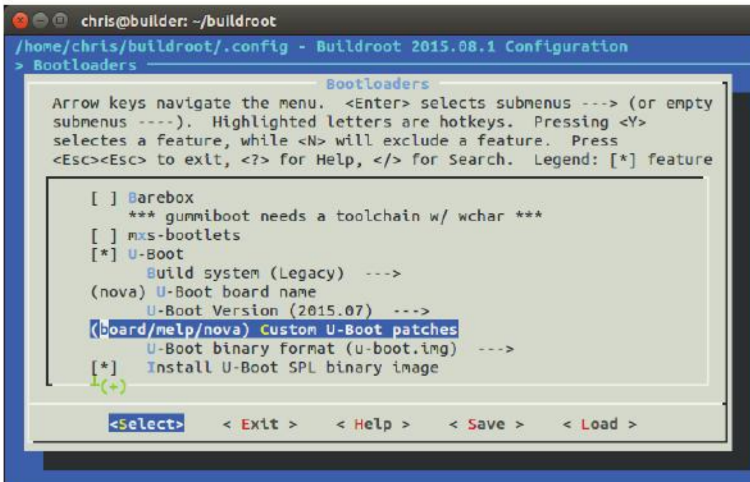
```
$ make clean # При смене целевой платформы всегда следует выполнять очистку
$ make beaglebone_defconfig
```

Сейчас файл `.config` содержит конфигурацию для BeagleBone. Далее создадим каталог для конфигурации нашей платы:

```
$ mkdir -p board/melp/nova
```

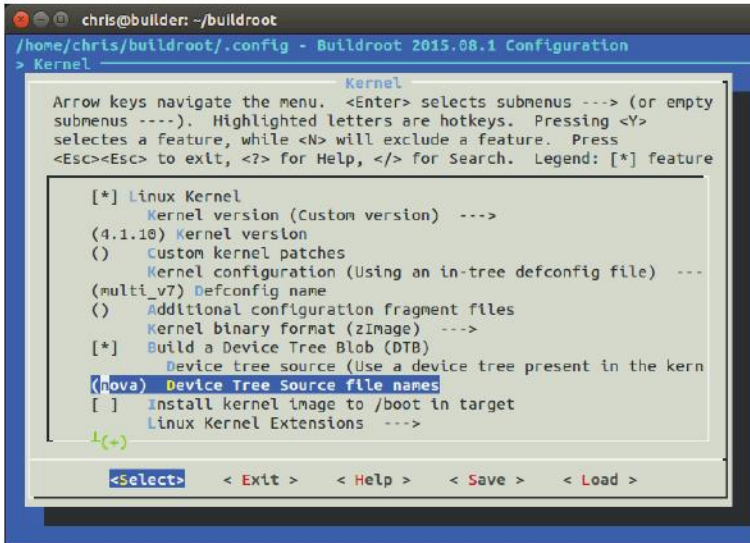
U-Boot

В главе 3 мы создали начальный загрузчик специально для платы Nova, основанный на версии U-Boot 2015.07, и файл заплаты для него. Мы можем сконфигурировать Buildroot, так чтобы выбиралась та же самая версия, и применить заплату. Сначала скопируйте файл заплаты в каталог `board/melp/nova`, а затем выполните команду `make menuconfig`, чтобы задать версию U-Boot 2015.07, каталог заплат `board/melp/nova` и имя платы `nova`, как показано на снимке экрана ниже.



Linux

В главе 4 мы собрали ядро на основе версии Linux 4.1.10 и подготовили новое дерево устройств в файле `nova.dts`. Скопируйте этот файл в каталог `board/melp/nova` и измените конфигурацию ядра в Buildroot, чтобы использовались эта версия и дерево устройств `nova`.



Сборка

Теперь, чтобы собрать систему для платы Nova, нужно просто набрать `make`, в результате чего в каталоге `output/images` будут созданы следующие файлы:

```
MLO    nova.dtb    rootfs.ext2    u-boot.img    uEnv.txt    zImage
```

Последний шаг – сохранение копии конфигурации, чтобы ее можно было использовать снова:

```
$ make savedefconfig BR2_DEFCONFIG=configs/nova_defconfig
```

Теперь у вас есть конфигурация Buildroot для платы Nova.

Добавление своего кода

Допустим, вы разработали какие-то программы и хотите включить их в процесс сборки. Есть два варианта. Первый: собрать их отдельно, пользуясь собственными средствами сборки, а затем наложить двоичные файлы на конечный результат сборки. Второй: создать пакет Buildroot, который можно выбрать из меню и построить как любой другой.

Наложение

Наложением называется структура каталогов, которая копируется поверх корневой файловой системы, построенной Buildroot, на последнем этапе сборки. В него могут входить исполняемые файлы, библиотеки и вообще все, что угодно. Отметим, что любой откомпилированный код должен быть совместим с библиотеками, развернутыми во время выполнения, а это означает, что он должен компилироваться тем же набором инструментов, который применяет Buildroot. Использовать набор инструментов Buildroot просто: нужно лишь добавить его в список путей:

```
$ PATH=<path_to_buildroot>/output/host/usr/bin:$PATH
```

Префикс набора инструментов – `<ARCH>-linux-`.

Наложение определяется параметром `BR2_ROOTFS_OVERLAY`, который содержит список каталогов через пробел, накладываемых на построенную корневую файловую систему. В `menuconfig` этот параметр находится в разделе **System configuration → Root filesystem overlay directories**.

Например, чтобы добавить программу `helloworld` в каталог `bin`, а также скрипт для ее запуска на этапе загрузки, следует создать наложение с такой структурой каталогов:

```
board/melp/nova/overlay
├ bin
│  └── helloworld
└ etc
   └── init.d
        └── S99helloworld
```

Затем каталог `board/melp/nova/overlay` нужно прописать в конфигурации.

Структура корневой файловой системы контролируется каталогом `system/skeleton`, а права доступа задаются в файлах `device_table_dev.txt` и `device_table.txt`.

Добавление пакета

Пакеты Buildroot хранятся в каталоге `package`; их больше 1000, и у каждого свой подкаталог. Пакет состоит как минимум из двух файлов: `Config.in`, содержащего фрагмент кода `Kconfig`, необходимый, чтобы пакет стал видимым в меню **configuration**, и make-файла с именем `<package_name>.mk`. Отметим, что пакеты не содержат никакого кода, а только инструкции о том, как получить код в виде архивного файла, из репозитория git и т. д.

Make-файл написан в формате, который ожидает Buildroot, и содержит директивы скачивания исходного кода, конфигурирования, компиляции и установки программы. Написание make-файла для нового пакета – сложная задача, детально рассмотренная в руководстве пользователя Buildroot. В примере ниже показано, как создать пакет для простой локальной программы, например `helloworld`.

Сначала создайте подкаталог `package/helloworld` и поместите в него такой конфигурационный файл `Config.in`:

```
config BR2_PACKAGE_HELLOWORLD
bool "helloworld"
help
  A friendly program that prints Hello World! every 10s
```

Первая строка должна иметь вид BR2_PACKAGE_<имя пакета заглавными буквами>. Далее идет строка, начинающаяся словом bool, за которым идет имя пакета так, как оно должно отображаться в меню **configuration**. Эта строка отвечает за возможность выбора пакета пользователем. Раздел *Help* факультативный (но, очевидно, полезный).

Затем свяжите новый пакет с меню **Target Packages**, для чего откройте файл package/Config.in и включите в него строку source с указанием описанного выше конфигурационного файла. Можно было бы добавить ее в уже существующее подменю, но в данном случае кажется более естественным создать новое подменю, содержащее только наш пакет:

```
menu "My programs"
  source "package/helloworld/Config.in"
endmenu
```

Далее создайте make-файл package/helloworld/helloworld.mk, который поставляет данные, необходимые Buildroot:

```
HELLOWORLD_VERSION:= 1.0.0
HELLOWORLD_SITE:= /home/chris/MELP/helloworld/
HELLOWORLD_SITE_METHOD:=local
HELLOWORLD_INSTALL_TARGET:=YES

define HELLOWORLD_BUILD_CMDS
  $(MAKE) CC="$(TARGET_CC)" LD="$(TARGET_LD)" -C $(@D) all
endef

define HELLOWORLD_INSTALL_TARGET_CMDS
  $(INSTALL) -D -m 0755 $(@D)/helloworld $(TARGET_DIR)/bin
endef

$(eval $(generic-package))
```

Местоположение кода здесь зашито в локальный путь. В реальной ситуации вы, наверное, получали бы код из системы управления версиями или с какого-то центрального сервера; в руководстве пользователя приведено подробное описание, а в других пакетах нет недостатка в примерах.

Соответствие лицензионным требованиям

Система Buildroot основана на ПО с открытым исходным кодом, как и пакеты, которые она компилирует. В какой-то момент вы должны будете проверить соответствие лицензионным требованиям, выполнив команду:

```
$ make legal-info
```

Собранная информация записывается в каталог `output/legal-info`. Это краткий отчет о лицензиях, использованных при компиляции инструментов в исходной системе (файл `host-manifest.csv`) и в целевой системе (файл `manifest.csv`). Дополнительные сведения смотрите в файле `README` и в руководстве пользователя Buildroot.

Yocto Project

Yocto Project устроен посложнее Buildroot. Он может не только собирать набор инструментов, начальный загрузчик, ядро и корневую файловую систему, как Buildroot, но и генерировать полный дистрибутив Linux со всеми двоичными пакетами, которые устанавливаются на этапе выполнения.

Yocto Project представляет собой преимущественно группу рецептов, похожих на пакеты Buildroot, но написанных на комбинации Python и языка оболочки, и планировщика задач BitBake, который порождает все, что сконфигурировано по рецептам.

На сайте https://www.yoctoproject.org/ хватает документации.

История

Структура Yocto Project станет более понятной, если принять во внимание его историю. Корнями проект уходит в OpenEmbedded (http://openembedded.org/), который, в свою очередь, вырос из многочисленных проектов портирования Linux на различные наладонные компьютеры, включая Sharp Zaurus и Compaq iPaq. Проект OpenEmbedded был основан в 2003 году как система сборки для таких наладонников, но быстро распространился и на другие встраиваемые устройства. Он разрабатывался и продолжает разрабатываться сообществом программистов-энтузиастов.

Перед проектом OpenEmbedded стояла цель создать набор двоичных пакетов в компактном формате `.ipk`, которые затем можно было бы разными способами комбинироваться для получения конечной системы, устанавливаемой на целевую платформу. Для этого нужно было написать рецепты сборки всех программных компонентов и воспользоваться планировщиком задач BitBake. Система была и остается очень гибкой. Подготовив нужные метаданные, можно создать полный дистрибутив Linux по собственным спецификациям. Из таких дистрибутивов хорошо известен *The Ångström Distribution* (http://www.angstrom-distribution.org), но есть и много других.

В 2005 году Ричард Пэрдай (Richard Purdie), в то время разработчик в компании OpenedHand, создал ответвление от проекта OpenEmbedded, отличающееся более консервативным подходом к отбору пакетов, и выпустил несколько стабильных версий. Он назвал проект Poky (произносится «поки») в честь японской закуски. И хотя Poky был ответвлением, оба проекта продолжали развиваться параллельно, заимствуя друг у друга обновления и стремясь в какой-то степени синхронизировать архитектуру. В 2008 году Intel приобрел OpenedHand, а в 2010-м передал Poky Linux в фонд Linux Foundation, где проект лег в основу Yocto Project.

Начиная с 2010 года, общие компоненты OpenEmbedded и Poky были объединены и вынесены в отдельный проект OpenEmbedded core, или просто oe-core.

Таким образом, в Yocto Project собрано несколько компонентов, из которых наиболее значимы следующие:

- **Poky**: эталонный дистрибутив;
- **oe-core**: основные метаданные, общие с OpenEmbedded;
- **BitBake**: планировщик задач, общий с OpenEmbedded и другими проектами;
- **документация**: руководства пользователя и разработчика для каждого компонента;
- **Hob**: графический пользовательский интерфейс к OpenEmbedded и BitBake;
- **Toaster**: веб-интерфейс к OpenEmbedded и BitBake;
- **ADT Eclipse**: подключаемый модуль для Eclipse, который упрощает построение проектов с использованием Yocto Project SDK.

Строго говоря, Yocto Project является «зонтиком» для этих подпроектов. В нем используется OpenEmbedded в качестве системы сборки, а Poky – как эталонное окружение и конфигурация по умолчанию. Однако зачастую, говоря «Yocto Project», имеют в виду только систему сборки. Полагаю, что мне уже слишком поздно плыть против течения, поэтому поступлю точно так же. Заранее приношу извинения разработчикам OpenEmbedded.

Yocto Project – это стабильная база, которую можно использовать как есть или расширить, применяя метаслои, о которых я расскажу ниже. Многие поставщики SoC-систем предлагают пакеты поддержки своих плат, устроенные именно таким образом. Метаслои можно также использовать для создания расширенных или просто других систем сборки. Некоторые из них являются открытыми, как Angstrom Project, другие – коммерческими, как MontaVista Carrier Grade Edition, Mentor Embedded Linux и Wind River Linux. В Yocto Project включена система проверки совместимости, гарантирующая возможность совместной работы компонентов. На многих веб-страницах можно встретить фразу типа «Yocto Project Compatible 1.7».

Таким образом, Yocto Project следует рассматривать как фундамент для целого спектра встраиваемых Linux-систем, а также как самостоятельную полнофункциональную систему сборки. Кстати говоря, *yocto* (иотта) – это префикс в системе единиц СИ, означающий 10^{-24}, точно так же, как «микро» означает 10^{-6}. Почему для проекта было выбрано такое название? Отчасти как намек на то, что с его помощью можно построить очень маленькую Linux-систему (хотя, если быть честным, это позволяют и другие системы сборки), а отчасти, возможно, чтобы утереть нос дистрибутиву Ångström, основанному на OpenEmbedded. Один ангстрем равен 10^{-10} м, просто огромная величина, по сравнению с иотта!

Стабильные версии и поддержка

Обычно новые версии Yocto Project выходят раз в шесть месяцев, в апреле и в октябре. У каждой версии есть кодовое название, но полезно также знать номера вер-

сий Yocto Project и Poky. Ниже перечислены последние четыре версии на момент написания книги.

Кодовое название	Дата выпуска	Версия Yocto	Версия Poky
Fido	Апрель 2015	1.8	13
Dizzy	Октябрь 2014	1.7	12
Daisy	Апрель 2014	1.6	11
Dora	Октябрь 2013	1.5	10

Стабильные версии поддерживаются в течение текущего и следующего циклов выпуска (вносятся исправления обнаруженных уязвимостей и критических ошибок), т. е. примерно в течение года после выхода. В обновлениях не допускаются никакие изменения набора инструментов или версии ядра. Как и в случае Buildroot, если требуется продолжение поддержки, то нужно либо переходить на следующую стабильную версию, либо самостоятельно портировать изменения в неподдерживаемую версию. Есть также возможность купить коммерческую поддержку на несколько лет у таких поставщиков операционных систем, как Mentor Graphics, Wind River и многие другие.

Установка Yocto Project

Для получения кода Yocto Project можно клонировать репозиторий, выбрав в качестве метки кодовое название (в данном случае fido):

```
$ git clone -b fido git://git.yoctoproject.org/poky.git
```

А можно скачать архив по адресу http://downloads.yoctoproject.org/ releases/yocto/yocto-1.8/poky-fido-13.0.0.tar.bz2.

В первом случае все файлы окажутся в каталоге poky, во втором – в каталоге poky-fido-13.0.0/.

Кроме того, прочитайте раздел «System Requirements» (Требования к системе) справочного руководства Yocto Project Reference Manual (http://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#detailed-supported-distros) и проверьте, что все перечисленные там пакеты установлены на вашу машину разработки.

Конфигурирование

Как и в случае Buildroot, начнем со сборки для эмулятора ARM QEMU. Прежде всего загрузите в оболочку скрипт для настройки окружения:

```
$ cd poky
$ source oe-init-build-env
```

При этом создается и делается текущим рабочий каталог build. В этот каталог помещаются все конфигурационные, промежуточные и подлежащие развертыванию файлы. Этот скрипт необходимо подгружать в начале каждого сеанса работы над проектом.

Можно указать другой рабочий каталог, передав путь к нему скрипту oe-init-build-env в качестве параметра, например:

```
$ source oe-init-build-env build-qemuarm
```

Теперь все файлы будут создаваться в каталоге build-qemuarm. Таким образом, можно одновременно работать над несколькими проектами, параметр скрипта oe-init-build-env определяет, какой проект выбрать в данный момент.

Первоначально в каталоге build есть только один подкаталог conf, в котором хранятся конфигурационные файлы проекта:

- local.conf: содержит описание устройства, для которого собирается система, и окружение сборки;
- bblayers.conf: содержит список каталогов, содержащих слои, которые вы собираетесь использовать. Впоследствии могут быть добавлены дополнительные слои;
- templateconf.cfg: содержит имя каталога с различными conf-файлами. По умолчанию указывает на meta-yocto/conf.

Пока что в файле local.conf нужно присвоить значение только переменной MACHINE, сделав ее равной qemuarm. Для этого удалите символ комментария в начале следующей строки:

```
MACHINE ?= "qemuarm"
```

Сборка

Для выполнения собственно сборки нужно запустить программу bitbake, уточнив, какой образ корневой файловой системы создавать. Ниже перечислены некоторые распространенные образы:

- **core-image-minimal**: небольшая консольная система, полезная для тестов и как основа для заказных образов;
- **core-image-minimal-initramfs**: как core-image-minimal, но с ram-диском;
- **core-image-x11**: базовый образ с поддержкой графики в объеме сервера X11 с терминальным приложением xterminal;
- **core-image-sato**: полноценная графическая система, основанная на Sato, мобильной графической среде на базе X11, и GNOME. Образ включает несколько приложений, в том числе терминал, редактор и диспетчер файлов.

Зная конечную цель, BitBake сначала строит все зависимости, начиная с набора инструментов. Для начала создадим минимальный образ – просто чтобы посмотреть, работает или нет:

```
$ bitbake core-image-minimal
```

Сборка займет некоторое время, возможно, больше часа. По завершении в каталоге build появится несколько новых каталогов, в том числе build/downloads, где находится весь скачанный исходный код, и build/tmp, содержащий большую часть артефактов сборки. В каталоге tmp вы найдете следующие подкаталоги:

- work: содержит каталог сборки и область технологической подготовки для всех компонентов, включая корневую файловую систему;

- `deploy`: содержит готовые двоичные файлы, подлежащие развертыванию на целевом устройстве:
  - `deploy/images/[machine name]`: содержит образы начального загрузчика, ядра и корневой файловой системы;
  - `deploy/rpm`: содержит RPM-пакеты, включенные в образы;
  - `deploy/licenses`: содержит файлы с лицензиями, извлеченными из каждого пакета.

Выполнение

При сборке для QEMU генерируется внутренняя версия QEMU, что избавляет от необходимости устанавливать пакет QEMU для своего дистрибутива, а значит, устраняет зависимость от версии. Для запуска этого внутреннего QEMU предназначен скрипт `runqemu`.

Чтобы запустить эмуляцию под управлением QEMU, сначала загрузите скрипт `oe-init-build-env`, а затем выполните такую команду:

```
$ runqemu qemuarm
```

В данном случае при конфигурировании QEMU была включена графическая консоль, поэтому сообщения о загрузке и приглашение к входу появляются на черном экране, соответствующем видеобуферу.

```
QEMU
md: Scanned 0 and added 0 devices.
md: autorun ...
md: ... autorun DONE.
EXT4-fs (sda): recovery complete
EXT4-fs (sda): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 8:0.
devtmpfs: mounted
Freeing unused kernel memory: 364K (c08cb000 - c0926000)
INIT: version 2.88 booting

Please wait: booting...
Starting udev
udev: Not using udev cache because of changes detected in the following files:
udev:     /proc/version /proc/cmdline /proc/devices
udev:     lib/udev/rules.d/* etc/udev/rules.d/*
udev: The udev cache will be regenerated. To identify the detected changes,
udev: compare the cached sysconf at   /etc/udev/cache.data
udev: against the current sysconf at  /dev/shm/udev.cache
udevd[73]: starting version 182
EXT4-fs (sda): re-mounted. Opts: data=ordered
random: dd urandom read with 117 bits of entropy available
Populating dev cache
random: nonblocking pool is initialized
INIT: Entering runlevel: 5
Configuring network interfaces... done.
Starting syslogd/klogd: done

Poky (Yocto Project Reference Distro) 1.8.1 qemuarm /dev/tty1

qemuarm login:
```

Можете войти как `root` без пароля. Для выхода из QEMU нужно закрыть окно видеобуфера. Можно запустить QEMU без графического окна, добавив параметр `nographic` в командную строку:

```
$ runqemu qemuarm nographic
```

В таком случае для выхода из QEMU нужно нажать **Ctrl**+**A**+**X**.

У скрипта `runqemu` много других параметров, для получения дополнительных сведений введите `runqemu help`.

Слои

Метаданные в Yocto Project разбиты на слои, и, по соглашению, имя каждого слоя начинается словом `meta`. Вот основные слои Yocto Project:

- **meta**: ядро OpenEmbedded;
- **meta-yocto**: метаданные, специфичные для Yocto Project, включая дистрибутив Poky;
- **meta-yocto-bsp**: пакеты поддержки плат для эталонных машин, поддерживаемых Yocto Project.

Список слоев, в которых BitBake ищет рецепты, хранится в файле `<ваш каталог сборки>/conf/bblayers.conf` и по умолчанию включает три вышеупомянутых слоя.

Такая структура рецептов и других конфигурационных данных позволяет очень легко расширять Yocto Project путем добавления новых слоев. Дополнительные слои есть в самом проекте Yocto Project, их предлагают изготовители SoC-систем и многочисленные желающие внести свой вклад в Yocto Project и OpenEmbedded. Полезный список слоев имеется на сайте http://layers.openembedded.org. Вот лишь несколько примеров:

- **meta-angstrom**: дистрибутив Ångström;
- **meta-qt5**: библиотеки и утилиты Qt5;
- **meta-fsl-arm**: пакеты поддержки для SoC-систем с архитектурой ARM компании Freescale;
- **meta-fsl-ppc**: пакеты поддержки для SoC-систем с архитектурой PowerPC компании Freescale;
- **meta-intel**: пакеты поддержки для процессоров и SoC-систем компании Intel;
- **meta-ti**: пакеты поддержки для SoC-систем с архитектурой ARM компании TI.

Для добавления слоя нужно всего лишь скопировать каталог с метаданными в подходящее место, обычно рядом со слоями по умолчанию, и прописать его в файле `bblayers.conf`. Только убедитесь, что он совместим с той версией Yocto Project, которой вы пользуетесь.

Для иллюстрации создадим слой для платы Nova, который будем использовать до конца этой главы. В каждом метаслое должен быть, по меньшей мере, один конфигурационный файл, `conf/layer.conf`, а также файл `README` и лицензия. Существует вспомогательный скрипт, который выполняет эти простые действия:

```
$ cd poky
$ scripts/yocto-layer create nova
```

Скрипт просит ввести приоритет слоя и спрашивает, хотим ли мы создать образцы рецептов. Ниже я согласился с предлагаемыми по умолчанию ответами:

```
Please enter the layer priority you'd like to use for the layer: [default: 6]
Would you like to have an example recipe created? (y/n) [default: n]
Would you like to have an example bbappend file created? (y/n) [default: n]
New layer created in meta-nova.
Don't forget to add it to your BBLAYERS (for details see meta-nova\README).
```

В результате будет создан слой с именем meta-nova, содержащий файл conf/layer.conf, заготовку файла README и лицензию MIT в файле COPYING.MIT. Файл layer.conf выглядит следующим образом:

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "nova"
BBFILE_PATTERN_nova = "^${LAYERDIR}/"
BBFILE_PRIORITY_nova = "6"
```

Он добавляет путь к слою в переменную BBPATH, а содержащиеся в слое рецепты – в переменную BBFILES. Глядя на код, мы понимаем, что рецепты ищутся в каталогах с именами, начинающимися с recipes-, и представляют собой файлы с расширением .bb (обычные рецепты BitBake) или .bbappend (рецепты, расширяющие имеющиеся обычные рецепты путем добавления или замены инструкций). Этот слой называется nova, так что это имя добавляется в список слоев в переменной BBFILE_COLLECTIONS, а приоритет слоя устанавливается равным 6. Приоритет используется в случае, когда один и тот же рецепт встречается в нескольких слоях: предпочтение отдается слою с более высоким приоритетом.

Поскольку мы строим новую конфигурацию, лучше начать с создания нового каталога сборки с именем build-nova:

```
$ cd ~/poky
$ . oe-init-build-env build-nova
```

Теперь нужно добавить этот слой в конфигурационный файл сборки conf/bblayers.conf:

```
LCONF_VERSION = "6"

BBPATH = "${TOPDIR}"
BBFILES ?= ""

BBLAYERS ?= " \
  /home/chris/poky/meta \
```

```
  /home/chris/poky/meta-yocto \
  /home/chris/poky/meta-yocto-bsp \
  /home/chris/poky/meta-nova \
  "
BBLAYERS_NON_REMOVABLE ?= " \
  /home/chris/poky/meta \
  /home/chris/poky/meta-yocto \
"
```

Убедиться в том, что все сделано правильно, помогает еще один вспомогательный скрипт:

```
$ bitbake-layers show-layers
layer                path                            priority
==========================================================
meta             /home/chris/poky/meta               5
meta-yocto       /home/chris/poky/meta-yocto         5
meta-yocto-bsp   /home/chris/poky/meta-yocto-bsp     5
meta-nova        /home/chris/poky/meta-nova          6
```

Как видим, появился новый слой. Он имеет приоритет 6, т. е. мы можем переопределять одноименные рецепты в других слоях с более низким приоритетом.

Теперь стоит выполнить сборку с этим, еще пустым слоем. Конечной целью будет плата Nova, но пока соберем систему для BeagleBone Black, для чего удалим комментарий перед строкой `MACHINE ?= "beaglebone"` в файле `conf/local.conf`. Затем соберем минимальный образ командой `bitbake core-image-minimal`, как и прежде.

Как и рецепты, слои могут содержать классы BitBake, конфигурационные файлы для машин, дистрибутивы и многое другое. Сначала я рассмотрю рецепты и покажу, как создать специализированный образ и пакет.

BitBake и рецепты

BitBake обрабатывает метаданные нескольких типов, в том числе:

- **рецепты**: файлы с расширением `.bb`. Содержат информацию о построении одной единицы ПО, в том числе о том, как получить исходный код, о зависимостях от других компонентов, о сборке и установке;
- **дополнения**: файлы с расширением `.bbappend`. Позволяют дополнить рецепт или частично заменить инструкции в нем. Инструкции, содержащиеся в `bbappend`-файле, просто дописываются в конец рецепта, хранящегося в одноименном `bb`-файле;
- **включения**: файлы с расширением `.inc`. Содержат информацию, общую для нескольких рецептов. Такие файлы можно включать с помощью ключевых слов `include` или `require`. Разница между ними в том, что `require` выдает ошибку, если файл не найден, а `include` – нет.
- **классы**: файлы с расширением `.bbclass`. Содержат общеизвестную информацию о сборке, например как собрать ядро или проект на основе `autotools`. Классы можно наследовать и расширять в рецептах и в других классах с по-

мощью ключевого слова inherit. Класс classes/base.bbclass неявно наследуется в каждом рецепте;

- **конфигурация**: файлы с расширением .conf. В них определены различные конфигурационные переменные, управляющие процессом сборки проекта.

Рецепт – это набор заданий, написанный на комбинации Python и языка оболочки. Заданиям присваиваются имена вида do_fetch, do_unpack, do_patch, do_configure, do_compile, do_install и т. д. Для их исполнения служит программа BitBake.

По умолчанию выполняется задание do_build, т. е. рецепт производит сборку. Чтобы получить список заданий, представленных в рецепте, выполните такую команду:

```
$ bitbake -c listtasks core-image-minimal
```

Флаг -c позволяет указать имя задания, опуская префикс do_, например -c fetch служит для получения кода, необходимого рецепту:

```
$ bitbake -c fetch busybox
```

Можно также указать задание fetchall, если нужно получить код как самой цели, так и всех ее зависимостей:

```
$ bitbake -c fetchall core-image-minimal
```

Файлам рецептов обычно даются имена вида <имя-пакета>_версия.bb. Они могут зависеть от других рецептов, что позволяет BitBake выполнить все подзадания, необходимые для завершения задачи верхнего уровня. К сожалению, в этой книге мне не хватит места, чтобы описать механизм зависимостей, однако все это изложено в документации по проекту Yocto Project.

Например, чтобы создать рецепт для нашей программы helloworld в слое meta-nova, нужно организовать такую структуру каталогов:

```
meta-nova/recipes-local/helloworld
├── files
│ └── helloworld.c
└── helloworld_1.0.bb
```

Файл рецепта будет называться helloworld_1.0.bb, а исходный код мы поместим в подкаталог files. Тогда рецепт будет содержать такие инструкции:

```
DESCRIPTION = "A friendly program that prints Hello World!"
PRIORITY = "optional"
SECTION = "examples"

LICENSE = "GPLv2"
LIC_FILES_CHKSUM = "file://${COMMON_LICENSE_DIR}/GPL-2.0;
md5=801f80980d171dd6425610833a22dbe6"

SRC_URI = "file://helloworld.c"
S = "${WORKDIR}"

do_compile() {
```

```
  ${CC} ${CFLAGS} -o helloworld helloworld.c
}

do_install() {
  install -d ${D}${bindir}
  install -m 0755 helloworld ${D}${bindir}
}
```

Местоположение исходного кода задается переменной SRC_URI: в данном случае будут просматриваться подкаталоги helloworld и helloworld-1.0 каталога рецепта. Определить нужно только задания do_compile и do_install, которые служат соответственно для компиляции единственного исходного файла и установки исполняемого файла в целевую корневую файловую систему. Результатом расширения переменной ${D} является область технологической подготовки, а переменной ${bindir} – подразумеваемый по умолчанию каталог исполняемых файлов /usr/bin.

С любым рецептом связана лицензия, указываемая в переменной LICENSE, в данном случае GPLv2. Имя файла с текстом лицензии и его контрольная сумма определены в переменной LIC_FILES_CHKSUM. BitBake прекращает сборку, если контрольные суммы не совпадают, поскольку это означает, что лицензия каким-то образом изменена. Имя файла лицензии должно либо указывать на файл, являющийся частью пакета, либо на текст одной из стандартных лицензий в каталоге meta/files/common-licenses, как в данном случае.

По умолчанию коммерческие лицензии запрещены, но их легко разрешить. Для этого нужно задать тип лицензии в рецепте:

```
LICENSE_FLAGS = "commercial"
```

Затем в файле conf/local.conf нужно будет явно разрешить лицензии такого типа:

```
LICENSE_FLAGS_WHITELIST = "commercial"
```

Чтобы удостовериться, что программа корректно компилируется, попросите BitBake собрать ее:

```
$ bitbake helloworld
```

Если все пройдет хорошо, то будет создан рабочий каталог tmp/work/cortexa8hf-vfp-neon-poky-linux-gnueabi/helloworld/.

Кроме того, появится RPM-пакет tmp/deploy/rpm/cortexa8hf_vfp_neon/helloworld-1.0-r0.cortexa8hf_vfp_neon.rpm.

Но пока этот пакет еще не стал частью конечного образа. Список подлежащих установке пакетов хранится в переменной IMAGE_INSTALL. Чтобы добавить свой пакет в конец этого списка, нужно дописать такую строку в файл conf/local.conf:

```
IMAGE_INSTALL_append = " helloworld"
```

Обратите внимание на пробел между открывающей двойной кавычкой и именем первого пакета – он обязателен. Теперь пакет будет включен в любой образ, собираемый bitbake:

```
$ bitbake core-image-minimal
```

Поинтересовавшись содержимым архива tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.tar.bz2, вы обнаружите там файл /usr/bin/helloworld.

Настройка образов с помощью local.conf

Часто на стадии разработки требуется добавить в образ пакет или еще как-то изменить его. Как было показано выше, мы можем просто расширить список устанавливаемых пакетов, дописав строку вида:

```
IMAGE_INSTALL_append = " strace helloworld"
```

Неудивительно, что можно сделать и обратную операцию – удалить пакет, – дописав такую строку:

```
IMAGE_INSTALL_remove = "someapp"
```

Можно внести и более масштабные изменения с помощью переменной EXTRA_IMAGE_FEATURES. Их слишком много, чтобы перечислять здесь, поэтому рекомендую заглянуть в раздел «Image Features» справочного руководства по Yocto Project и в код класса meta/classes/core-image.bbclass. Но краткий список, который позволит составить предложение о различных свойствах образов, я все же приведу:

- dbg-pkgs: установить пакеты отладочных символов для всех пакетов, включенных в образ;
- debug-tweaks: разрешить вход от имени root без пароля и другие изменения, упрощающие разработку;
- package-management: установить средства управления пакетами и сохранить базу данных диспетчера пакетов;
- read-only-rootfs: сделать корневую файловую систему доступной только для чтения. Этот режим мы подробнее рассмотрим в главе 7;
- x11: установить X-сервер;
- x11-base: установить X-сервер с минимальным окружением;
- x11-sato: установить окружение OpenedHand Sato.

Рецепт создания образа

При изменении файла local.conf возникает проблема из-за того, что эти изменения локальны. Если требуется создать образ, общий для нескольких разработчиков или предназначенный для загрузки в производственную систему, то эти изменения следует поместить в рецепт создания образа.

Рецепт создания образа содержит инструкции о том, как создать файлы образов для целевой системы: начального загрузчика, ядра и корневой файловой системы. Получить список доступных образов можно такой командой:

```
$ ls meta*/recipes*/images/*.bb
```

Рецепт создания образа core-image-minimal находится в файле meta/recipes-core/images/core-image-minimal.bb.

Проще всего взять существующий рецепт создания образа и модифицировать его по аналогии с файлом local.conf.

Пусть, например, мы хотим создать такой же образ, как core-image-minimal, но еще добавить туда нашу программу helloworld и утилиту strace. Для этого понадобится файл рецепта из двух строчек, который включает (с помощью ключевого слова require) базовый образ и добавляет указанные пакеты. По соглашению образ помещают в каталог images, поэтому добавим в каталог meta-nova/recipes-local/images рецепт такого содержания:

```
require recipes-core/images/core-image-minimal.bb
IMAGE_INSTALL += "helloworld strace"
```

Теперь можно удалить строку IMAGE_INSTALL_append из файла local.conf и собрать образ командой:

```
$ bitbake nova-image
```

Если вы хотите пойти дальше и полностью взять на себя контроль над корневой файловой системой, то можете начать с нуля, т. е. самостоятельно присвоить значение переменной IMAGE_INSTALL, как показано ниже:

```
SUMMARY = "A small image with helloworld and strace packages"
IMAGE_INSTALL = "packagegroup-core-boot helloworld strace"
IMAGE_LINGUAS = " "
LICENSE = "MIT"
IMAGE_ROOTFS_SIZE ?= "8192"
inherit core-image
```

Переменная IMAGE_LINGUAS содержит список локалей glibc, которые должны присутствовать в целевой системе. Локали могут занимать очень много места, поэтому в данном случае мы сделали список пустым, и это допустимо, коль скоро мы не будем пользоваться зависящими от локали библиотечными функциями. В переменной IMAGE_ROOTFS_SIZE хранится размер результирующего образа диска в КиБ. Основная часть работы возлагается на класс core-image, которому мы наследуем в самом конце.

Создание SDK

Очень полезно иметь возможность создания автономного набора инструментов, который могут установить другие разработчики. Тогда не придется всем членам команды полностью устанавливать Yocto Project. В идеале набор инструментов должен включать все библиотеки, устанавливаемые в целевую систему, и файлы-заголовки для них. Это можно сделать для любого образа с помощью задания populate_sdk:

```
$ bitbake nova-image -c populate_sdk
```

В результате в каталоге tmp/deploy/sdk создается самоустанавливаемый скрипт оболочки с именем вида

```
poky-<c_library>-<host_machine>-<target_image><target_machine>-toolchain-<version>.sh
```

Вот пример такого имени:

```
poky-glibc-x86_64-nova-image-cortexa8hf-vfp-neon-toolchain-1.8.1.sh
```

Отметим, что по умолчанию набор инструментов не включает статические библиотеки. Их можно добавить отдельно, дописав в `local.conf` или в рецепт создания образа строки вида:

```
TOOLCHAIN_TARGET_TASK_append = " glibc-staticdev"
```

Можно также активировать режим включения статических библиотек глобально:

```
SDKIMAGE_FEATURES_append = " staticdev-pkgs"
```

Если требуется только базовый набор инструментов, содержащий лишь кросс-компиляторы C и C++, библиотеку C и файлы-заголовки, то можно вместо этого выполнить команду:

```
$ bitbake meta-toolchain
```

Для установки SDK достаточно выполнить созданный скрипт оболочки. По умолчанию он устанавливается в каталог `/opt/poky`, но скрипт установки позволяет его изменить:

```
$ tmp/deploy/sdk/poky-glibc-x86_64-nova-image-cortexa8hf-vfp-neon-toolchain-1.8.1.sh
Enter target directory for SDK (default: /opt/poky/1.8.1):
You are about to install the SDK to "/opt/poky/1.8.1". Proceed[Y/n]?
[sudo] password for chris:
Extracting SDK...done
Setting it up...done
SDK has been successfully set up and is ready to be used.
```

Чтобы воспользоваться набором инструментов, сначала загрузите скрипт подготовки окружения:

```
. /opt/poky/1.8.1/environment-setup-cortexa8hf-vfp-neon-poky-linux-gnueabi
```

В сгенерированных таким образом наборах инструментов `sysroot` не сконфигурирован:

```
$ arm-poky-linux-gnueabi-gcc -print-sysroot
/not/exist
```

Следовательно, при попытке выполнить кросс-компиляцию, как в предыдущих главах, произойдет ошибка:

```
$ arm-poky-linux-gnueabi-gcc helloworld.c -o helloworld
helloworld.c:1:19: fatal error: stdio.h: No such file or directory
```

```
#include <stdio.h>
                    ^
compilation terminated.
```

Объясняется это тем, что на этапе конфигурирования задан обобщенный компилятор, поддерживающий широкий спектр процессоров с архитектурой ARM, а точная настройка производится в момент запуска путем использования подходящих флагов gcc. Так что если использовать в качестве команды компиляции переменную $CC, то все пройдет нормально:

```
$ $CC helloworld.c -o helloworld
```

Контроль лицензий

Yocto Project настаивает на том, чтобы в каждом пакете была лицензия. Копия лицензии помещается в файл tmp/deploy/licenses/[имя пакета] для каждого пакета на этапе его сборки. Кроме того, в каталоге <имя образа>-<имя машины>-<временная метка> сохраняются сводки всех пакетов и лицензий, включенных в образ.

```
$ ls tmp/deploy/licenses/nova-image-beaglebone-20151104150124
license.manifest      package.manifest
```

В первом файле хранится список лицензий для каждого пакета, во втором – список одних лишь имен пакетов.

Дополнительная литература

Дополнительные сведения имеются в следующих источниках:

- Руководство пользователя Buildroot. URL: http://buildroot.org/downloads/manual/manual.html.
- Документация по проекту Yocto Project: девять справочных руководств плюс десятое, содержащее все прочие (так называемое «Мегаруководство»), https://www.yoctoproject.org/documentation.
- *Vizuete D. M.* Instant Buildroot. Packt Publishing, 2013.
- *Salvador O., Angolini D.* Embedded Linux Development with Yocto Project. Packt Publishing, 2014.

Резюме

Система сборки берет на себя тяжелый труд по созданию встраиваемой Linux-системы, и это почти всегда лучше, чем ручная сборка системы. В настоящее время доступно много систем сборки с открытым исходным кодом: Buildroot и Yocto Project представляют два разных подхода. Buildroot – простая и быстрая система, которая идеально подходит для сравнительно несложных узкоспециализированных устройств: традиционных встраиваемых Linux-систем, как я привык их называть.

Yocto Project сложнее и обладает большей гибкостью. Он основан на пакетах, а значит, имеется возможность установить менеджер пакетов и производить обновление отдельных пакетов после того, как устройство уже куплено. Структура метаслоев упрощает расширение метаданных и пользуется широкой поддержкой со стороны сообщества и индустрии. Недостаток – сложность изучения: чтобы освоить систему профессионально, придется потратить несколько месяцев, и даже тогда можно столкнуться с неожиданностями, по крайней мере у меня так и было.

Не забывайте, что все системы, созданные с помощью этих инструментов, необходимо в течение какого-то времени, часто многих лет, сопровождать уже после того, как они куплены. Для Yocto Project выпускаются обновления в течение примерно одного года после выхода версии, для Buildroot обычно не выпускается никаких обновлений. В любом случае вам либо придется поддерживать версию, использованную для сборки своей системы, самостоятельно, либо платить за коммерческую поддержку. Третий вариант – игнорировать проблему – вообще не следует рассматривать!

В следующей главе мы рассмотрим файловое хранилище и файловые системы, а также вопрос о том, как сделанный выбор влияет на стабильность и удобство сопровождения встраиваемой Linux-системы.

Глава 7

Выбор стратегии хранения

Характеристики массовой памяти встраиваемого устройства существенно влияют на остальные части системы в плане надежности, быстродействия и обновления в месте эксплуатации.

В большинстве устройств используется флэш-память того или иного вида. За последние годы флэш-память сильно подешевела, а ее емкость увеличилась с десятков мегабайт до десятков гигабайт.

Я начну эту главу с подробного обзора технологии флэш-памяти и расскажу, как организация памяти влияет на управляющий ей низкоуровневый драйвер, не оставив без внимания интерфейс устройств на основе технологий памяти (memory technology device – MTD) в Linux.

Для каждой технологии флэш-памяти имеются различные файловые системы. Я опишу те, что чаще всего используются во встраиваемых устройствах, и закончу обзор сводкой вариантов для всех типов флэш-памяти.

В последних разделах рассматривается вопрос о том, как оптимально использовать флэш-память, как обновлять устройства в месте эксплуатации и как объединить все вместе в сбалансированную стратегию хранения.

Типы запоминающих устройств

Встраиваемым устройствам необходимо запоминающее устройство, которое потребляло бы мало электроэнергии, было компактным и надежным в эксплуатации на протяжении всего срока службы, который может достигать десяти лет. Почти во всех случаях это означает применение полупроводниковой (твердотельной) памяти, уже много лет используемой в постоянных запоминающих устройствах (ПЗУ). Но в последние 20 лет используются различные виды флэш-памяти. За это время сменилось несколько поколений флэш-памяти – от NOR- и NAND-приборов до управляемой флэш-памяти типа eMMC.

Флэш-память типа NOR стоит дорого, но надежна и допускает отображение на адресное пространство процессора, что позволяет непосредственно исполнять

код, находящийся в такой памяти. Емкость микросхем NOR-памяти невелика – от нескольких мегабайт до одного гигабайта.

Флэш-память типа NAND гораздо дешевле, а емкость одной микросхемы может быть выше – от десятков мегабайт до десятков гигабайт. Но чтобы превратить ее в полезную среду хранения, нужно много дополнительного оборудования и программного обеспечения.

Управляемая флэш-память состоит из одного или нескольких NAND-приборов и контроллера, который берет на себя все сложности работы с флэш-памятью и предоставляет примерно такой же аппаратный интерфейс, как у жесткого диска. Достоинство заключается в том, что сложность переносится из драйвера на другой уровень, а конструктор системы освобождается от учета частой смены технологий флэш-памяти. Почти во всех современных смартфонах и планшетах используется eMMC-память, и с большой вероятностью эта тенденция распространится на другие категории встраиваемых устройств.

Жесткие диски используются во встраиваемых системах редко. Заметное исключение составляет запись цифрового видео в абонентских приставках и телевизорах Smart TV, где необходима внешняя память большого объема с малым временем записи.

В любом случае эксплуатационная надежность стоит на первом месте: нам нужно, чтобы устройство могло загрузиться и перейти в рабочее состояние, несмотря на сбои электропитания и неожиданные сбросы. Файловая система должна сохранять работоспособность при таких условиях.

Флэш-память типа NOR

Ячейки NOR-памяти организованы в стираемые блоки размером, к примеру 128 КиБ. Стирание блока означает, что все его биты устанавливаются в 1. Доступ возможен на уровне одного слова (длиной 8, 16 или 32 бит в зависимости от ширины шины данных). Каждый цикл стирания приводит к небольшой деградации ячеек памяти, и после определенного числа циклов стираемый блок теряет надежность и не может больше использоваться. Максимальное число циклов стирания обычно указывается в техническом описании микросхемы и обычно варьируется от 100 тысяч до 1 миллиона.

Данные можно читать по одному слову. Обычно микросхема отображается на адресное пространство процессора, а это значит, что можно непосредственно исполнять код, хранящийся в NOR-памяти. Поэтому NOR-память – подходящее место для размещения кода начального загрузчика, так как не нуждается ни в какой инициализации, кроме фиксированной настройки отображения адресов. SoC-системы с поддержкой NOR-памяти конфигурируются так, что вектор сброса процессора находится в области отображаемых на нее адресов.

Во флэш-памяти можно размещать также ядро и даже корневую файловую систему, это позволяет избежать копирования их в ОЗУ, а значит, открывает возможность создавать устройства с небольшим объемом оперативной памяти. Эта техника, называемая «выполнением на месте» (**eXecute In Place** – **XIP**), очень

специализирована, и я ее рассматривать не буду. В конце главы имеются ссылки на дополнительную литературу.

Для микросхем NOR-памяти разработан стандартный регистровый интерфейс, который называется «стандартный интерфейс флэш-памяти» (**common flash interface** – CFI) и поддерживается всеми современными микросхемами.

NAND-память

NAND-память гораздо дешевле NOR-памяти, а ее объем может быть больше. В микросхемах NAND-памяти первого поколения в каждой ячейке хранился один бит, теперь такая организация называется одноуровневой (**single level cell** – **SLC**). Позже в ячейке стали хранить два бита – многоуровневые ячейки (**multi-level cell** – **MLC**), а теперь уже и три бита – трехуровневые ячейки (**tri-level cell** – **TLC**). С увеличением числа битов в ячейке падает надежность, и для компенсации требуются более сложные контроллеры и программное обеспечение.

Как и NOR-память, NAND-память организована в стираемые блоки размером от 16 до 512 КиБ, и операция стирания также означает установку всех битов в 1. Однако ненадежность наступает после меньшего числа циклов стирания, обычно 1000 циклов для TLC-микросхем и 100 000 – для SLC-микросхем. NAND-память допускает чтение и запись только страницами, обычно размером 2 или 4 КиБ. Поскольку доступ на уровне байтов невозможен, то такую память нельзя отобразить на адресное пространство, и, следовательно, данные нужно предварительно скопировать в ОЗУ.

Во время передачи данных с микросхемы в ОЗУ и обратно возможно инвертирование битов, которое можно обнаружить и исправить с помощью кодов, исправляющих ошибки (ECC). В SLC-микросхемах обычно применяются коды Хэмминга, допускающие эффективную программную реализацию и способные исправлять ошибки в одном бите при чтении страницы. Для MLC- и TLC-микросхем необходимы более сложные коды, например **БЧХ** (код Боуза–Чоудхури–Хоквингема), способный исправлять до 8 битовых ошибок на странице. Для них требуется аппаратная поддержка.

Дополнительную информацию для исправления ошибок необходимо где-то хранить, поэтому на каждой странице имеется запасная (**out of band** – **OOB**) область. В SLC-микросхемах обычно отводят 1 запасной байт на каждые 32 байта основной памяти, т. е. в случае устройства со страницей размером 2 КиБ под запасную область будет отведено 64 байта на каждой странице, а если размер страницы равен 4 КиБ, то 128 байтов. В MLC- и TLC-микросхемах запасная область пропорционально больше, поскольку алгоритмы исправления ошибок сложнее. На рисунке ниже показана организация микросхемы со стираемым блоком размером 128 КиБ и страницей размером 2 КиБ.

В процессе производства проверяются все блоки, и для тех, которые не прошли проверку, поднимается флаг в OOB-области каждой страницы блока. Нередко можно встретить совершенно новые микросхемы, в которых помечено до 2% дефектных блоков. Более того, согласно техническому описанию, примерно такова же доля блоков, дающих ошибку при стирании еще до достижения максимального числа циклов стирания. Драйвер NAND-памяти должен обнаруживать такие ошибки и помечать блок как дефектный.

Помимо памяти для флага дефектного блока и байтов для исправления ошибок, в OOB-области остается еще несколько байтов. В некоторых файловых системах для флэш-памяти эти свободные байты используются для хранения метаданных файловой системы. Поэтому структура OOB-области интересна многим программам, в том числе коду в ПЗУ SoC-системы, начальному загрузчику, драйверу MTD в ядре, коду файловой системы и инструментам для создания образов файловых систем. Но эта часть технологии слабо стандартизована, поэтому легко оказаться в ситуации, когда начальный загрузчик записывает в OOB-область данные в формате, непонятном драйверу MTD. Согласование форматов – ваша обязанность.

Для доступа к микросхемам NAND-памяти необходим контроллер, который обычно является частью SoC-системы. Также нужен соответствующий драйвер в начальном загрузчике и в ядре. Контроллер NAND-памяти управляет аппаратным интерфейсом микросхемы, т. е. передает данные в страницу и из нее и может также включать оборудование для исправления ошибок.

Существует стандартный регистровый интерфейс с микросхемами NAND-памяти – **open NAND flash interface (ONFi)**, поддерживаемый в большинстве современных микросхем. См. http://www.onfi.org.

Управляемая флэш-память

Бремя поддержки флэш-памяти, и особенно NAND, со стороны операционной системы было бы не столь тяжким при наличии точно определенного аппаратного интерфейса и стандартного контроллера флэш-памяти, скрывающего ее сложность. В этом и состоит идея управляемой флэш-памяти, которая получает все большее распространение. По сути дела, она сводится к объединению одной или нескольких микросхем флэш-памяти с микроконтроллером, и в итоге получается идеальное запоминающее устройство с небольшим размером сектора, совместимое с традиционными файловыми системами. Из управляемых микросхем флэш-памяти для встраиваемых систем наиболее важными являются карты **Secure Digital (SD)** и встраиваемый вариант технологии MMC, называемый **eMMC**.

Карты MultiMediaCard и Secure Digital

Технология **MultiMediaCard (MMC)** была предложена в 1997 году компаниями SanDisk и Siemens в форме портативного запоминающего устройства на базе флэш-памяти. Вскоре после этого, в 1999 году, компании SanDisk, Matsushita и Toshiba разработали карту SD на основе MMC с добавлением шифрования и технических средств защиты авторских прав – DRM (отсюда и слово «secure» в названии). Оба вида устройств предназначались для использования в бытовой электронике: цифровых камерах, музыкальных плеерах и т. п. В настоящее время карты SD доминируют на рынке управляемой флэш-памяти для бытовой и встраиваемой электроники, хотя встроенные в них средства шифрования используются редко. В последних редакциях технических характеристик карт SD уменьшен размер физического носителя (mini SD и micro SD, или uSD) и увеличена емкость: SDHC (большой емкости) – до 32 ГБ и SDXC (расширенной емкости) – до 2 ТБ.

Аппаратные интерфейсы карт MMC и SD очень похожи, так что можно вставить полноразмерную карту MMC в разъем для полноразмерной карты SD (но не наоборот). В ранних вариантах использовался 1-разрядный последовательный периферийный интерфейс (Serial Peripheral Interface – **SPI**); позже стал применяться 4-разрядный интерфейс. Существует набор команд для чтения и записи в память 512-байтовыми секторами. Внутри карты имеются микроконтроллер и одна или несколько микросхем NAND-памяти, как показано на рисунке ниже.

Микроконтроллер реализует набор команд и управляет флэш-памятью, выполняя функцию уровня флэш-преобразования, описанного ниже в этой главе. Карты SD заранее отформатированы под файловую систему

FAT: FAT16 для SDSC, FAT32 для SDHC и exFAT для SDXC. Качество микросхем NAND-памяти и программного обеспечения микроконтроллера сильно зависит от конкретной карты. Не очевидно, все ли они достаточно надежны для использования в ответственных встраиваемых системах, но уж, безусловно, не с файловой системой FAT, уязвимой для повреждения файлов. Напомним, что основное применение карты MMC и SD находят в камерах, планшетах и смартфонах.

Карты eMMC

eMMC, или **Embedded MMC** – это просто MMC-память в форме, допускающей напаивание на материнскую плату, с использованием 4- или 8-разрядного интерфейса передачи данных. Однако их назначение – служить внешней памятью для операционной системы, поэтому все компоненты подготовлены к решению данной задачи. Обычно микросхемы поставляются неформатированными.

Другие типы управляемой флэш-памяти

Одной из первых технологий управляемой флэш-памяти была **CompactFlash** (**CF**), в ней было реализовано подмножество интерфейса **PCMCIA** (**Personal Computer Memory Card International Association**). В CF доступ к памяти организован через параллельный интерфейс ATA, т. е. операционная система видит устройство как стандартный жесткий диск. Такие карты часто применяются в одноплатных компьютерах с архитектурой x86, а также в профессиональных видео- и фотокамерах.

Еще один формат, с которым все мы сталкиваемся ежедневно, – флэш-диск с интерфейсом USB. В этом случае доступ к памяти организован через интерфейс USB, а контроллер реализует как спецификацию массового запоминающего USB-устройства, так и уровень флэш-преобразования и интерфейс с одной или несколькими микросхемами флэш-памяти. Протокол массового запоминающего USB-устройства, в свою очередь, основан на наборе команд работы с диском SCSI. Как и карты MMC и SD, флэш-диски обычно форматируются под файловую систему FAT. Во встраиваемых системах они применяются в основном для обмена данными с ПК.

Недавно арсенал управляемой флэш-памяти пополнился универсальными флэш-накопителями (universal flash storage – **UFS**). Как и eMMC, UFS-память изготовлена в виде микросхемы, монтируемой на материнской плате. Она обладает высокоскоростным последовательным интерфейсом и может достигать более высокой скорости передачи данных, чем eMMC. Поддерживает набор команд SCSI.

Доступ к флэш-памяти из начального загрузчика

В главе 3 я упоминал, что начальный загрузчик должен уметь загружать двоичные образы ядра и других компонентов с различных устройств флэш-памяти, а также выполнять такие задачи обслуживания системы, как стирание и перепрограммирование флэш-памяти. Отсюда следует, что в составе начального загрузчика

должны быть драйверы и инфраструктура для поддержки операций чтения, стирания и записи, ориентированные на имеющийся тип памяти: NOR, NAND или управляемую. В примере ниже я буду говорить о U-Boot, но остальные загрузчики в этом отношении похожи.

U-Boot и флэш-память типа NOR

U-Boot включает драйверы для NOR-микросхем с интерфейсом CFI в каталоге `drivers/mtd` и поддерживает команды `erase` для стирания памяти и `cp.b` для побайтового копирования данных с целью программирования флэш-памяти. Предположим, что NOR-память отображена на диапазон адресов 0x40000000– 0x48000000, из которого область длиной 4 МиБ, начинающаяся с адреса 0x40040000, отведена под образ ядра. Тогда для загрузки нового ядра во флэш-память нужно выполнить такие команды:

```
U-Boot# tftpboot 100000 uImage
U-Boot# erase 40040000 403fffff
U-Boot# cp.b 100000 40040000 $(filesize)
```

Здесь команда `tftpboot` присваивает переменной `filesize` значение, равное размеру только что загруженного файла.

U-Boot и флэш-память типа NAND

В случае NAND-памяти необходим драйвер для контроллера NAND-памяти в SoC-системе, он находится в каталоге `drivers/mtd/nand`. Для управления памятью применяется команда `nand` с подкомандами `erase`, `write` и `read`. В примере ниже показано, как загрузить образ ядра в ОЗУ, начиная с адреса 0x82000000, а затем переместить во флэш-память со смещением 0x280000:

```
U-Boot# tftpboot 82000000 uImage
U-Boot# nand erase 280000 400000
U-Boot# nand write 82000000 280000 $(filesize)
```

U-Boot умеет также читать файлы из файловых систем типа JFFS2, YAFFS2 и UBIFS.

U-Boot и карты MMC, SD и eMMC

В U-Boot имеются драйверы для нескольких контроллеров MMC в каталоге `drivers/mmc`. Для доступа к неформатированным данным служат команды `mmc read` и `mmc write` на уровне пользовательского интерфейса, это позволяет работать с образами ядра и файловой системы.

U-boot умеет также читать файлы из файловых систем типа FAT32 и ext4, организованных в MMC-накопителе.

Доступ к флэш-памяти из Linux

Для доступа к неуправляемой флэш-памяти типа NOR и NAND используется подсистема устройств на основе технологий памяти (MTD), предоставляющая базовые

интерфейсы для чтения, стирания и записи блоков флэш-памяти. Для NAND-памяти имеются функции работы с OOB-областью и идентификации дефектных блоков.

Для доступа к управляемой флэш-памяти нужны драйверы для работы с конкретным аппаратным интерфейсом. Для карт MMC/SD и eMMC используется драйвер mmcblk; для CompactFlash и жестких SCSI-дисков – драйвер sd; для флэш-дисков с интерфейсом USB – драйвер usb_storage совместно с драйвером sd.

Устройства на основе технологии памяти

Разработка подсистемы устройств на основе технологии памяти (memory technology devices – **MTD**) была начата Дэвидом Вудхаусом (David Woodhouse) в 1999 году и впоследствии активно развивалась. В этом разделе мы поговорим о том, как в ней обрабатываются две основные технологии флэш-памяти: NOR и NAND.

Подсистема MTD состоит из трех уровней: основной набор функций, набор драйверов для микросхем разных типов и драйверы пользовательского уровня, которые представляют флэш-память в виде символьного или блочного устройства, как показано на рисунке ниже.

Драйверы микросхем находятся на самом нижнем уровне и реализуют интерфейс с микросхемами флэш-памяти. Для NOR-микросхем нужно совсем немного драйверов – только чтобы охватить стандарт CFI и его вариации, а также несколько нестандартных микросхем, которые теперь по большей части устарели. В случае NAND необходим драйвер для контроллера используемой микросхемы, обычно он входит в пакет поддержки платы. В текущей стержневой ветви ядра таких драйверов около 40, все они находятся в каталоге `drivers/mtd/nand`.

Разделы в MTD

В большинстве случаев флэш-память желательно разделить на несколько областей, например для начального загрузчика, образа ядра и корневой файловой

системы. В MTD предусмотрено несколько способов задания размера и местоположения разделов, перечислим основные:

- в командной строке ядра (параметр `CONFIG_MTD_CMDLINE_PARTS`);
- посредством дерева устройств (параметр `CONFIG_MTD_OF_PARTS`);
- с помощью драйвера отображения платформы.

В первом случае в командной строке ядра задается параметр `mtdparts`, который в исходном коде Linux (файл `drivers/mtd/cmdlinepart.c`) определен следующим образом:

```
mtdparts=<mtddef>[;<mtddef]
<mtddef>  := <mtd-id>:<partdef>[,<partdef>]
<mtd-id>  := unique name for the chip
<partdef> := <size>[@<offset>][<name>][ro][lk]
<size>    := размер раздела ИЛИ "-", обозначающий все оставшееся место
<offset>  := смещение начала раздела; оставить пустым, если раздел
             должен начинаться сразу после предыдущего
<name>    := '(' NAME ')'
```

Разобраться в этом поможет пример. Пусть имеется одна микросхема флэш-памяти емкостью 128 МиБ, которую следует разбить на пять разделов. Типичная команда выглядит так:

```
mtdparts=:512k(SPL)ro,780k(U-Boot)ro,128k(U-BootEnv),
4m(Kernel),-(Filesystem)
```

Первое поле, предшествующее двоеточию, – `mtd-id`, оно идентифицирует микросхему либо по номеру, либо по имени, присвоенному в пакете поддержки платы. Если микросхема всего одна, как в данном случае, то поле можно оставить пустым. Если же микросхем несколько, то блоки информации, относящиеся к каждой, отделяются друг от друга точкой с запятой. Затем для каждой микросхемы указывается список разделов через запятую: размер в байтах, килобайтах (`k`) или мегабайтах (`m`) – и имя в скобках. Суффикс `ro` означает, что с точки зрения MTD раздел доступен только для чтения, это часто используется, чтобы предотвратить случайную перезапись начального загрузчика. Вместо размера последнего раздела можно указать минус (`-`), тогда он займет все оставшееся место.

Во время выполнения сводную конфигурацию можно прочитать из файла `/proc/mtd`:

```
# cat /proc/mtd
dev:    size   erasesize   name
mtd0: 00080000 00020000  "SPL"
mtd1: 000C3000 00020000  "U-Boot"
mtd2: 00020000 00020000  "U-BootEnv"
mtd3: 00400000 00020000  "Kernel"
mtd4: 07A9D000 00020000  "Filesystem"
```

Подробная информация о каждом разделе, в том числе о размерах стираемого блока и страницы, имеется в файле `/sys/class/mtd` и представлена в отформатированном виде командой `mtdinfo`:

```
# mtdinfo /dev/mtd0
mtd0
Name:                           SPL
Type:                           nand
Eraseblock size:                131072 bytes, 128.0 KiB
Amount of eraseblocks:          4 (524288 bytes, 512.0 KiB)
Minimum input/output unit size: 2048 bytes
Sub-page size:                  512 bytes
OOB size:                       64 bytes
Character device major/minor:   90:0
Bad blocks are allowed:         true
Device is writable:             false
```

Эквивалентную информацию о разделе можно включить в дерево устройств:

```
nand@0,0 {
  #address-cells = <1>;
  #size-cells = <1>;
  partition@0 {
    label = "SPL";
    reg = <0 0x80000>;
  };
  partition@80000 {
    label = "U-Boot";
    reg = <0x80000 0xc3000>;
  };
  partition@143000 {
    label = "U-BootEnv";
    reg = <0x143000 0x20000>;
  };
  partition@163000 {
    label = "Kernel";
    reg = <0x163000 0x400000>;
  };
  partition@563000 {
    label = "Filesystem";
    reg = <0x563000 0x7a9d000>;
  };
};
```

Третий способ – представить информацию о разделе в виде платформенных данных в структуре `mtd_partition`, как показано в следующем примере, взятом из файла `arch/arm/mach-omap2/board-omap3beagle.c` (константа `NAND_BLOCK_SIZE` определена в другом месте как 128K):

```
static struct mtd_partition omap3beagle_nand_partitions[] = {
  {
    .name = "X-Loader",
    .offset = 0,
    .size = 4 * NAND_BLOCK_SIZE,
```

```
    .mask_flags = MTD_WRITEABLE, /* force read-only */
  },
  {
    .name = "U-Boot",
    .offset = 0x80000;
    .size = 15 * NAND_BLOCK_SIZE,
    .mask_flags = MTD_WRITEABLE, /* force read-only */
  },
  {
    .name = "U-Boot Env",
    .offset = 0x260000;
    .size = 1 * NAND_BLOCK_SIZE,
  },
  {
    .name = "Kernel",
    .offset = 0x280000;
    .size = 32 * NAND_BLOCK_SIZE,
  },
  {
    .name = "File System",
    .offset = 0x680000;
    .size = MTDPART_SIZ_FULL,
  },
};
```

Драйверы устройств в MTD

На верхнем уровне подсистемы MTD находятся два драйвера устройств.

- Драйвер символьного устройства со старшим номером 90. Для каждого раздела с номером `N` существуют два узла устройств: `/dev/mtdN` (младший номер = N*2) и `/dev/mtdNro` (младший номер = N*2 + 1). Второй из них – вариант первого, доступный только для чтения.
- Драйвер блочного устройства со старшим номером 31 и младшим номером `N`. Узлы устройств имеют вид `/dev/mtdblockN`.

Символьное устройство mtd

Символьные устройства – самые важные: они позволяют обращаться к флэш-памяти как к массиву байтов, т. е. читать и записывать (программу). Кроме того, драйвер реализует несколько функций `ioctl`, которые дают возможность стирать блоки и управлять OOB-областью в NAND-микросхемах. Приведенный ниже список функций взят из файла `include/uapi/mtd/mtd-abi.h`:

IOCTL	Описание
`MEMGETINFO`	Получить основные характеристики MTD
`MEMERASE`	Стереть блоки в разделе MTD
`MEMWRITEOOB`	Записать данные в OOB-область страницы
`MEMREADOOB`	Прочитать данные из OOB-области страницы

IOCTL	Описание
MEMLOCK	Заблокировать микросхему (если поддерживается)
MEMUNLOCK	Разблокировать микросхему (если поддерживается)
MEMGETREGIONCOUNT	Получить количество стираемых регионов: отлично от 0, если в разделе имеются стираемые блоки разного размера, что типично для NOR-приборов, но редко встречается в NAND-приборах
MEMGETREGIONINFO	Если MEMGETREGIONCOUNT не равно 0, то эту функцию можно использовать, чтобы получить смещение, размер и количество блоков в стираемом регионе
MEMGETOOBSEL	Не поддерживается
MEMGETBADBLOCK	Получить флаг дефектного блока
MEMSETBADBLOCK	Установить флаг дефектного блока
OTPSELECT	Установить режим OTP (однократно программируемый), если микросхема его поддерживает
OTPGETREGIONCOUNT	Получить количество OTP-регионов
OTPGETREGIONINFO	Получить информацию об OTP-регионе
ECCGETLAYOUT	Не поддерживается

Существует набор утилит `mtd-utils` для манипулирования флэш-памятью, в котором используются эти функции `ioctl`. Их исходный код можно получить из репозитория http://git.infradead.org/mtd-utils.git, он доступен также в качестве пакета в проектах Yocto Project и Buildroot. Наиболее важные утилиты перечислены в списке ниже. Пакет включает также утилиты для работы с файловыми системами JFFS2 и UBI/UBIFS, которые будут рассмотрены ниже. Для каждой утилиты одним из параметров является символьное устройство:

- **flash_erase**: стереть диапазон блоков;
- **flash_lock**: заблокировать диапазон блоков;
- **flash_unlock**: разблокировать диапазон блоков;
- **nanddump**: получить дамп NAND-памяти, факультативно включив OOB-область. Дефектные блоки пропускаются;
- **nandtest**: тестирование и диагностика NAND-памяти;
- **nandwrite**: записать (запрограммировать) данные из файла в NAND-память, пропуская дефектные блоки.

Перед записью во флэш-память ее обязательно нужно стирать. Для этой цели предназначена утилита `flash_erase`.

Для программирования NOR-памяти нужно просто скопировать байты на MTD-устройство командой `cp` или ей подобной.

К сожалению, для NAND-памяти так не получится, потому что копирование завершится с ошибкой на первом же дефектном блоке. Вместо этого нужно использовать команду `nandwrite`, которая пропускает дефектные блоки. А для чтения NAND-памяти предназначена команда `nanddump`, также пропускающая дефектные блоки.

Блочное устройство mtdblock

Драйвер mtdblock используется редко. Его назначение – представить флэш-память в виде блочного устройства, которое можно форматировать и монтировать

как файловую систему. Однако у него имеется ряд серьезных ограничений: он не умеет обрабатывать дефектные блоки NAND-памяти, не производит выравнивания износа, не учитывает различия в размерах блока файловой системы и стираемого блока. Иными словами, в нем отсутствует уровень флэш-преобразования, крайне важный для организации надежного файлового хранилища. Единственная ситуация, в которой драйвер mtdblock полезен, – монтирование защищенных от записи файловых систем типа Squashfs поверх надежной флэш-памяти, например NOR.

Чтобы получить защищенную от записи файловую систему в NAND-памяти, следует использовать драйвер UBI, описанный ниже в этой главе.

Запись сообщений об ошибках ядра в журнал на MTD-устройстве

Ошибки ядра, или oops'ы, обычно протоколируются с помощью демонов `klogd` и `syslogd` в кольцевом буфере или в файле. После перезагрузки журнал, хранящийся в кольцевом буфере, теряется, но даже при хранении в файле может случиться, что файл не сброшен на диск перед крахом системы.

Надежнее записывать информацию об ошибках и панике ядра в раздел MTD-устройства, организованный как кольцевой буфер. Этот режим включается, если задать конфигурационный параметр `CONFIG_MTD_OOPS` и добавить параметр `console=ttyMTDN` в командную строку ядра (здесь `N` – номер MTD-устройства, на которое записываются сообщения).

Эмуляция NAND-памяти

Эмулятор NAND имитирует микросхему NAND с помощью системной оперативной памяти. Его основное назначение – тестирование кода, который должен работать с NAND-памятью, не имея доступа к физической NAND-памяти. В частности, он способен эмулировать дефектные блоки, инвертирование разрядов и другие ошибки, которые нужно протестировать, но трудно воспроизвести при работе с реальной флэш-памятью. Для получения дополнительных сведений лучше всего заглянуть в исходный код, где подробно описаны все способы конфигурирования драйвера. Этот код находится в файле `drivers/mtd/nand/nandsim.c`. Он активируется заданием конфигурационного параметра ядра `CONFIG_MTD_NAND_NANDSIM`.

Драйвер блочного устройства MMC

Доступ к картам MMC/SD и микросхемам eMMC опосредован драйвером блочного устройства mmcblk. Необходимо, чтобы хост-контроллер соответствовал используемому адаптеру MMC, который входит в состав пакета поддержки платы. Драйверы находятся в каталоге `drivers/mmc/host` исходного кода Linux.

MMC-накопитель имеет точно такую же таблицу разделов, как жесткие диски, и разбивается на разделы с помощью утилиты fdisk или ей подобной.

Файловые системы для флэш-памяти

Для эффективного использования флэш-памяти в качестве массового запоминающего устройства нужно разрешить несколько проблем: несоответствие между размерами стираемого блока и сектора диска, ограниченное количество циклов стирания для каждого стираемого блока, необходимость учитывать дефектные блоки при работе с NAND-микросхемами. Эти различия разрешаются уровнем флэш-преобразования (Flash Translation Layer – **FTL**).

Уровень флэш-преобразования

Уровень флэш-преобразования реализует следующие функции.

- **Подвыделение.** Файловая система работает оптимально, когда единица выделения мала, обычно это сектор длиной 512 байтов. Это гораздо меньше, чем стираемый блок флэш-памяти размером 128 КиБ и более. Поэтому стираемые блоки необходимо разбить на меньшие, чтобы место в памяти не расходовалось впустую.
- **Сборка мусора.** Следствием подвыделения является тот факт, что в процессе работы файловой системы в стираемых блоках накапливаются как актуальные, так и устаревшие данные. Поскольку освободить можно только стираемый блок целиком, единственный способ вернуть неиспользуемое пространство в оборот – перенести актуальные данные в другое место и поместить пустой стираемый в список свободных. Этот процесс называется сборкой мусора и обычно реализуется фоновым потоком.
- **Выравнивание износа.** Для каждого блока определено максимальное число циклов стирания. Чтобы продлить срок службы микросхемы, важно перемещать данные, так чтобы каждый блок стирался примерно одинаковое число раз.
- **Обработка дефектных блоков.** При работе с NAND-микросхемами следует избегать использования блоков, помеченных как дефектные, и помечать хорошие блоки, если их не удалось стереть.
- **Эксплуатационная надежность.** Отключение от питания или сброс встраиваемых устройств может производиться без предупреждения, поэтому файловая система должна выдерживать такие события без повреждения данных. Обычно для этой цели используется журнал транзакций.

Существует несколько вариантов размещения уровня флэш-преобразования.

- **В файловой системе.** Так делается в файловых системах JFFS2, YAFFS2 и UBIFS.
- **В драйвере блочного устройства.** Драйвер UBI, лежащий в основе файловой системы UBIFS, реализует некоторые функции уровня флэш-преобразования.
- **В контроллере устройства.** Так делается в управляемых флэш-устройствах.

Если уровень флэш-преобразования размещен в файловой системе или в драйвере блочного устройства, то его код является частью ядра и, следова-

тельно, открыт, т. е. мы можем посмотреть, как он работает, и надеяться, что со временем он будет становиться лучше. Если же этот уровень реализован внутри управляемого флэш-устройства, то он скрыт от нас, и мы не можем проверить, так ли он работает, как мы хотим, или нет. Хуже того, размещение уровня FTL в дисковом контроллере означает, что ему недоступна информация, хранящаяся на уровне файловой системы, в частности о том, какие сектора принадлежат уже удаленным файлам и, следовательно, не содержат полезных данных. Эта проблема решается путем добавления команд для передачи информации между файловой системой и устройством, которые я опишу ниже в разделе, посвященном команде `TRIM`. Но вопрос о видимости кода по-прежнему остается. Если вы собираетесь использовать управляемую флэш-память, то выбирайте производителя, которому доверяете.

Файловые системы для флэш-памяти типа NOR и NAND

Чтобы использовать «голые» микросхемы флэш-памяти в качестве массового запоминающего устройства, нужна файловая система, учитывающая особенности технологии. Таких файловых систем три:

- **Journaling Flash File System 2, JFFS2.** Исторически первая файловая система для флэш-накопителей в Linux, но используется до сих пор. Работает как для NOR-, так и для NAND-памяти, но монтируется долго.
- **Yet Another Flash File System 2, YAFFS2.** Похожа на JFFS2, но разработана специально для NAND-накопителей. Принята Google в качестве предпочтительной файловой системы для устройств на платформе Android.
- **Unsorted Block Image File System, UBIFS.** Новейшая файловая система, работающая с NOR- и NAND-накопителями, используется в сочетании с драйвером блочного устройства UBI. Как правило, обеспечивает более высокую производительность, по сравнению с JFFS2 и YAFFS2, рекомендуется использовать в новых проектах.

Все они используют подсистему MTD в качестве единого интерфейса к флэш-памяти.

JFFS2

Файловая система Journaling Flash File System впервые появилась в программном обеспечении для сетевой камеры Axis 2100 в 1999 году. Много лет она была единственной файловой системой для флэш-накопителей в Linux и установлена на многих тысячах устройств разных типов. Сегодня это не лучший выбор, но я все же расскажу о ней, потому что она стоит в начале пути развития.

JFFS2 – журнально-структурированная файловая система, в которой для доступа к флэш-памяти используется подсистема MTD. В такой системе изменения записываются во флэш-память последовательно в виде узлов. Узел может содержать изменение каталога, например создание или удаление имен файлов, или из-

менение данных в файле. Со временем информация в одном узле может быть замещена информацией в последующих узлах, и тогда узел становится устаревшим.

Определены три типа стираемых блоков:

- **свободный**: вообще не содержит узлов;
- **чистый**: содержит только актуальные узлы;
- **измененный**: содержит хотя бы один устаревший узел.

В каждый момент времени обновления записываются только в один блок, который называется открытым. В случае выключения электропитания или сброса системы потерян будет только результат последней операции записи в открытый блок. Кроме того, при записи узлы сжимаются, что повышает эффективность хранения во флэш-памяти – вещь немаловажная, когда речь идет о дорогой памяти типа NOR.

Когда число свободных блоков оказывается ниже порогового значения, запускается сборщик мусора – поток ядра, который ищет измененные блоки, копирует находящиеся в них актуальные узлы в открытый блок, а затем делает измененный блок свободным.

Попутно сборщик мусора реализует примитивный вариант выравнивания износа, так как переносит актуальные данные из одного блока в другой. Поскольку в качестве места назначения выбирается открытый блок, каждый блок будет стираться примерно одинаковое число раз при условии, что данные, которые он содержит, время от времени изменяются. Иногда сборщик мусора выбирает чистый блок, чтобы блоки, содержащие статические, редко изменяемые данные тоже подвергались выравниванию износа.

В файловой системе JFFS2 имеется кэш со сквозной записью; это означает, что данные записываются во флэш-память синхронно, как если бы файловая система была смонтирована с флагом `-o sync`. Это повышает надежность, но вместе с тем увеличивает время записи данных. Проблема осложняется, когда записывается небольшая порция данных: если размер данных сравним с размером заголовка узла (40 байтов), то накладные расходы оказываются чрезмерно велики. Распространенный крайний случай – файлы журналов, формируемые, скажем, демоном syslogd.

Сводные узлы

У JFFS2 есть один важный недостаток: поскольку на кристалле нет индекса, структуру каталогов приходится вычислять на этапе монтирования, читая журнал от начала до конца. В конце просмотра мы имеем полное представление об актуальных узлах в каталоге, но необходимое для этого время пропорционально размеру раздела. Нередко приходится видеть, что время монтирования составляет порядка одной секунды на мегабайт, т. е. общее время варьируется от десятков до сотен секунд.

Чтобы уменьшить время просмотра на этапе монтирования, в версии Linux 2.6.15 были введены сводные узлы. Сводный узел (summary node) записывается в конец открытого стираемого блока непосредственно перед его закрытием. Он содержит всю информацию, необходимую для просмотра на этапе монтирования, вследствие чего уменьшается объем просматриваемых данных. Сводные узлы позволяют уменьшить время монтирования в 2–5 раз ценой примерно пятипро-

центного перерасхода объема используемой памяти. Этот режим включается при установке конфигурационного параметра ядра CONFIG_JFFS2_SUMMARY.

Маркер очистки

Стертый блок, в котором все биты равны 1, неотличим от блока, в который записаны одни единицы, однако в последнем случае ячейки памяти не были регенерированы, и, следовательно, их нельзя перепрограммировать до стирания. В JFFS2 используется механизм маркеров очистки (clean markers), позволяющий различить эти случаи. После успешного стирания блока записывается маркер очистки – либо в начале блока, либо в OOB-области первой страницы блока. Если маркер очистки присутствует, значит, блок чистый.

Создание файловой системы JFFS2

Для создания пустой файловой системы JFFS2 на этапе выполнения достаточно стереть раздел MTD, поставив маркеры очистки, а затем смонтировать его. Форматировать ничего не нужно, потому что пустая файловая система JFFS2 содержит только свободные блоки. Например, для создания JFFS2 в разделе 6 MTD-устройства следует выполнить на устройстве такие команды:

```
# flash_erase -j /dev/mtd6 0 0
# mount -t jffs2 mtd6 /mnt
```

Флаг -j команды flash_erase расставляет маркеры очистки, а монтирование с указанием типа jffs2 представляет раздел как пустую файловую систему. Отметим, что монтируемое устройство задано в виде mtd6, а не /dev/mtd6. Можно было бы также указать узел блочного устройства /dev/mtdblock6. Такая вот особенность у JFFS2. Смонтированную файловую систему можно использовать как обычно: при следующей загрузке и монтировании файлы никуда не денутся.

Создать образ файловой системы можно прямо по области технологической подготовки на исходной машине, воспользовавшись командой mkfs.jffs2, чтобы записать файлы в формате JFFS2, и командой sumtool, чтобы добавить сводные узлы. Обе команды входят в пакет mtd-utils.

Например, создадим образ, содержащий файлы из rootfs, для флэш-накопителя типа NAND с размером стираемого блока 128 КиБ (0x20000), включив сводные узлы:

```
$ mkfs.jffs2 -n -e 0x20000 -p -d ~/rootfs -o ~/rootfs.jffs2
$ sumtool -n -e 0x20000 -p -i ~/rootfs.jffs2 -o ~/rootfs-sum.jffs2
```

Флаг -p означает, что файл образа нужно дополнить до целого числа стираемых блоков. Флаг -n подавляет создание маркеров очистки в образе, это нормально для NAND-приборов, поскольку маркер очистки находится в OOB-области. Для NOR-приборов флаг -n следует опустить. В команде mkfs.jffs2 можно с помощью флага -D [device table] указать таблицу устройств, что позволит задать для каждого файла владельца и права доступа. Разумеется, Buildroot и Yocto Project все это делают за вас.

Записать образ во флэш-память позволит начальный загрузчик. Например, если образ файловой системы загружен в ОЗУ начиная с адреса 0x82000000 и мы хотим загрузить его в раздел флэш-накопителя со смещением 0x163000 байтов от начала флэш-памяти и длиной 0x7a9d000 байтов, то нужно будет выполнить такие команды U-Boot:

```
nand erase clean 163000 7a9d000
nand write 82000000 163000 7a9d000
```

То же самое можно сделать из Linux с помощью драйвера mtd:

```
# flash_erase -j /dev/mtd6 0 0
# nandwrite /dev/mtd6 rootfs-sum.jffs2
```

Чтобы загрузиться с корневой файловой системы JFFS2, нужно указать соответствующий узел устройства `mtdblock` в командной строке ядра, а также тип корневой файловой системы, потому что автоматически JFFS2 не распознается:

```
root=/dev/mtdblock6 rootfstype=jffs2
```

YAFFS2

Файловую систему YAFFS написал Чарльз Мэннинг (Charles Manning) в 2001 году специально для поддержки NAND-накопителей, когда JFFS2 еще не умела этого делать. Впоследствии были внесены изменения для работы со страницами увеличенного размера (2 КиБ), и новая версия получила название YAFFS2. Файловой системе YAFFS посвящен сайт http://www.yaffs.net.

YAFFS – журнально-структурированная система, построенная на основе тех же принципов, что и JFFS2. Но благодаря другим проектным решениям она быстрее выполняет просмотр на этапе монтирования, обладает более простым и быстрым сборщиком мусора и не поддерживает сжатия, что ускоряет чтение и запись ценой менее эффективного использования места.

YAFFS не ограничивается Linux; она была перенесена и на другие операционные системы. Распространяется по двойной схеме лицензирования: GPLv2 для совместимости с Linux и коммерческая лицензия для других операционных систем. К сожалению, код YAFFS так никогда и не был включен в стержневую ветвь Linux, поэтому вам придется наложить на ядро заплату, как показано ниже:

```
$ git clone git://www.aleph1.co.uk/yaffs2
$ cd yaffs2
$ ./patch-ker.sh c m <path to your link source>
```

Затем сконфигурируйте ядро, задав параметр `CONFIG_YAFFS_YAFFS2`.

Создание файловой системы YAFFS2

Как и в случае JFFS2, для создания файловой системы YAFFS2 на этапе выполнения нужно лишь стереть раздел и смонтировать его, однако на этот раз записывать маркеры очистки не следует:

```
# flash_erase /dev/mtd/mtd6 0 0
# mount -t yaffs2 /dev/mtdblock6 /mnt
```

Для создания образа файловой системы проще всего воспользоваться программой mkyaffs2, доступной по адресу https://code.google.com/p/yaffs2utils:

```
$ mkyaffs2 -c 2048 -s 64 rootfs rootfs.yaffs2
```

Здесь флаг -c задает размер страницы, а -s – размер OOB-области. В составе кода YAFFS имеется программа mkyaffs2image, но у нее есть два недостатка. Во-первых, размеры страницы и OOB-области зашиты в код: если для вашего устройства они отличаются от 2048 и 64 соответственно, то придется отредактировать и перекомпилировать программу. Во-вторых, структура OOB-области несовместима с MTD: в MTD для маркера дефектного блока зарезервированы первые два байта, а в mkyaffs2image эти байты используются для хранения метаданных YAFFS.

Для копирования образа в MTD-раздел из Linux выполните следующие команды:

```
# flash_erase /dev/mtd6 0 0
# nandwrite -a /dev/mtd6 rootfs.yaffs2
```

Чтобы загрузиться из корневой файловой системы YAFFS2, добавьте в командную строку ядра такие параметры:

```
root=/dev/mtdblock6 rootfstype=yaffs2
```

UBI и UBIFS

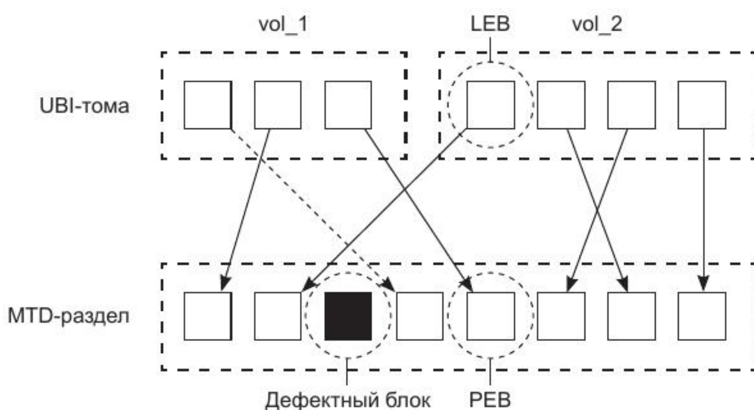
Драйвер образа с несортированными блоками (unsorted block image – **UBI**) представляет собой диспетчер тома для флэш-накопителей, он берет на себя обработку дефектных блоков и выравнивание износа. Написан Артемом Битюцким и впервые был включен в ядро Linux 2.6.22. Параллельно программисты из Nokia работали над файловой системой, которая пользовалась возможностями UBI. Эта файловая система была названа UBIFS и появилась в версии Linux 2.6.27. Подобное разделение уровня флэш-преобразования делает код более модульным и позволяет другим файловым системам также воспользоваться драйвером UBI, что мы и увидим ниже.

UBI

UBI дает идеализированное представление о микросхеме флэш-памяти как о надежном устройстве, отображая физические стираемые блоки (physical erase blocks – **PEB**) на логические стираемые блоки (logical erase blocks – **LEB**). Дефектные блоки не отображаются на LEB и, следовательно, никогда не используются. Если блок невозможно стереть, то он помечается как дефектный и исключается из отображения. В заголовке логического стираемого блока, соответствующего физическому, UBI запоминает, сколько раз этот физический блок стирался, и изменяет отображение, так чтобы все физические блоки стирались одинаковое число раз.

UBI получает доступ к флэш-памяти посредством подсистемы MTD. В качестве дополнительного бонуса он может разбить MTD-раздел на несколько UBI-томов,

что повышает качество выравнивания износа следующим образом. Представьте, что есть две файловые системы: в одной данные относительно статические, как, например, в корневой файловой системе, а в другой постоянно изменяются. Если бы они хранились в разных MTD-разделах, то выравнивание износа оказывало бы влияние только на второй. А если мы разместим их в двух UBI-томах, находящихся в одном MTD-разделе, то выравнивание износа распространится на обе области накопителя, поэтому срок службы флэш-памяти увеличится. Эта ситуация иллюстрируется на рисунке ниже.



Таким образом, драйвер UBI удовлетворяет два требования к уровню флэш-преобразования: выравнивание износа и обработка дефектных блоков.

Для подготовки MTD-раздела к работе с UBI используется не утилита `flash_erase`, как в случае JFFS2 и YAFFS2, а `ubiformat`, которая сохраняется счетчики стираний, хранящиеся в заголовках PEB. Утилита `ubiformat` должна знать минимальную единицу ввода-вывода, которая для большинства микросхем NAND-памяти совпадает с размером страницы, однако некоторые микросхемы производят чтение и запись подстраницами, размер которых равен половине или четверти размера страницы. Справьтесь с техническим описанием микросхемы, а если сомневаетесь, указывайте размер страницы. В примере ниже устройство `mtd6` подготавливается в предположении, что размер страницы равен 2048 байтов:

```
# ubiformat /dev/mtd6 -s 2048
```

Для присоединения драйвера UBI к подготовленному таким образом MTD-разделу служит команда `ubiattach`:

```
# ubiattach -p /dev/mtd6 -O 2048
```

В результате создается узел устройства `/dev/ubi0`, с помощью которого мы можем обращаться к UBI-томам. Команду `ubiattach` можно использовать и для других MTD-разделов, тогда они будут доступны как `/dev/ubi1`, `/dev/ubi2` и т. д.

Отображение между PEB и LEB загружается в память на этапе присоединения, и это занимает время, пропорциональное числу физических блоков, обычно несколько секунд. В версию Linux 3.7 добавлена новая функция – быстрое отображение UBI (UBI fastmap), – которая периодически сохраняет текущее состояние отображения во флэш-памяти и тем самым уменьшает время присоединения. Соответствующий конфигурационный параметр ядра называется CONFIG_MTD_UBI_FASTMAP.

При первом после выполнения ubiformat присоединении к MTD-разделу томов еще нет. Для создания томов служит команда ubimkvol. Пусть, например, имеется MTD-раздел размером 128 МиБ, и мы хотим разбить его на два тома размером 32 и 96 МиБ. В предположении, что размер стираемого блока равен 128 КиБ, а размер страницы – 2 КиБ, нужно выполнить такие команды:

```
# ubimkvol /dev/ubi0 -N vol_1 -s 32MiB
# ubimkvol /dev/ubi0 -N vol_2 -s 96MiB
```

В результате появятся узлы устройств /dev/ubi0_0 и /dev/ubi0_1. Убедиться в этом позволит команда ubinfo:

```
# ubinfo -a /dev/ubi0
ubi0
Volumes count:                           2
Logical eraseblock size:                 15360 bytes, 15.0 KiB
Total amount of logical eraseblocks:     8192 (125829120 bytes, 120.0 MiB)
Amount of available logical eraseblocks: 0 (0 bytes)
Maximum count of volumes                 89
Count of bad physical eraseblocks:       0
Count of reserved physical eraseblocks:  160
Current maximum erase counter value:     1
Minimum input/output unit size:          512 bytes
Character device major/minor:            250:0
Present volumes:                         0, 1
Volume ID:  0 (on ubi0)
Type:       dynamic
Alignment:  1
Size:       2185 LEBs (33561600 bytes, 32.0 MiB)
State:      OK
Name:       vol_1
Character device major/minor: 250:1
-----------------------------------
Volume ID:  1 (on ubi0)
Type:       dynamic
Alignment:  1
Size:       5843 LEBs (89748480 bytes, 85.6 MiB)
State:      OK
Name:       vol_2
Character device major/minor: 250:2
```

Отметим, что поскольку у каждого LEB имеется заголовок, содержащий метаданные, необходимые драйверу UBI, размер LEB на одну страницу меньше, чем PEB. Например, для микросхемы с размером PEB 128 КиБ и страницами размером 2 КиБ размер LEB будет равен 126 КиБ. Это важная информация, которую следует иметь в виду при создании образа UBIFS.

UBIFS

В файловой системе UBIFS для повышения эксплуатационной надежности используется UBI-том, к которому она добавляет подвыделение и сборку мусора, чтобы удовлетворить оставшимся требованиям к уровню флэш-преобразования. В отличие от JFFS2 и YAFFS2, на кристалле хранится индексная информация, поэтому монтирование производится быстро, но не забывайте, что до монтирования производится операция присоединения UBI-тома, которая может занимать довольно много времени. UBIFS поддерживает также механизм кэширования с отложенной записью, как и обычная дисковая файловая система, а это означает, что операции записи производятся гораздо быстрее, но возникает типичная проблема потенциальной потери данных, которые не были сброшены из кэша во флэш-память в момент выключения питания. Эту проблему можно решить путем аккуратного использования функций `fsync(2)` и `fdatasync(2)` для принудительного сброса данных в критически важных точках программы.

UBIFS ведет журнал для быстрого восстановления после выключения питания. Для журнала требуется место, обычно 4 МиБ и больше, поэтому UBIFS не годится для флэш-накопителей очень малого объема.

Созданный UBI-том можно смонтировать, указав соответствующий узел устройства, `/dev/ubi0_0`, или узел устройства для всего раздела, уточненный именем тома:

```
# mount -t ubifs ubi0:vol_1 /mnt
```

Создание образа файловой системы UBIFS – двухшаговый процесс: сначала создается образ UBIFS командой `mkfs.ubifs`, а затем он вкладывается в UBI-том командой `ubinize`.

Команде `mkfs.ubifs` нужно сообщить размер страницы с помощью флага `-m`, размер логического стираемого блока UBI с помощью флага `-e` (не забыв, что LEB обычно на одну страницу короче PEB) и максимальное число стираемых блоков в томе с помощью флага `-c`. Если размер первого тома равен 32 МиБ, а размер стираемого блока – 128 КиБ, то число стираемых блоков равно 256. Поэтому, чтобы создать образ UBIFS с именем `rootfs.ubi`, поместив в него все содержимое каталога `rootfs`, нужно выполнить такую команду:

```
$ mkfs.ubifs -r rootfs -m 2048 -e 126KiB -c 256 -o rootfs.ubi
```

На втором шаге понадобится создать конфигурационный файл для команды `ubinize`, описав в нем характеристики каждого тома в образе. В справке (`ubinize h`) приведены подробные сведения о формате. В примере ниже создаются два тома:

vol_1 и vol_2:

```
[ubifsi_vol_1]
mode=ubi
image=rootfs.ubi
vol_id=0
vol_name=vol_1
vol_size=32MiB
vol_type=dynamic

[ubifsi_vol_2]
mode=ubi
image=data.ubi
vol_id=1
vol_name=vol_2
vol_type=dynamic
vol_flags=autoresize
```

Для второго тома поднят флаг автоматического изменения размера (autoresize), поэтому он займет все оставшееся место в MTD-разделе. Этот флаг может быть поднят только для одного тома. Пользуясь этой информацией, ubinize создает файл образа, имя которого задано флагом -o, размер PEB – флагом -p, размер страницы – флагом -m, а размер подстраницы – флагом -s:

```
$ ubinize -o ~/ubi.img -p 128KiB -m 2048 -s 512 ubinize.cfg
```

Для установки этого образа в целевую систему нужно выполнить в ней следующие команды:

```
# ubiformat /dev/mtd6 -s 2048
# nandwrite /dev/mtd6 /ubi.img
# ubiattach -p /dev/mtd6 -O 2048
```

Чтобы загрузиться из корневой файловой системы UBIFS, добавьте в командную строку ядра такие параметры:

```
ubi.mtd=6 root=ubi0:vol_1 rootfstype=ubifs
```

Файловые системы для управляемой флэш-памяти

Поскольку тенденция к переходу на технологии управляемой флэш-памяти, особенно eMMC, набирает силу, нужно рассмотреть вопрос о ее эффективном использовании. На первый взгляд, управляемые флэш-накопители обладают такими же характеристиками, как жесткие диски, но у некоторых микросхем NAND-памяти есть ограничения: большие стираемые блоки, ограниченное число циклов стирания и учет дефектных блоков. И, разумеется, нам нужна надежность в случае выключения питания.

В принципе, можно использовать любую из традиционных дисковых файловых систем, но желательно выбрать такую, которая минимизирует количество опера-

ций записи на диск и быстро восстанавливается после неожиданной остановки; последнее обычно обеспечивается журналом.

Flashbench

Для оптимального использования флэш-памяти необходимо знать размер стираемого блока и размер страницы. Производители обычно не публикуют этих данных, но их можно определить, наблюдая за поведением микросхемы или карты.

Одним из таких инструментов является программа Flashbench. Автор первоначальной версии – Арнд Бергман (Arnd Bergman), как следует из статьи на сайте LWN по адресу http://lwn.net/Articles/428584. Получить исходный код можно по адресу https://github.com/bradfa/flashbench.

Вот результат типичного прогона программы для карты SanDisk GiB SDHC:

```
$ sudo ./flashbench -a /dev/mmcblk0 --blocksize=1024
align 536870912 pre 4.38ms  on 4.48ms   post 3.92ms  diff 332µs
align 268435456 pre 4.86ms  on 4.9ms    post 4.48ms  diff 227µs
align 134217728 pre 4.57ms  on 5.99ms   post 5.12ms  diff 1.15ms
align 67108864  pre 4.95ms  on 5.03ms   post 4.54ms  diff 292µs
align 33554432  pre 5.46ms  on 5.48ms   post 4.58ms  diff 462µs
align 16777216  pre 3.16ms  on 3.28ms   post 2.52ms  diff 446µs
align 8388608   pre 3.89ms  on 4.1ms    post 3.07ms  diff 622µs
align 4194304   pre 4.01ms  on 4.89ms   post 3.9ms   diff 940µs
align 2097152   pre 3.55ms  on 4.42ms   post 3.46ms  diff 917µs
align 1048576   pre 4.19ms  on 5.02ms   post 4.09ms  diff 876µs
align 524288    pre 3.83ms  on 4.55ms   post 3.65ms  diff 805µs
align 262144    pre 3.95ms  on 4.25ms   post 3.57ms  diff 485µs
align 131072    pre 4.2ms   on 4.25ms   post 3.58ms  diff 362µs
align 65536     pre 3.89ms  on 4.24ms   post 3.57ms  diff 511µs
align 32768     pre 3.94ms  on 4.28ms   post 3.6ms   diff 502µs
align 16384     pre 4.82ms  on 4.86ms   post 4.17ms  diff 372µs
align 8192      pre 4.81ms  on 4.83ms   post 4.16ms  diff 349µs
align 4096      pre 4.16ms  on 4.21ms   post 4.16ms  diff 52.4µs
align 2048      pre 4.16ms  on 4.16ms   post 4.17ms  diff 9ns
```

Flashbench читает блоки, размер которых в данном случае равен 1024 байта, расположенные непосредственно до и непосредственно после смещения, равного степени двойки. Если блок пересекает границу страницы или стираемого блока, то операция чтения занимает больше времени. В самом правом столбце показана разность двух чисел, она-то и интересна. Если смотреть снизу вверх, то самый большой скачок имеет место при 4 КиБ, это и есть наиболее вероятный размер страницы. Второй скачок – от 52,4 мкс до 349 мкс – имеет место при 8 КиБ. Это типичный результат, он показывает, что в карте реализован многоплановый доступ, т. е. она может читать две страницы по 4 КиБ одновременно. Дальше различия не так сильно выражены, но имеется очевидный скачок от 485 мкс до 805 мкс при 512 КиБ, это, вероятно, размер стираемого блока. Учитывая, что тестируемая карта довольно старая, как раз таких чисел и следовало ожидать.

Discard и TRIM

Обычно при удалении файла в файловую систему записывается только узел модифицированного каталога, а сектора, содержащие данные файла, не изменяются. Если уровень флэш-преобразования реализован в дисковом контроллере, как в случае управляемой флэш-памяти, то он не знает, что эта группа секторов содержит уже ненужные данные, поэтому продолжает их копировать.

В последние годы ситуация улучшилась благодаря добавлению транзакций, в которых информация об удаленных секторах передается от операционной системы контроллеру. В спецификациях SCSI и SATA описана команда `TRIM`, а в спецификации MMC имеется аналогичная команда `ERASE`. В Linux эта функция называется `discard`.

Чтобы можно было воспользоваться `discard`, ее должны поддерживать запоминающее устройство (большинство современных микросхем eMMC поддерживают) и его драйвер в Linux. Проверить, так ли это, можно, посмотрев на параметры системной очереди блочного устройства в `/sys/block/<блочное устройство>/queue/`. Интерес представляют следующие:

- `discard_granularity`: размер внутренней единицы выделения устройства;
- `discard_max_bytes`: какое максимальное число байтов можно отбросить за одну операцию;
- `discard_zeroes_data`: если равен 1, то отброшенные данные обнуляются.

Если устройство или драйвер не поддерживает функцию `discard`, то все эти значения равны 0. Вот какие параметры установлены для микросхемы eMMC на плате BeagleBone Black:

```
# grep -s "" /sys/block/mmcblk0/queue/discard_*
/sys/block/mmcblk0/queue/discard_granularity:2097152
/sys/block/mmcblk0/queue/discard_max_bytes:2199023255040
/sys/block/mmcblk0/queue/discard_zeroes_data:1
```

Дополнительные сведения можно найти в документации по ядру (файл `Documentation/block/queue-sysfs.txt`).

Чтобы включить функцию `discard` на этапе монтирования файловой системы, добавьте параметр `-o discard` при вызове команды `mount`. Его поддерживают файловые системы ext4 и F2FS.

Прежде чем задавать параметр `-o discard` в команде `mount`, проверьте, что запоминающее устройство поддерживает функцию `discard`, иначе можно потерять данные.

Есть также возможность принудительно выполнить `discard` из командной строки независимо от того, как был смонтирован раздел. Для этого служит команда `fstrim` из пакета `util-linux`, которая обычно выполняется периодически, скажем раз в неделю, чтобы освободить неиспользуемое место. Команда `fstrim` работает для смонтированной файловой системы, поэтому, чтобы почистить корневую файловую систему /, нужно ввести:

```
# fstrim -v /
/: 2061000704 bytes were trimmed
```

Здесь мы задали флаг `-v`, так что печатается, сколько байтов освобождено. В данном случае 2 061 000 704 – приблизительный размер свободного места в файловой системе, т. е. верхняя оценка числа освобожденных байтов.

Ext4

Расширенная файловая система, ext, является основной файловой системой для ПК под управлением Linux с 1992 года. Текущая версия, ext4, очень стабильна, отлично протестирована и поддерживает журнал, который обеспечивает быстрое и в большинстве случае безболезненное восстановление после неожиданной остановки. Это хороший выбор для управляемой флэш-памяти, она выбрана в качестве предпочтительной файловой системы для устройств на платформе Android, оснащенных микросхемой eMMC. Если устройство поддерживает функцию `discard`, то файловую систему следует монтировать с параметром `-o discard`.

Чтобы отформатировать устройство и создать на нем файловую систему ext4 на этапе выполнения, введите такие команды:

```
# mkfs.ext4 /dev/mmcblk0p2
# mount -t ext4 -o discard /dev/mmcblk0p1 /mnt
```

Чтобы создать образ файловой системы, можно воспользоваться утилитой `genext2fs`, доступной по адресу `http://genext2fs.sourceforge.net`. В примере ниже я задал размер блока с помощью флага `-B` и количество блоков в образе с помощью флага `-b`:

```
$ genext2fs -B 1024 -b 10000 -d rootfs rootfs.ext4
```

Команда `genext2fs` может воспользоваться таблицей устройств для задания владельцев и прав доступа к файлам (см. главу 5); для этого нужно задать флаг `-D [file table]`.

Как следует из названия, команда на самом деле генерирует образ в формате `ext2`. Для модификации до версии `ext4` можно воспользоваться командой `tune2fs` (ее флаги подробно описаны в странице руководства по `tune2fs`):

```
$ tune2fs -j -J size=1 -O filetype,extents,uninit_bg,dir_index rootfs.ext4
$ e2fsck -pDf rootfs.ext4
```

И в Yocto Project, и в Buildroot именно эти шаги выполняются при создании образов в формате `.ext4`.

Конечно, журнал – спасение для устройств, подверженных внезапному выключению питания, но он добавляет дополнительные операции записи к каждой транзакции, что приводит к более быстрому изнашиванию флэш-памяти. Если устройство питается от аккумулятора и особенно если аккумулятор несъемный, вероятность неожиданного выключения питания мала, поэтому имеет смысл отключить журнал.

F2FS

Файловая система **Flash-Friendly File System (F2FS)** является журнально-структурированной и предназначена для устройств с управляемой флэш-памятью, особенно eMMC и SD. Она разработана компанией Samsung и включена в стержневую ветвь Linux, начиная с версии 3.8. Код помечен как экспериментальный, это означает, что он еще не внедрен достаточно широко, но, похоже, используется в некоторых Android-устройствах.

В F2FS принимаются во внимание размеры страницы и стираемого блока и делается попытка выровнять данные на эти границы. Формат журнала защищает от внезапного выключения питания и одновременно обеспечивает неплохую производительность записи, в некоторых тестах она вдвое превосходила ext4. Дизайн F2FS хорошо описан в документации по ядру (файл `Documentation/filesystems/f2fs.txt`), имеются также ссылки на дополнительную литературу в конце этой главы.

Утилита `mfs2.fs2` с флагом `-l` создает пустую файловую систему F2FS:

```
# mkfs.f2fs -l rootfs /dev/mmcblock0p1
# mount -t f2fs /dev/mmcblock0p1 /mnt
```

Пока еще не существует инструмента для создания образов файловой системы F2FS.

FAT16/32

Старые файловые системы Microsoft, FAT16 и FAT32, сохраняют свое значение как общий формат, понятный большинству операционных систем. Только что купленная карта SD или флэш-диск с интерфейсом USB почти наверняка форматированы под FAT32, а в некоторых случаях размещенный на карте микроконтроллер оптимизирован для работы с FAT32. Кроме того, некоторые загрузочные ПЗУ требуют, чтобы вторичный начальный загрузчик находился в FAT-разделе, примером могут служить микросхемы на базе TI OMAP. Однако формат FAT определенно не годится для хранения ответственных файлов, потому что легко повреждается и неэкономно расходует место.

Linux поддерживает формат FAT16 в виде файловой системы msdos, а также форматы FAT32 и FAT16 в виде файловой системы `vfat`. В большинстве случаев необходимо использовать драйвер `vfat`. Тогда для монтирования устройства, скажем карты SD, на втором аппаратном адаптере `mmc` следует выполнить команду:

```
# mount -t vfat /dev/mmcblock1p1 /mnt
```

В прошлом существовали лицензионные проблемы, связанные с использованием драйвера `vfat`, который возможно (а возможно, и нет) нарушал права Microsoft, защищенные патентом.

У FAT32 имеется ограничение на размер раздела – 32 ГиБ. Для форматирования устройств большей емкости можно воспользоваться форматом Microsoft exFAT, а для карт SDXC это обязательное требование. В ядре не существует драйвера для exFAT, но его можно поддержать с помощью драйвера, работающего в пользовательском пространстве. Поскольку формат exFAT защищен авторским правом Microsoft, то, решив поддержать его на своем устройстве, вы столкнетесь с проблемами лицензирования.

Сжатые неизменяемые файловые системы

Сжатие данных полезно, если не хватает места для размещения всего необходимого. И JFFS2, и UBIFS по умолчанию производят сжатие «на лету». Однако если не предполагается записывать файлы, как это обычно бывает с корневой файловой системой, то можно повысить коэффициент сжатия, воспользовавшись сжатой файловой системой, допускающей только чтение. Linux поддерживает несколько таких систем: `romfs`, `cramfs` и `squashfs`. Первые две теперь считаются устаревшими, поэтому я опишу только `squashfs`.

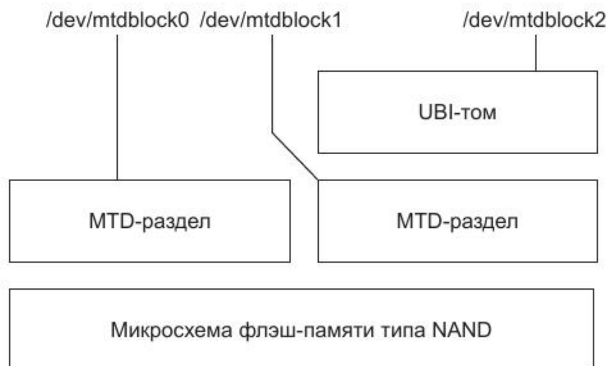
squashfs

Файловую систему `squashfs` написал Филлип Лоуджер (Phillip Lougher) в 2002 году как замену `cramfs`. Долгое время она существовала в виде заплаты к ядру, пока, наконец, в 2009 году не была включена в стержневую ветвь 2.6.29. Пользоваться ей очень просто: нужно создать образ файловой системы командой `mksquashfs` и установить ее на флэш-накопитель:

```
$ mksquashfs rootfs rootfs.squashfs
```

Созданная таким образом файловая система неизменяема, т. е. никаким способом невозможно модифицировать хранящиеся в ней файлы во время выполнения. Чтобы обновить файловую систему `squashfs`, придется стереть весь раздел и записать в него новый образ.

Файловая система `squashfs` ничего не знает о дефектных блоках, поэтому использовать ее следует только с надежной флэш-памятью, например типа NOR. С NAND-памятью ее можно использовать только в сочетании с драйвером UBI, который эмулирует надежный MTD-том. При этом нужно будет включить конфигурационный параметр ядра `CONFIG_MTD_UBI_BLOCK`, в результате чего для каждого UBI-тома будет создаваться доступное только для чтения блочное MTD-устройство. На рисунке ниже изображены два MTD-раздела и соответствующие им устройства `mtdblock`. Для второго раздела создан также UBI-том, видный как третье надежное устройство `mtdblock`, которое можно использовать для размещения любой неизменяемой файловой системы, ничего не знающей о дефектных блоках.



Временные файловые системы

Всегда существуют файлы, которые живут очень недолго или могут не сохраняться после перезагрузки. Часто такие файлы помещают в каталог /tmp, поэтому имеет смысл сделать так, чтобы они не попадали в долговременное хранилище.

Для этой цели идеальна файловая система tmpfs. Чтобы создать временную файловую систему в ОЗУ, нужно просто смонтировать tmpfs:

```
# mount -t tmpfs tmp_files /tmp
```

Для tmpfs, как и для procfs и sysfs, не существует узла устройства, поэтому нужно указывать вместо него фиктивный параметр – в примере выше это строка tmp_files.

Объем занятой памяти будет расти и уменьшаться по мере создания и удаления файлов. По умолчанию максимальный размер составляет половину физической оперативной памяти. Но в большинстве случаев увеличение tmpfs до такого размера стало бы катастрофой, поэтому настоятельно рекомендуется ограничить его с помощью параметра -o size. Размер size можно задавать в байтах, КиБ (k), МиБ (m) или ГиБ (g), например:

```
mount -t tmpfs -o size=1m tmp_files /tmp
```

Помимо /tmp, есть несколько подкаталогов в /var, которые тоже содержат недолговечные данные, и их тоже было бы разумно разместить в tmpfs, либо создав для них отдельную файловую систему, либо – что экономнее – воспользовавшись символическими ссылками. Buildroot делает это следующим образом:

```
/var/cache -> /tmp
/var/lock -> /tmp
/var/log -> /tmp
/var/run -> /tmp
/var/spool -> /tmp
/var/tmp -> /tmp
```

В Yocto Project на каталоги /run и /var/volatile смонтирована tmpfs, а символические ссылки установлены следующим образом:

```
/tmp -> /var/tmp
/var/lock -> /run/lock
/var/log -> /var/volatile/log
/var/run -> /run
/var/tmp -> /var/volatile/tmp
```

Превращение корневой файловой системы в неизменяемую

Необходимо сделать так, чтобы целевое устройство могло пережить неожиданные события, в том числе повреждение файлов, не теряя возможности загрузиться и сохранить хотя бы минимальный уровень функционирования. Важнейшим шагом на пути к достижению этой цели является неизменяемость корневой файловой системы, поскольку это исключает случайную перезапись файла. Сделать ее неизменяемой просто: нужно лишь заменить `rw` на `ro` в командной строке ядра или воспользоваться принципиально неизменяемой файловой системой типа `squashfs`. Однако существует несколько файлов и каталогов, которые традиционно допускают запись.

- `/etc/resolv.conf`. В этот файл скрипты конфигурирования сети записывают адреса DNS-серверов. Эта информация может изменяться, поэтому нужно просто создать символическую ссылку с этого файла на файл в каталоге, допускающем запись, например: `/etc/resolv.conf -> /var/run/resolv.conf`.
- `/etc/passwd`. В этом файле, а также в файлах `/etc/group`, `/etc/shadow` и `/etc/gshadow` хранятся имена и пароли пользователей и групп. Для них также нужно создать символические ссылки, ведущие на каталог, допускающий запись.
- `/var/lib`. Многие приложения ожидают, что в этот каталог можно записывать, и хранят там долговечные данные. Одно из возможных решений – скопировать базовый набор файлов в файловую систему `tmpfs` на этапе загрузки, а затем перемонтировать `/var/lib` с флагом `bind` на новый каталог с помощью такой последовательности команд:

```
mkdir -p /var/volatile/lib
cp -a /var/lib/* /var/volatile/lib
mount --bind /var/volatile/lib /var/lib
```

- `/var/log`. В этом месте хранят журналы syslog и другие демоны. Вообще говоря, хранить журналы во флэш-памяти нежелательно из-за большого числа мелких операций записи. Простое решение – смонтировать `tmpfs` на `/var/log`, сделав все сообщения недолговечными. В случае `syslogd` BusyBox предлагает вариант, в котором сообщения записываются в кольцевой буфер.

При использовании Yocto Project для создания неизменяемой корневой файловой системы нужно добавить строку `IMAGE_FEATURES = "read-only-rootfs"` в файл `conf/local.conf` или в рецепт создания образа.

Варианты выбора файловой системы

Мы рассмотрели технологии изготовления флэш-памяти и различные типы файловых систем. Пришло время подвести итоги.

В большинстве случаев требования к внешнему запоминающему устройству можно отнести к одной из трех категорий:

- **долговечные данные, допускающие чтение и запись:** конфигурационные данные, задаваемые на этапе выполнения, сетевые параметры, пароли, журналы и пользовательские данные;
- **долговечные данные, допускающие** только чтение: программы, библиотеки и постоянные конфигурационные файлы, например корневая файловая система;
- **недолговечные данные**: область временных файлов, например `/tmp`.

Для создания хранилища, допускающего чтение и запись, у нас есть такие варианты:

- **NOR**: UBIFS или JFFS2;
- **NAND**: UBIFS, JFFS2 или YAFFS2;
- **eMMC**: ext4 или F2FS.

В случае хранилища, допускающего только чтение, можно использовать все вышеперечисленное, но монтировать с параметром `ro`. Кроме того, если требуется сэкономить место, то можно взять файловую систему `squashfs`, а в случае NAND-памяти использовать комбинацию драйвера UBI и устройства `mtdblock`, чтобы эмулировать обработку дефектных блоков.

Наконец, для организации хранилища недолговечных данных есть только один вариант – `tmpfs`.

Обновление в месте эксплуатации

В СМИ была широко распространена информация о некоторых уязвимостях, в том числе Heartbleed (ошибка в библиотеках OpenSSL) и Shellshock (ошибка в оболочке bash), которые могут иметь серьезные последствия для уже развернутых встраиваемых Linux-систем. Уже по одной лишь этой причине крайне желательно иметь механизм обновления устройств в месте эксплуатации, чтобы можно было исправлять связанные с безопасностью ошибки по мере их обнаружения. Есть и другие веские причины: исправление других ошибок и обновление функциональных возможностей.

Основополагающий принцип любого механизма обновления – не навредить – в полном соответствии с законом Мэрфи: если что-то может сломаться, то рано или поздно сломается. Любой механизм обновления должен обладать следующими свойствами:

- **эксплуатационная надежность**: он не должен приводить к неработоспособности устройства. Обновления должны быть атомарными: либо система обновлена успешно, либо не обновлена вовсе и продолжает работать, как и раньше;

- **отказоустойчивость**: корректное поведение в случае прерванного обновления;
- **безопасность**: запрет несанкционированного обновления, в противном случае обновление может превратиться в механизм атаки.

Атомарность можно обеспечить за счет хранения дубликатов обновляемых объектов и переключения на новый экземпляр только тогда, когда это безопасно.

Для обеспечения отказоустойчивости должны существовать механизм обнаружения незавершенного обновления, например аппаратная сторожевая схема, и заведомо исправная копия программного обеспечения, к которой можно вернуться в случае ошибки.

Безопасность можно обеспечить в случае локальных обновлений, инициируемых оператором, который вводит пароль или ПИН-код. Если же обновление производится удаленно и автоматически, то необходим какой-то способ аутентификации в сети. Может даже оказаться так, что придется добавить безопасный начальный загрузчик и подписанные двоичные файлы обновления.

Одни компоненты обновить проще, другие сложнее. Очень трудно обновить начальный загрузчик, потому что обычно имеются аппаратные ограничения, разрешающие существование только одного начального загрузчика, поэтому невозможно хранить резервную копию на случай, если обновление пройдет неудачно. С другой стороны, начальные загрузчики редко становятся причиной ошибок на этапе выполнения. Лучшее, что можно порекомендовать, – избегать обновления начального загрузчика в месте эксплуатации.

Степень детализации: файл, пакет или образ?

Это серьезный вопрос, и ответ на него зависит от общего дизайна системы и желаемого уровня эксплуатационной надежности.

Атомарно обновить файл легко: нужно просто записать новую версию во временный файл в той же самой файловой системе, а затем воспользоваться описанной в POSIX командой `rename(2)`, чтобы заменить старый файл новым. Это работает, потому что команда `rename` гарантированно атомарна. Однако это лишь часть проблемы, так как существуют зависимости между файлами, которые необходимо учитывать.

Обновлять на уровне пакетов (`RPM`, `dpkg` или `ipk`) лучше, если только имеется менеджер пакетов, доступный на этапе выполнения. В конце концов, именно так дистрибутивы для настольных ПК обновляются уже много лет. У менеджера пакетов имеется база данных об обновлениях, и он может отследить, какие пакеты обновлены, а какие – нет. В каждом пакете есть скрипт обновления, написанный так, чтобы обеспечить атомарность. Важное преимущество – возможность обновлять существующие пакеты, устанавливать новые и удалять ненужные. Если корневая файловая система смонтирована как неизменяемая, то на время обновления ее придется перемонтировать в режиме чтения-записи, в результате чего открывается непродолжительное окно, в течение которого возможно повреждение файлов.

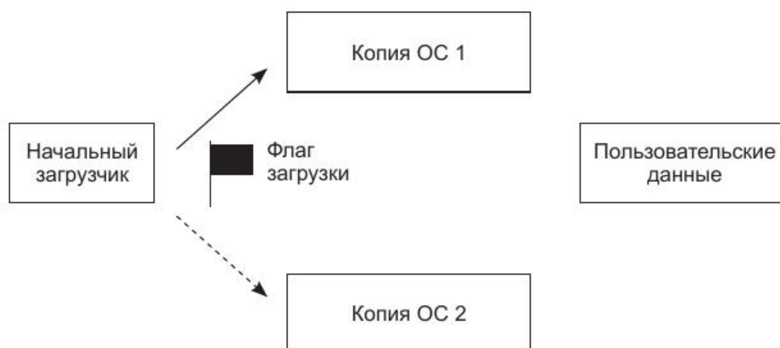
Но у менеджеров пакетов есть и недостатки. Они не способны обновить ядро и другие образы в неуправляемой флэш-памяти. После неоднократного обновления устройства в месте эксплуатации на нем может оказаться одна из многочисленных комбинаций пакетов и их версий, что усложняет контроль качества, предшествующий очередному циклу обновления. Менеджеры пакетов не дают стопроцентной гарантии надежности в случае выключения питания во время обновления.

Третий вариант – обновить образ системы целиком: ядро, корневую файловую систему, пользовательские приложения и т. д.

Атомарное обновление образа

Чтобы сделать обновление атомарным, нужны две вещи: копия операционной системы, которую можно использовать во время обновления, и механизм, позволяющий начальному загрузчику выбрать, какую копию ОС загружать. Обе копии могут совпадать, что обеспечивает полное резервирование ОС, но может быть и так, что копия представляет собой сокращенную операционную систему, специально предназначенную для обновления главной.

В первом случае имеются две идентичные копии операционной системы, каждая из которых включает ядро, корневую файловую систему и системные приложения, как показано на рисунке ниже.



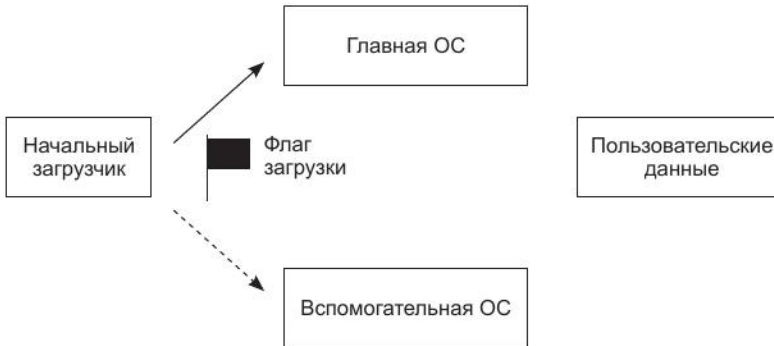
Первоначально флаг загрузки сброшен, поэтому начальный загрузчик загружает копию 1. Чтобы установить обновление, компонент обновления, входящий в состав операционной системы, перезаписывает копию 2. По завершении он поднимает флаг загрузки и перезагружает систему. После этого начальный загрузчик загружает новую ОС. При выполнении следующего обновления компонент обновления в копии 2 перезаписывает копию 1 и сбрасывает флаг загрузки, так что копии меняются местами.

Если обновление оказалось неудачно, то флаг загрузки не изменяется и используется последняя хорошая ОС. Даже если обновляется несколько компонентов: образ ядра, двоичное дерево устройств, корневая файловая система и файловая систе-

ма, содержащая системные приложения, все равно обновление в целом атомарно, потому что флаг загрузки поднимается только после завершения всех обновлений.

Основной недостаток этой схемы – необходимость иметь достаточно места для хранения двух копий операционной системы.

Уменьшать требования к размеру памяти можно, если хранить только минимальную операционную систему, предназначенную исключительно для обновления главной. Это показано на следующем рисунке.



Если требуется установить обновление, мы поднимаем флаг загрузки и перезагружаемся. Вспомогательная система запускает компонент обновления, который перезаписывает образ главной ОС. По завершении он очищает флаг загрузки и перезагружает систему, но теперь загружается новая главная ОС.

Обычно вспомогательная операционная система гораздо меньше главной и может занимать всего несколько мегабайтов, так что накладные расходы невелики. Именно такая схема принята в Android. Размер главной ОС составляет несколько сотен мегабайтов, а вспомогательная ОС – это простой ram-диск размером несколько мегабайтов.

Дополнительная литература

Ниже перечислены ресурсы, в которых можно найти дополнительные сведения о темах, затронутых в этой главе.

- *Wool V.* XIP: The past, the present... the future? Презентация на конференции FOSDEM 2007: https://archive.fosdem.org/2007/slides/devrooms/embedded/ Vitaly_Wool_XIP.pdf.
- Общая документация по MTD. URL: http://www.linux-mtd.infradead.org/doc/ general.html.
- *Bergmann A.* Optimizing Linux with cheap flash drives. URL: http://lwn.net/Articles/428584/.
- Flash memory card design. URL: https://wiki.linaro.org/WorkingGroups/ KernelArchived/Projects/FlashCardSurvey.

- eMMC/SSD File System Tuning Methodology. URL: http://elinux.org/images/b/ b6/EMMC-SSD_File_System_Tuning_Methodology_v1.0.pdf.
- Flash-Friendly File System (F2FS). URL: http://elinux.org/images/1/12/Elc2013_Hwang.pdf.
- An f2fS teardown. URL: http://lwn.net/Articles/518988/.
- *Ben-Yossef G.* Building Murphy-compatible embedded Linux systems. URL: https://www.kernel.org/doc/ols/2005/ols2005v1-pages-21-36.pdf.

Резюме

Флэш-память была основной технологией хранения во встраиваемых Linux-системах с самого начала, и с годами в Linux была создана очень хорошая поддержка: от низкоуровневых драйверов до файловых систем, учитывающих особенности флэш-памяти, последней из таких стала UBIFS.

Однако темпы появления новых технологий флэш-памяти настолько высоки, что становится все труднее поспевать за последними изменениями. Проектировщики систем все чаще обращаются к управляемой флэш-памяти в форме микросхем eMMC, чтобы обеспечить стабильный аппаратный и программный интерфейс, независимый от того, какие микросхемы памяти находятся внутри. Разработчики встраиваемых Linux-систем начинают привыкать к этим новым картам. Поддержка команды TRIM в файловых системах ext4 и F2FS уже «устаканилась» и постепенно перемещается на уровень самих микросхем. И еще одним желанным продвижением стало появление новых файловых систем, оптимизированных для работы с управляемой флэш-памятью, например F2FS.

Однако факт остается фактом: флэш-память – не то же самое, что жесткий диск. Необходимо стремиться к минимизации операций записи в файловую систему – особенно при использовании новых TLC-микросхем высокой плотности, которые, возможно, поддерживают всего-то 1000 циклов стирания.

Наконец, важно выработать стратегию обновления хранящихся в устройстве файлов и образов прямо в месте эксплуатации. Один из важных аспектов такой стратегии – использовать менеджер пакетов или нет. Менеджер пакетов обеспечивает гибкость, но не может обеспечить полную защищенность обновления от действия закона Мэрфи. Вам предстоит найти компромисс между удобством и эксплуатационной надежностью.

В следующей главе рассматривается вопрос об управлении аппаратными компонентами системы с помощью драйверов устройств – как традиционных, находящихся в ядре, так и работающих в пользовательском пространстве.

Глава 8

Введение в драйверы устройств

Драйверы устройств, находящиеся в ядре, – это механизм, посредством которого оборудование становится видно остальным частям системы. Разработчик встраиваемых систем должен понимать, какое место занимают драйверы в общей архитектуре и как обращаться к ним из программ, работающих в пользовательском пространстве. Вероятно, в вашей системе имеется какое-то инновационное оборудование, и вам предстоит разработать способ доступа к нему. Часто оказывается, что драйверы уже существуют и желаемого можно добиться, не дописывая код ядра. Например, контактами интерфейса ввода-вывода общего назначения (GPIO) и светодиодами можно манипулировать с помощью файлов в `sysfs`, а для доступа к последовательным шинам, в том числе SPI и I2C, есть библиотеки.

О разработке драйверов устройств можно прочитать во многих местах, но мало где написано, зачем это делать и какие есть варианты. Именно этот вопрос я и хочу рассмотреть. Однако имейте в виду, что это не книга о написании драйверов устройств и что приведенная здесь информация поможет ориентироваться в этой местности, но вряд ли покажет дорогу к дому. Есть много хороших книг и статей, которые научат вас писать драйверы, некоторые из них перечислены в конце главы.

Роль драйверов устройств

В главе 4 мы отмечали, что одна из функций ядра – инкапсулировать многочисленные аппаратные интерфейсы вычислительной системы и представить их в единообразном виде пользовательским программам. Существуют каркасы, цель которых – упростить кодирование логики взаимодействия с устройством и интеграцию ее с ядром, т. е. написание драйвера устройства, расположенного между ядром сверху и оборудованием снизу. Драйвер может управлять как физическими устройствами типа UART или контроллера MMC, так и виртуальными, например «черной дырой» (`/dev/null`) или ram-диском. Один драйвер может управлять несколькими однотипными устройствами.

Драйвер устройства, как и все ядро, работает с высоким уровнем привилегий. Он имеет полный доступ ко всему адресному пространству процессора и аппаратным регистрам. Он может обрабатывать прерывания и осуществлять прямую передачу данных между устройством и памятью (DMA). Ему разрешено пользоваться развитой инфраструктурой ядра для синхронизации и управления памятью. Но у всего этого есть и обратная сторона – если в драйвере имеется ошибка, то он может «положить» всю систему. Поэтому драйвер должен быть максимально прост, его задача – лишь предоставить информацию приложению, которое и принимает решения. Этот принцип часто формулируют фразой: «никаких политик в ядре».

В Linux есть три основных типа драйверов.

- **Драйверы символьных устройств.** Предназначены для небуферизованного ввода-вывода. На этом уровне имеется много функций, и он представляет собой тонкий слой между кодом приложения и драйвером. Решая, как реализовать собственный драйвер устройства, нужно в первую очередь рассмотреть этот вариант.
- **Драйверы блочных устройств.** Имеют интерфейс, ориентированный на блочный ввод-вывод для массовых запоминающих устройств. Существует толстый слой буферизации, цель которого – максимальное ускорение чтения и записи, поэтому ни для какой другой цели такие драйверы непригодны.
- **Драйверы сетевых устройств.** Похожи на драйверы блочных устройств, но используются для приема и передачи сетевых пакетов, а не дисковых блоков.

Существует еще четвертый тип, который снаружи представляется в виде группы файлов в одной из псевдофайловых систем. Например, к драйверу GPIO можно обращаться как к группе файлов в каталоге `/sys/class/gpio`, как будет описано ниже в этой главе. Начнем с детального рассмотрения трех основных типов устройств.

Символьные устройства

В пользовательском пространстве эти устройства идентифицируются именем файла: если мы хотим читать из универсального асинхронного приемопередатчика (UART), то открываем соответствующий узел устройства; например, первый последовательный порт на плате из семейства ARM Versatile Express будет называться `/dev/ttyAMA0`. В ядре драйвер идентифицируется иначе: с помощью старшего номера, который в примере выше равен 204. Поскольку драйвер UART способен поддерживать несколько устройств UART, существует второй – младший – номер, который идентифицирует конкретный интерфейс, в данном случае 64:

```
# ls -l /dev/ttyAMA*
crw-rw----   1  root  root   204, 64 Jan  1 1970 /dev/ttyAMA0
crw-rw----   1  root  root   204, 65 Jan  1 1970 /dev/ttyAMA1
crw-rw----   1  root  root   204, 66 Jan  1 1970 /dev/ttyAMA2
crw-rw----   1  root  root   204, 67 Jan  1 1970 /dev/ttyAMA3
```

Список стандартных старших и младших номеров приведен в документации по ядру (файл Documentation/devices.txt). Он обновляется довольно редко и не включает устройство ttyAMA. Тем не менее, заглянув в исходный код драйвера в файле drivers/tty/serial/amba-pl011.c, мы увидим, как объявлены старший и младший номера устройства:

```
#define SERIAL_AMBA_MAJOR 204
#define SERIAL_AMBA_MINOR 64
```

Если может существовать несколько экземпляров устройства, то по соглашению для именования узлов устройств применяется схема <базовое имя><номер интерфейса>, например: ttyAMA0, ttyAMA1 и т. д.

В главе 5 отмечалось, что узлы устройств можно создавать несколькими способами:

- devtmpfs: узел, который создается, когда драйвер устройства регистрирует новый интерфейс устройства, используя базовое имя, прописанное в драйвере (ttyAMA), и номер экземпляра;
- udev или mdev (без devtmpfs): по существу, то же, что devtmpfs, с тем отличием, что работающая в пользовательском пространстве программа-демон должна извлечь имя устройства из sysfs и создать узел. О sysfs я расскажу ниже;
- mknod: статические узлы устройств создаются вручную с помощью программы mknod.

Возможно, глядя на приведенные выше значения старшего и младшего номеров, вы подумали, что это 8-разрядные числа в диапазоне от 0 до 255. Но на самом деле начиная с версии Linux 2.6 старший номер имеет длину 12 разрядов и выбирается из диапазона от 1 до 4095, а младший – 20 разрядов, что дает диапазон от 0 до 1 048 575.

При открытии узла устройства ядро проверяет, попадают ли старший и младший номера в диапазоны, зарегистрированные драйвером устройства этого типа (символьного или блочного). Если да, то вызов передается драйверу, иначе завершается с ошибкой. Драйвер устройства может извлечь младший номер и по нему понять, какой аппаратный интерфейс использовать. Если младший номер выходит за границы допустимого диапазона, возвращается код ошибки.

Для написания программы, обращающейся к драйверу устройства, нужно понимать, как он работает. Иными словами, драйвер – не то же самое, что файл: выполняемые операции изменяют состояния устройства. Простой пример – генератор псевдослучайных чисел urandom, который возвращает случайные байты при каждой операции чтения из него. Ниже приведена соответствующая программа.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(void)
```

```
{
  int f;
  unsigned int rnd;
  int n;
  f = open("/dev/urandom", O_RDONLY);
  if (f < 0) {
    perror("Ошибка при открытии urandom");
    return 1;
  }
  n = read(f, &rnd, sizeof(rnd));
  if (n != sizeof(rnd)) {
    perror("Ошибка при чтении urandom");
    return 1;
  }
  printf("Случайное число = 0x%x\n", rnd);
  close(f);
  return 0;
}
```

Модель драйверов в Unix хороша тем, что если мы знаем, что существует устройство с именем `urandom` и что при каждом чтении из него возвращается новый набор псевдослучайных данных, то больше ничего о нем знать не нужно. Можно использовать обычные функции `open(2)`, `read(2)` и `close(2)`.

Мы могли бы воспользоваться функциями потокового ввода-вывода `fopen(3)`, `fread(3)` и `fclose(3)`, но встроенная в них буферизация часто приводит к неожиданному поведению. Например, функция `fwrite(3)` обычно пишет только в буфер в пользовательском адресном пространстве, а не на устройство. Чтобы сбросить буферы, нужно вызвать функцию `fflush(3)`.

Не используйте функции потокового ввода-вывода, в частности `fread(3)` и `fwrite(3)`, при обращении к драйверам устройств.

Блочные устройства

Блочные устройства также ассоциированы с узлом устройства, имеющим старший и младший номера.

Хотя как символьные, так и блочные устройства идентифицируются старшим и младшим номерами, они находятся в разных пространствах имен. Символьное устройство со старшим номером 4 никак не связано с блочным устройством, имеющим тот же номер 4.

В случае блочных устройств старший номер идентифицирует драйвер, а младший – раздел. Рассмотрим в качестве примера драйвер MMC:

```
# ls -l /dev/mmcblk*

brw-------    1 root    root 179,   0 Jan 1 1970 /dev/mmcblk0
brw-------    1 root    root 179,   1 Jan 1 1970 /dev/mmcblk0p1
```

```
brw-------   1 root   root 179,  2 Jan 1 1970 /dev/mmcblk0p2
brw-------   1 root   root 179,  8 Jan 1 1970 /dev/mmcblk1
brw-------   1 root   root 179,  9 Jan 1 1970 /dev/mmcblk1p1
brw-------   1 root   root 179, 10 Jan 1 1970 /dev/mmcblk1p2
```

Старший номер равен 179 (загляните в файл `devices.txt`!). Младшие номера выделяются диапазонами и идентифицируют различные устройства `mmc` и разделы устройства. В случае драйвера mmcblk на каждое устройство выделяется по восемь младших номеров: номера от 0 до 7 относятся к первому устройству, от 8 до 15 – ко второму и т. д. В пределах одного диапазона первый младший номер представляет все устройство в виде последовательности секторов, а остальные – до семи разделов.

Вероятно, вы знаете о драйвере SCSI-дисков с именем sd, который служит для управления дисками, понимающими набор команд SCSI, к каковым относятся SCSI, SATA, массовые запоминающие устройства с интерфейсом USB и универсальные флэш-накопители **UFS** (universal flash storage). Старший номер этого драйвера равен 8, а младшие выделяются диапазонами по 16 номеров на каждый интерфейс (или диск). Номера от 0 до 15 относятся к первому диску, и им соответствуют узлы устройств с именами от `sda` до `sda15`, номера от 16 до 31 – ко второму диску (узлы устройств от `sdb` до `sdb15`) и т. д. Так продолжается до шестнадцатого диска с младшими номерами от 240 до 255 и именем узла `sdp`. Для SCSI-дисков в силу их популярности зарезервированы и другие старшие номера, но нас это сейчас не волнует.

Разделы создаются такими утилитами, как `fdisk`, `sfidsk` и `parted`. Исключением является неуправляемая флэш-память: информация о разделе для драйвера MTD передается в командной строке ядра, или в дереве устройств, или еще каким-то способом, описанным в главе 7.

Программы, работающие в пользовательском пространстве, могут открывать блочное устройство по имени его узла и затем взаимодействовать с ним. Это не слишком распространенная практика, обычно она применяется для таких административных операций, как разбиение на разделы, форматирование под какую-то файловую систему и монтирование. После того как файловая система смонтирована, взаимодействие с блочным устройством происходит опосредованно – через файлы.

Сетевые устройства

Доступ к сетевым устройствам производится не через узлы устройств, и у них нет ни старших, ни младших номеров. Имя сетевому устройству присваивает ядро, составляя его из некоторой строки и номера экземпляра. Вот пример того, как драйвер регистрирует сетевой интерфейс:

```
my_netdev = alloc_netdev(0, "net%d", NET_NAME_UNKNOWN, netdev_setup);
ret = register_netdev(my_netdev);
```

В результате при первом обращении создается сетевое устройство `net0`, при втором – `net1` и т. д. Чаще, впрочем, употребляются имена `lo`, `eth0` и `wlan0`.

Обратите внимание, что это начальное имя устройства; диспетчеры устройств, в частности udev, могут впоследствии заменить его другим.

Обычно имя сетевого интерфейса используется только на этапе конфигурирования сети с помощью таких утилит, как ip и ifconfig, когда задается адрес сети и маршрут. После этого взаимодействие с драйвером производится путем открытия сокетов, а сетевой уровень решает, какому интерфейсу направить обращения.

Однако существует возможность обращаться к сетевым устройствам напрямую из пользовательского пространства, для чего нужно создать сокет и пользоваться командами ioctl, перечисленными в файле include/linux/sockios.h. Например, следующая программа выполняет команду SIOCGIFHWADDR, чтобы запросить у драйвера аппаратный MAC-адрес:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/sockios.h>
#include <net/if.h>

int main (int argc, char *argv[])
{
  int s;
  int ret;
  struct ifreq ifr;
  int i;

  if (argc != 2) {
    printf("Usage %s [network interface]\n", argv[0]);
    return 1;
  }
  s = socket(PF_INET, SOCK_DGRAM, 0);
  if (s < 0) {
    perror("socket");
    return 1;
  }
  strcpy(ifr.ifr_name, argv[1]);
  ret = ioctl(s, SIOCGIFHWADDR, &ifr);
  if (ret < 0) {
    perror("ioctl");
    return 1;
  }
  for (i = 0; i < 6; i++)
    printf("%02x:", (unsigned char)ifr.ifr_hwaddr.sa_data[i]);
  printf("\n");
  close(s);
  return 0;
}
```

Это стандартная команда `ioctl`, которую сетевой уровень отрабатывает от имени драйвера, но можно также определить собственные команды `ioctl` и обрабатывать их в своем сетевом драйвере.

Получение информации о драйверах на этапе выполнения

После того как система Linux начала работать, полезно узнать, какие драйверы устройств загружены и в каком состоянии они находятся. Многое можно выяснить, читая файлы в каталогах `/proc` и `/sys`.

Прежде всего можно распечатать список загруженных и активных в данный момент драйверов символьных и блочных устройств, прочитав файл `/proc/devices`:

```
# cat /proc/devices

Character devices:

  1 mem
  2 pty
  3 ttyp
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  5 /dev/console
  5 /dev/ptmx
  7 vcs
 10 misc
 13 input
 29 fb
 81 video4linux
 89 i2c
 90 mtd
116 alsa
128 ptm
136 pts
153 spi
180 usb
189 usb_device
204 ttySC
204 ttyAMA
207 ttymxc
226 drm
239 ttyLP
240 ttyTHS
241 ttySiRF
242 ttyPS
243 ttyWMT
```

```
244 ttyAS
245 ttyO
246 ttyMSM
247 ttyAML
248 bsg
249 iio
250 watchdog
251 ptp
252 pps
253 media
254 rtc

Block devices:

259 blkext
  7 loop
  8 sd
 11 sr
 31 mtdblock
 65 sd
 66 sd
 67 sd
 68 sd
 69 sd
 70 sd
 71 sd
128 sd
129 sd
130 sd
131 sd
132 sd
133 sd
134 sd
135 sd
179 mmc
```

Для каждого драйвера показаны старший номер и базовое имя устройства. Но это ничего не говорит о том, сколько устройств присоединено к каждому драйверу. Мы видим лишь имя `ttyAMA`, но не знаем, что в действительности существуют четыре приемопередатчика UART. Я вернусь к этому вопросу позже, когда буду обсуждать `sysfs`. Если используется диспетчер устройств, например `mdev`, `udev` или `devtmpfs`, то получить список интерфейсов символьных и блочных устройств можно, заглянув в каталог `/dev`.

Список сетевых устройств выводят программы `ifconfig` или `ip`:

```
# ip link show

1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```
2: eth0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN mode DE-
FAULT qlen 1000
    link/ether 54:4a:16:bb:b7:03 brd ff:ff:ff:ff:ff:ff
3: usb0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP mode DE-
FAULT qlen 1000
    link/ether aa:fb:7f:5e:a8:d5 brd ff:ff:ff:ff:ff:ff
```

Можно также получить информацию об устройствах, подключенных к шинам USB или PCI, для этого служат хорошо известные команды lsusb и lspci. Подробнее о них написано на страницах руководства и в многочисленных документах в сети, поэтому я не стану этим заниматься.

По-настоящему интересная информация есть в псевдофайловой системе sysfs, которую мы далее и рассмотрим.

Получение информации из sysfs

Строго говоря, sysfs определяется как представление объектов ядра, их атрибутов и связей между ними. Объектом ядра является, например, каталог, атрибутом – файл, а связью – символическая ссылка с одного объекта на другой.

Но, с практической точки зрения, после реализации модели драйверов устройств в версии Linux 2.6 все устройства и драйверы представлены в виде объектов ядра. Взгляд на систему с точки зрения ядра можно получить, заглянув в каталог /sys:

```
# ls /sys
block   bus   class   dev   devices   firmware   fs   kernel   module
power
```

Поскольку сейчас нас интересует получение информации об устройствах и драйверах, я рассмотрю только три каталога: devices, class и block.

Устройства: /sys/devices

Это взгляд ядра на устройства, обнаруженные с момента загрузки, и их связи между собой. На верхнем уровне находится системная шина, поэтому в разных системах представление различно. Ниже показано, как устройство Versatile Express выглядит в эмуляторе QEMU:

```
# ls
/sys/devices
armv7_cortex_a9    platform    system
breakpoint         software    virtual
```

В любой системе имеются три каталога:

- system: содержит основные системные устройства, в том числе процессоры и генераторы тактовых импульсов;
- virtual: содержит устройства, относящиеся к памяти. В virtual/mem мы обнаружим устройства, представляемые как /dev/null, /dev/random и /dev/zero. Возвратный сетевой интерфейс lo находится в virtual/net;

- platform: здесь собраны все устройства, не подключенные к традиционной аппаратной шине. Во встраиваемом устройстве это может быть практически все оборудование.

Прочие устройства помещены в каталоги, соответствующие фактическим системным шинам. Например, корневая шина PCI, если таковая имеется, представлена как pci0000:00.

Перемещаться по этой иерархии трудно, потому что необходимы знания о топологии системы, а очень длинные пути сложно запомнить. Чтобы упростить жизнь, в каталогах /sys/class и /sys/block множество устройств представлено двумя другими способами.

Драйверы: /sys/class

Это представление драйверов устройств по типу, иными словами, программное, а не аппаратное. Каждый подкаталог представляет один класс драйверов и реализован некоторым компонентом драйверной инфраструктуры. Например, устройства UART управляются уровнем tty, поэтому находятся в каталоге /sys/class/tty. Аналогично сетевые устройства находятся в каталоге /sys/class/net, устройства ввода – клавиатура, сенсорный экран и мышь – в каталоге /sys/class/input и т. д.

В каждом подкаталоге для каждого экземпляра типа устройства существует символическая ссылка на его представление в каталоге /sys/device.

В качестве примера рассмотрим каталог /sys/class/tty/ttyAMA0:

```
# cd /sys/class/tty/ttyAMA0/
# ls
close_delay     flags            line       uartclk
closing_wait    io_type          port       uevent
custom_divisor  iomem_base       power      xmit_fifo_size
dev             iomem_reg_shift  subsystem
device          irq              type
```

Ссылка device ведет на аппаратный узел данного устройства, а subsystem – обратная ссылка на /sys/class/tty. Прочие элементы – это атрибуты устройства. Одни из них специфичны для UART, например xmit_fifo_size, другие относятся ко многим типам устройств, например номер прерывания irq и номер устройства dev. Некоторые файлы атрибутов допускают запись, т. е. позволяют настраивать параметры на этапе выполнения.

Атрибут dev особенно интересен. Выведем его значение:

```
# cat /sys/class/tty/ttyAMA0/dev
204:64
```

Это старший и младший номера устройства. Этот атрибут создается, когда драйвер регистрирует интерфейс, и именно из этого файла udev и mdev читают информацию, если используются без поддержки со стороны devtmpfs.

Блочные устройства: /sys/block

Существует еще одно важное представление модели устройств: драйверы блочных устройств в каталоге /sys/block. Каждому блочному устройству здесь соответствует отдельный подкаталог. Следующий пример относится к плате BeagleBone Black:

```
# ls /sys/block/
loop0  loop4  mmcblk0       ram0   ram12  ram2  ram6
loop1  loop5  mmcblk1       ram1   ram13  ram3  ram7
loop2  loop6  mmcblk1boot0  ram10  ram14  ram4  ram8
loop3  loop7  mmcblk1boot1  ram11  ram15  ram5  ram9
```

Зайдя в каталог mmcblk1, соответствующий микросхеме eMMC на данной плате, мы увидим атрибуты интерфейса и перечень разделов:

```
# cd /sys/block/mmcblk1
# ls
alignment_offset   ext_range     mmcblk1p1  ro
bdi                force_ro      mmcblk1p2  size
capability         holders       power      slaves
dev                inflight      queue      stat
device             mmcblk1boot0  range      subsystem
discard_alignment  mmcblk1boot1  removable  uevent
```

Таким образом, мы приходим к выводу, что благодаря sysfs можно многое узнать об устройствах (оборудовании) и драйверах (программном обеспечении), присутствующих в системе.

Поиск подходящего драйвера устройства

Типичная встраиваемая плата основана на эталонной конструкции изготовителя с изменениями, делающими ее пригодной для конкретного приложения. Устройство может быть оснащено датчиком температуры, подключенным к интерфейсной шине I2C, лампочками и кнопками, подключенными к контактам GPIO, внешним MAC-адресом, индикаторной панелью с интерфейсом MIPI и много чем еще. Ваша задача – собрать специальное ядро, которое будет всем этим управлять. Так с чего же начать?

Некоторые вещи настолько просты, что для работы с ними можно написать код, исполняемый в пользовательском пространстве. Ниже я объясню, что контактами GPIO и простыми периферийными устройствами, подключенными к шине I2C или SPI, легко управлять из пользовательского пространства.

Для другого оборудования нужен драйвер в ядре, поэтому необходимо знать, как его найти, и включить в сборку. Простого ответа тут не существует, но есть места, где стоит поискать.

Самое очевидное из них – страница поддержки драйверов на сайте изготовителя оборудования, можно также прямо задать вопрос изготовителю. Мой опыт по-

казывает, что это редко дает желаемый результат; производители оборудования не особо сведущи в Linux и часто дают дезориентирующую информацию. Они могут располагать драйвером только в двоичном виде или иметь исходный код, но не для той версии ядра, что у вас. Впрочем, попробовать в любом случае стоит. Но лично я всегда стараюсь найти подходящий драйвер с открытым исходным кодом.

Возможно, в ядре уже имеется поддержка: в стержневой версии Linux тысячи драйверов, и еще множество в версиях ядра от различных поставщиков. Запустите `menuconfig` (или `xconfig`) и поищите по названию или номеру изделий. Если не найдете точного совпадения, попробуйте расширенный поиск с учетом того факта, что большинство драйверов рассчитано на несколько изделий из одного и того же семейства. Затем попробуйте поискать в исходном коде в каталоге `drivers` (grep вам в помощь). Всегда выбирайте последнюю версию ядра для своей платы: чем старше версия, тем больше в ней драйверов устройств.

Если ничего так и не нашлось, попробуйте поискать в сети и поспрашивать на тематических форумах, не знает ли кто-нибудь о драйвере для другой версии Linux. Если такой есть, то вам придется выполнить его обратное портирование на свою версию ядра. Если версии ядра похожи, то это может оказаться просто, но если их разделяет от 12 до 18 месяцев, то велики шансы, что интерфейсы поменялись настолько сильно, что придется переписывать значительный кусок драйвера. Возможно, вы захотите отдать эту работу на сторону. Если ничто из вышеперечисленного не привело к успеху, то решение придется искать самостоятельно.

Драйверы устройств в пользовательском пространстве

Прежде чем бросаться писать драйвер устройства, подумайте, так ли это необходимо. Для многих распространенных типов устройств существуют обобщенные драйверы, позволяющие взаимодействовать с оборудованием прямо из пользовательского пространства, и при этом не придется писать ни строчки кода в ядре. Код, работающий в пользовательском пространстве, проще писать и отлаживать. К тому же он не подпадает под действие лицензии GPL, хотя я не думаю, что эта причина сама по себе достаточна.

Такие драйверы можно разбить на две категории: контролируемые посредством файлов в `sysfs` (к ним относятся, в частности, GPIO и светодиоды) и последовательные шины, раскрывающие обобщенный интерфейс через какой-то узел устройства, например интерфейсная шина I2C.

GPIO

Интерфейс ввода-вывода общего назначения (General Purpose Input/Output – **GPIO**) – простейший вид цифрового интерфейса, поскольку дает прямой доступ к отдельным аппаратным контактам, каждый из которых можно сконфигуриро-

вать для ввода или для вывода. GPIO можно даже использовать для создания интерфейсов более высокого уровня, например I2C или SPI, путем программного манипулирования отдельными битами, эта техника называется bit bang. Основное ограничение – скорость выполнения программных циклов и невозможность гарантировать точное время их работы. Вообще говоря, трудно достичь разрешающей способности таймера меньше одной миллисекунды, если ядро откомпилировано с параметром `CONFIG_PREEMPT`, и меньше 100 микросекунд – если с параметром `RT_PREEMPT`, о чем мы еще будем говорить в главе 14. GPIO чаще всего используют для чтения состояния нажимаемых кнопок и цифровых датчиков, а также для управления светодиодами, электромоторами и реле.

В большинстве SoC-систем много битов GPIO, которые сгруппированы в регистры GPIO, обычно по 32 бита в регистре. Биты, расположенные на кристалле GPIO, управляются с помощью контактов GPIO, выведенных из микросхемы, посредством мультиплексора, который я опишу ниже. Могут существовать дополнительные биты GPIO вне кристалла, находящиеся в микросхеме управления питанием и в специализированных расширителях GPIO, подключенных к шине I2C или SPI. Всеми этими разнообразными возможностями управляет подсистема ядра, которая называется `gpiolib`, хотя это и не библиотека вовсе, а инфраструктура, используемая драйверами GPIO для единообразного представления ввода-вывода.

Детали реализации `gpiolib` можно найти в документации по ядру (`Documentation/gpio`) и в коде самих драйверов в каталоге `drivers/gpio`.

Приложения могут взаимодействовать с `gpiolib` посредством файлов в каталоге `/sys/class/gpio`. Ниже показано, что находится в этом каталоге на типичной встраиваемой плате (BeagleBone Black):

```
# ls /sys/class/gpio
export  gpiochip0  gpiochip32  gpiochip64  gpiochip96  unexport
```

Каталоги от `gpiochip0` до `gpiochip96` представляют регистры GPIO, по 32 бита в каждом. Заглянув в какой-нибудь из каталогов `gpiochip`, мы увидим следующие файлы:

```
# ls /sys/class/gpio/gpiochip96/
base  label  ngpio  power  subsystem  uevent
```

Файл `base` содержит номер первого контакта GPIO в регистре, а `ngpio` – количество битов в регистре. В данном случае `gpiochip96/base` содержит 96, `gpiochip96/ngpio` – 32, т. е. данный регистр содержит биты GPIO с номерами от 96 до 127. Возможен разрыв между номером последнего бита GPIO в одном регистре и номером первого бита в следующем.

Для управления контактом GPIO из пользовательского пространства необходимо сначала экспортировать его из пространства ядра, для чего нужно записать номер контакта в файл `/sys/class/gpio/export`. В примере ниже показано, как это делается для контакта GPIO 48:

```
# echo 48 > /sys/class/gpio/export
# ls /sys/class/gpio
export      gpio48    gpiochip0    gpiochip32    gpiochip64
gpiochip96  unexport
```

Теперь появился новый каталог gpio48, содержащий файлы, необходимые для управления контактом. Отметим, что если битом GPIO уже управляет ядро, то экспортировать его таким способом не получится.

В каталоге gpio48 находятся следующие файлы:

```
# ls /sys/class/gpio/gpio48
active_low  direction  edge  power  subsystem  uevent  value
```

Первоначально контакт считается предназначенным для ввода. Чтобы переназначить его на вывод, нужно записать значение в файл direction. В этом файле находится текущее состояние контакта: 0 – низкий уровень, 1 – высокий. Если контакт предназначен для вывода, то нужно поменять значение на противоположное. Иногда оборудование инвертирует смысл низкого и высокого уровней (конструкторы обожают развлекаться подобным образом), тогда нужно записать значение 1 в файл active_low – в результате низкому напряжению будет соответствовать 1, а высокому – 0.

Чтобы запретить управление контактом GPIO из пользовательского пространства, нужно записать его номер в файл /sys/class/gpio/unexport.

Обработка прерывания от GPIO

Во многих случаях можно сконфигурировать систему так, что при изменении состояния ввода от GPIO будет генерироваться прерывание. Это дает возможность ожидать прерывания вместо опроса состояния в неэффективном программном цикле. Если контакт GPIO может генерировать прерывания, то существует файл edge. Первоначально в нем находится значение none, т. е. прерывания не генерируются. Чтобы разрешить прерывания, нужно записать в файл одно из следующих значений:

- **rising**: прерывание по переднему фронту сигнала;
- **falling**: прерывание по заднему фронту сигнала;
- **both**: прерывание по обоим фронтам сигнала;
- **none**: прерывания запрещены (режим по умолчанию).

Для ожидания прерывания можно использовать функцию poll() с событием POLLPRI. Если мы хотим ожидать прерывания по переднему фронту от контакта GPIO 48, то сначала разрешим прерывания:

```
# echo 48 > /sys/class/gpio/export
# echo rising > /sys/class/gpio/gpio48/edge
```

Затем вызовем функцию poll(), которая будет ждать изменения состояния, как показано в коде ниже:

```
#include <stdio.h>
#include <unistd.h>
```

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <poll.h>

int main (int argc, char *argv[])
{
  int f;
  struct pollfd poll_fds [1];
  int ret;
  char value[4];
  int n;
  f = open("/sys/class/gpio/gpio48", O_RDONLY);
  if (f == -1) {
    perror("Can't open gpio48");
    return 1;
  }
  poll_fds[0].fd = f;
  poll_fds[0].events = POLLPRI | POLLERR;
  while (1) {
    printf("Waiting\n");
    ret = poll(poll_fds, 1, -1);
    if (ret > 0) {
        n = read(f, &value, sizeof(value));
        printf("Button pressed: read %d bytes, value=%c\n", n, value[0]);
    }
  }
  return 0;
}
```

Светодиоды

Светодиоды часто управляются с помощью контакта GPIO, но в ядре существует и специализированная подсистема для этой цели. Подсистема `leds` добавляет средства для задания яркости, если светодиод обладает такой возможностью, и может управлять светодиодами, подключенными не к контакту GPIO, а как-то иначе. Ее можно сконфигурировать так, чтобы она зажигала светодиод, когда происходит некое событие, например доступ к блочному устройству, или просто периодически, чтобы показать, что устройство работает. Дополнительные сведения имеются в файле `Documentation/leds/` и в коде драйверов в каталоге `drivers/leds/`.

Как и GPIO, светодиоды управляются с помощью интерфейса `sysfs`, а точнее каталога `/sys/class/leds`. Светодиодам присваиваются имена вида `devicename: colour:function`, например:

```
# ls /sys/class/leds
beaglebone:green:heartbeat        beaglebone:green:usr2
beaglebone:green:mmc0             beaglebone:green:usr3
```

Вот как выглядит каталог, соответствующий конкретному светодиоду:

```
# ls /sys/class/leds/beaglebone:green:usr2
brightness      max_brightness      subsystem      uevent
device          power               trigger
```

Файл brightness управляет яркостью светодиода и может содержать число от 0 (выключен) до max_brightness (горит в полную силу). Если светодиод не поддерживает промежуточных уровней яркости, то любое ненулевое значение означает «включен», а 0 – «выключен». В файле trigger перечислены события, включающие светодиод. Список таких триггеров зависит от реализации. Вот пример:

```
# cat /sys/class/leds/beaglebone:green:heartbeat/trigger
none mmc0 mmc1 timer oneshot [heartbeat] backlight gpio cpu0 default-on
```

Выбранный в данный момент триггер заключен в квадратные скобки. Его можно изменить, записав в файл имя другого триггера. Если вы хотите управлять светодиодом только с помощью файла brightness, выберите значение none. Если установить триггер timer, то появляются еще два файла, позволяющие задать время, в течение которого светодиод остается включенным и выключенным, в миллисекундах:

```
# echo timer > /sys/class/leds/beaglebone:green:heartbeat/trigger
# ls /sys/class/leds/beaglebone:green:heartbeat
brightness   delay_on  max_brightness   subsystem   uevent
delay_off    device    power            trigger
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_on
500
# cat /sys/class/leds/beaglebone:green:heartbeat/delay_off
500
#
```

Если в светодиод включен аппаратный таймер, то мигание происходит без вмешательства со стороны процессора.

Шина I2C

I2C – простая низкоскоростная двухпроводная шина, которая часто встречается во встраиваемых платах и обычно используется для доступа к периферийным устройствам, находящимся вне SoC-системы, например дисплейным контроллерам, датчикам камеры, расширителям GPIO и т. п. Существует также стандарт SMBus (системная управляющая шина), который применяется в ПК и служит для доступа к датчикам температуры и напряжения. SMBus – подмножество I2C.

I2C построена на основе протокола «ведущий-ведомый», причем в качестве ведущих устройств выступает один или несколько контроллеров на плате SoC-системы. Ведомым устройствам производитель назначает 7-разрядные адреса (читайте техническое описание), что позволяет подключать к шине до 128 устройств, но 16 из них зарезервированы, так что на практике остается 112. Скорость шины составляет 100 Кбит/с в стандартном режиме и до 400 Кбит/с – в скоростном. Протокол допускает транзакции чтения и записи между ведущим и ведомым узлами длиной до 32 байтов. Часто первый байт используется для задания регистра

периферийного устройства, а остальные содержат данные, прочитанные из этого регистра или записанные в него.

Каждому контроллеру соответствует один узел устройства. Вот пример SoC-системы с четырьмя контроллерами:

```
# ls -l /dev/i2c*
crw-rw---- 1 root i2c 89, 0 Jan  1 00:18 /dev/i2c-0
crw-rw---- 1 root i2c 89, 1 Jan  1 00:18 /dev/i2c-1
crw-rw---- 1 root i2c 89, 2 Jan  1 00:18 /dev/i2c-2
crw-rw---- 1 root i2c 89, 3 Jan  1 00:18 /dev/i2c-3
```

В интерфейсе устройства предусмотрен набор функций `ioctl`, позволяющих опрашивать контроллер и посылать команды `read` и `write` ведомым устройствам. Существует пакет `i2c-tools`, в котором этот интерфейс используется для реализации командных средств взаимодействия с I2C-устройствами. В него входят следующие команды:

- `i2cdetect`: выводит список I2C-адаптеров и зондирует шину;
- `i2cdump`: выводит данные из всех регистров периферийного устройства на шине I2C;
- `i2cget`: читает данные из ведомого устройства на шине I2C;
- `i2cset`: записывает данные в ведомое устройство на шине I2C.

Пакет `i2c-tools` включен в Buildroot, в Yocto Project и в большинство основных дистрибутивов. Если известен адрес и протокол ведомого устройства, то написать программу для взаимодействия с ним очень просто:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <i2c-dev.h>
#include <sys/ioctl.h>
#define I2C_ADDRESS 0x5d
#define CHIP_REVISION_REG 0x10

void main (void)
{
  int f_i2c;
  int val;
  /* Открыть адаптер и задать адрес устройства на шине I2C */
  f_i2c = open ("/dev/i2c-1", O_RDWR);
  ioctl (f_i2c, I2C_SLAVE, I2C_ADDRESS);

  /* Читать 16-разрядные данные из регистра */
  val = i2c_smbus_read_word_data(f, CHIP_REVISION_REG);
  printf ("Версия микросхемы датчика %d\n", val);
  close (f_i2c);
}
```

Отметим, что заголовок `i2c-dev.h` взят из пакета `i2c-tools`, а не из ядра Linux. Функция `i2c_smbus_read_word_data()` в файле `i2c-dev.h` объявлена встраиваемой.

Дополнительные сведения о работе с шиной I2C в Linux см. в файле `Documentation/i2c/dev-interface`. Драйверы контроллеров находятся в каталоге `drivers/ i2c/busses`.

Шина SPI

Шина последовательного периферийного интерфейса SPI похожа на I2C, но значительно быстрее, она может достигать скоростей порядка Мбит/с. В интерфейсе четыре провода с раздельными линиями приема и передачи, что позволяет работать в полнодуплексном режиме. Для выбора подключенной к шине схемы используется отдельная линия выбора. Обычно используется для подключения датчиков сенсорного экрана, дисплейных контроллеров и последовательных устройств флэш-памяти типа NOR.

Как и в случае I2C, используется протокол «ведущий-ведомый», причем в большинстве SoC-систем присутствует один или несколько ведущих контроллеров. Существует обобщенный драйвер SPI, который включается в ядро, если задан конфигурационный параметр `CONFIG_SPI_SPIDEV`. Драйвер создает узел устройства для каждого контроллера на шине SPI, что позволяет обращаться к устройствам на шине из пользовательского пространства. Узлы устройств именуются `spidev[шина].[номер схемы]`:

```
# ls -l /dev/spi*
crw-rw---- 1 root root 153, 0 Jan 1 00:29 /dev/spidev1.0
```

Примеры работы с интерфейсом `spidev` приведены в файле `Documentation/spi`.

Написание драйвера устройства

Если все возможности управления устройством из пользовательского пространства рассмотрены и ни одна не подошла, то для доступа к оборудованию, входящему в состав вашего устройства, придется писать драйвер. Вдаваться в детали у нас нет ни времени, ни места, но упомянуть о различных возможностях все же стоит. Наибольшей гибкостью обладают символьные устройства, они закрывают 90% потребностей; сетевые устройства относятся к работе с сетевым интерфейсом, а блочные – к работе с массовыми запоминающими устройствами. Задача написания драйвера в ядре сложна и выходит за рамки этой книги. В конце главы есть несколько ссылок на соответствующие ресурсы. В этом разделе я опишу варианты взаимодействия с драйвером – эта тема обычно не рассматривается – и объясню, из каких частей состоит драйвер.

Проектирование интерфейса символьного устройства

В основе интерфейса символьного устройства лежит поток байтов, как в случае последовательного порта. Однако многие устройства не подпадают под это описание: например, контроллеру роботизированной руки необходимы функции для поступательного перемещения и вращения каждого сочленения. К счастью,

существуют и другие способы взаимодействия с драйвером устройства, помимо read(2) и write(2).

- ioctl. Функция ioctl позволяет передать драйверу два аргумента, семантику которых вы можете определить любым удобным способом. По соглашению первый аргумент – команда, выбирающая одну из нескольких функций драйвера, а второй – указатель на структуру, служащую для хранения входных и выходных параметров. Это холст, на котором проектировщик может нарисовать любой программный интерфейс, и очень часто драйвер и приложение бывают тесно связаны и разрабатываются одной и той же командой. Однако функция ioctl объявлена нерекомендуемой, и вряд ли вы встретите в стержневой ветви хотя бы один новый драйвер, включающий ioctl. Разработчики ядра не любят ioctl, потому что из-за нее коды ядра и приложений оказываются слишком сильно зависимы, и становится трудно синхронизировать их код при смене версии или архитектуры.
- sysfs. Сейчас этот способ считается предпочтительным, и примером может служить описанный выше интерфейс GPIO. Он самодокументированный, при условии что выбираются смысловые имена файлов. Он также допускает использование в скриптах, потому что содержимое файлов – строки в кодировке ASCII. С другой стороны, требование о том, что каждый файл должен содержать только одно значение, затрудняет обеспечение атомарности, когда нужно за одну операцию изменить несколько значений. Например, если требуется задать два значения, а затем инициировать действия, то придется писать в три файла: два содержат значения, а третий играет роль триггера. Но тогда нет гарантии, что пока идет запись в один файл, два других не будут изменены кем-то еще. В этом смысле у ioctl есть преимущество: все аргументы передаются в структуре, так что требуется только один вызов функции.
- mmap. Получить прямой доступ к буферам ядра и аппаратным регистрам можно, отобразив память ядра в пользовательское адресное пространство в обход ядра. Правда, все равно может понадобиться код в ядре для обработки прерываний и организации прямого доступа к памяти. Существует подсистема uio (сокращение от «user I/O» – пользовательский ввод-вывод), инкапсулирующая эту идею. Документация по ней содержится в файле Documentation/DocBook/uio-howto, а примеры драйверов можно найти в каталоге drivers/uio.
- sigio. Функция kill_fasync() позволяет послать из драйвера сигнал, чтобы уведомить приложение о событии, например о том, что появились входные данные или что произошло прерывание. По соглашению для этой цели используется сигнал SIGIO, но можно послать любой другой. Примеры можно найти в драйвере UIO (файл drivers/uio/uio.c) и в драйвере RTC (файл drivers/char/rtc.c). Основная проблема – в трудности написания надежного обработчика сигналов, из-за чего эта возможность остается слабо востребованной.

- debugfs. Это еще одна псевдофайловая система, в которой данные ядра представлены в виде файлов и каталогов, как в proc и sysfs. Основное отличие заключается в том, что debugfs не должна содержать информацию, необходимую для нормальной работы системы; она предназначена только для отладочных и трассировочных данных. Монтируется командой mount t debugfs debug /sys/kernel/debug. Хорошее описание debugfs имеется в файле Documentation/filesystems/debugfs.txt.
- proc. Файловую систему proc не рекомендуется использовать в новых программах, если только речь не идет о процессах, для чего она изначально и задумывалась. Однако вы можете публиковать в proc любую информацию, какую сочтете нужным. И, в отличие от sysfs и debugfs, ее разрешено использовать в модулях, на которые не распространяется лицензия GPL.
- netlink. Это семейство протоколов, основанных на сокетах. Если задать параметр AF_NETLINK, то будет создан сокет, связывающий адресное пространство ядра с пользовательским пространством. Первоначально задумывалось для того, чтобы сетевые инструментальные программы могли взаимодействовать с сетевой подсистемой Linux: получать доступ к таблицам маршрутизации и т. д. Также используется диспетчером устройств udev для передачи событий от ядра демону udev. Очень редко применяется в драйверах общего назначения.

В исходном коде ядра много примеров работы с описанными файловыми системами, так что вы можете спроектировать весьма интересные интерфейсы с кодом своего драйвера. Главное – придерживаться универсального принципа наименьшего удивления. Иными словами, авторы приложений, в которых используется ваш драйвер, не должны сталкиваться с сюрпризами или странностями – все должно быть последовательно и логично.

Анатомия драйвера устройства

Настало время соединить разрозненные концы и познакомиться с кодом простого драйвера устройства.

Ниже приведен полный код драйвера с именем dummy, который создает четыре устройства, доступных по именам от /dev/dummy0 до /dev/dummy3.

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#define DEVICE_NAME "dummy"
#define MAJOR_NUM 42
#define NUM_DEVICES 4

static struct class *dummy_class;
static int dummy_open(struct inode *inode, struct file *file)
{
```

```
  pr_info("%s\n", __func__);
  return 0;
}

static int dummy_release(struct inode *inode, struct file *file)
{
  pr_info("%s\n", __func__);
  return 0;
}

static ssize_t dummy_read(struct file *file,
    char *buffer, size_t length, loff_t * offset)
{
  pr_info("%s %u\n", __func__, length);
  return 0;
}

static ssize_t dummy_write(struct file *file,
    const char *buffer, size_t length, loff_t * offset)
{
  pr_info("%s %u\n", __func__, length);
  return length;
}

struct file_operations dummy_fops = {
  .owner = THIS_MODULE,
  .open = dummy_open,
  .release = dummy_release,
  .read = dummy_read,
  .write = dummy_write,
};

int __init dummy_init(void)
{
  int ret;
  int i;
  printk("Dummy loaded\n");
  ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &dummy_fops);
  if (ret != 0)
    return ret;
  dummy_class = class_create(THIS_MODULE, DEVICE_NAME);
  for (i = 0; i < NUM_DEVICES; i++) {
    device_create(dummy_class, NULL,
    MKDEV(MAJOR_NUM, i), NULL, "dummy%d", i);
  }
  return 0;
}

void __exit dummy_exit(void)
{
```

```
  int i;
  for (i = 0; i < NUM_DEVICES; i++) {
    device_destroy(dummy_class, MKDEV(MAJOR_NUM, i));
  }
  class_destroy(dummy_class);
  unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
  printk("Dummy unloaded\n");
}

module_init(dummy_init);
module_exit(dummy_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Chris Simmonds");
MODULE_DESCRIPTION("A dummy driver");
```

В самом конце находятся макросы `module_init` и `module_exit`, которые задают функции, вызываемые при загрузке и выгрузке модуля. Следующие три макроса добавляют базовую информацию о модуле, которую можно извлечь из откомпилированного кода модуля командой `modinfo`.

Сразу после загрузки модуля вызывается функция `dummy_init()`.

Для того чтобы показать, что это драйвер символьного устройства, вызывается функция `register_chrdev`, которой передается структура `file_operations`, содержащая указатели на четыре функции, реализованные в драйвере. Хотя `register_chrdev` сообщает ядру о существовании драйвера со старшим номером 42, она ничего не говорит о типе этого драйвера, поэтому не создается подкаталог в `/sys/class`. Не видя ничего в `/sys/class`, диспетчер устройств не сможет создать узлы устройств. Поэтому в следующих строках создаются класс устройства, `dummy`, и четыре устройства этого класса с именами от `dummy0` до `dummy3`. В результате образуются каталог `/sys/class/dummy` и в нем подкаталоги `dummy0`, ..., `dummy3`, в каждом из которых находится файл `dev`, содержащий старший и младший номера устройства. Это все, что необходимо диспетчеру устройств для создания узлов `/dev/dummy0`, ..., `/dev/dummy3`.

Функция `exit` должна освободить ресурсы, захваченные функцией `init`, в данном случае это означает освобождение класса устройства и старшего номера.

Файловые операции этого драйвера реализованы функциями `dummy_open()`, `dummy_read()`, `dummy_write()` и `dummy_release()`, они вызываются, когда программа в пользовательском пространстве вызывает `open(2)`, `read(2)`, `write(2)` и `close(2)`. Каждая функция просто выводит сообщение ядра, чтобы было видно, что к ней обращались. Продемонстрировать это можно, выполнив команду `echo`:

```
# echo hello > /dev/dummy0

[ 6479.741192] dummy_open
[ 6479.742505] dummy_write 6
[ 6479.743008] dummy_release
```

В данном случае сообщения видны, потому что я вошел в систему с консоли, а сообщения ядра по умолчанию выводятся на консоль.

Полный исходный код драйвера насчитывает меньше 100 строк, но и этого достаточно, чтобы продемонстрировать связь между узлом устройства и кодом драйвера, создание класса устройства, благодаря чему диспетчер устройств может автоматически создавать узлы после загрузки драйвера, и перемещение данных между адресными пространствами ядра и пользователя. Следующий шаг – сборка драйвера.

Компиляция и загрузка

Итак, мы имеем код драйвера, который хотим откомпилировать и протестировать в целевой системе. Мы можем скопировать его в дерево исходного кода ядра и модифицировать make-файлы, так чтобы он собирался, а можем откомпилировать как отдельно стоящий модуль. Начнем с последнего.

Нам понадобится простой make-файл, в котором для выполнения всей сложной работы используется система сборки ядра:

```
LINUXDIR := $(HOME)/MELP/build/linux

obj-m := dummy.o
all:
	make ARCH=arm CROSS_COMPILE=arm-cortex_a8-linux-gnueabihf- \
	    -C $(LINUXDIR) M=$(shell pwd)
clean:
	make -C $(LINUXDIR) M=$(shell pwd) clean
```

Присвоим переменной `LINUXDIR` путь к каталогу ядра для целевого устройства, на котором будет работать модуль. Строка `obj-m := dummy.o` означает, что будет вызвано правило сборки ядра, которое преобразует исходный файл `dummy.c` в модуль ядра `dummy.ko`. Отметим, что модули не являются двоично совместимыми с различными версиями и конфигурациями ядра, модуль загрузится только в то ядро, вместе с которым компилировался.

Конечным результатом сборки является файл `dummy.ko`, который можно скопировать в целевую систему и загрузить, как показано в следующем разделе.

Если мы хотим строить драйвер в составе дерева исходного кода ядра, то процедура еще проще. Выберите каталог, соответствующий типу драйвера. Поскольку мы имеем простой драйвер символьного устройства, то я помещу файл `dummy.c` в каталог `drivers/char`. Затем отредактируйте находящийся в этом каталоге make-файл, добавив в него строку для безусловной сборки драйвера в виде модуля:

```
obj-m += dummy.o
```

Или строку для безусловного встраивания в ядро:

```
obj-y += dummy.o
```

Если вы хотите сделать драйвер факультативным, то можете добавить пункт меню в файл `Kconfig` и сделать компиляцию зависящей от соответствующего конфигурационного параметра, как было описано в главе 4.

Загрузка модулей ядра

Для загрузки, выгрузки и перечисления модулей существуют простые команды insmod, rmmod и lsmod. Вот как загружается драйвер dummy:

```
# insmod /lib/modules/4.1.10/kernel/drivers/dummy.ko
# lsmod
dummy 1248 0 - Live 0xbf009000 (O)
# rmmod dummy
```

Если модуль помещен в подкаталог /lib/modules/<версия ядра>, как в этом примере, то можно создать базу данных о зависимостях между модулями, выполнив команду depmod:

```
# depmod -a
# ls /lib/modules/4.1.10/
kernel             modules.builtin.bin    modules.order
modules.alias      modules.dep            modules.softdep
modules.alias.bin  modules.dep.bin        modules.symbols
modules.builtin    modules.devname        modules.symbols.bin
```

Информация в файлах module.* используется командой modprobe для поиска модуля по имени, а не по полному пути. У modprobe много других возможностей, все они описаны в руководстве.

Сведения о зависимости между модулями используются также диспетчерами устройств и, в частности, udev. Когда обнаруживается новое оборудование, к примеру USB-устройство, демон udevd получает уведомление, сопровождаемое названием производителя и идентификаторами изделия, полученными от оборудования. Демон udevd просматривает файлы с данными о зависимостях в поисках модуля, который зарегистрировал эти идентификаторы. Если таковой найден, то он загружается с помощью modprobe.

Определение конфигурации оборудования

На примере драйвера dummy мы продемонстрировали структуру драйвера устройства, но в его коде отсутствует взаимодействие с реальным оборудованием, он лишь манипулирует структурами данных в памяти. Обычно драйверы пишутся для работы с оборудованием, и одна из первых задач – умение определить наличие оборудования при том, что его адрес может зависеть от конфигурации.

В некоторых случаях оборудование само предоставляет информацию о себе. Устройства, подключенные к шине, допускающей обнаружение, например PCI или USB, имеют режим опроса, в котором возвращают требования к ресурсам и уникальный идентификатор. Ядро сравнивает идентификатор и, возможно, другие характеристики с тем, что зарегистрировал драйвер, и сопоставляет одно с другим.

Однако у большинства аппаратных модулей SoC-системы таких идентификаторов нет. Вы должны сами предоставить информацию в виде дерева устройств или структур C, называемых платформенными данными.

В стандартной модели драйверов в Linux драйвер регистрируется в соответствующей ему подсистеме: PCI, USB, открытая прошивка (дерева устройств), платформенное устройство и т. д. При регистрации указываются идентификатор и функция обратного вызова, называемая функцией зондирования (probe function), которая вызывается, когда идентификатор драйвера и идентификатор оборудования совпадают. Для шин PCI и USB идентификатор строится из названия поставщика и идентификатора изделия, а для дерева устройств и платформенных устройств это просто некоторое имя (строка в кодировке ASCII).

Деревья устройств

С деревьями устройств мы познакомились в главе 3. Здесь же я покажу, как драйверы устройств пользуются этой информацией.

В качестве примера возьму плату ARM Versatile, `arch/arm/boot/dts/ versatile-ab.dts`, для которой адаптер Ethernet определен следующим образом:

```
net@10010000 {
  compatible = "smsc,lan91c111";
  reg = <0x10010000 0x10000>;
  interrupts = <25>;
};
```

Платформенные данные

Если дерево устройств не поддерживается, то есть резервный метод описания оборудования в виде структур C, известных под названием «платформенные данные».

Каждая единица оборудования описывается структурой `struct platform_device`, в которой хранятся имя и указатель на массив ресурсов. Тип ресурса определяется флагами:

- `IORESOURCE_MEM`: физический адрес области памяти;
- `IORESOURCE_IO`: физический адрес или номер порта регистров ввода-вывода;
- `IORESOURCE_IRQ`: номер прерывания.

Ниже приведен пример платформенных данных для адаптера Ethernet, взятый из файла `arch/arm/mach-versatile/core.c` и немного отредактированный, чтобы было понятнее:

```
#define VERSATILE_ETH_BASE 0x10010000
#define IRQ_ETH 25
static struct resource smc91x_resources[] = {
  [0] = {
    .start    = VERSATILE_ETH_BASE,
    .end      = VERSATILE_ETH_BASE + SZ_64K - 1,
    .flags    = IORESOURCE_MEM,
  },
  [1] = {
    .start    = IRQ_ETH,
    .end      = IRQ_ETH,
    .flags    = IORESOURCE_IRQ,
```

```
  },
};
static struct platform_device smc91x_device = {
  .name          = "smc91x",
  .id            = 0,
  .num_resources = ARRAY_SIZE(smc91x_resources),
  .resource      = smc91x_resources,
};
```

Здесь определены область памяти размером 64 КиБ и прерывание. Платформенные данные необходимо зарегистрировать в ядре, обычно на этапе инициализации платы:

```
void __init versatile_init(void)
{
  platform_device_register(&versatile_flash_device);
  platform_device_register(&versatile_i2c_device);
  platform_device_register(&smc91x_device);
  [ ...]
```

Связывание оборудования с драйверами

В предыдущем разделе было показано, как адаптер Ethernet описывается в виде дерева устройств и платформенных данных. Соответствующий драйвер находится в файле `drivers/net/ethernet/smsc/smc91x.c` и способен работать как с тем, так и с другим. Вот как выглядит код инициализации (после незначительного форматирования):

```
static const struct of_device_id smc91x_match[] = {
  { .compatible = "smsc,lan91c94", },
  { .compatible = "smsc,lan91c111", },
  {},
};
MODULE_DEVICE_TABLE(of, smc91x_match);
static struct platform_driver smc_driver = {
  .probe = smc_drv_probe,
  .remove = smc_drv_remove,
  .driver = {
    .name = "smc91x",
    .of_match_table = of_match_ptr(smc91x_match),
  },
};
static int __init smc_driver_init(void)
{
  return platform_driver_register(&smc_driver);
}
static void __exit smc_driver_exit(void) \
{
  platform_driver_unregister(&smc_driver);
}
```

```
module_init(smc_driver_init);
module_exit(smc_driver_exit);
```

На этапе инициализации драйвер вызывает функцию `platform_driver_register()`, передавая ей указатель на структуру `struct platform_driver`, в которой хранятся указатель на функцию зондирования, имя драйвера (`smc91x`) и указатель на структуру `struct of_device_id`.

Если драйвер конфигурируется с помощью дерева устройств, то ядро будет искать совпадения свойства `compatible` в узле дерева устройств со строкой, на которую указывает элемент структуры `compatible`. Всякий раз, обнаружив совпадение, ядро вызывает функцию `probe`.

Если же драйвер конфигурируется с помощью платформенных данных, то функция `probe` вызывается при каждом совпадении со строкой, на которую указывает `driver.name`.

Функция `probe` извлекает информацию об интерфейсе:

```
static int smc_drv_probe(struct platform_device *pdev)
{
  struct smc91x_platdata *pd = dev_get_platdata(&pdev->dev);
  const struct of_device_id *match = NULL;
  struct resource *res, *ires;
  int irq;

  res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
  ires = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
  [...]

  addr = ioremap(res->start, SMC_IO_EXTENT);
  irq = ires->start;
  [...]
}
```

Функция `platform_get_resource()` читает сведения о памяти и `irq` либо из дерева устройств, либо из платформенных данных. В задачу драйвера входят отображение памяти и установка обработчика прерываний. Третий параметр, равный нулю в обоих показанных выше случаях, вступает в игру, если существует несколько ресурсов одного типа.

Деревья устройств позволяют конфигурировать не только простые диапазоны памяти и прерывания. В следующем фрагменте функции `probe` читаются дополнительные параметры из дерева устройств. В данном случае мы читаем свойство `register-io-width`:

```
match = of_match_device(of_match_ptr(smc91x_match), &pdev->dev);
if (match) {
  struct device_node *np = pdev->dev.of_node;
  u32 val;
  [...]
  of_property_read_u32(np, "reg-io-width", &val);
  [...]
}
```

Привязки большинства драйверов документированы в файле Documentation/devicetree/bindings. Но информация об этом конкретном драйвере находится в файле Documentation/devicetree/bindings/net/smsc911x.txt.

Главное, что нужно помнить, – тот факт, что драйвер должен зарегистрировать функцию probe и предоставить ядру достаточно информации для вызова probe при обнаружении оборудования, о котором оно знает. Связь между оборудованием, описанным в дереве устройств, и драйвером устанавливается по свойству compatible. Связь между платформенными данными и драйвером устанавливается по имени.

Дополнительная литература

Ниже перечислены ресурсы, в которых можно найти дополнительные сведения по вопросам, затронутым в данной главе.

- *McKellar J., Rubini A., Corbet J., Kroah-Hartman G.* Linux Device Drivers. 4-е изд. На момент написания этой книги четвертое издание еще не вышло, но если будет столь же хорошим, как предыдущее, то это отличный выбор. Само же третье издание слишком устарело, поэтому я не могу его рекомендовать.
- *Love R.* Linux Kernel Development. 3rd ed. Addison-Wesley Professional, 2010. ISBN-10: 0672329468.
- Linux Weekly News. URL: lwn.net.

Резюме

Задача драйвера – взаимодействие с устройством, обычно с физическим оборудованием, но иногда и с виртуальными интерфейсами, и представление его верхним уровням системы в единообразном и полезном виде. Драйверы в Linux можно отнести к одной из трех категорий: символьных, блочных и сетевых устройств. Из них наибольшей гибкостью обладает интерфейс драйверов символьных устройств, а потому он и встречается чаще всего. В Linux существует специальная инфраструктура, которая называется моделью драйверов и раскрывается через псевдофайловую систему sysfs. Из каталога /sys можно получить целостное представление об устройствах и драйверах.

У любой встраиваемой системы имеется уникальный набор аппаратных интерфейсов и требований. Linux предоставляет драйверы для большинства стандартных интерфейсов, поэтому, правильно сконфигурировав ядро, вы сможете очень быстро заставить устройство работать. Остаются нестандартные компоненты, для которых придется добавить поддержку самостоятельно.

В некоторых случаях проблему удается обойти с помощью обобщенных драйверов для GPIO, I2C и т. д. и ограничиться написанием кода, работающего в пользовательском пространстве. Я рекомендую начинать именно с этого, поскольку это даст возможность лучше познакомиться с оборудованием без написания кода ядра. Писать драйверы устройств не особенно трудно, но если уж вы решитесь на

это, то кодировать нужно аккуратно, чтобы не поставить под угрозу стабильность системы.

Я начал говорить о том, как писать код драйвера, работающего в ядре: если вы пойдете по этому пути, то неизбежно захотите узнать, правильно ли он работает, и выловить ошибки. Этим вопросом мы займемся в главе 12 «Отладка в GDB».

Следующая глава посвящена инициализации в пользовательском пространстве и различным вариантам программы `init` – от простого комплекта BusyBox до сложной системы systemd.

Глава 9

Инициализация системы – программа init

В главах 4, 5 и 6 я рассмотрел процесс загрузки ядра до момента запуска первой программы, init. Мы видели, как создавать корневые файловые системы различной сложности, и все они содержали программу init. Пришло время поговорить о init подробнее и разобраться, почему она так важна для всей системы.

Есть много реализаций init. Я опишу три: BusyBox init, System V init и systemd. В каждом случае я расскажу, как программа работает и для каких типов систем подходит лучше всего. Частично решение основано на компромиссе между сложностью и гибкостью.

После того как ядро загрузилось

В главе 4 мы видели, что начальный загрузчик ядра ищет корневую файловую систему – либо в виде initramfs, либо в каталоге, указанном в параметре root= командной строки ядра, – а затем выполняет программу /init в случае initramfs или /sbin/init в случае обычной файловой системы. Программа init работает с привилегиями root, а поскольку это первый исполняемый процесс, то его идентификатор (PID) равен 1. Если по какой-то причине выполнить init не удается, ядро паникует.

Программа init является предком всех остальных процессов, как показывает команда pstree, в большинстве дистрибутивов входящая в состав пакета psmisc:

```
# pstree -gn

init(1)-+-syslogd(63)
        |-klogd(66)
        |-dropbear(99)
        `-sh(100)---pstree(109)
```

Задача init – взять на себя управление системой и привести ее в работоспособное состояние. Это может быть простой скрипт оболочки (такой пример был приведен в начале главы 5), но в большинстве случаев используется специальная программа-демон init. Перечислим ее задачи.

- На этапе загрузки она запускает демоны, настраивает параметры системы и выполняет другие действия, необходимые для приведения системы в рабочее состояние.
- Факультативно на терминалах запускаются демоны типа getty, позволяющие войти в систему и получить начальную оболочку.
- Усыновляет процессы, осиротевшие из-за того, что их непосредственный родитель завершился, а других потоков в той же группе не осталось.
- Реагирует на завершение непосредственных потомков init, перехватывая сигнал SIGCHLD и получая код возврата, тем самым предотвращает появление процессов-зомби. О том, что такое зомби, мы будем говорить в главе 10.
- Факультативно перезапускает завершившиеся демоны.
- Обрабатывает остановку системы.

Иными словами, init отвечает за весь жизненный цикл системы от загрузки до остановки. В настоящее время превалирует мнение, что init удачно расположена для того, чтобы обрабатывать и другие события во время выполнения, как, например, добавление нового оборудования и загрузку и выгрузку модулей. Именно этим и занимается systemd.

Введение в программы init

Скорее всего, при работе со встраиваемыми системами вам придется столкнуться с тремя программами init: BusyBox init, System V init и systemd. Buildroot поддерживает построение всех трех, по умолчанию отдавая предпочтение BusyBox init. Yocto Project позволяет выбрать между System V init и systemd, по умолчанию выбирается System V init.

В следующей таблице приведены некоторые характеристики всех трех программ.

	BusyBox init	System V init	systemd
Сложность	Малая	Средняя	Большая
Время загрузки	Быстрая	Медленная	Средняя
Требуемая оболочка	ash	ash или bash	Нет
Число исполняемых файлов	0	4	50 (*)
libc	Любая	Любая	glibc
Размер (МиБ)	0	0.1	34 (*)

(*) При той конфигурации systemd, которая используется в Buildroot.

Таким образом, при движении от BusyBox init к systemd сложность и гибкость возрастают.

BusyBox init

В комплект BusyBox включена минимальная программа init, которая пользуется конфигурационным файлом /etc/inittab, где определены правила запуска про-

грамм при загрузке и их завершения при остановке системы. Основную работу обычно выполняют скрипты оболочки, которые по соглашению находятся в каталоге /etc/init.d.

Первым делом init читает конфигурационный файл /etc/inittab. Он содержит перечень запускаемых программ, по одной на строку, в следующем формате:

```
<id>::<action>:<program>
```

где:

- id: управляющий терминал для данной команды;
- action: условия выполнения этой команды, описанные ниже;
- program: запускаемая программа.

Параметр action (действие) может принимать следующие значения:

- sysinit: выполнять программу при запуске init до обработки действий любых других типов;
- respawn: выполнить эту программу и перезапускать в случае завершения. Используется для запуска программ-демонов;
- askfirst: то же, что respawn, но предварительно на консоли печатается сообщение **Please press Enter to activate this console** (Нажмите Enter, чтобы активировать эту консоль), и программа запускается только после нажатия **Enter**. Применяется для запуска интерактивной оболочки на терминале без запроса имени и пароля пользователя;
- once: выполнить программу и не пытаться перезапустить ее в случае завершения;
- wait: выполнить программу и ждать ее завершения;
- restart: выполнить программу, когда init получает сигнал SIGHUP, означающий, что необходимо перечитать файл inittab;
- ctrlaltdel: выполнить программу, когда init получает сигнал SIGINT, обычно после нажатия **Ctrl**+**Alt**+**Del** на консоли;
- shutdown: выполнить программу, когда init завершается.

В следующем примере монтируются файловые системы proc и sysfs и запускается оболочка на консоли, подключенной к последовательному интерфейсу:

```
null::sysinit:/bin/mount -t proc proc /proc
null::sysinit:/bin/mount -t sysfs sysfs /sys
console::askfirst:-/bin/sh
```

В простых проектах, где нужно запустить всего несколько демонов и, быть может, начальную оболочку на последовательном терминале, скрипт легко написать вручную, так что такой подход вполне годится для «самопальной» встраиваемой Linux-системы. Но по мере того как количество подлежащих конфигурированию аспектов возрастает, поддерживать «рукописные» скрипты инициализации становится все труднее. Они, как правило, оказываются не очень модульными, а значит, нуждаются в обновлении при добавлении новых компонентов.

Скрипты инициализации в Buildroot

В Buildroot уже много лет эффективно используется программа `init` из BusyBox. В Buildroot есть два скрипта в каталоге `/etc/init.d`: `rcS` и `rcK`. Первый работает на этапе загрузки: находит все скрипты, имена которых начинаются с заглавной буквы `S` и двух последующих цифр, и выполняет их в алфавитном порядке. Это стартовые скрипты. Скрипт `rcK` работает на этапе остановки системы: находит все скрипты, имена которых начинаются с заглавной буквы `K` и двух последующих цифр, и выполняет их в алфавитном порядке. Это завершающие скрипты (K означает «kill» – убить).

При такой схеме в пакет Buildroot легко включить стартовый и завершающий скрипты, используя две цифры для задания порядка выполнения, так что система становится расширяемой. Если вы пользуетесь Buildroot, то все это делается незаметно для вас. Если нет, то можете написать собственные скрипты для BusyBox `init`, следуя этому образцу.

System V init

Эта программа `init` своими корнями уходит в систему UNIX System V, датируемую серединой 1980-х годов. Версия, чаще всего встречающаяся в дистрибутивах Linux, первоначально была написана Микелем ван Смуренбургом (Miquel van Smoorenburg). До недавнего времени она считалась основным способом загрузки Linux, включая и встраиваемые системы, а BusyBox `init` представляет собой урезанный вариант System V `init`.

По сравнению с BusyBox `init`, у System V `init` два преимущества. Во-первых, загрузочные скрипты написаны в хорошо известном модульном стиле, что позволяет легко добавлять новые пакеты на этапе сборки или выполнения. Во-вторых, определено понятие уровня выполнения, что дает возможность запускать или останавливать целый набор программ одним движением, переключаясь с одного уровня на другой.

Существуют 8 уровней выполнения с номерами от 0 до 6 плюс S:

- **S**: однопользовательский режим;
- **0**: остановить систему;
- **1–5**: для общего пользования;
- **6**: перезагрузить систему.

Уровни 1–5 можно использовать как угодно. В дистрибутивах Linux для ПК им обычно назначают такую семантику:

- **1**: однопользовательский режим;
- **2**: многопользовательский режим без сети;
- **3**: многопользовательский режим с сетью;
- **4**: не используется;
- **5**: многопользовательский режим с графическим входом в систему.

Программа init инициализирует систему на уровне выполнения, который задается в строке initdefault файла /etc/inittab. Уровень можно изменить на этапе выполнения с помощью команды telinit [уровень выполнения], которая посылает сообщение init. Узнать текущий и предыдущий уровни выполнения позволяет команда runlevel, например:

```
# runlevel
N 5
# telinit 3
INIT: Switching to runlevel: 3
# runlevel
5 3
```

При первом запуске runlevel напечатала N 5, это означает, что предыдущего уровня выполнения не было, так как уровень не изменялся после загрузки системы, а текущий уровень равен 5. После смены уровня выполнения runlevel печатает 5 3, сообщая, что имел место переход от уровня 5 к уровню 3. Команды halt и reboot изменяют уровень выполнения на 0 и 6 соответственно. Чтобы изменить уровень выполнения по умолчанию, нужно указать его в командной строке ядра: цифру от 0 до 6 или S для загрузки в однопользовательском режиме. Вот, например, как выглядит командная строка ядра для загрузки в однопользовательском режиме:

```
console=ttyAMA0 root=/dev/mmcblk1p2 S
```

Для каждого уровня выполнения существуют завершающие скрипты, призванные остановить работу каких-то программ, и стартовые скрипты, которые запускают определенные программы. При переходе на новый уровень выполнения init сначала выполняет завершающие скрипты, а затем стартовые. Работающим демонам, для которых на новом уровне нет ни завершающего, ни стартового скриптов, посылается сигнал SIGTERM. Иными словами, по умолчанию при переходе на новый уровень выполнения демоны завершаются, если явно не указано противное.

По правде говоря, во встраиваемых Linux-системах уровни выполнения используются редко: большинство устройств после загрузки оказывается на уровне выполнения по умолчанию и там и остаются. У меня такое ощущение, что отчасти по этой причине народ про них ничего не знает.

Уровни выполнения – простой и удобный способ переключения режимов, например из производственного в режим обслуживания.

Программу System V init можно выбрать как в Buildroot, так и в Yocto Project. В обоих случаях из скриптов инициализации убрана специфика bash, так что они работают и с оболочкой ash из BusyBox. Однако Buildroot жульничает: заменяет BusyBox init на SystemV init и добавляет inittab, чтобы имитировать поведение BusyBox. В Buildroot уровни выполнения не реализованы в полном объеме, но переключение на уровень 0 или 6 соответственно останавливает и перезагружает систему.

Далее мы рассмотрим некоторые детали. Приведенные ниже примеры взяты из версии fido проекта Yocto Project. В других дистрибутивах скрипты `init` могут выглядеть несколько иначе.

inittab

Программа `init` начинает работу с чтения файла `/etc/inttab`, в котором определено, что должно происходить на каждом уровне выполнения. С точки зрения формата, это обобщение файла `inittab` из комплекта BusyBox, описанного в предыдущем разделе, что и неудивительно, поскольку BusyBox заимствовал `init` как раз из System V!

Строки `inittab` устроены следующим образом:

```
id:runlevels:action:process
```

где:

- `id`: уникальный идентификатор, содержащий до 4 символов;
- `runlevels`: уровни выполнения, на которых должна выполняться данная строка (в BusyBox `inittab` это поле оставлено пустым);
- `action`: одно из описанных ниже ключевых слов;
- `process`: запускаемая команда.

Поле `action` может принимать те же значения, что в BusyBox `init`: `sysinit`, `respawn`, `once`, `wait`, `restart`, `ctrlaltdel` и `shutdown`. Но в System V `init` нет действия `askfirst`, являющегося спецификой BusyBox.

Ниже приведен пример полного файла `inttab`, включенного в образ core-image-minimal из Yocto Project:

```
# /etc/inittab: init(8) configuration.
# $Id: inittab,v 1.91 2002/01/25 13:35:21 miquels Exp $

# The default runlevel.
id:5:initdefault:

# Boot-time system configuration/initialization script.
# This is run first except when booting in emergency (-b) mode.
si::sysinit:/etc/init.d/rcS

# What to do in single-user mode.
~~:S:wait:/sbin/sulogin
# /etc/init.d executes the S and K scripts upon change
# of runlevel.
#
# Runlevel 0 is halt.
# Runlevel 1 is single-user.
# Runlevels 2-5 are multi-user.
# Runlevel 6 is reboot.

l0:0:wait:/etc/init.d/rc 0
l1:1:wait:/etc/init.d/rc 1
l2:2:wait:/etc/init.d/rc 2
```

```
l3:3:wait:/etc/init.d/rc 3
l4:4:wait:/etc/init.d/rc 4
l5:5:wait:/etc/init.d/rc 5
l6:6:wait:/etc/init.d/rc 6

# Normally not reached, but fallthrough in case of emergency.
z6:6:respawn:/sbin/sulogin
AMA0:12345:respawn:/sbin/getty 115200 ttyAMA0
# /sbin/getty invocations for the runlevels.
#
# The "id" field MUST be the same as the last
# characters of the device (after "tty").
#
# Format:
# <id>:<runlevels>:<action>:<process>
#

1:2345:respawn:/sbin/getty 38400 tty1
```

В первой строке `id:5:initdefault` устанавливается уровень выполнения по умолчанию 5. В следующей строке `si::sysinit:/etc/init.d/rcS` описывается скрипт `rcS`, который выполняется на этапе загрузки. Мы еще вернемся к этой строке ниже. Немного дальше расположена группа из шести строк, начинающаяся строкой `l0:0:wait:/etc/init.d/rc 0`. В каждой из них запускается скрипт `/etc/init.d/rc` при смене уровня выполнения: этот скрипт отвечает за выполнение стартовых и завершающих скриптов. Существует также строка для уровня выполнения `S`, в которой описан запуск однопользовательской программы входа.

В конце `inittab` мы видим две строки для запуска программы `getty`: при переходе на уровень выполнения от 1 до 5 она выводит приглашение на устройства `/dev/ttyAMA0` и `/dev/tty1`. Это позволяет войти в систему и получить интерактивную оболочку.

```
AMA0:12345:respawn:/sbin/getty 115200 ttyAMA0
1:2345:respawn:/sbin/getty 38400 tty1
```

Устройство `ttyAMA0` – это последовательная консоль на плате ARM Versatile, эмулируемой в QEMU; для других макетных плат устройство будет другим. `Tty1` – виртуальная консоль, которой часто сопоставляется графический экран, если ядро собрано с одним из параметров `CONFIG_FRAMEBUFFER_CONSOLE` или `VGA_CONSOLE`. В Linux для ПК обычно запускаются шесть процессов `getty` на виртуальных терминалах с номерами от 1 до 6, на которые можно переключиться нажатием комбинаций клавиш **Ctrl**+**Alt**+**F1**, ..., **Ctrl**+**Alt**+**F6**, а виртуальный терминал 7 зарезервирован для графического экрана. Во встраиваемых системах виртуальные терминалы используются редко.

В строке `sysinit` запускается скрипт `/etc/init.d/rcS`, который просто переходит на уровень выполнения `S`:

```
#!/bin/sh

[...]
exec /etc/init.d/rc S
```

Таким образом, сначала производится переход на уровень выполнения S, а уже потом – на уровень по умолчанию 5. Отметим, что уровень S не запоминается и никогда не отображается, как предыдущий, командой runlevel.

Скрипты init.d

Каждому компоненту, который должен реагировать на изменение уровня выполнения, соответствует скрипт в каталоге /etc/init.d, отрабатывающий изменение. Скрипт ожидает передачи параметра, который может принимать одно из двух значений: start и stop. Пример будет приведен ниже.

Скрипт /etc/init.d/rc принимает новый уровень выполнения в качестве параметра. Каждому уровню выполнения соответствует каталог с именем rc<уровень выполнения>.d:

```
# ls -d /etc/rc*
/etc/rc0.d   /etc/rc2.d   /etc/rc4.d   /etc/rc6.d
/etc/rc1.d   /etc/rc3.d   /etc/rc5.d   /etc/rcS.d
```

В этих каталогах находятся скрипты с именами, начинающимися заглавной буквой S с двумя последующими цифрами, а также скрипты с именами, начинающимися заглавной буквой K. Это стартовые и завершающие скрипты: в Buildroot заимствована та же идея.

```
# ls /etc/rc5.d
S01networking   S20hwclock.sh   S99rmnologin.sh   S99stop-bootlogd
S15mountnfs.sh  S20syslog
```

Все это на самом деле символические ссылки на соответствующие скрипты в каталоге init.d. Скрипт rc сначала выполняет K-скрипты, передавая им параметр stop, а затем S-скрипты с параметром start. Две цифры, как и раньше, служат для задания порядка выполнения.

Добавление нового демона

Допустим, что имеется программа simpleserver, написанная как традиционный Unix-демон, т. е. она разветвляется и остается работать в фоновом режиме. Для нее потребуется примерно такой скрипт в каталоге init.d:

```
#! /bin/sh

case "$1" in
  start)
    echo "Starting simpelserver"
    start-stop-daemon -S -n simpleserver -a /usr/bin/simpleserver
    ;;
  stop)
    echo "Stopping simpleserver"
    start-stop-daemon -K -n simpleserver
    ;;
  *)
```

```
    echo "Usage: $0 {start|stop}"
  exit 1
esac

exit 0
```

Здесь start-stop-daemon – вспомогательная функция, упрощающая действия с фоновыми процессами. Первоначальная версия появилась в установщике пакетов Debian, dpkg, но в большинстве встраиваемых систем используется версия из BusyBox. С параметром -S функция запускает демон, гарантируя, что одновременно не будут запущены два экземпляра, а при задании параметра -K находит демон по имени и посылает ему сигнал – по умолчанию SIGTERM. Поместите этот скрипт в каталог /etc/init.d/simpleserver и сделайте его исполняемым.

Затем добавьте на него символические ссылки из каталогов, соответствующих тем уровням выполнения, на которых должна запускаться эта программа. В данном случае это только каталог, соответствующий уровню по умолчанию 5:

```
# cd /etc/init.d/rc5.d
# ln -s ../init.d/simpleserver S99simpleserver
```

Число 99 означает, что эта программа выполняется одной из последних. Имейте в виду, что могут существовать и другие скрипты, начинающиеся с S99, тогда скрипт rc будет выполнять их в лексикографическом порядке.

Во встраиваемых системах редко приходится беспокоиться об операциях остановки, но если что-то необходимо сделать на этом этапе, добавьте символические ссылки на завершающие скрипты в каталоги, соответствующие уровням 0 и 6:

```
# cd /etc/init.d/rc0.d
# ln -s ../init.d/simpleserver K01simpleserver
# cd /etc/init.d/rc6.d
# ln -s ../init.d/simpleserver K01simpleserver
```

Запуск и остановка служб

Скрипты в каталоге /etc/init.d можно вызывать и напрямую, например скрипт syslog, который управляет демонами syslogd и klogd:

```
# /etc/init.d/syslog --help
Usage: syslog { start | stop | restart }

# /etc/init.d/syslog stop
Stopping syslogd/klogd: stopped syslogd (pid 198)
stopped klogd (pid 201)
done

# /etc/init.d/syslog start
Starting syslogd/klogd: done
```

Все скрипты понимают параметры start и stop и, как правило, параметр help. В некоторых реализован также параметр status, при задании которого скрипт сообщает, работает служба или нет. В основных дистрибутивах, где еще использу-

ется System V init, имеется команда service, которая запускает и останавливает службы, скрывая детали работы со скриптами.

systemd

Проект systemd определяет себя как менеджер системы и служб. Он был начат в 2010 году Леннартом Петтерингом (Lennart Poettering) и Кэем Сиверсом (Kay Sievers) с целью создать комплексный набор инструментов для управления системой Linux и в том числе демоном init. Среди прочего включает диспетчер устройств (udev) и средства протоколирования. Бытует мнение, что это не просто программа init, а целый стиль жизни. В настоящее время эта система считается самой передовой и продолжает быстро развиваться. systemd типична в дистрибутивах Linux для ПК и серверов и становится популярной также во встраиваемых системах, особенно когда речь идет о достаточно сложных устройствах. Так почему же она лучше, чем System V init, для встраиваемых систем?

- Конфигурирование проще и логичнее (после того как вы поняли идею). Вместо не всегда понятных скриптов оболочки для System V init в systemd используются конфигурационные файлы-агрегаты для задания параметров.
- Между службами имеются явные зависимости, а не двузначные коды, которые всего лишь определяют последовательность выполнения скриптов.
- Для каждой службы легко задать разрешения и лимиты ресурсов, это важно с точки зрения безопасности.
- systemd умеет наблюдать за службами и при необходимости перезапускать их.
- Для каждой службы и для самого демона systemd определены сторожевые таймеры.
- Службы запускаются параллельно, что сокращает время загрузки.

Приводить здесь полное описание systemd неуместно, да и невозможно. Как и для System V init, я расскажу лишь о применении во встраиваемых системах, а примеры будут основаны на конфигурации, созданной в Yocto Fido, куда включена systemd версии 219. После краткого обзора я познакомлю вас с несколькими конкретными примерами.

Сборка systemd в Yocto Project и Buildroot

По умолчанию в версии Yocto Fido используется System V init. Чтобы выбрать systemd, добавьте в конфигурационный файл, например conf/local.conf, такие строки:

```
DISTRO_FEATURES_append = " systemd"
VIRTUAL-RUNTIME_init_manager = "systemd"
```

Обратите внимание на начальный пробел, он необходим! Затем пересоберите.

В Buildroot systemd включена как третий вариант init. Для нее необходимо выбирать glibc в качестве библиотеки C и ядро версии не ниже 3.7, собранное с опре-

деленными конфигурационными параметрами. Полный перечень зависимостей приведен в файле README на верхнем уровне исходного кода systemd.

Цели, службы и агрегаты

Прежде чем описывать, как работает systemd init, необходимо определить три ключевых понятия.

Целью (target) называется группа служб, это понятие похоже на уровень выполнения System V, но является более общим. Существует цель по умолчанию, это группа служб, запускаемых на этапе загрузки системы.

Службой (service) называется демон, который можно запустить и остановить; очень похоже на службу в смысле System V.

Наконец, агрегатом (unit) называется конфигурационный файл, в котором определена цель, служба или что-то еще. Агрегаты – это текстовые файлы, содержащие свойства и значения.

Изменить состояние и узнать, что происходит, позволяет команда systemctl.

Агрегаты

Основным элементом конфигурации является файл-агрегат. Агрегаты могут находиться в трех местах:

- /etc/systemd/system: локальная конфигурация;
- /run/systemd/system: конфигурация времени выполнения;
- /lib/systemd/system: дистрибутивная конфигурация.

В поисках агрегата systemd просматривает каталоги именно в таком порядке и останавливается, как только обнаружит подходящий файл. Это позволяет переопределить поведение дистрибутивного агрегата, поместив одноименный агрегат в каталог /etc/systemd/system. Чтобы полностью подавить агрегат, достаточно создать локальный файл, который пуст или ссылается на /dev/null.

Все агрегаты начинаются секцией [Unit], содержащей основную информацию и зависимости, например:

```
[Unit]
Description=D-Bus System Message Bus
Documentation=man:dbus-daemon(1)
Requires=dbus.socket
```

Зависимости в агрегатах определяются с помощью свойств Requires, Wants и Conflicts:

- Requires: список агрегатов, от которых зависит данный и которые должны быть запущены вместе с запуском данного;
- Wants: более слабая форма Requires: перечисленные агрегаты запускаются, но запуск данного не отменяется, если какой-то из них завершится с ошибкой;
- Conflicts: отрицательная зависимость: перечисленные агрегаты останавливаются при запуске данного и наоборот – если один из них запускается, то данный останавливается.

В процессе обработки зависимостей порождается список агрегатов, которые необходимо запустить (или остановить). Ключевые слова `Before` и `After` определяют порядок запуска. Порядок остановки противоположен порядку запуска.

- `Before`: данный агрегат должен быть запущен раньше перечисленных;
- `After`: данный агрегат должен быть запущен после перечисленных.

В примере ниже директива `After` гарантирует, что веб-сервер будет запущен после сети:

```
[Unit]
Description=Lighttpd Web Server
After=network.target
```

Если директивы `Before` и `After` отсутствуют, то агрегаты запускаются и останавливаются параллельно в произвольном порядке.

Службы

Службой называется демон, который можно запустить или остановить, это эквивалент службы в смысле System V. Службе соответствует файл-агрегат с именем, оканчивающимся на `.service`, например `lighttpd.service`.

В агрегате службы имеется секция `[Service]`, в которой описывается, что запустить. Вот как выглядит соответствующая секция в файле `lighttpd.service`:

```
[Service]
ExecStart=/usr/sbin/lighttpd -f /etc/lighttpd/lighttpd.conf -D
ExecReload=/bin/kill -HUP $MAINPID
```

Это команды, которые выполняются при запуске и перезапуске службы. Здесь может быть много других элементов конфигурации, все они описаны в странице руководств по `systemd.service`.

Цели

Цель – еще один тип агрегата, в котором группируются службы (или агрегаты других типов). В таких агрегатах есть только зависимости. Им соответствуют файлы с именами, заканчивающимися на `.target`, например `multi-user.target`. Цель – это желаемое состояние, оно играет ту же роль, что уровень выполнения в System V.

Как systemd загружает систему

Теперь посмотрим, как `systemd` реализует начальную загрузку системы. Ядро выполняет `systemd`, поскольку с `/sbin/init` ведет символическая ссылка на `/lib/systemd/systemd`. При этом запускается цель по умолчанию, `default.target`, которая всегда является ссылкой на желаемую цель, например `multi-user.target` в случае текстового входа в систему или `graphical.target` в случае графического. Так, если целью по умолчанию является `multi-user.target`, то имеется такая символическая ссылка:

```
/etc/systemd/system/default.target -> /lib/systemd/system/multi-user.target
```

Цель по умолчанию можно переопределить, передав в командной строке ядра параметр system.unit=<новая цель>. Чтобы узнать, какова цель по умолчанию, можно воспользоваться командой systemctl:

```
# systemctl get-default
multi-user.target
```

Запуск цели, например multi-user.target, создает дерево зависимостей, которые переводят систему в рабочее состояние. В типичной системе multi-user.target зависит от basic.target, та – от sysinit.target, а та – от служб, которые должны быть запущены на ранних стадиях. Команда systemctl list-dependencies строит и печатает граф зависимостей.

Команда systemctl list-units --type service выводит список всех служб с указанием текущего состояния, а команда systemctl list-units --type target – список целей.

Добавление своей службы

Продолжая пример simpleserver, выпишем агрегат службы:

```
[Unit]
Description=Simple server

[Service]
Type=forking
ExecStart=/usr/bin/simpleserver

[Install]
WantedBy=multi-user.target
```

В секции [Unit] находится только описание, чтобы служба корректно отображалась systemctl и другими командами. Зависимостей нет; как я и говорил, служба очень простая.

Секция [Service] указывает на исполняемую программу и содержит флаг, показывающий, что она разветвляется. Если бы программа была проще и работала в приоритетном режиме, то systemd сама превратила бы ее в демона, и флаг Type=forking был бы не нужен.

Благодаря секции [Install] служба оказывается зависимой от цели multi-user.target, так что наш сервер запускается, когда система переходит в многопользовательский режим.

Сохранив агрегат в файле /etc/systemd/system/simpleserver.service, мы можем запускать и останавливать его командами systemctl start simpleserver и systemctl stop simpleserver. Для получения текущего состояния службы выполним такую команду:

```
# systemctl status simpleserver
simpleserver.service - Simple server
Loaded: loaded (/etc/systemd/system/simpleserver.service; disabled)
```

```
Active: active (running) since Thu 1970-01-01 02:20:50 UTC; 8s ago
Main PID: 180 (simpleserver)
CGroup: /system.slice/simpleserver.service
        └─180 /usr/bin/simpleserver -n

Jan 01 02:20:50 qemuarm systemd[1]: Started Simple server.
```

Пока что служба запускается и останавливается только по команде. Чтобы это происходило автоматически, нужно добавить постоянную зависимость от цели. В этом и состоит назначение секции [Install] в агрегате: она говорит, что если данная служба включена, то она зависима от цели multi-user.target и, следовательно, будет запущена на этапе загрузки. Для включения службы используется команда systemctl enable:

```
# systemctl enable simpleserver
Created symlink from /etc/systemd/system/multi-user.target.wants/simpleserver.service to /
etc/systemd/system/simpleserver.service.
```

Теперь вы видите, как зависимости добавляются на этапе выполнения без необходимости редактировать файлы-агрегаты. Для цели может существовать каталог с именем <имя_цели>.target.wants, в котором определены ссылки на службы. Это в точности то же самое, что добавление зависимого агрегата в список [Wants] в файле цели. В данном случае мы обнаружим, что создана такая ссылка:

```
/etc/systemd/system/multi-user.target.wants/simpleserver.service
/etc/systemd/system/simpleserver.service
```

Если бы эта служба была важна, то имело бы смысл перезапустить ее в случае завершения. Для этого нужно добавить такое свойство в секцию [Service]:

```
Restart=on-abort
```

Свойство Restart может принимать также значения on-success, on-failure, on-abnormal, on-watchdog, on-abort, always.

Добавление сторожевого таймера

Наличие сторожевых таймеров – стандартное требование во встраиваемых устройствах: необходимо предпринять какие-то действия, если ответственная служба перестает работать; обычно перезагружают систему. В большинстве встраиваемых SoC-систем имеется аппаратный сторожевой таймер, доступ к которому дает узел устройства /dev/watchdog. При инициализации сторожевого таймера указывается тайм-аут, и до истечения тайм-аута таймер должен быть переустановлен, иначе он сработает и система перезагрузится. Интерфейс драйвера сторожевого таймера описан в файле Documentation/watchdog в исходном коде ядра, а его код находится в каталоге drivers/watchdog.

Проблема возникает, когда есть две и более ответственные службы, которые нужно защитить сторожевым таймером. В systemd имеется полезное средство, позволяющее разделить один таймер между несколькими службами.

Программу systemd можно настроить так, что она будет ожидать регулярного контрольного сообщения от службы и предпринимать определенные действия, если такое сообщение не поступит вовремя. Иначе говоря, организовать программный сторожевой таймер. Чтобы это работало, в код демона нужно добавить логику отправки контрольных сообщений. Демон должен проверять, находится ли в переменной окружения WATCHDOG_USEC ненулевое значение, и, если да, вызывать функцию sd_notify(false, "WATCHDOG=1") с интервалом не больше указанного количества микросекунд (рекомендуется, чтобы интервал был даже не больше половины этого значения). Примеры имеются в исходном коде systemd.

Чтобы включить сторожевой таймер в агрегат службы, добавьте в секцию [Service] строки вида:

```
WatchdogSec=30s
Restart=on-watchdog
StartLimitInterval=5min
StartLimitBurst=4
StartLimitAction=reboot-force
```

Здесь служба ожидает получать контрольные сообщения не реже, чем раз в 30 с. Если сообщение не придет, служба будет перезапущена, но если это происходит более чем четыре раза в течение пяти минут, то systemd принудительно перезагрузит систему. Полное описание всех параметров имеется в странице руководства по systemd.

Такой сторожевой таймер применим к отдельным службам, но что, если «упадет» сам демон systemd, или произойдет крах ядра, или зависнет оборудование? В таких случаях мы должны сказать systemd, что необходимо использовать драйвер сторожевого таймера: добавьте в файл /etc/systemd/system.conf.systemd строку RuntimeWatchdogSec=NN, означающую, что сторожевой таймер должен быть переустановлен в течение указанного времени; тогда если по какой-то причине systemd этого не сделает, то система перезагрузится.

Применение во встраиваемых Linux-системах

У systemd немало функций, полезных во встраиваемых Linux-системах, в том числе не упомянутых в предыдущем кратком описании, например: управление ресурсами с помощью срезов (см. страницы руководства systemd.slice(5) и systemd.resource-control(5)), управление устройствами (udev(7)) и средства системного протоколирования (journald(5)).

Но нужно соотносить все это с размером: даже минимальная сборка, в которую включены только базовые компоненты systemd, udevd и journald, занимает около 10 МиБ во внешней памяти, включая разделяемые библиотеки.

Не забывайте также, что разработка systemd тесно связана с разработкой ядра, поэтому система не будет работать с ядром, которое на год-два старше systemd.

Дополнительная литература

В указанном ниже ресурсе можно найти дополнительные сведения по вопросам, затронутым в данной главе:

- systemd system and Service Manager. URL: http://www.freedesktop.org/wiki/Software/systemd/ (в конце страницы много полезных ссылок).

Резюме

Любому устройству под управлением Linux нужна та или иная программа `init`. Если вы проектируете систему, которая должна запустить всего несколько демонов в самом начале, а затем остается относительно статической, то достаточно BusyBox `init`. Обычно это хороший выбор для систем, собираемых с помощью Buildroot.

Если же в системе имеются сложные зависимости между службами на этапе загрузки или выполнения и внешней памяти достаточно, то лучше обратиться к `systemd`. Даже в простейшем виде `systemd` обладает рядом полезных возможностей: управление сторожевыми таймерами, удаленное протоколирование и т. д., поэтому определенно заслуживает пристального внимания.

Трудно придумать, в каком случае следует предпочесть System V `init`, потому что у нее мало преимуществ, по сравнению с более простой BusyBox `init`. Тем не менее она еще проживет долго – просто потому, что уже существует. Например, если вы собираете систему с помощью Yocto Project и почему-либо настроены против `systemd`, то можно взять System V `init`.

Если говорить о времени загрузки, то `systemd` быстрее System V `init` при аналогичной рабочей нагрузке. Но если требуется что-то по-настоящему быстрое, то никто не может составить конкуренцию простой BusyBox `init` с минимальными загрузочными скриптами.

Эта глава была посвящена одному, хотя и очень важному процессу – `init`. В следующей главе я расскажу, что же такое процесс на самом деле, как он соотносится с потоками, как они взаимодействуют между собой и как планируется их выполнение. Эти вещи необходимо понимать, если вы собираетесь построить надежную встраиваемую систему, удобную для сопровождения.

Глава 10

Процессы и потоки

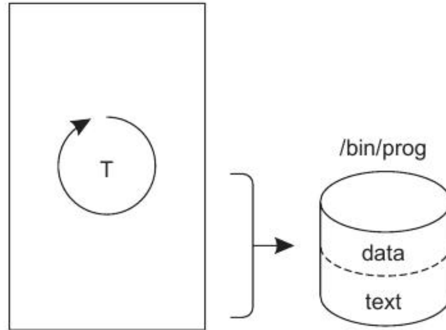
В предыдущих главах мы рассмотрели различные аспекты создания встраиваемой Linux-платформы. Пришло время взглянуть, как воспользоваться этой платформой для получения рабочего устройства. В этой главе мы поговорим о модели процессов в Linux и о том, как в нее укладываются многопоточные программы. Я объясню плюсы и минусы однопоточных и многопоточных процессов. Я также остановлюсь на планировании и различиях между политиками разделения времени и реального времени.

Хотя эти вопросы не являются спецификой встраиваемых систем, конструктор встраиваемого устройства должен иметь о них хотя бы общее представление. На эту тему есть немало отличных работ, некоторые из которых упомянуты в конце главы, но, вообще говоря, встраиваемые сценарии в них не затрагиваются. Поэтому я сосредоточу внимание на концепциях и проектных решениях, а не на описании вызовов функций и коде.

Процесс или поток?

Многие разработчики встраиваемых систем, знакомые с операционными системами реального времени (ОСРВ), считают модель процессов в Unix чрезмерно громоздкой. С другой стороны, они видят сходство между задачами в ОСРВ и потоками Linux, а потому испытывают соблазн перенести существующий проект, полагая, что имеется взаимно однозначное соответствие между тем и другим. Мне встречались проекты, в которых все приложение было реализовано в виде одного процесса, содержащего 40 и более потоков. Я хочу потратить некоторое время на то, чтобы разобраться, хороша эта идея или не очень. Начнем с определений.

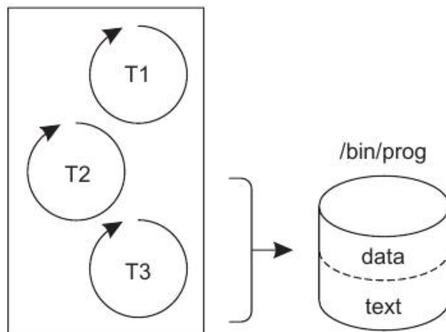
Процесс состоит из адресного пространства в памяти и потока выполнения, как показано на рисунке ниже. Адресное пространство является собственностью процесса и недоступно потокам, работающим в других процессах. Такое разграничение памяти обеспечивает подсистема управления памятью в ядре, которая хранит отображение страниц памяти для каждого процесса и перепрограммирует блок управления памятью при каждом контекстном переключении. Подробнее о том, как это работает, я расскажу в главе 11. Часть адресного пространства отображается на файл, содержащий код и статические данные исполняемой программы.



Во время работы программа захватывает ресурсы: место в стеке, память в куче, ссылки на файлы и т. д. По завершении процесса все эти ресурсы возвращаются системе: память освобождается, а дескрипторы файлов закрываются.

Процессы могут взаимодействовать с помощью средств межпроцессного взаимодействия **(IPC)**, например локальных сокетов. Об IPC я еще расскажу.

Говоря «поток», имеют в виду поток выполнения внутри процесса. В начале работы в каждом процессе есть единственный поток, в котором выполняется функция `main()`, поэтому он называется главным потоком. Для создания дополнительных потоков POSIX применяется функция `pthread_create(3)`. Дополнительные потоки исполняются в том же адресном пространстве, как показано на следующем рисунке. Поскольку все потоки работают в одном процессе, они могут сообща использовать ресурсы. Потоки могут читать и записывать одни и те же области памяти и пользоваться одними и теми же дескрипторами файлов, поэтому обмениваться данными между потоками легко, если только не забывать о синхронизации и проблемах блокировки.



Итак, зная только лишь это, мы можем представить себе два полярно противоположных подхода к переносу гипотетической системы с 40 задачами из ОСРВ в Linux.

Можно отобразить задачи на процессы и таким образом получить 40 отдельных программ, сообщающихся между собой посредством механизмов IPC, например обменивающихся сообщениями через сокеты. Повреждение памяти при этом оказалось бы маловероятным, поскольку главный поток в каждом процессе защищен от всех остальных, да и утечки памяти удалось бы снизить, потому что после завершения процесса все занятые им ресурсы освобождаются. Однако интерфейс передачи сообщений между процессами довольно сложен, и в случае, когда имеется тесная кооперация между группой процессов, сообщений может оказаться очень много, что приведет к снижению производительности всей системы. Кроме того, любой из 40 процессов может завершиться, быть может, из-за ошибки, приведшей к краху, а остальные будут продолжать работать. Поэтому каждый процесс должен принимать во внимание возможность, что его соседи прекратили существование, и корректно обрабатывать этот случай.

Другая крайность – отобразить задачи на потоки и реализовать систему, в которой есть один процесс, содержащий 40 потоков. Кооперация значительно упрощается, потому что адресное пространство и дескрипторы файлов общие. Накладные расходы на передачу сообщений снижаются или устраняются вовсе, а контекстное переключение между потоками производится быстрее, чем между процессами. Недостаток же в том, что теперь одна задача может повредить кучу или стек другой. Если какой-нибудь поток столкнется с фатальной ошибкой, то завершится весь процесс, потянув за собой все потоки. Наконец, отладка сложной многопоточной программы может превратиться в кошмар.

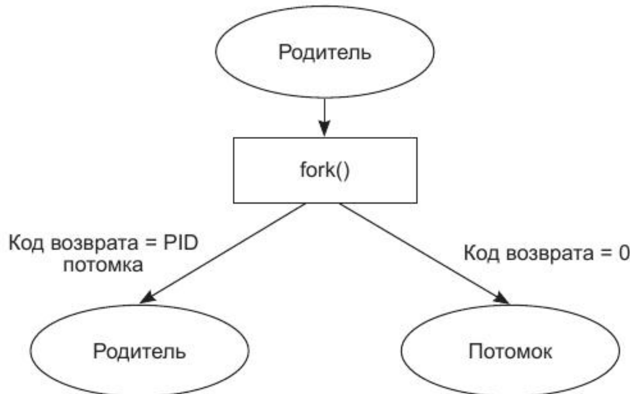
Мы приходим к выводу, что ни тот, ни другой подходы не идеальны и что должны быть более удачные решения. Но прежде чем переходить к делу, я хотел бы немного углубиться в API и поведение процессов и потоков.

Процессы

Процесс является владельцем окружения, в котором работают потоки: он хранит отображение страниц памяти, дескрипторы файлов, идентификаторы пользователя и группы и многое другое. Первым процессом в системе является `init`, его создает ядро на этапе загрузки, и он имеет идентификатор 1. Последующие процессы создаются путем дублирования, эта операция называется разветвлением.

Создание нового процесса

В стандарте `POSIX` определена функция создания процесса `fork(2)`. Это функция необычна тем, что в случае успеха возвращает два значения: одно – в процессе, из которого была вызвана (он называется родителем), а второе – во вновь созданном процессе (потомке). Это показано на рисунке ниже.



Сразу после вызова потомок является точной копией родителя: у него такой же стек, такая же куча, такие же файловые дескрипторы, и выполняет он ту же строку кода, которая следует за вызовом `fork(2)`. Различить их программист может только по значению, возвращенному `fork`: в потомке оно равно 0, а в родителе больше нуля. На самом деле родителю возвращается идентификатор (PID) вновь созданного процесса-потомка. Есть и третья возможность – отрицательный код возврата, это означает, что вызов `fork` завершился ошибкой и по-прежнему существует только один процесс.

Хотя первоначально оба процесса идентичны, они находятся в разных адресных пространствах. Если один процесс изменит какую-то переменную, то другой этого не увидит. В действительности ядро не производит физического копирования всей памяти родителя, поскольку это было бы очень долгой операцией и бессмысленной тратой памяти. Вместо этого память остается общей, но помечается флагом копирования при записи. Если родитель или потомок модифицирует такую память, то ядро сначала сделает копию, а затем произведет запись в эту копию. В результате и функция разветвления работает эффективно, и логическое разграничение адресного пространства налицо. Мы еще вернемся к копированию при записи в главе 11.

Завершение процесса

Процесс может завершиться добровольно, вызвав функцию `exit(3)`, или быть снят принудительно – из-за того, что не обработал полученный сигнал. Сигнал `SIGKILL` вообще нельзя обработать, поэтому он всегда завершает процесс. В любом случае при завершении процесса останавливаются все потоки, закрываются все файловые дескрипторы и освобождается вся память. Система посылает родителю завершившегося процесса сигнал `SIGCHLD`, чтобы уведомить его о смерти потомка.

У завершившегося процесса есть код возврата, равный аргументу функции `exit(3)` в случае нормального завершения или номеру сигнала, если он был завершен принудительно. Код возврата используется прежде всего в скриптах оболоч-

ки: он позволяет узнать, как завершилась программа. По соглашению 0 означает успешное завершение, любое другое значение – ошибку.

Родитель может получить код возврата, вызвав функцию wait(2) или waitpid(2). Но тут возникает проблема: между завершением потомка и получением его кода возврата родителем проходит некоторое время. В это время код возврата нужно где-то хранить, а PID уже «мертвого» процесса нельзя использовать повторно. Процесс, находящийся в таком состоянии, называется «зомби», в программах ps и top его состояние обозначается Z. Если процесс-родитель вызывает wait(2) или waitpid(2) как только получает уведомление о завершении потомка (посредством сигнала SIGCHLD, о деталях обработки сигналов см. Robert Love «Linux System Programming», O'Reilly Media или Michael Kerrisk «The Linux Programming Interface», No Starch Press), то зомби существует так недолго, что мы не успеваем увидеть его в списке процессов. Неприятности начинаются, когда родитель забывает получить код возврата, потому что рано или поздно станет невозможно создавать новые процессы.

В примере ниже показаны создание и завершение процесса:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
{
  int pid;
  int status;
  pid = fork();
  if (pid == 0) {
    printf("Я потомок, PID %d\n", getpid());
    sleep(10);
    exit(42);
  } else if (pid > 0) {
    printf("Я родитель, PID %d\n", getpid());
    wait(&status);
    printf("Потомок завершился, код возврата %d\n",
    WEXITSTATUS(status));
  } else
    perror("fork:");
    return 0;
}
```

Функция wait(2) блокирует программу до тех пор, пока процесс-потомок не завершится, и сохраняет его код возврата. Если запустить эту программу, она напечатает что-то типа:

```
Я родитель, PID 13851
Я потомок, PID 13852
Потомок завершился, код возврата 42
```

Процесс-потомок наследует большинство атрибутов родителя, в том числе идентификаторы пользователя и группы (UID и GID), все открытые дескрипторы файлов, диспозицию обработки сигналов и характеристики планирования.

Выполнение другой программы

Функция `fork` создает копию работающей программы, но не запускает никакую другую программу. Для этого нужна одна из функций семейства `exec`:

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Каждая из них принимает путь к файлу программы, который нужно загрузить и выполнить. В случае успеха ядро освобождает все ресурсы текущего процесса, включая память и дескрипторы файлов, и выделяет память вновь загруженной программе. Когда поток, в котором вызвана `exec*`, возвращает управление, выполнение продолжается не со строки, следующей за вызовом, а с начала функции `main()` новой программы. В примере ниже показана программа запуска других программ: она запрашивает команду, например `/bin/ls`, разветвляется и выполняет введенную команду:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
  char command_str[128];
  int pid;
  int child_status;
  int wait_for = 1;
  while (1) {
    printf("sh> ");
    scanf("%s", command_str);
    pid = fork();
    if (pid == 0) {
        /* потомок */
        printf("cmd '%s'\n", command_str);
        execl(command_str, command_str, (char *)NULL);
        /* Мы не должны вернуться из execl, так что следующая строка */
        /* выполняется, только если execl завершится с ошибкой */
        perror("exec");
        exit(1);
```

```
    }
    if (wait_for) {
        waitpid(pid, &child_status, 0);
        printf("Готово, код возврата %d\n", child_status);
    }
  }
  return 0;
}
```

Может показаться странным, что одна функция дублирует существующий процесс, а вторая освобождает его ресурсы и загружает в память другую программу, особенно если принять во внимание, что очень часто за fork почти сразу следует exec. Во многих операционных системах эти две операции объединены в одну функцию.

Но у такого решения есть и неоспоримые преимущества. Например, оно позволяет очень легко реализовать перенаправление и конвейеры в оболочке. Допустим, что мы хотим получить распечатку каталога. Последовательность событий будет такой:

1. Мы вводим ls в ответ на приглашение оболочки.
2. Оболочка разветвляется и порождает свою копию.
3. Потомок вызывает exec, чтобы выполнить программу /bin/ls.
4. Программа ls выводит содержимое каталога в стандартный вывод stdout (файл с дескриптором 1), присоединенный к терминалу. Мы видим содержимое.
5. Программа ls завершается, и оболочка снова получает управление.

Теперь допустим, что содержимое каталога требуется записать в файл, перенаправив вывод с помощью символа >. Тогда последовательность будет такой:

1. Мы вводим ls > listing.txt.
2. Оболочка разветвляется и порождает свою копию.
3. Потомок открывает и усекает файл listing.txt, а затем вызывает функцию dup2(2), чтобы скопировать дескриптор этого файла в дескриптор файла 1 (stdout).
4. Потомок вызывает exec, чтобы выполнить программу /bin/ls.
5. Программа печатает содержимое каталога, как и раньше, то теперь вывод направляется в файл listing.txt.
6. Программа ls завершается, и оболочка снова получает управление.

Отметим, что на шаге 3 есть возможность модифицировать окружение процесса-потомка до выполнения программы. Программе ls нет нужды знать, что она осуществляет вывод в файл, а не на терминал. Дескриптор stdout мог бы быть присоединен не к файлу, а к конвейеру, и тогда ls – по-прежнему, безо всяких изменений – передавала бы данные другой программе. Это часть философии Unix: комбинирование небольших компонентов, каждый из которых хорошо умеет делать свою работу. Подробнее см. книгу Eric Steven Raymond «The Art of Unix Programming»[1], Addison Wesley, 2003, ISBN 978-0131429017, особенно главу «Конвейеры, перенаправление и фильтры».

---

[1] *Рэймонд Э. С.* Искусство программирования для Unix. М.: Вильямс, 2016.

Демоны

Мы уже несколько раз упоминали демонов. Демоном называется процесс, который работает в фоновом режиме, является потомком процесса `init`, имеющего идентификатор PID 1, и не присоединен к управляющему терминалу. Ниже перечислены шаги создания демона.

1. Родитель вызывает `fork()` для создания нового процесса, после чего завершается, оставляя тем самым процесс-сироту, которого «усыновит» `init`.
2. Процесс-потомок вызывает `setsid(2)`, в результате чего создаются новый сеанс и группа процессов, в которой он будет единственным членом. Детали сейчас не важны, нужно лишь понимать, что таким образом процесс отсоединяется от управляющего терминала.
3. Потомок делает корневой каталог своим рабочим каталогом.
4. Потомок перенаправляет `stdin`, `stdout` и `stderr` (дескрипторы 0, 1 и 2) в `/dev/null`, чтобы не получать ничего из стандартного вывода и подавить весь вывод. Все остальные файловые дескрипторы закрываются.

К счастью, все это можно сделать, вызвав одну-единственную функцию `daemon(3)`.

Межпроцессное взаимодействие

Каждый процесс – островок в море памяти. Передать информацию от одного процесса другому можно двумя способами. Во-первых, скопировать ее из одного адресного пространства в другое. Во-вторых, создать область памяти, доступную обоим процессам, и таким образом организовать общий доступ к данным.

Первый подход обычно сочетается с очередью или буфером, так что сообщения, передаваемые между процессами, выстраиваются в последовательность. Это означает, что сообщение приходится копировать дважды: сначала в буфер, а затем получателю. Примерами таких механизмов служат сокеты, конвейеры, очереди сообщений, определенные в POSIX.

При втором подходе нужен не только метод отображения одной области памяти сразу на два (или более) адресных пространства, но и средства синхронизации доступа к этой памяти, например с помощью семафоров или мьютексов. В POSIX для всего этого определены функции.

Существует более старый набор интерфейсов, System V IPC, в котором определены очереди сообщений, разделяемая память и семафоры, но он обладает меньшей гибкостью, чем эквивалентные средства в POSIX, поэтому описывать его я не буду. Обзор этих средств можно найти на странице руководства `svipc(7)`, а также в книгах Michael Kerrisk «The Linux Programming Interface», No Starch Press и W. Richard Stevens «Unix Network Programming», том 2.

Программы, основанные на протоколах обмена сообщениями, обычно проще писать и отлаживать, чем программы на базе разделяемой памяти, но если сообщения длинные, то механизм работает медленно.

Межпроцессное взаимодействие на основе передачи сообщений

Существует несколько вариантов, различающихся следующими характеристиками:

- является ли поток сообщений односторонним или двусторонним;
- представляют ли передаваемые данные поток байтов, не разделенный на сообщения, или поток дискретных сообщений с сохранением границ. В последнем случае важен размер сообщения;
- можно ли задавать приоритет сообщения.

В таблице ниже показано, какими из перечисленных свойств обладают именованные каналы (FIFO), сокеты и очереди сообщений:

Свойство	Именованный канал	Сокет Unix, потоковый	Сокет Unix, дейтаграммный	Очередь сообщений POSIX
Сохранение границ сообщений	Поток байтов	Поток байтов	Дискретное	Дискретное
Односторонний или двусторонний	Односторонний	Двусторонний	Односторонний	Односторонний
Максимальный размер сообщения	Не ограничен	Не ограничен	В диапазоне от 100 КиБ до 250 КиБ	По умолчанию 8 КиБ, абсолютный максимум 1 МиБ
Уровни приоритета	Нет	Нет	Нет	от 0 до 32767

Unix-сокеты

Unix-сокеты удовлетворяют большинству требований, а благодаря знакомому API являются самым распространенным механизмом.

Для создания Unix-сокета указываются адресное семейство `AF_UNIX` и привязка к пути в файловой системе. Доступ к сокету определяется правами доступа к ассоциированному с ним файлу. Как и для интернет-сокетов, тип сокета может принимать значения `SOCK_STREAM` или `SOCK_DGRAM`, причем в первом случае мы получаем двусторонний поток байтов, а во втором дискретные сообщения с сохранением границ. Дейтаграммные Unix-сокеты надежны в том смысле, что не могут быть потеряны или переупорядочены. Максимальный размер дейтаграммы зависит от системы, узнать его можно из файла `/proc/sys/net/core/wmem_max`. Обычно он равен 100 КиБ или больше.

У Unix-сокетов нет механизма задания приоритета сообщения.

FIFO и именованные каналы

FIFO и именованный канал – два разных названия одного и того же. Это обобщение анонимных каналов, которые позволяют взаимодействовать родителю и потомку и используются для реализации конвейеров в оболочке.

Именованный канал – это файл специального вида, который создается командой `mkfifo(1)`. Как и для Unix-сокетов, права доступа к файлу определяют, кто может читать и писать в канал. Каналы односторонние, т. е. существует один читатель

и, как правило, один писатель, хотя писателей может быть и несколько. Данные передаются в виде потока байтов без границ, но существует гарантия атомарности сообщений, длина которых меньше размера буфера, ассоциированного с каналом. Иными словами, если длина сообщения меньше этого размера, то сообщение не будет разбито на части, и читатель получит его целиком при условии, что размер буфера на стороне читателя достаточно велик. По умолчанию размер буфера FIFO в современных ядрах равен 64 КиБ, а вызов функции `fcntl(2)` с параметром `F_SETPIPE_SZ` позволяет увеличить его до значения, указанного в файле `/proc/sys/fs/pipe-max-size`, обычно до 1 МиБ.

Понятия приоритета нет.

Очереди сообщений POSIX

Очередь сообщений идентифицируется именем, которое должно начинаться символом / и содержать только один символ /. В действительности очереди сообщений хранятся в псевдофайловой системе типа `mqueue`. Чтобы создать очередь или получить ссылку на существующую очередь, нужно вызвать функцию `mq_open(3)`, которая возвращает дескриптор файла. У каждого сообщения есть приоритет, сообщения читаются из очереди в порядке приоритетов, а при равных приоритетах – в порядке возраста. Максимальная длина сообщения в байтах определяется параметром, значение которого можно получить из файла `/proc/sys/kernel/msgmax`. По умолчанию она равна 8 КиБ, но можно установить любое другое значение от 128 байтов до 1 МиБ, записав новое значение в `/proc/sys/kernel/msgmax`. Поскольку ссылка на очередь – дескриптор файла, мы можем использовать функции `select(2)`, `poll(2)` и им подобные для ожидания событий очереди.

Дополнительные сведения см. на странице руководства `mq_overview(7)`.

Заключительные замечания о межпроцессном взаимодействии на основе передачи сообщений

Unix-сокеты используются чаще всего, потому что предлагают все необходимое, за исключением приоритетов сообщений. Они реализованы в большинстве операционных систем, поэтому обеспечивают максимальную переносимость.

Именованные каналы используются реже, в основном потому, что им недостает механизма, эквивалентного дейтаграммам. С другой стороны, API очень прост и состоит из обычных файловых функций `open(2)`, `close(2)`, `read(2)` и `write(2)`.

Очереди сообщений встречаются реже всего. Относящийся к ним код в ядре не так хорошо оптимизирован, как для сокетов (сетевая подсистема) и именованных каналов (файловая система).

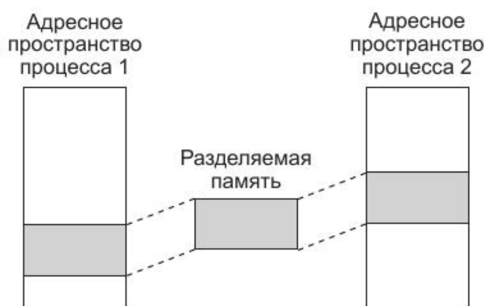
Существуют также абстракции более высокого уровня, в частности dbus, которые постепенно начинают использоваться не только в основных приложениях Linux, но и во встраиваемых системах. В dbus под капотом используются Unix-сокеты и разделяемая память.

Межпроцессное взаимодействие на основе разделяемой памяти

Совместное использование памяти устраняет необходимость в копировании данных между разными адресными пространствами, зато появляется проблема синхронизации доступа. Обычно для синхронизации процессов применяются семафоры.

Разделяемая память в POSIX

Для разделения памяти между процессами нужно сначала создать область памяти, а затем отобразить ее на адресное пространство каждого процесса, как показано на рисунке ниже.



В POSIX разделяемая память устроена по тому же образцу, что и очереди сообщений. Сегмент идентифицируется именем, которое должно начинаться символом / и содержать только такой символ. Функция `shm_open(3)` принимает имя и возвращает дескриптор соответствующего ему файла. Если сегмент не существует и задан флаг `O_CREAT`, то создается новый сегмент. Первоначально размер сегмента равен 0. Чтобы увеличить его до нужного размера, пользуйтесь функцией `ftruncate(2)` (хотя ее имя вводит в заблуждение).

Зная дескриптор сегмента разделяемой памяти, мы можем отобразить его на адресное пространство процесса, вызвав функцию `mmap(2)`, и таким образом потоки в разных процессах получают доступ к этой памяти.

Ниже приведен пример.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h> /* Ради констант mode */
#include <fcntl.h>
#include <sys/types.h>
#include <errno.h>
#include <semaphore.h>
```

```
#define SHM_SEGMENT_SIZE 65536
#define SHM_SEGMENT_NAME "/demo-shm"
#define SEMA_NAME "/demo-sem"

static sem_t *demo_sem;
/*
 * Если сегмент разделяемой памяти еще не существует, создать его.
 * Возвращает указатель на сегмент или NULL в случае ошибки.
 */

static void *get_shared_memory(void)
{
  int shm_fd;
  struct shared_data *shm_p;
  /* Пытаемся создать сегмент разделяемой памяти */
  shm_fd = shm_open(SHM_SEGMENT_NAME, O_CREAT | O_EXCL | O_RDWR, 0666);
  if (shm_fd > 0) {
    /* удачно: увеличим сегмент до нужного размера (Примечание: не делайте
      это несколько раз, поскольку ftruncate заполняет сегмент нулями) */
    printf ("Создается разделяемая память размером=%d\n",
            SHM_SEGMENT_SIZE);

    if (ftruncate(shm_fd, SHM_SEGMENT_SIZE) < 0) {
      perror("ftruncate");
      exit(1);
    }
    /* Создаем также семафор */
    demo_sem = sem_open(SEMA_NAME, O_RDWR | O_CREAT, 0666, 1);

    if (demo_sem == SEM_FAILED)
      perror("ошибка sem_open\n");
  }
  else if (shm_fd == -1 && errno == EEXIST) {
    /* Уже существует: открыть снова без O_CREAT */
    shm_fd = shm_open(SHM_SEGMENT_NAME, O_RDWR, 0);
    demo_sem = sem_open(SEMA_NAME, O_RDWR);
    if (demo_sem == SEM_FAILED)
      perror("ошибка sem_open\n");
  }

  if (shm_fd == -1) {
    perror("shm_open " SHM_SEGMENT_NAME);
    exit(1);
  }
  /* отобразить разделяемую память*/
  shm_p = mmap(NULL, SHM_SEGMENT_SIZE, PROT_READ | PROT_WRITE,
      MAP_SHARED, shm_fd, 0);

  if (shm_p == NULL) {
    perror("mmap");
    exit(1);
```

```
  }
  return shm_p;
}

int main(int argc, char *argv[])
{
  char *shm_p;
  printf("%s PID=%d\n", argv[0], getpid());
  shm_p = get_shared_memory();
  while (1) {
    printf("Нажмите Enter для просмотра текущего содержимого shm\n");
    getchar();
    sem_wait(demo_sem);
    printf("%s\n", shm_p);
    /* Записываем нашу сигнатуру в разделяемую память */
    sprintf(shm_p, "Привет от процесса %d\n", getpid());
    sem_post(demo_sem);
  }
  return 0;
}
```

В Linux память берется из файловой системы `tmpfs`, смонтированной на `/dev/shm` или `/run/shm`.

Потоки

Настало время познакомиться с многопоточными процессами. Для программирования потоков применяется описанный в POSIX API потоков, который первоначально был определен в стандарте IEEE POSIX 1003.1c (1995), больше известном под названием Pthreads. Он был реализован как дополнение к библиотеке C – `libpthread.so`. Последние 15 лет или что-то около того существуют две версии Pthreads: Linux Threads и **Native POSIX Thread Library (NPTL)**. Вторая точнее соответствует спецификации, особенно в части обработки сигналов и идентификаторов процессов. Сейчас она преобладает, но все еще можно встретить старые версии uClibc, в которых используется реализация Linux Threads.

Создание нового потока

Поток создается функцией `pthread_create(3)`:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

Новый поток начинает выполнение в функции `start_routine`, а его дескриптор записывается в переменную типа `pthread_t`, на которую указывает аргумент `thread`. Поток наследует параметры планирования вызывающего потока, но их можно переопределить, передав указатель на атрибуты потока в аргументе `attr`. Поток начинает выполнение немедленно.

Тип pthread_t – основной способ сослаться на поток внутри программы, а для того чтобы увидеть его извне, можно выполнить команду ps -eLf:

```
UID     PID  PPID   LWP  C  NLWP     STIME       TTY          TIME CMD
...
chris  6072  5648  6072  0     3     21:18  pts/0 00:00:00 ./thread-demo
chris  6072  5648  6073  0     3     21:18  pts/0 00:00:00 ./thread-demo
```

В программе thread-demo два потока. Из столбцов PID и PPID видно, что они принадлежат одному процессу и имеют одного родителя – что и неудивительно. Интересен столбец LWP. Акроним LWP означает «Light Weight Process» (облегченный процесс), в данном контексте это синоним потока. Числа в этом столбце называются идентификаторами потоков, или **TID**. TID главного потока совпадает с PID'ом процесса, а TID'ы остальных потоков больше. Некоторые функции принимают TID там, где в документации сказано, что нужно передать PID, но имейте в виду, что это поведение специфично для Linux и не переносимо. Ниже приведен код программы thread-demo:

```
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>

static void *thread_fn(void *arg)
{
  printf("Запущен новый поток, PID %d TID %d\n",
  getpid(), (pid_t)syscall(SYS_gettid));
  sleep(10);
  printf("Новый поток завершается\n");
  return NULL;
}

int main(int argc, char *argv[])
{
  pthread_t t;
  printf("Главный поток, PID %d TID %d\n",
  getpid(), (pid_t)syscall(SYS_gettid));
  pthread_create(&t, NULL, thread_fn, NULL);
  pthread_join(t, NULL);
  return 0;
}
```

На странице руководства по gettid(2) сказано, что в Linux нужно вызывать функцию syscall напрямую, потому что в библиотеке C для нее нет обертки. Так мы и сделали.

Существует ограничение на общее количество потоков, планируемых ядром. Предел масштабируется в зависимости от размера системы: для небольших систем он может быть равен примерно 1000, а в больших встраиваемых системах достигать десятков тысяч. Точное значение можно узнать из файла /proc/sys/ker-

nel/threads-max. Если этот предел достигнут, то функции fork() и pthread_create() завершаются с ошибкой.

Завершение потока

Поток завершается при следующих условиях:

- выполнение достигло конца функции start_routine;
- была вызвана функция pthread_exit(3);
- другой поток отменил данный, вызвав функцию pthread_cancel(3);
- объемлющий процесс завершился, например потому, что была вызвана функция exit(3) или получен сигнал, который процесс не обработал, не замаскировал и не проигнорировал.

Отметим, что если многопоточная программа вызывает функцию fork(2), то в процессе-потомке будет продублирован только поток, из которого она была вызвана. Разветвление не приводит к репликации всех потоков.

Поток возвращает значение – указатель на void. Один поток может дождаться завершения другого и получить возвращенное им значение, вызвав функцию pthread_join(2). Пример имеется в коде thread-demo из предыдущего раздела. При этом возникает проблема, похожая на возникновение зомби в случае процессов: ресурсы потока, например стек, нельзя освободить, пока другой поток не сольется с ним, вызвав pthread_join. Если поток завершается без слияния, то в программе образуется утечка ресурсов.

Компиляция многопоточной программы

Поддержка потоков POSIX входит в состав библиотеки C и реализована в файле libpthread.so. Однако для сборки программы, в которой используются потоки, недостаточно компоновки с этой библиотекой. Необходимо еще, чтобы компилятор генерировал код так, чтобы экземпляры некоторых глобальных переменных, к примеру errno, существовали в каждом потоке, а не на уровне процесса в целом.

При сборке многопоточной программы необходимо задавать флаг -pthread на этапах компиляции и компоновки.

Межпроцессное взаимодействие

Преимущество потоков состоит в том, что они разделяют общее адресное пространство и потому могут пользоваться общими переменными. Но это и огромный недостаток, потому что для обеспечения согласованности данных необходима синхронизация. Ситуация похожа на доступ к сегментам разделяемой памяти – с тем отличием, что в случае потоков разделяется вся память процесса. Поток может создать поточно-локальную память, к которой только он будет иметь доступ.

В интерфейсе pthreads определены базовые примитивы синхронизации: мьютексы и условные переменные. Если нужно что-то большее, придется написать самостоятельно. Стоит отметить, что все описанные выше методы межпроцессного взаимодействия работают не только для процессов, но и для потоков.

Мьютексы

Чтобы программа работала надежно, любой разделяемый ресурс необходимо защищать мьютексом и следить за тем, чтобы любой код, который хочет прочитать или изменить ресурс, сначала захватывал ассоциированный с ним мьютекс. Если не изменять этому правилу, то большинство проблем будет решено. Оставшиеся же связаны с фундаментальными особенностями поведения мьютексов. Я кратко упомяну их, но вдаваться в детали не стану.

- **Взаимоблокировка**. Это происходит, когда мьютекс оказывается захвачен навечно. Классическая ситуация – «смертельные объятия», когда каждому из двух потоков необходимы два мьютекса, но захватить удается только один. Теперь каждый поток ждет, когда другой освободит свой мьютекс, и это ожидание никогда не заканчивается. Простое правило, позволяющее избежать смертельных объятий, – всегда захватывать мьютексы в одном и том же порядке. Другие решения предполагают введение тайм-аутов и повторные попытки со случайной задержкой.
- **Инверсия приоритетов.** Задержки, вызванные ожиданием мьютекса, могут привести к тому, что поток реального времени пропустит критический срок обслуживания. Частный случай инверсии приоритетов возникает, когда высокоприоритетный поток оказывается блокирован в ожидании мьютекса, захваченного низкоприоритетным потоком. Если затем низкоприоритетный поток будет вытеснен другими потоками с промежуточными приоритетами, то высокоприоритетному потоку придется ждать неопределенно долго. Существуют протоколы работы мьютексов – наследования приоритета и предельного приоритета, – которые решают эту проблему ценой дополнительных накладных расходов в ядре при каждом захвате и освобождении мьютекса.
- **Низкая производительность.** Мьютексы оказывают минимальное влияние на код, если потоки не ожидают их освобождения большую часть времени. Если же в программе имеется ресурс, востребованный многими потоками, то коэффициент конкуренции становится велик. Обычно это проблема проектирования, которую можно решить, разбив защищаемый мьютексом ресурс на несколько более мелких или применив другой алгоритм.

Изменение условий

Кооперативным потокам необходим метод, позволяющий уведомить другой поток о произошедшем событии, которое требует внимания. Событие в этом контексте называется условием, а уведомление отправляется с помощью условной переменной, `condvar`.

Условие – это нечто, что можно проверить, получив результат `true` или `false`. Простой пример: буфер, который может содержать элементы или быть пустым. Один поток выбирает элементы из буфера и засыпает, если буфер пуст. Другой поток помещает элементы в буфер и сигнализирует первому о том, что сделал, поскольку изменилось условие, которого этот поток ожидает. Если поток спал, он должен проснуться и что-то сделать. Единственная сложность заключается в том,

что условие – по определению разделяемый ресурс, поэтому должно быть защищено мьютексом. Ниже приведен простой код, в котором реализована описанная связь между производителем и потребителем:

```
pthread_cond_t cv = PTHREAD_COND_INITIALIZER;
pthread_mutex_t mutx = PTHREAD_MUTEX_INITIALIZER;

void *consumer(void *arg)
{
  while (1) {
    pthread_mutex_lock(&mutx);
    while (buffer_empty(data))
        pthread_cond_wait(&cv, &mutx);
    /* Получены данные: забрать их из буфера */
    pthread_mutex_unlock(&mutx);
    /* Обработать элемент данных */
  }
  return NULL;
}

void *producer(void *arg)
{
  while (1) {
    /* Произвести элемент данных */
    pthread_mutex_lock(&mutx);
    add_data(data);
    pthread_mutex_unlock(&mutx);
    pthread_cond_signal(&cv);
  }
  return NULL;
}
```

Отметим, что когда поток потребителя блокируется по переменной `condvar`, он удерживает захваченный мьютекс, а это прямой путь к взаимоблокировке в следующий раз, когда поток производителя попытается обновить условную переменную. Во избежание такой ситуации `pthread_condwait(3)` освобождает мьютекс, заблокировав поток, и снова захватывает его, перед тем как пробудить поток и вернуться из состояния ожидания.

Разбиение проблемы на части

Разобравшись с основными понятиями процессов и потоков, а также со способами их взаимодействия, мы можем перейти к вопросу о том, что с ними делать.

Вот несколько правил, которых я придерживаюсь при создании систем.

- **Правило 1.** Организуйте задачи, в которых много внутренних взаимодействий. Минимизируйте накладные расходы, собирая тесно взаимодействующие потоки в один процесс.
- **Правило 2.** Не складывайте все потоки в одну корзину. С другой стороны, постарайтесь разнести компоненты с ограниченным взаимодействием по

разным процессам. Тем самым вы повысите отказоустойчивость и модульность системы.

- **Правило 3.** Не смешивайте в одном процессе особо ответственные и менее важные потоки. Это развитие правила 2: критическая часть системы, например программы управления станком, должна быть максимально проста и написана более тщательно, чем прочие части. Она должна быть способна продолжать функционирование, даже если все остальные процессы «свалились». Если в системе имеются потоки реального времени, то они, по определению, являются критическими и должны работать в отдельном процессе.
- **Правило 4.** Потоки не должны быть связаны слишком тесно. При написании многопоточных программ возникает соблазн смешать в одну кучу код и переменные, используемые в разных потоках, потому что программа-то одна и сделать это очень просто. Старайтесь делать потоки модульными и точно определять взаимодействия между ними.
- **Правило 5.** Не думайте, что потоки ничего не стоят. Создавать потоки легко, но у всего есть цена, и это не в последнюю очередь дополнительная синхронизация, необходимая для координации действий.
- **Правило 6.** Потоки могут работать параллельно. На многоядерном процессоре потоки могут работать одновременно, и это повышает пропускную способность системы. Если имеется задача с большим объемом вычислений, то мы можем создать по одному потоку на каждое ядро и в максимальной степени задействовать оборудование. Существуют библиотеки, которые помогают это сделать, например OpenMP. Вряд ли стоит кодировать алгоритмы параллельной обработки с нуля.

Хороший пример дает дизайн системы Android. Каждое приложение в ней – отдельный процесс Linux, что способствует организации модульного управления памятью, но – что гораздо важнее – гарантирует, что крах одного приложения не отразится на системе в целом. Модель процессов используется и для управления доступом: процесс может обращаться только к тем файлам и ресурсам, которые разрешены его UID'у и GID'у. В каждом процессе имеется группа потоков. Один из них занимается пользовательским интерфейсом, другой обрабатывает сигналы от операционной системы, несколько потоков управляют динамическим выделением и освобождением памяти для объектов Java, и еще есть пул рабочих потоков (не менее двух) для получения сообщений от других частей системы по протоколу Binder.

Подведем итоги. Процессы обеспечивают отказоустойчивость, потому что у каждого процесса имеется собственное адресное пространство, а когда процесс завершается, все выделенные ему ресурсы, включая память и дескрипторы файлов, освобождаются, что предотвращает утечку ресурсов. С другой стороны, потоки сообща пользуются ресурсами и потому могут легко обмениваться информацией с помощью общих переменных, а также получать доступ к одним и тем же файлам и другим ресурсам. Потоки обеспечивают параллелизм благодаря пулам потоков и другим абстракциям, полезным в многоядерных процессорах.

Планирование

Вторая важная тема, которую я хочу рассмотреть в этой главе, – планирование. Планировщик Linux управляет очередью потоков, готовых к выполнению, его задача – распределить потоки по свободным процессорам. С каждым потоком связана политика планирования: с разделением времени или реального времени. Для потоков с разделением времени определена любезность (niceness), которая повышает или понижает право потока на владение процессором. Для потоков реального времени определен статический приоритет (priority), причем потоки с более высоким приоритетом вытесняют низкоприоритетные потоки. Планировщик работает на уровне потоков, а не процессов. Потоки планируются вне зависимости от того, какому процессу принадлежат.

Планировщик получает управление в следующих случаях:

- поток приостанавливает свое выполнение, обратившись к `sleep()` или выполнив блокирующую операцию ввода-вывода;
- поток с разделением времени исчерпал выделенный ему квант;
- прерывание привело к разблокированию потока, например вследствие завершения операции ввода-вывода.

Для ознакомления с планировщиком в Linux рекомендую главу о планировании процессов в книге Robert Love «Linux Kernel Development», 3-е изд., Addison-Wesley Professional, 2010, ISBN-10: 0672329468.

Справедливость и детерминированность

Я выделил две категории политик планирования: с разделением времени и реального времени. Политики с разделением времени основаны на принципе справедливости. Их цель – сделать так, чтобы каждый поток получал справедливую долю процессорного времени и чтобы никакой поток не смог захватить все системные ресурсы. Если поток работает слишком долго, то он помещается в конец очереди, чтобы дать шанс остальным. В то же время политика справедливости должна подстраиваться под потоки, которые выполняют много работы, и предоставлять им необходимые для дела ресурсы. Планирование с разделением времени хорошо тем, что автоматически адаптируется к широкому спектру рабочих нагрузок.

С другой стороны, в программах реального времени справедливость – не критерий. Нам нужно, чтобы политика была детерминированной, дающей минимальные гарантии того, что потоки реального времени будут запланированы вовремя и не пропустят критического срока обслуживания. Это означает, что поток реального времени должен вытеснять потоки с разделением времени. Потокам реального времени назначается статический приоритет, чтобы планировщик мог выбрать какой-то один, когда к выполнению готово сразу несколько. Планировщик реального времени в Linux реализует сравнительно стандартный алгоритм, который отдает предпочтение потоку реального времени с наивысшим приоритетом. Большинство планировщиков в ОСРВ написано так же.

Потоки обоих типов могут сосуществовать. Потоки, которым требуется детерминированное планирование, планируются первыми, а оставшееся время делится между потоками с разделением времени.

Политики с разделением времени

Политики с разделением времени проектируются с учетом справедливости. Начиная с версии Linux 2.6.23 используется абсолютно справедливый планировщик (Completely Fair Scheduler – **CFS**). В нем нет временных квантов в обычном понимании этого слова. Вместо этого динамически вычисляется время, на которое поток имел бы право, если бы получил справедливую долю процессора, и это время сравнивается с фактически проработанным временем. Если поток проработал дольше, чем имел право, и существуют другие потоки с разделением времени, ожидающие своей очереди, то планировщик приостанавливает текущий поток и отдает процессор одному из ожидающих.

Определены следующие политики с разделением времени.

- SCHED_NORMAL (или SCHED_OTHER). Это политика по умолчанию. Большинство потоков в Linux планируется именно так.
- SCHED_BATCH. Аналогична SCHED_NORMAL, но гранулярность больше, т. е. поток работает дольше, но и ждать в очереди ему тоже приходится дольше. Идея в том, чтобы уменьшить количество контекстных переключений при фоновой обработке (пакетные задания) и тем самым уменьшить интенсивность вытеснения из кэша процессора.
- SCHED_IDLE. Такие потоки работают только тогда, когда нет готовых к выполнению потоков с другой политикой. У них наименьший возможный приоритет.

Есть две пары функций для получения и установки политики планирования и приоритета потока. Первые две функции принимают в качестве параметра PID и применяются к главному потоку процесса:

```
struct sched_param {
  ...
  int sched_priority;
  ...
};
int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
int sched_getscheduler(pid_t pid);
```

Другие две функции применяются к pthread_t и потому могут изменять параметры любых потоков процесса:

```
pthread_setschedparam(pthread_t thread, int policy,
                      const struct sched_param *param);
pthread_getschedparam(pthread_t thread, int *policy,
                      struct sched_param *param);
```

Любезность

Некоторые потоки с разделением времени важнее других. Мы можем сообщить об этом с помощью «любезности» `nice`, определяющей коэффициент, на который умножается право потока на ЦП. Название происходит от имени функции `nice(2)`, которая существует в Unix с самых первых дней. Поток становится любезным, когда уменьшает создаваемую им нагрузку на систему, и нелюбезным – в противном случае. Значения выбираются в диапазоне от 19 (воистину любезный) до –20 (совсем нелюбезный). Значение 0 соответствует умеренной любезности, без крайностей.

Величину `nice` можно изменять для потоков с политикой `SCHED_NORMAL` и `SCHED_BATCH`. Для уменьшения любезности и, стало быть, повышения нагрузки на процессор необходимо обладать привилегией `CAP_SYS_NICE`, доступной только пользователю `root`.

Почти во всей документации по функциям и командам, изменяющим значение `nice` (`nice(2)`, а также команды `nice` и `renice`), речь идет о процессах. Но на самом деле любезность – это свойство потока. Как уже отмечалось выше, мы можем использовать TID вместо PID, чтобы изменить любезность конкретного потока. В стандартных определениях `nice` есть еще одно несоответствие: часто о `nice` говорят как о приоритете потока (иногда процесса, но это вообще ошибка). Я полагаю, что это неправильно и только создает путаницу с приоритетом реального времени – совершенно другим понятием.

Политики реального времени

Политики реального времени предназначены для обеспечения детерминированности. Планировщик реального времени всегда выбирает поток реального времени с наивысшим приоритетом из числа готовых к выполнению. Потоки реального времени всегда вытесняют потоки с разделением времени. По сути дела, задавая политику реального времени вместо разделения времени, вы говорите, что обладаете сакральным знанием о том, как должен планироваться этот поток, и желаете отменить встроенные в планировщик предположения.

Существуют две политики реального времени.

- `SCHED_FIFO`. Это алгоритм выполнения до конца, который означает, что если поток начал работать, то он так и будет продолжать работу, пока его не вытеснит поток реального времени с более высоким приоритетом или пока он не выполнит блокирующий системный вызов или пока не завершится.
- `SCHED_RR`. Это круговой алгоритм, который в цикле перебирает потоки с одинаковым приоритетом, переходя к следующему, когда текущий исчерпает свой временной квант, по умолчанию равный 100 мс. Начиная с версии Linux 3.9 стало возможно изменять значение `timeslice` с помощью файла `/proc/sys/kernel/sched_rr_timeslice_ms`. В остальном эта политика ведет себя так же, как `SCHED_FIFO`.

Для каждого потока реального времени определен приоритет в диапазоне от 1 до 99, причем 99 считается наивысшим.

Для назначения потоку политики реального времени необходимо обладать привилегией `CAP_SYS_NICE`, по умолчанию доступной только пользователю `root`.

С планированием реального времени не только в Linux, но и в любой другой системе, связана одна проблема: если поток не освобождает процессор, часто из-за ошибки, приведшей к зацикливанию, то ни потоки реального времени с меньшим приоритетом, ни потоки с разделением времени не могут исполняться. Система начинает вести себя странно и может вообще зависнуть. Есть два способа защититься от этой неприятности.

Во-первых, начиная с версии Linux 2.6.25 планировщик по умолчанию резервирует 5% процессорного времени для потоков, планируемых не в реальном времени, поэтому вышедший из-под контроля поток реального времени не может привести к полному останову системы. Для конфигурирования этого поведения предназначены два параметра:

- `/proc/sys/kernel/sched_rt_period_us`;
- `/proc/sys/kernel/sched_rt_runtime_us`.

По умолчанию они равны 1 000 000 (1 секунда) и 950 000 (950 мс), т. е. из каждой секунды 50 мс резервируется для потоков, планируемых не в реальном времени. Если вы хотите, чтобы потоки реального времени могли получить все 100%, присвойте параметру `sched_rt_runtime_us` значение –1.

Вторая возможность – воспользоваться сторожевым таймером, аппаратным или программным, для мониторинга выполнения важнейших потоков и принятия каких-то мер, если они начинают пропускать критические сроки обслуживания.

Выбор политики

На практике политики с разделением времени подходят для большинства рабочих нагрузок. Потоки, ограниченные скоростью ввода-вывода, проводят большую часть времени в заблокированном состоянии, поэтому у них всегда есть неиспользованное право на процессор. Будучи разблокированными, они получают процессор почти сразу. Ну а счетные потоки естественным образом получают в свое распоряжение оставшееся процессорное время. Кроме того, для менее важных потоков можно задать положительное значение любезности, а для более важных – отрицательное.

Разумеется, это лишь поведение в среднем, нет никаких гарантий, что так будет всегда. Если требуется более детерминированное поведение, то к нашим услугам политики реального времени. Вот несколько случаев, когда имеет смысл задать для потока политику планирования реального времени:

- имеется критический срок обслуживания, в течение которого поток обязан выдать результат;
- пропуск критического срока обслуживания ставит под сомнение полезность системы;
- поток управляется событиями;
- поток не является счетным, т. е. не занимает всех ресурсов процессора.

В качестве примеров задач реального времени можно привести классический сервоконтроллер роботизированной руки, обработку мультимедийных данных и обработку данных, поступающих по линиям связи.

Выбор приоритета реального времени

Задание приоритетов реального времени, которые обеспечили бы нормальную работу для любой ожидаемой нагрузки, – дело тонкое, и уже одного этого достаточно, чтобы по возможности избегать политик реального времени.

Чаще всего для выбора приоритетов применяют частотно-монотонный анализ (ЧМА), впервые описанный в 1973 году в статье Лиу и Лейленда. Он применим к очень важному классу систем реального времени с периодическими потоками. Для каждого потока определены период и коэффициент использования, т. е. доля периода, в течение которой он будет исполняться. Цель состоит в том, чтобы все потоки могли завершить фазу исполнения до наступления следующего периода. Теория ЧМА утверждает, что эта цель достижима, если:

- наивысшие приоритеты назначаются потокам с самыми короткими периодами;
- полный коэффициент использования меньше 69%.

Полный коэффициент использования равен сумме всех коэффициентов использования. Предполагается также, что время, потраченное на взаимодействие между потоками, в ожидании мьютекса и т. п., пренебрежимо мало.

Дополнительная литература

Ниже перечислены ресурсы, в которых можно найти дополнительные сведения по вопросам, затронутым в данной главе.

- *Raymond E. S.* The Art of Unix Programming. Addison Wesley, 2003. ISBN 978-0131429017.
- *Love R.* Linux System Programming. 2nd ed. O'Reilly Media, 2013. ISBN-10: 1449339530.
- *Love R.* Linux Kernel Development. 3rd ed. Addison-Wesley Professional, 2010. ISBN-10: 0672329468.
- *Kerrisk M.* The Linux Programming Interface. No Starch Press, 2010. ISBN 978-1-59327-220-3.
- *Stevens W. R.* UNIX Network Programming: v. 2: Interprocess Communications. 2nd ed. Prentice Hall, 1998. ISBN-10: 0132974290.
- *Butenhof D. R.* Programming with POSIX Threads. Addison-Wesley Professional.
- *Liu C. L., Layland J. W.* Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment // Journal of ACM. 1973. Vol. 20. № 1. P. 46–61.

Резюме

В богатом наследии Unix, вошедшем в состав Linux и сопутствующих библиотек C, есть почти все, что может понадобиться для написания стабильных отказоустойчивых встраиваемых приложений. Проблема в том, что любую задачу можно решить, по меньшей мере, двумя способами.

Эту главу я посвятил двум аспектам проектирования системы: разделению ее на отдельные процессы, в каждом из которых работает один или несколько потоков, и планированию этих потоков. Надеюсь, что мне удалось пролить свет на эти вопросы и заложить фундамент для дальнейшего изучения.

В следующей главе я рассмотрю еще одну важную сторону системы: управление памятью.

Глава 11

Управление памятью

В этой главе рассматриваются вопросы, относящиеся к управлению памятью, которое играет важную роль в любой системе на базе Linux, а особенно во встраиваемых системах, где память обычно ограничена. После краткого введения в механизм виртуальной памяти я объясню, как измерять потребление памяти, как находить ошибки в работе с памятью, в том числе утечки, и что происходит, когда памяти не хватает. Вы должны понимать, какие инструменты имеются в вашем распоряжении, от простейших типа `free` и `top` до таких сложных, как `mtrace` и `Valgrind`.

Основы виртуальной памяти

Напомню, что Linux конфигурирует блок управления памятью, входящий в состав ЦП, так чтобы он представлял программе виртуальное адресное пространство как начинающееся с адреса 0 и заканчивающееся максимальным адресом `0xffffffff` (в случае 32-разрядного процессора). Это адресное пространство делится на страницы размером 4 КиБ (изредка встречаются системы с другим размером страницы).

Linux разбивает виртуальное адресное пространство на две области: для приложений (пользовательское пространство) и для ядра (пространство ядра). Граница между ними определяется конфигурационным параметром ядра `PAGE_OFFSET`. В типичной 32-разрядной встраиваемой системе `PAGE_OFFSET` равен `0xc0000000`, т. е. три нижних гигабайта отведены под пользовательское пространство, а один верхний гигабайт – под ядро. Память из пользовательского адресного пространства выделяется на уровне процесса, так что каждый процесс работает в песочнице, отделенный от всех остальных. Адресное пространство ядра общее для всех процессов, так как существует всего одно ядро.

Страницы этого виртуального адресного пространства отображаются на физические адреса блоком управления памятью (БУП), который пользуется для этой цели таблицами страниц.

Страница виртуальной памяти может находиться в одном из следующих состояний:

- не отображена, попытка обратиться к ней приводит к сигналу `SIGSEGV`;
- отображена на страницу физической памяти, принадлежащую конкретному процессу;

- отображена на страницу физической памяти, разделяемую несколькими процессами;
- отображена и является разделяемой, установлен флаг копирования при записи: попытка записи перехватывается ядром, которое создает копию страницы и отображает ее в адресное пространство процесса вместо исходной, после чего разрешает запись;
- отображена на физическую страницу в пространстве ядра.

Ядро может также отображать страницы на зарезервированные области памяти, например для доступа к регистрам и буферам в драйверах устройств.

Возникает очевидный вопрос: зачем все это, если можно просто обращаться непосредственно к физической памяти, как в типичной ОСРВ?

У виртуальной памяти много преимуществ, я перечислю лишь некоторые.

- Доступ к несуществующей памяти перехватывается, и приложение уведомляется об этом с помощью сигнала `SIGSEGV`.
- Каждый процесс работает в собственном адресном пространстве и изолирован от других процессов.
- Эффективное использование памяти за счет совместного доступа к общему коду и данным, например библиотекам.
- Возможность увеличить кажущийся объем физической памяти путем добавления файлов подкачки, хотя во встраиваемых системах подкачка встречается редко.

Все это сильные аргументы, но не следует отворачиваться и от недостатков. Трудно определить, сколько памяти в действительности нужно приложению, а это как раз один из основных вопросов в данной главе. По умолчанию выделяется больше памяти, чем есть в наличии, что ведет к неприятным ситуациям с нехваткой памяти, которые мы обсудим ниже. Наконец, задержки, связанные с обработкой исключений – страничных отказов – в коде управления памятью, делают систему менее предсказуемой, что плохо для программ реального времени. Эту тему мы обсудим в главе 14.

В ядре и в пользовательском пространстве управление памятью осуществляется по-разному. В следующих разделах мы опишем существенные различия и особенности, о которых следует знать.

Структура памяти ядра

Управление памятью в ядре организовано довольно просто. Здесь нет подкачки страниц по запросу, т. е. при каждом обращении к функции `kmalloc()` или ей подобной выделяется физическая память. Память ядра никогда не замещается и не выгружается на диск.

Для некоторых архитектур распределение памяти ядра на этапе загрузки выводится в журнал. Показанная ниже распечатка сформирована 32-разрядным устройством с архитектурой ARM (платой BeagleBone Black):

```
Memory: 511MB = 511MB total
Memory: 505980k/505980k available, 18308k reserved, 0K highmem
```

```
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)
    vmalloc : 0xe0800000 - 0xff000000   ( 488 MB)
    lowmem  : 0xc0000000 - 0xe0000000   ( 512 MB)
    pkmap   : 0xbfe00000 - 0xc0000000   (   2 MB)
    modules : 0xbf800000 - 0xbfe00000   (   6 MB)
      .text : 0xc0008000 - 0xc0763c90   (7536 kB)
      .init : 0xc0764000 - 0xc079f700   ( 238 kB)
      .data : 0xc07a0000 - 0xc0827240   ( 541 kB)
      .bss  : 0xc0827240 - 0xc089e940   ( 478 kB)
```

Фраза «505980k available» означает, что именно такой объем памяти ядро видит в самом начале выполнения, когда еще не было ни одного динамического выделения.

К потребителям памяти ядра относятся:

- само ядро, т. е. код и данные, загруженные из образа ядра на этапе загрузки. В распечатке выше это память в сегментах `.text`, `.init`, `.data` и `.bss`. Сегмент `.init` освобождается по завершении инициализации ядра;
- память, выделяемая распределителем слябов (slab), которая используется для различных структур данных ядра. Сюда входит память, выделяемая функцией `kmalloc()`. Слябы находятся в области `lowmem`;
- память, выделяемая с помощью функции `vmalloc()`, обычно для более крупных блоков, чем может выделить `kmalloc()`. Эта память берется из области `vmalloc`;
- отображение в интересах драйверов устройств, которым нужно обращаться к регистрам и памяти, принадлежащей различным частям оборудования (см. файл `/proc/iomem`). Эти адреса находятся в области `vmalloc`, но поскольку они отображены на физическую память за пределами основной памяти системы, то никакой реальной памяти не занимают;
- модули ядра, которые загружаются в область `modules`;
- другие низкоуровневые механизмы выделения памяти.

Сколько памяти потребляет ядро?

К сожалению, точно ответить на этот вопрос нельзя, но ниже дан настолько близкий к реальности ответ, насколько возможно.

Для начала можно посмотреть, сколько памяти занято кодом и данными ядра, – в журнале, как описано выше, или с помощью команды `size`:

```
$ arm-poky-linux-gnueabi-size vmlinux
text      data     bss      dec        hex     filename
9013448   796868   8428144  18238460   1164bfc vmlinux
```

Обычно этот размер мал, по сравнению с общим объемом памяти. Если это не так, то имеет смысл открыть программу конфигурирования ядра и удалить ненужные компоненты. Не прекращаются старания упростить создание небольших

ядер, задайте поисковику запрос «Linux-tiny» или «Linux Kernel Tinification». Этой задаче посвящен проект по адресу https://tiny.wiki.kernel.org/.

Дополнительные сведения об использовании памяти можно получить из файла /proc/meminfo:

```
# cat /proc/meminfo
MemTotal:        509016 kB
MemFree:         410680 kB
Buffers:           1720 kB
Cached:           25132 kB
SwapCached:           0 kB
Active:           74880 kB
Inactive:          3224 kB
Active(anon):     51344 kB
Inactive(anon):    1372 kB
Active(file):     23536 kB
Inactive(file):    1852 kB
Unevictable:          0 kB
Mlocked:              0 kB
HighTotal:            0 kB
HighFree:             0 kB
LowTotal:        509016 kB
LowFree:         410680 kB
SwapTotal:            0 kB
SwapFree:             0 kB
Dirty:               16 kB
Writeback:            0 kB
AnonPages:        51248 kB
Mapped:           24376 kB
Shmem:             1452 kB
Slab:             11292 kB
SReclaimable:      5164 kB
SUnreclaim:        6128 kB
KernelStack:       1832 kB
PageTables:        1540 kB
NFS_Unstable:         0 kB
Bounce:               0 kB
WritebackTmp:         0 kB
CommitLimit:     254508 kB
Committed_AS:    734936 kB
VmallocTotal:    499712 kB
VmallocUsed:      29576 kB
VmallocChunk:    389116 kB
```

Все эти поля описаны на странице руководства proc(5). Объем памяти, используемой ядром, складывается из следующих величин:

- **Slab**: сколько памяти выделено распределителем слябов;
- **KernelStack**: размер стека во время исполнения кода ядра;

- **PageTables**: сколько памяти отведено под таблицы страниц;
- **VmallocUsed**: объем памяти, выделенной функцией vmalloc().

Что касается выделения памяти для слябов, дополнительная информация есть в файле /proc/slabinfo. Точно так же в файле /proc/vmallocinfo имеется детальная информация об области vmalloc. В обоих случаях нужно очень хорошо разбираться в ядре и его подсистемах, чтобы понять, какая именно подсистема выделила память и для чего. Это выходит далеко за рамки нашего обсуждения.

Команда lsmod сообщает о том, сколько памяти отведено под код и данные модулей:

```
# lsmod
Module              Size  Used by
g_multi            47670  2
libcomposite       14299  1 g_multi
mt7601Usta        601404  0
```

Здесь не учтены низкоуровневые операции выделения памяти, которые нигде не регистрируются, так что мы не можем получить точного отчета об использовании памяти в пространстве ядра. Если объединить все, что нам известно о выделенной памяти в ядре и пользовательском пространстве, то сложится впечатление, будто часть памяти куда-то делась.

Структура памяти в пользовательском пространстве

В Linux применяется стратегия отложенного выделения памяти в пользовательском пространстве, т. е. отображение физической страницы происходит лишь при попытке доступа к ней. Например, в ответ на запрос о выделении буфера размером 1 МиБ функция malloc(3) возвращает указатель на область памяти, но никакой физической памяти за ним не стоит. В элементах таблицы страниц установлен флаг, означающий, что любая попытка чтения или записи страницы перехватывается ядром. Это называется страничным отказом. И лишь в этот момент ядро пытается найти страницу физической памяти и добавить ее в таблицу отображения страниц для данного процесса. Продемонстрируем это на примере простой программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#define BUFFER_SIZE (1024 * 1024)

void print_pgfaults(void)
{
  int ret;
  struct rusage usage;
```

```
  ret = getrusage(RUSAGE_SELF, &usage);
  if (ret == -1) {
    perror("getrusage");
  } else {
    printf ("Число страничных отказов (жесткий) %ld\n", usage.ru_majflt);
    printf ("Число страничных отказов (мягкий) %ld\n", usage.ru_minflt);
  }
}

int main (int argc, char *argv[])
{
  unsigned char *p;
  printf("Начальное состояние\n");
  print_pgfaults();
  p = malloc(BUFFER_SIZE);
  printf("После malloc\n");
  print_pgfaults();
  memset(p, 0x42, BUFFER_SIZE);
  printf("После memset\n");
  print_pgfaults();
  memset(p, 0x42, BUFFER_SIZE);
  printf("После второго memset\n");
  print_pgfaults();
}
```

Выполнение этой программы дает такой результат:

```
Начальное состояние
Число страничных отказов (жесткий) 0
Число страничных отказов (мягкий) 172
После malloc
Число страничных отказов (жесткий) 0
Число страничных отказов (мягкий) 186
После memset
Число страничных отказов (жесткий) 0
Число страничных отказов (мягкий) 442
После второго memset
Число страничных отказов (жесткий) 0
Число страничных отказов (мягкий) 442
```

На стадии инициализации программы было зарегистрировано 172 мягких страничных отказа и еще 14 в результате вызова `getrusage(2)` (числа зависят от архитектуры и версии библиотеки C). Обратите внимание на прирост при заполнении памяти данными: 442 – 186 = 256. Размер буфера равен 1 МиБ, т. е. 256 страниц. При втором обращении к `memset(3)` число страничных отказов не увеличивается, потому что все страницы уже отображены.

Как видим, страничный отказ генерируется, когда ядро перехватывает попытку доступа к еще не отображенной странице. На самом деле есть два типа страничных отказов: мягкий и жесткий. В случае мягкого отказа ядро просто должно найти

страницу физической памяти и отобразить ее в адресное пространство процесса, как показано в примере выше. Жесткий страничный отказ возникает, когда виртуальная память отображена на файл, например с помощью функции mmap(2), которую я опишу ниже. Чтение из такой памяти означает, что ядро должно не только найти и отобразить страницу памяти, но еще и заполнить ее данными из файла. Следовательно, жесткие отказы обходятся гораздо дороже с точки зрения времени и системных ресурсов.

Карта памяти процесса

Карту памяти процесса можно посмотреть в файловой системе proc. Вот, например, карта памяти процесс init с идентификатором 1:

```
# cat /proc/1/maps
00008000-0000e000 r-xp 00000000 00:0b 23281745  /sbin/init
00016000-00017000 rwxp 00006000 00:0b 23281745  /sbin/init
00017000-00038000 rwxp 00000000 00:00 0         [heap]
b6ded000-b6f1d000 r-xp 00000000 00:0b 23281695  /lib/libc-2.19.so
b6f1d000-b6f24000 ---p 00130000 00:0b 23281695  /lib/libc-2.19.so
b6f24000-b6f26000 r-xp 0012f000 00:0b 23281695  /lib/libc-2.19.so
b6f26000-b6f27000 rwxp 00131000 00:0b 23281695  /lib/libc-2.19.so
b6f27000-b6f2a000 rwxp 00000000 00:00 0
b6f2a000-b6f49000 r-xp 00000000 00:0b 23281359  /lib/ld-2.19.so
b6f4c000-b6f4e000 rwxp 00000000 00:00 0
b6f4f000-b6f50000 r-xp 00000000 00:00 0         [sigpage]
b6f50000-b6f51000 r-xp 0001e000 00:0b 23281359  /lib/ld-2.19.so
b6f51000-b6f52000 rwxp 0001f000 00:0b 23281359  /lib/ld-2.19.so
beea1000-beec2000 rw-p 00000000 00:00 0         [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0         [vectors]
```

В первых трех столбцах показаны начальный и конечный виртуальные адреса и права доступа для каждой отображенной области. Права интерпретируются следующим образом:

- r = разрешено читать;
- w = разрешено писать;
- x = разрешено выполнять;
- s = разделяемая память;
- p = частная (копировать при записи).

Если отображенная область связана с файлом, то в последнем столбце будет находиться имя файла, а в четвертом, пятом и шестом столбцах – смещение от начала файла, номер блочного устройства и индексный узел (inode) файла. По большей части, это отображения самой программы и библиотек, с которыми она скомпонована. Есть две области, из которых программа может выделять память, они помечены как [heap] и [stack]. Память, которую выделяет функция malloc(3), берется из кучи (область [heap]) (кроме очень больших блоков, о которых мы поговорим ниже), а память в стеке – из области [stack]. Максимальный размер обеих областей в процессе управляется программой ulimit:

- **куча**: `ulimit -d`, по умолчанию не ограничена;
- **стек**: `ulimit -s`, по умолчанию 8 МиБ.

Попытка превысить пределы пресекается сигналом `SIGSEGV`.

Если памяти не хватает, то ядро может попытаться заместить страницы, отображенные на файл и доступные только для чтения. При повторном обращении к странице произойдет жесткий страничный отказ, и страница будет снова прочитана.

Подкачка

Идея подкачки, или свопинга, заключается в том, чтобы зарезервировать часть внешней памяти, куда ядро сможет поместить страницы памяти, не отображенной на файл, и тем самым освободить место для других целей. В результате эффективный размер физической памяти увеличивается на размер файла подкачки. Это не панацея: копирование страниц между памятью и файлом подкачки обходится не даром, и это становится очевидно, когда объем физической памяти системы слишком мал для возлагаемой на нее нагрузки, и диск начинает пробуксовывать.

Подкачка редко применяется во встраиваемых устройствах, потому что постоянная запись приводит к быстрому износу флэш-памяти. Однако можно рассмотреть возможность выгрузки страниц в сжатую оперативную память (zram).

Выгрузка страниц в сжатую память (zram)

Драйвер zram создает блочные устройства в ОЗУ с именами `/dev/zram0`, `/dev/zram1` и т. д. Перед записью на такое устройство страницы сжимаются. Если коэффициент сжатия составляет от 30 до 50%, то можно ожидать, что объем свободной памяти увеличится примерно на 10% – ценой дополнительной обработки и соответственного потребления энергии. Эта техника используется в некоторых Android-устройствах с небольшим объемом памяти.

Для включения zram нужно задать следующие конфигурационные параметры:

```
CONFIG_SWAP
CONFIG_CGROUP_MEM_RES_CTLR
CONFIG_CGROUP_MEM_RES_CTLR_SWAP
CONFIG_ZRAM
```

Затем смонтируйте zram на этапе загрузки, добавив следующую строку в файл `/etc/fstab`:

```
/dev/zram0 none swap defaults zramsize=<size in bytes>,
swapprio=<swap partition priority>
```

Для включения и выключения подкачки предназначены такие команды:

```
# swapon /dev/zram0
# swapoff /dev/zram0
```

Отображение памяти с помощью mmap

В начале жизни процессу выделена память, отображенная на сегменты text (код) и data (данные) файла программы, а также библиотек, с которыми программа скомпонована. Во время работы процесс может выделять память из кучи, вызывая malloc(3), и из стека, размещая в нем локальные переменные, а также блоки, выделенные функцией alloca(3). Кроме того, он может динамически загружать библиотеки с помощью функции dlopen(3). Отображение памяти во всех этих случаях берет на себя ядро. Но процесс может манипулировать своей картой памяти и напрямую, обращаясь к функции mmap(2):

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

Эта функция отображает length байтов из файла с дескриптором fd, начиная со смещения offset, и в случае успеха возвращает указатель на отображенную область памяти. Поскольку оборудование работает только со страницами, length округляется с избытком до ближайшего числа полных страниц. Параметр prot представляет собой комбинацию прав на чтение, запись и исполнение, а параметр flags содержит, по меньшей мере, один из флагов MAP_SHARED или MAP_PRIVATE. Есть много других флагов, все они описаны на странице руководства.

У функции mmap много применений. Перечислим некоторые из них.

Использование mmap для выделения частной памяти

Функцию mmap можно использовать для выделения частной области памяти, установив флаг MAP_ANONYMOUS и передав –1 в качестве дескриптора файла fd. Это похоже на выделение памяти из кучи с помощью malloc(3) с тем отличием, что память выровнена на границу страницы и выделяется блоками по нескольку страниц. Память выделяется из той же области, что и для библиотек. По этой причине ее иногда называют областью mmap.

Анонимное отображение лучше использовать для выделения больших блоков, потому что при этом снижается риск фрагментации кучи. Интересно, что если у функции malloc(3) (по крайней мере, в библиотеке glibc) запрашивается блок размером свыше 128 КиБ, то память выделяется не из кучи, а с помощью mmap, так что можно не разделять эти случаи самостоятельно, а всецело положиться на malloc – система сама выберет оптимальный способ удовлетворить запрос.

Использование mmap для разделения памяти

В главе 10 мы видели, что в стандарте POSIX для доступа к сегменту разделяемой памяти предполагается использование mmap. В таком случае мы поднимаем флаг MAP_SHARED и используем дескриптор, возвращенный функцией shm_open():

```
int shm_fd;
char *shm_p;

shm_fd = shm_open("/myshm", O_CREAT | O_RDWR, 0666);
ftruncate(shm_fd, 65536);
shm_p = mmap(NULL, 65536, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
```

Использование mmap для доступа к памяти устройства

В главе 8 я отмечал, что драйвер может разрешить приложению использовать mmap для управляемого им устройства. Точная реализация зависит от драйвера.

Пример дает драйвер видеобуфера в Linux: /dev/fb0. Его интерфейс, определенный в файле /usr/include/linux/fb.h, включает функцию ioctl, которая получает размер экрана и количество бит на пиксель. Затем можно воспользоваться функцией mmap, попросив видеодрайвер разделить видеобуфер с приложением, после чего никто не мешает читать и записывать пиксели:

```
int f;
int fb_size;
unsigned char *fb_mem;

f = open("/dev/fb0", O_RDWR);
/* Используем FBIOGET_VSCREENINFO, чтобы получить размеры экрана и вычислить fb_size */
fb_mem = mmap(0, fb_size, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
/* читаем и пишем пиксели с помощью указателя fb_mem */
```

Другой пример – интерфейс потокового видео, Video 4 Linux, версия 2, или V4L2, определенный в файле /usr/include/linux/videodev2.h. Каждому видеоустройству соответствует узел вида /dev/videoN, начиная с /dev/video0. Существует функция ioctl, позволяющая запросить у драйвера выделение некоторого числа видеобуферов, которые отображаются в пользовательское пространство с помощью mmap. Затем нужно в цикле перебирать буферы, заполняя их данными или опустошая, в зависимости от выполняемой операции: воспроизведение или захват видеопотока.

Сколько памяти потребляет мое приложение?

Как и в случае ядра, из-за наличия разных способов выделения, отображения и разделения памяти в пользовательском пространстве очень трудно ответить на этот вроде бы простой вопрос.

Для начала можно спросить у ядра, сколько, по его мнению, памяти доступно; для этого служит команда free. Вот что она выводит:

```
             total       used       free     shared    buffers     cached
Mem:        509016     504312       4704          0      26456     363860
-/+ buffers/cache:     113996     395020
Swap:            0          0          0
```

На первый взгляд, у системы почти закончилась память: осталось всего 4704 КиБ из 509 016 КиБ – меньше 1%. Заметим, однако, что 26 456 КиБ занято буферами и целых 363 860 КиБ отведено под кэш. Linux полагает, что свободная память – это память, расходуемая напрасно, поэтому ядро использует свободную память под буферы и кэши, зная, что ее можно будет отдать, если понадобится. Если не учитывать буферы и кэша, то мы получим действительный размер свободной памяти – 395 020 КиБ, т. е. 77% от общего объема. Так что в выдаче free важны именно числа во второй строке -/+ buffers/cache.

Мы можем заставить ядро освободить кэши, записав число от 1 до 3 в файл /proc/sys/vm/drop_caches:

```
# echo 3 > /proc/sys/vm/drop_caches
```

Это число на самом деле является битовой маской, которая показывает, какой из двух типов кэшей освобождать: 1 – кэш страниц, 2 – комбинированный кэш структур dentry и inode. Точное назначение этих кэшей не так интересно, важно лишь, что это память, которую ядро использует, но может в любой момент освободить.

Потребление памяти на уровне процесса

Существует несколько метрик, позволяющих измерить объем потребляемой процессом памяти. Начну с двух самых простых: размер виртуального набора (virtual set size – **vss**) и размер резидентной памяти (resident memory size – **rss**). Обе величины показываются в большинстве реализаций программ ps и top:

- **Vss**: ps называет эту величину VSZ, а top – VIRT. Она равна суммарному объему памяти, отображенной на адресное пространство процесса. Это сумма размеров всех областей, показанных в файле /proc/<PID>/map. Она не слишком интересна, потому что в каждый момент времени только части виртуальной памяти соответствует физическая;
- **Rss**: ps называет эту величину RSS, a top – RES. Она равна суммарному размеру страниц, которым сопоставлена физическая память. Это уже ближе к фактическому потреблению памяти процессом, но есть проблема: если сложить величины Rss для всех процессов, то полученная оценка будет завышена, потому что некоторые страницы разделяются между несколькими процессами.

Использование top и ps

Версии программ top и ps из комплекта BusyBox дают очень ограниченную информацию. В примерах ниже использовались полные версии из пакета procps.

Команда ps показывает величины Vss (VSZ) и Rss (RSS) при задании флагов Aly или пользовательского формата, включающего параметры vsz и rss, как в примере ниже:

```
# ps -eo pid,tid,class,rtprio,stat,vsz,rss,comm

PID  TID CLS RTPRIO STAT     VSZ   RSS COMMAND
  1    1 TS       - Ss      4496  2652 systemd
...
205  205 TS       - Ss      4076  1296 systemd-journal
228  228 TS       - Ss      2524  1396 udevd
581  581 TS       - Ss      2880  1508 avahi-daemon
584  584 TS       - Ss      2848  1512 dbus-daemon
590  590 TS       - Ss      1332   680 acpid
594  594 TS       - Ss      4600  1564 wpa_supplicant
```

Команда top тоже показывает сводные сведения о свободной памяти и потреблении памяти каждым процессом:

```
top - 21:17:52 up 10:04, 1 user, load average: 0.00, 0.01, 0.05
Tasks: 96 total, 1 running, 95 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.7 us, 2.2 sy, 0.0 ni, 95.9 id, 0.0 wa, 0.0 hi, 0.2 si, 0.0 st
KiB Mem: 509016 total, 278524 used, 230492 free, 25572 buffers
KiB Swap: 0 total, 0 used, 0 free, 170920 cached

 PID USER   PR NI  VIRT  RES  SHR  S %CPU %MEM  TIME+    COMMAND
1098 debian 20  0 29076  16m  8312 S  0.0  3.2  0:01.29  wicd-client
 595 root   20  0 64920 9.8m  4048 S  0.0  2.0  0:01.09  node
 866 root   20  0 28892 9152  3660 S  0.2  1.8  0:36.38  Xorg
```

Эти простые команды дают представление об использовании памяти, а если величина Rss процесса постоянно растет, то это может считаться первым признаком утечки памяти. Но с точки зрения точности измерения они оставляют желать лучшего.

Использование smem

В 2009 году Мэтт Маколл (Matt Mackall) начал изучать проблему подсчета разделяемых страниц при измерении потребления памяти процессом и ввел две новые метрики: собственный размер набора (unique set size – **Uss**) и пропорциональный размер набора (proportional set size – **Pss**).

- **Uss**: это суммарный размер страниц, которым сопоставлена физическая память и которые принадлежат только этому процессу и больше никому. То есть это тот объем памяти, который будет освобожден при завершении данного процесса.
- **Pss**: эта метрика распределяет учет разделяемых страниц, которым сопоставлена физическая память, между всеми процессами, на пространство которых они отображены. Например, если имеется область библиотечного кода длиной 12 страниц, которая разделяется шестью процессами, то на Pss каждого процесса придется по две страницы. Следовательно, если сложить величины Pss для всех процессов, то получится фактический объем памяти, используемой этими процессами. Иными словами, Pss – как раз то число, которое нас интересует.

Эта информация имеется в файле /proc/<PID>/smaps наряду с прочими данными об отображении памяти, дополняющими то, что есть в файле /proc/<PID>/maps. Вот одна из секций этого файла, содержащая сведения об отображении сегмента кода из библиотеки libc:

```
b6e6d000-b6f45000 r-xp 00000000 b3:02 2444 /lib/libc-2.13.so
Size:                864 kB
Rss:                 264 kB
Pss:                   6 kB
Shared_Clean:        264 kB
Shared_Dirty:          0 kB
```

```
Private_Clean:        0 kB
Private_Dirty:        0 kB
Referenced:         264 kB
Anonymous:            0 kB
AnonHugePages:        0 kB
Swap:                 0 kB
KernelPageSize:       4 kB
MMUPageSize:          4 kB
Locked:               0 kB
VmFlags: rd ex mr mw me
```

Обратите внимание, что Rss равно 264 КиБ, но поскольку эта область разделяется между многими процессами, то Pss составляет всего 6 КиБ.

Существует программа smem, которая собирает информацию из файлов smaps и представляет ее разными способами, в том числе в виде секторных и столбчатых диаграмм. Страница проекта находится по адресу https://www.selenic.com/smem. Программа включена в виде пакета в большинство дистрибутивов для ПК. Но поскольку она написана на Python, для установки ее во встраиваемую систему придется установить все окружение Python, а это слишком много хлопот для одного инструмента. На выручку приходит небольшая утилита smemcap, которая собирает данные о состоянии из каталога /proc в целевой системе и сохраняет их в TAR-файле для последующего анализа на исходном компьютере. Она входит в состав BusyBox, но может быть также скомпилирована из исходного кода smem.

Запустив smem от имени root, мы увидим такие результаты:

```
# smem -t
PID  User  Command                Swap   USS     PSS     RSS
 610 0     /sbin/agetty -s tty00 11  0    128     149     720
1236 0     /sbin/agetty -s ttyGS0 1  0    128     149     720
 609 0     /sbin/agetty tty1 38400   0    144     163     724
 578 0     /usr/sbin/acpid           0    140     173     680
 819 0     /usr/sbin/cron            0    188     201     704
 634 103   avahi-daemon: chroot hel  0    112     205     500
 980 0     /usr/sbin/udhcpd -S /etc  0    196     205     568
 ...
 836 0     /usr/bin/X :0 -auth /var  0   7172    7746    9212
 583 0     /usr/bin/node autorun.js  0   8772    9043   10076
1089 1000  /usr/bin/python -O /usr/  0   9600   11264   16388
-------------------------------------------------------------------
  53 6                               0  65820   78251  146544
```

В последней строке видно, что в данном случае суммарный Pss составляет примерно половину от суммарного Rss.

Если вы не хотите устанавливать Python в целевую систему, то можете собрать данные с помощью smemcap, также от имени root:

```
# smemcap > smem-bbb-cap.tar
```

Затем скопируйте TAR-файл в исходную систему и прочитайте его командой smem -S, для этого привилегии root не нужны:

```
$ smem -t -S smem-bbb-cap.tar
```

Результат получится такой же, как при запуске smem в целевой системе.

Другие инструменты

Еще один способ вывести Pss дает программа ps_mem (https://github.com/pixelb/ps_mem), которая печатает ту же информацию, но в более простом формате. Она также написана на Python.

Для Android имеется еще утитита procrank, которую можно кросс-компилировать для встраиваемой Linux-системы, внеся небольшие изменения. Ее исходный код находится по адресу https://github.com/csimmonds/procrank_linux.

Обнаружение утечек памяти

Утечка памяти возникает, если память выделена, но не освобождена, когда в ней отпала необходимость. Это явление встречается в любых системах, но во встраиваемых особенно неприятно, потому что, во-первых, в них вообще не слишком много памяти, а во-вторых, они часто работают долго без перезагрузки, так что натечь может целая лужа.

Обнаружить утечку памяти позволяют программы free и top, если объем свободной памяти постоянно уменьшается, даже если отказаться от кэшей, как описано в предыдущем разделе. Чтобы найти виновника (или виновников), нужно проанализировать изменение Uss и Rss для всех процессов.

Существует несколько инструментов для поиска утечек памяти в программе. Я рассмотрю два: mtrace и Valgrind.

mtrace

Функция mtrace входит в состав glibc и предназначена для трассировки обращений к malloc(3), free(3) и родственным функциям. Она находит области памяти, которые не были освобождены к моменту завершения программы. Необходимо до запуска программы записать в переменную окружения MALLOC_TRACE путь к файлу, куда будут помещены данные о трассировке, и в начале программы вызвать mtrace(). Если файл MALLOC_TRACE невозможно открыть, то mtrace ничего не делает. Информация о трассировке записывается в текстовом виде, но обычно ее просматривают с помощью программы mtrace.

Вот пример программы с трассировкой:

```
#include <mcheck.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
```

```
  int j;
  mtrace();
  for (j = 0; j < 2; j++)
    malloc(100);   /* Не освобождается: утечка памяти */
  calloc(16, 16);  /* Не освобождается: утечка памяти */
  exit(EXIT_SUCCESS);
}
```

А вот как запустить эту программу и вывести трассу:

```
$ export MALLOC_TRACE=mtrace.log
$ ./mtrace-example
$ mtrace mtrace-example mtrace.log
```

```
Memory not freed:
-----------------
           Address     Size    Caller
0x0000000001479460     0x64 at /home/chris/mtrace-example.c:11
0x00000000014794d0     0x64 at /home/chris/mtrace-example.c:11
0x0000000001479540    0x100 at /home/chris/mtrace-example.c:15
```

К сожалению, `mtrace` не поможет узнать об утечках памяти, пока программа работает. Сначала ее нужно остановить.

Valgrind

Valgrind – очень эффективный инструмент для обнаружения различных ошибок управления памятью, в том числе утечек. Одно из его достоинств состоит в том, что не нужно перекомпилировать исследуемые программы и библиотеки, хотя результаты все же будут лучше, если программа откомпилирована с флагом `-g`, т. е. содержит отладочную информацию. Valgrind запускает программу в эмулированном окружении и в определенных точках перехватывает выполнение. В этом причина колоссального недостатка Valgrind: программа работает гораздо медленнее, чем в нормальном режиме, поэтому тестировать таким образом программы реального времени вряд ли возможно.

Кстати говоря, название программы часто произносят неверно: в Valgrind FAQ сказано, что *grind* произносится «гринд», а не «грайнд». Этот FAQ, документация и сама программа находятся по адресу http://valgrind.org.

В состав Valgrind входит несколько диагностических средств:

- **memcheck**: подразумевается по умолчанию, обнаруживает утечки памяти и общие ошибки работы с памятью;
- **cachegrind**: вычисляет коэффициент попадания в кэш процессора;
- **callgrind**: вычисляет стоимость каждого вызова функции;
- **helgrind**: обнаруживает неправильное использование Pthread API, потенциальные взаимоблокировки и состояния гонки;
- **DRD**: еще один инструмент анализа Pthread;
- **massif**: строит профили использования кучи и стека.

Для выбора нужного инструмента задайте флаг -tool. Valgrind работает на большинстве встраиваемых платформ: ARM (Cortex A), PPC, MIPS и x86 в 32- и 64-разрядных вариантах. Соответствующий пакет имеется в Yocto Project и в Buildroot.

Для поиска утечек памяти следует воспользоваться средством по умолчанию memcheck, задав также флаг --leakcheck=full для печати строк, в которых обнаружена утечка:

```
$ valgrind --leak-check=full ./mtrace-example
==17235== Memcheck, a memory error detector
==17235== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==17235== Using Valgrind-3.10.0.SVN and LibVEX; rerun with -h for copyright info
==17235== Command: ./mtrace-example
==17235==
==17235==
==17235== HEAP SUMMARY:
==17235== in use at exit: 456 bytes in 3 blocks
==17235== total heap usage: 3 allocs, 0 frees, 456 bytes allocated
==17235==
==17235== 200 bytes in 2 blocks are definitely lost in loss record 1 of 2
==17235== at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-linux.so)
==17235== by 0x4005FA: main (mtrace-example.c:12)
==17235==
==17235== 256 bytes in 1 blocks are definitely lost in loss record 2 of 2
==17235== at 0x4C2CC70: calloc (in /usr/lib/valgrind/vgpreload_memcheck-linux.so)
==17235== by 0x400613: main (mtrace-example.c:14)
==17235==
==17235== LEAK SUMMARY:
==17235== definitely lost: 456 bytes in 3 blocks
==17235== indirectly lost: 0 bytes in 0 blocks
==17235== possibly lost: 0 bytes in 0 blocks
==17235== still reachable: 0 bytes in 0 blocks
==17235== suppressed: 0 bytes in 0 blocks
==17235==
==17235== For counts of detected and suppressed errors, rerun with: -v
==17235== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
```

Нехватка памяти

Стандартно память выделяется с избытком (over-commit), т. е. ядро позволяет приложениям выделить больше памяти, чем имеется в наличии. Как правило, это проходит потому, что приложения действительно часто заказывают больше памяти, чем им необходимо. Это заодно упрощает реализацию fork(2): делать копию большой программы безопасно, потому что страницы памяти разделяются благодаря флагу копирования при записи. В большинстве случаев за fork сразу следует вызов exec, который кладет конец совместному владению памятью и загружает новую программу.

Однако всегда остается вероятность, что при определенной рабочей нагрузке группа процессов потребует оплатить выданный ранее чек и окажется, что столько памяти-то и нет. В результате возникает ситуация нехватки памяти (out of memory – OOM). Тогда не остается ничего другого, как принудительно завершать процессы, пока проблема не разрешится. Это задача процесса «out of memory killer».

Прежде чем переходить к деталям, отметим, что существует настроечный параметр в файле `/proc/sys/vm/overcommit_memory`, который управляет политикой выделения памяти ядром:

- `0`: эвристическое выделение с избытком (по умолчанию);
- `1`: всегда выделять с избытком, никогда не проверять;
- `2`: всегда проверять, никогда не выделять с избытком.

Значение 1 полезно, только если имеются программы, которые работают с большими разреженными массивами, т. е. выделяют большие области памяти, но пишут в небольшие их участки. Во встраиваемых системах такие программы – редкость.

Значение 2 (никогда не выделять с избытком) представляется разумным выбором, если вас беспокоит возможность нехватки памяти в особо ответственном или важном для обеспечения безопасности приложении. При такой политике ядро откажется выделять память сверх предела, который равен размеру области подкачки плюс размер физической памяти, умноженный на коэффициент избытка. Коэффициент избытка управляется параметром `/proc/sys/vm/overcommit_ratio` и по умолчанию равен 50%.

Предположим, к примеру, что имеется устройство с ОЗУ объемом 512 МиБ и задан консервативный коэффициент 25%:

```
# echo 25 > /proc/sys/vm/overcommit_ratio
# grep -e MemTotal -e CommitLimit /proc/meminfo
MemTotal:    509016 kB
CommitLimit: 127252 kB
```

Подкачки нет, поэтому предельный объем выделенной памяти равен 25% от `MemTotal`, как и ожидалось.

В файле `/proc/meminfo` есть еще одна важная переменная: `Committed_AS`. Это общий объем памяти, необходимой для выполнения всех данных обязательств по ее выделению. Как-то в одной системе я встретил такую ситуацию:

```
# grep -e MemTotal -e Committed_AS /proc/meminfo
MemTotal:     509016 kB
Committed_AS: 741364 kB
```

Иными словами, ядро уже пообещало больше памяти, чем физически доступно. Поэтому задание `overcommit_memory=2` означает, что любая попытка выделить память окажется неудачной вне зависимости от значения `overcommit_ratio`. Чтобы получить работоспособную систему, придется либо удвоить объем ОЗУ, либо сильно сократить число работающих процессов, каковых на тот момент было 40.

В любом случае последним рубежом обороны является убийца процессов «OOM killer». Этот компонент ядра с помощью эвристической методики назначает каждому процессу оценку от 0 до 1000, а затем завершает процессы с наибольшей оценкой, пока не освободится достаточно памяти. В журнале ядра появятся сообщения такого вида:

```
[44510.490320] eatmem invoked oom-killer: gfp_mask=0x200da, order=0, oom_score_adj=0
...
```

Можно принудительно вызвать событие нехватки памяти, выполнив команду `echo f > /proc/sysrq-trigger`.

Повлиять на оценку процесса можно, записав поправочное значение в файл `/proc/<PID>/oom_score_adj`. Значение `-1000` означает, что оценка данного процесса никогда не превысит 0, поэтому он не будет завершен; значение `+1000` означает, что оценка будет заведомо больше 1000, так что процесс будет завершен непременно.

Дополнительная литература

Ниже перечислены ресурсы, в которых можно найти дополнительные сведения по вопросам, затронутым в данной главе.

- *Love R.* Linux Kernel Development. 3rd ed. Addison Wesley; O'Reilly Media, 2010. ISBN-10: 0672329468.
- *Love R.* Linux System Programming. 2nd ed. O'Reilly Media, 2013. ISBN-10: 1449339530.
- *Gorman M.* Understanding the Linux VM Manager. URL: https://www.kernel.org/doc/gorman/pdf/understand.pdf.
- *Seward J., Nethercote N., Weidendorfer J.* Valgrind 3.3 – Advanced Debugging and Profiling for Gnu/Linux Applications. Network Theory Ltd, 2008. ISBN 978-0954612054.

Резюме

Учесть каждый байт памяти в системе с виртуальной памятью попросту невозможно. Но команда `free` дает достаточно точную оценку общего объема свободной памяти, за исключением занятой буферами и кэшем. Последив за ней в течение некоторого времени при разных рабочих нагрузках, можно с большой долей уверенности решить, будет ли занятая память оставаться в заданных пределах.

Если понадобится точно настроить использование памяти или выявить источники неожиданного потребления, то существуют инструменты для получения детальной информации. В том, что касается ядра, наиболее полезные сведения дает каталог `/proc`: `meminfo`, `slabinfo` и `vmallocinfo`.

Для точного измерения памяти в пользовательском пространстве лучше всего подходит метрика Pss, которую вычисляют программа `smem` и другие инструмен-

ты. Для отладки ошибок работы с памятью есть как простые трассировщики типа `mtrace`, так и тяжеловесный и всеобъемлющий инструмент Valgrind memcheck.

Если вас беспокоят последствия нехватки памяти, то можете настроить механизм ее выделения с помощью параметра `/proc/sys/vm/overcommit_memory` и контролировать шансы на принудительное завершение отдельных процессов, задавая параметр `oom_score_adj`.

Следующая глава посвящена отладке кода пользовательских приложений и ядра с помощью отладчика GNU. Я расскажу также о том, какие знания, в том числе об управлении памятью, можно почерпнуть, наблюдая за исполнением кода.

Глава 12

Отладка в GDB

Ошибки случаются. Их поиск и исправление – часть процесса разработки. Существует много методов нахождения и классификации дефектов программы, в том числе статический и динамический анализы, экспертиза кода, трассировка, профилирование и интерактивная отладка. Средства трассировки и профилирования будут рассмотрены в следующей главе, а здесь мы сосредоточимся на традиционном подходе к наблюдению за кодом во время выполнения в отладчике, в данном случае **GDB**. Отладчик GDB – мощный и гибкий инструмент. Его можно использовать для отладки приложений, анализа посмертных дампов памяти (core-файлов), создаваемых после аварийного завершения программы, и даже для пошагового выполнения кода ядра.

В этом разделе я покажу, как использовать GDB для отладки приложений, как анализировать файлы дампов памяти и как отлаживать код ядра, акцентируя внимание на аспектах, относящихся к встраиваемым Linux-системам.

Отладчик GNU

GDB – отладчик исходного кода для компилируемых языков, главным образом C и C++, хотя поддерживаются и другие языки, например Go и Objective. В замечаниях к используемой версии GDB вы найдете сведения о текущем положении дел с поддержкой различных языков. На сайте проекта по адресу http://www.gnu.org/software/gdb есть много полезной информации, в том числе руководство.

Родной интерфейс GDB – командная строка – многим внушает отвращение, хотя, немного приноровившись, с ним очень легко работать. Если командные интерфейсы – не ваш конек, то есть немало графических пользовательских интерфейсов к GDB, и некоторые из них я опишу ниже.

Подготовка к отладке

Отлаживаемую программу необходимо откомпилировать, включив отладочные символы. В GCC для этого есть два флага: `-g` и `-ggdb`. Второй добавляет отладочную информацию, специфичную для GDB, а первый порождает ее в формате, пригодном для любой целевой операционной системы, т. е. является более пере-

носимым. В нашем случае целевая ОС – всегда Linux, поэтому безразлично, какой флаг использовать: `-g` или `-ggdb`. Интереснее, что оба флага позволяют задать еще и уровень отладочной информации от 0 до 3:

- 0: не добавляется никакой отладочной информации, то же самое, что вообще не задавать флаг `-g` или `-ggdb`;
- 1: добавляется минимум информации – только имена функций и внешних переменных, этого достаточно, чтобы сгенерировать обратную трассировку вызовов;
- 2: режим по умолчанию, добавляется информация о локальных переменных и номерах строк, так что появляется возможность отлаживать исходный код и выполнять его в пошаговом режиме;
- 3: включается дополнительная информация, в том числе позволяющая GDB корректно расширять макросы.

В большинстве случаев хватает флага `-g`, но вспомните о режиме `-g3` или `-ggdb3`, если возникнут проблемы при пошаговом выполнении кода, особенно содержащего макросы.

Следующий вопрос – уровень оптимизации кода. В процессе оптимизации компилятором часто уничтожается связь между строками исходного кода и машинным кодом, в результате чего пошаговое выполнение становится непредсказуемым. Столкнувшись с такими проблемами, перекомпилируйте программу без оптимизации, убрав флаг `-O` вообще или хотя бы понизив уровень до 1 (`-O1`).

Сюда же относится проблема указателей на кадры стека, которые нужны GDB для порождения обратной трассировки вызовов функций. В некоторых архитектурах GCC не генерирует указатели на кадры стека при высоких уровнях оптимизации (`O2`). Если вы оказались в ситуации, когда необходимо откомпилировать программу с флагом `O2`, сохранив, однако, возможность обратной трассировки, то это поведение можно переопределить, задав флаг `-fno-omit-frame-pointer`. Обращайте также внимание на код, который был вручную оптимизирован путем добавления флага `-fomit-frame-pointer`, подавляющего генерацию указателей на кадры стека; возможно, этот флаг придется временно убрать.

Отладка приложений в GDB

Использовать GDB для отладки приложений можно двумя способами. Если вы разрабатываете программу для работы на ПК или на сервере, да и вообще в любом окружении, где можно компилировать и исполнять код на одной и той же машине, то естественно запускать GDB прямо на этой же машине. Но встраиваемые системы обычно разрабатываются с помощью перекрестных наборов инструментов, а потому требуется отлаживать код, работающий на устройстве, управляя им с машины, где находятся исходный код и инструменты. Я расскажу именно об этом сценарии, поскольку он плохо документирован, но является основным для разработчика встраиваемой системы. Я не стану описывать основы работы с GDB, потому что на эту тему есть много достойных материалов, включая само руководство по GDB и дополнительную литературу в конце главы.

Я начну с изложения некоторых деталей работы с программой gdbserver, а затем покажу, как настроить Yocto Project и Buildroot для удаленной отладки.

Удаленная отладка с помощью gdbserver

Основной компонент удаленной отладки – отладочный агент gdbserver, который работает на целевой платформе и управляет выполнением отлаживаемой программы. Gdbserver подключается к экземпляру GDB, работающему на исходном компьютере, по сети или через последовательный интерфейс RS-232.

Отладка через gdbserver производится почти, но не совсем так же, как локально. Различия связаны в основном с тем, что в процесс вовлечены два компьютера, которые должны находиться в нужном для отладки состоянии. Вот несколько моментов, на которые следует обращать внимание:

- в начале сеанса отладки необходимо загрузить отлаживаемую программу в gdbserver на целевой платформе и отдельно загрузить GDB из набора инструментов на исходной машине;
- GDB и gdbserver должны установить между собой соединение перед началом сеанса отладки;
- GDB, работающему на исходной машине, необходимо сообщить, где искать отладочные символы и исходный код, особенно для разделяемых библиотек;
- команда GDB `run` работает не так, как ожидается;
- gdbserver завершается по окончании сеанса отладки, и, чтобы начать новый сеанс, его необходимо запустить повторно;
- отладочные символы и исходный код отлаживаемой программы должны присутствовать на исходной машине, а на целевой их присутствие необязательно. Зачастую на целевой платформе для них недостаточно места, поэтому перед развертыванием необходимо удалить таблицу символов командой `strip`;
- комбинация GDB/gdbserver поддерживает не все возможности, доступные при локальной отладке, например gdbserver не может последовать за процессом-потомком после `fork()`, а локальный GDB может;
- если версии GDB и gdbserver различны или одинаковы, но по-разному сконфигурированы, то могут происходить странные вещи. В идеале их следует собирать из одного и того же исходного кода одной и той же системой сборки.

Наличие отладочных символов резко увеличивает размер исполняемого файла, бывает, что и в 10 раз. Как было сказано в главе 5, иногда полезно удалить таблицу символов без перекомпиляции. Для этого служит программа `strip`, входящая в набор инструментов. Ее поведением управляют следующие флаги:

- `--strip-all`: (по умолчанию) удалить все символы;
- `--strip-unneeded`: удалить символы, не нужные для процедуры перемещения в памяти;
- `--strip-debug`: удалить только отладочные символы.

Для приложений и разделяемых библиотек режим по умолчанию `--strip-all` безопасен, но если так поступить с модулем ядра, то он перестанет загружаться. В этом случае нужно задавать флаг `--strip-unneeded`. Я до сих пор не придумал применения флагу `--strip-debug`.

Памятуя об этом, рассмотрим особенности отладки в Yocto Project и Buildroot.

Настройка Yocto Project

Yocto Project собирает перекрестный GDB для исходной системы как часть SDK, но, чтобы включить gdbserver в состав образа целевой системы, нужно внести изменения в конфигурацию. Пакет можно добавить явно, например поместив такую строку в файл `conf/local.conf` (не забудьте о пробеле в начале строки):

```
IMAGE_INSTALL_append = " gdbserver"
```

А можно включить `tools-debug` в переменную `EXTRA_IMAGE_FEATURES`, тогда в образ целевой системы будут добавлены gdbserver и strace (об утилите `strace` я расскажу в следующей главе):

```
EXTRA_IMAGE_FEATURES = "debug-tweaks tools-debug"
```

Настройка Buildroot

В Buildroot необходимо включить сборку перекрестного GDB для исходной системы (в предположении, что используется внутренний набор инструментов Buildroot) и сборку gdbserver для целевой системы. Точнее, необходимо включить следующие параметры:

- `BR2_PACKAGE_HOST_GDB` в меню **Toolchain → Build cross gdb for the host**;
- `BR2_PACKAGE_GDB` в меню **Target packages → Debugging, profiling and benchmark → gdb**;
- `BR2_PACKAGE_GDB_SERVER` в меню **Target packages → Debugging, profiling and benchmark → gdbserver**.

Начало отладки

После того как gdbserver установлен на целевую систему, а перекрестный GDB – на исходную, можно начинать сеанс отладки.

Подключение GDB к gdbserver

Соединение между GDB и gdbserver можно установить по сети или по последовательному интерфейсу. В случае соединения по сети мы запускаем gdbserver, указав прослушиваемый порт TCP и факультативно IP-адрес, с которого принимать запросы на подключение. Обычно IP-адрес может быть любым, поэтому нужно указать лишь номер порта. В примере ниже gdbserver ожидает подключения к порту `10000` с любого компьютера:

```
# gdbserver :10000 ./hello-world
Process hello-world created; pid = 103
Listening on port 10000
```

Затем запустите экземпляр GDB из набора инструментов, указав в качестве аргумента ту же программу, чтобы GDB мог загрузить таблицу символов:

```
$ arm-poky-linux-gnueabi-gdb hello-world
```

В GDB введите команду `target remote`, чтобы установить соединение, указав IP-адрес или имя хоста целевой системы и номер порта:

```
(gdb) target remote 192.168.1.101:10000
```

Увидев запрос на подключение, gdbserver напечатает:

```
Remote debugging from host 192.168.1.1
```

Для подключения через последовательный интерфейс процедура аналогична. В целевой системе мы сообщаем gdbserver имя последовательного интерфейса:

```
# gdbserver /dev/ttyO0 ./hello-world
```

Предварительно можно настроить скорость передачи с помощью программы `stty` или ей подобной, например:

```
# stty -F /dev/ttyO1 115200
```

У `stty` есть еще много параметров, почитайте о них на странице руководства. Стоит отметить, что этот порт не должен использоваться ни для каких других целей, например в качестве системной консоли. В исходной системе для подключения к gdbserver нужно ввести команду `target remote`, указав последовательное устройство на локальной стороне кабеля. Обычно задается еще скорость передачи данных с помощью команды GDB `set remotebaud`:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
```

Задание sysroot

GDB должен знать, где искать отладочные символы и исходный код разделяемых библиотек. При локальной отладке пути известны заранее и встроены в GDB, но если используется перекрестный набор инструментов, то GDB понятия не имеет, где находится корень целевой файловой системы. Вы должны сообщить ему об этом, задав sysroot. В Yocto Project и в Buildroot библиотечные символы обрабатываются по-разному, поэтому и местоположение sysroot различается.

В Yocto Project отладочная информация помещается в образ целевой файловой системы, поэтому нужно будет распаковать tar-файл образа, созданный в каталоге `build/tmp/deploy/images`, выполнив примерно такие команды:

```
$ mkdir ~/rootfs
$ cd ~/rootfs
$ sudo tar xf ~/poky/build/tmp/deploy/images/beaglebone/core-image-minimal-beaglebone.tar.bz2
```

После этого можно прописать в sysroot корень дерева распакованных файлов:

```
(gdb) set sysroot /home/chris/MELP/rootfs
```

Buildroot компилирует библиотеки с минимальным или с полным набором отладочных символов в зависимости от параметра `BR2_ENABLE_DEBUG`, помещает их в каталог технологической подготовки, а затем удаляет таблицы символов при копировании в образ целевой системы. Поэтому в случае Buildroot sysroot всегда указывает на каталог технологической подготовки вне зависимости от того, куда будет распакована корневая файловая система.

Командные файлы GDB

Некоторые действия необходимо выполнять при каждом запуске GDB, например задавать значение sysroot. Удобно поместить такие команды в файл, который будет автоматически обрабатываться при запуске. GDB читает команды сначала из файла `$HOME/.gdbinit`, затем из файла `.gdbinit` в текущем каталоге, а потом из файлов, заданных в командной строке с помощью параметра `-x`. Но в последних версиях GDB по умолчанию перестал читать файл `.gdbinit` в текущем каталоге из соображений безопасности. Это поведение можно изменить для одного конкретного каталога, добавив в файл `$HOME/.gdbinit` строку вида:

```
add-auto-load-safe-path /home/chris/myprog/.gdbinit
```

Можно также подавить проверку глобально, добавив команду:

```
set auto-load safe-path /
```

Лично я предпочитаю указывать командный файл с помощью параметра `-x`, чтобы не забывать, где находится мой файл.

Для упрощения настройки GDB Buildroot создает командный файл `output/staging/usr/share/buildroot/gdbinit`, содержащий правильную команду задания sysroot вида:

```
set sysroot /home/chris/buildroot/output/host/usr/arm-buildroot-linux-gnueabi/sysroot
```

Обзор команд GDB

В GDB очень много команд, они описаны в онлайновом руководстве и в ресурсах, упомянутых в разделе «Дополнительная литература». Чтобы помочь вам в освоении, ниже приведен перечень наиболее употребительных команд. Как правило, для каждой команды имеется сокращенная форма, которая указана под полным названием команды.

Точки прерывания

В таблице ниже перечислены команды для работы с точками прерывания.

Команда	Назначение
break <location> b <location>	Установить точку прерывания на функции с указанным именем, на строке с указанным номером или на строке указанного файла, например: "main", "5" "sortbug.c:42"
info break i b	Вывести список точек прерывания
delete break <N> d b <N>	Удалить точку прерывания с номером N

Запуск и пошаговое выполнение

В таблице ниже перечислены команды для запуска и пошагового выполнения программы.

Команда	Назначение
run r	Загрузить программу в память и начать ее выполнение. Для удаленной отладки с помощью gdbserver не работает
continue c	Продолжить выполнение после остановки в точке прерывания
Ctrl-C	Остановить отлаживаемую программу
step s	Выполнить одну строку кода с заходом в вызываемую функцию
next n	Выполнить одну строку кода без захода в вызываемую функцию
finish	Продолжать выполнение до возврата из текущей функции

Информационные команды

В таблице ниже перечислены команды для получения различной информации.

Команда	Назначение
backtrace bt	Распечатать стек вызовов
info threads	Получить список потоков
info libs	Получить список библиотек
print <variable> p <variable>	Напечатать значение переменной, например: print foo
list	Напечатать строки вокруг исполняемой в данный момент

Выполнение до точки прерывания

Gdbserver загружает программу в память и устанавливает точку прерывания в первой строке, а затем ждет подключения GDB. После подключения начинается сеанс отладки. Но попытка сразу перейти в режим пошагового выполнения приводит к такому сообщению:

```
Cannot find bounds of current function
(Невозможно определить границы текущей функции)
```

Это связано с тем, что программа остановлена в написанном на ассемблере коде создания среды исполнения для программ на C и C++. В программах на этих языках первой исполняется функция main(). Если вы хотите остановиться в main(), то поставьте там точку прерывания, а затем выполните команду continue (сокращенно c), которая просит gdbserver продолжить выполнение до этой точки:

```
(gdb) break main
Breakpoint 1, main (argc=1, argv=0xbefffe24) at helloworld.c:8
8 printf("Hello, world!\n");
```

Если после этого выдается сообщение

```
warning: Could not load shared library symbols for 2 libraries, e.g. /lib/libc.so.6.
```

значит, вы забыли установить sysroot!

Эта процедура сильно отличается от запуска программы в локальном сеансе отладки, где достаточно просто набрать run. Если вы попробуете набрать run в удаленном сеансе, то либо увидите сообщение о том, что удаленная система не поддерживает run, либо (в более старых версиях) GDB просто зависнет без всяких объяснений.

Отладка разделяемых библиотек

Для отладки библиотек, построенных системой сборки, нужно будет внести несколько изменений в конфигурацию сборки. А если библиотека строится без помощи системы сборки, то придется проделать дополнительную работу.

Yocto Project

В Yocto Project отладочные варианты двоичных пакетов строятся и помещаются в файл build/tmp/deploy/<менеджер пакетов>/<целевая архитектура>. Например, для библиотеки C отладочный пакет называется так:

```
build/tmp/deploy/rpm/armv5e/libc6-dbg-2.21-r0.armv5e.rpm
```

Можно избирательно включить некоторые отладочные пакеты в образ целевой системы, добавив строку <package name-dbg> в рецепт. Для glibc пакет называется glibc-dbg. Можно также заказать установку всех отладочных пакетов, добавив строку dbg-pkgs в переменную EXTRA_IMAGE_FEATURES. Но имейте в виду, что это резко увеличит размер образа, иногда на несколько сотен мегабайтов.

Yocto Project помещает отладочные символы в скрытый каталог .debug, присутствующий в каталогах lib и usr/lib. GDB знает, что сведения о символах нужно искать в этих местах внутри sysroot.

Отладочные пакеты содержат также копию исходного кода в каталоге usr/src/debug/<имя пакета> в образе, и это одна из причин увеличения размера. Это можно предотвратить, добавив в рецепты строку

```
PACKAGE_DEBUG_SPLIT_STYLE = "debug-without-src"
```

Не забывайте, однако, что для удаленной отладки с помощью gdbserver отладочные символы и исходный код нужны только в исходной системе, а не в целевой. Так что вполне можно удалить каталоги `lib/.debug`, `usr/lib/.debug` и `usr/src` из образа, установленного на целевую систему.

Buildroot

Определяющей характеристикой Buildroot является простота. Нужно всего лишь пересобрать, включив параметр `BR2_ENABLE_DEBUG` в меню **Build options → build packages with debugging symbols**.

В результате будут созданы библиотеки с отладочными символами в каталоге `output/host/usr/<arch>/sysroot`, но из их копий в образе целевой системы таблицы символов все равно вырезаются. Если отладочные символы там нужны, например для запуска GDB локально, то можно подавить вырезание, присвоив параметру **Build options → strip command for binaries on target** значение `none`.

Другие библиотеки

Помимо сборки с отладочными символами, необходимо сообщить GDB, где искать исходный код. В GDB встроены пути поиска исходных файлов, их показывает команда `show directories`:

```
(gdb) show directories
Source directories searched: $cdir:$cwd
```

Это пути по умолчанию: `$cdir` – каталог, в котором компилировался исходный код, а `$cwd` – текущий рабочий каталог GDB.

Обычно этого достаточно, но если исходный код был куда-то перемещен, то нужно будет выполнить команду `directory`, как показано ниже:

```
(gdb) dir /home/chris/MELP/src/lib_mylib
Source directories searched: /home/chris/MELP/src/lib_mylib:$cdir:$cwd
```

Своевременная отладка

Иногда программа начинает сбоить, проработав какое-то время, и требуется понять, что происходит. Для этого предназначена функция GDB `attach`. Я называю этот режим своевременной (just-in-time) отладкой. Он доступен как для локальной, так и для удаленной отладки.

В случае удаленной отладки нужно узнать PID интересующего процесса и передать его команде gdbserver в качестве значения флага `--attach`. Так, если PID равен 109, то надо набрать:

```
# gdbserver --attach :10000 109
Attached; pid = 109
Listening on port 10000
```

Это приведет к остановке процесса, как если бы он встретил точку прерывания, после чего вы сможете запустить перекрестный GDB и подключиться к gdbserver.

По завершении отладки можно отсоединить отладчик от процесса, дав программе возможность нормально работать дальше:

```
(gdb) detach
Detaching from program: /home/chris/MELP/helloworld/helloworld, process 109
Ending remote debugging.
```

Отладка разветвлений и потоков

Что происходит, когда отлаживаемая программа разветвляется? За кем последует отладчик: за родителем или за потомком? Это поведение управляется параметром `follow-fork-mode`, который может принимать значение `parent` или `child`, причем по умолчанию подразумевается родитель (`parent`). К сожалению, последние версии gdbserver не поддерживают эту возможность, так что она работает только в локальном сеансе. Если все-таки необходимо отладить процесс-потомок, то можно проделать обходной маневр: модифицировать код, так что сразу после разветвления потомок крутится в бесконечном цикле по некоторой переменной. Это позволит присоединить к нему gdbserver, начав новый сеанс, в котором присвоить переменной такое значение, которое заставит программу выйти из цикла.

Если какой-то поток в многопоточной программе встречает точку прерывания, то по умолчанию останавливаются все потоки. В большинстве случаев это именно то, что нужно, потому что дает возможность проверить статические переменные, не опасаясь, что их изменят другие потоки. После возобновления текущего потока возобновляются и все остальные, даже если в этот момент вы проходили код в пошаговом режиме, и именно эта ситуация может привести к проблемам. Стратегию обработки остановленных потоков в GDB можно изменить с помощью параметра `scheduler-locking`. Обычно он равен `off`, но если установить его в `on`, то возобновляется только поток, остановленный в точке прерывания, а прочие остаются в приостановленном состоянии. Тем самым вы получаете возможность посмотреть, что поток будет делать в одиночестве, когда ему никто не мешает. Этот режим остается в силе, пока параметр `scheduler-locking` не будет сброшен в `off`. Gdbserver поддерживает эту функциональность.

Core-файлы

В core-файлах хранится состояние аварийно завершившейся программы в момент ее краха. Вполне возможно, что вы в этот момент не работали с отладчиком. Так что, увидев сообщение `Segmentation fault (core dumped)`, не впадайте в отчаяние, а просто покопайтесь в золотых россыпях информации, содержащейся в core-файле.

Прежде всего отметим, что core-файлы по умолчанию не создаются, для этого нужно, чтобы соответствующее ограничение на ресурсы было отлично от 0. В текущей оболочке для этого нужно выполнить команду `ulimit -c`. Чтобы снять любые ограничения на размер core-файлов, введите команду:

```
$ ulimit -c unlimited
```

По умолчанию core-файл называется core и создается в текущем рабочем каталоге процесса, путь к которому записан в файле /proc/<PID>/cwd. У такой схемы есть ряд недостатков. Во-первых, если на устройстве имеется несколько файлов core, то непонятно, какими программами они были сгенерированы. Во-вторых, текущий каталог вполне может находиться в неизменяемой файловой системе, или в файловой системе может оказаться недостаточно места для его создания, или процесс не имеет права писать в текущий каталог.

Есть два файла, контролирующие именование и размещение файлов core. Первый – /proc/sys/kernel/core_uses_pid. Если записать в него 1, то к имени core-файла будет дописан PID скончавшегося процесса; это полезно, только если можно связать PID с именем программы, посмотрев журналы.

Куда полезнее файл /proc/sys/kernel/core_pattern, который дает намного больший контроль над core-файлами. По умолчанию подразумевается образец имени core, но это можно изменить, добавив следующие метасимволы:

- %p: PID процесса;
- %u: истинный UID скончавшегося процесса;
- %g: истинный GID скончавшегося процесса;
- %s: номер сигнала, приведшего к сбросу дампа памяти;
- %t: время дампа в секундах от точки отсчета, 1970-01-01 00:00:00 +0000 (UTC);
- %h: имя хоста;
- %e: имя исполняемого файла;
- %E: путь к исполняемому файлу, в котором знаки косой черты (/) заменены восклицательными знаками (!);
- %c: мягкое ресурсное ограничение на размер core-файла, действовавшее в скончавшемся процессе.

Можно также включить в начало образца абсолютный путь к каталогу, так что все core-файлы будут собраны в одном месте. Например, следующий образец означает, что все core-файлы помещаются в каталог /corefiles и их имена включают имя программы и время краха:

```
# echo /corefiles/core.%e.%t > /proc/sys/kernel/core_pattern
```

После аварийного завершения программы в этом каталоге будет находиться файл вида:

```
$ ls /corefiles/
core.sort-debug.1431425613
```

Дополнительные сведения приведены на странице руководства *core(5)*.

Можно организовать и более хитрую схему обработки core-файлов, направив их по конвейеру программе постобработки. В этом случае образец имени core-файла должен начинаться символом конвейера |, за которым следуют имя и параметры программы. Так, в моей системе Ubuntu 14.04 задан такой образец:

```
|/usr/share/apport/apport %p %s %c %P
```

Apport – это программа формирования отчета о крахе, входящая в состав системы Canonical. Запущенный таким образом инструмент анализа работает, пока программа еще находится в памяти, и ядро передает ему данные из образа памяти через стандартный ввод. Поэтому программа может обработать образ: возможно, вырезать его части, чтобы уменьшить размер результирующего файла, или просто просмотреть в поисках определенной информации. Программа может попутно читать и другие системные данные, например искать в каталоге /proc сведения, относящиеся к упавшей программе, или выполнять системные вызовы ptrace для получения данных непосредственно от программы. Но после того, как образ памяти прочитан из стандартного ввода, ядро выполняет различные операции очистки, стирая всю информацию о почившем процессе.

Использование GDB для анализа core-файлов

Ниже приведен пример сеанса GDB, в котором анализируется core-файл:

```
$ arm-poky-linux-gnueabi-gdb sort-debug /home/chris/MELP/rootdirs/rootfs/corefiles/core.
sort-debug.1431425613
[...]
Core was generated by `./sort-debug'.
Program terminated with signal SIGSEGV, Segmentation fault.
#0  0x000085c8 in addtree (p=0x0, w=0xbeac4c60 "the") at sort-debug.c:41
41     p->word = strdup (w);
```

Мы видим, что программа остановилась в строке 43. Команда list показывает соседние строки:

```
(gdb) list
37  static struct tnode *addtree (struct tnode *p, char *w)
38  {
39    int cond;
40
41    p->word = strdup (w);
42    p->count = 1;
43    p->left = NULL;
44    p->right = NULL;
45
```

Команда backtrace (сокращенно bt) показывает, как мы попали в эту точку:

```
(gdb) bt
#0 0x000085c8 in addtree (p=0x0, w=0xbeac4c60 "the") at sort-debug.c:41
#1 0x00008798 in main (argc=1, argv=0xbeac4e24) at sort-debug.c:89
```

Ошибка очевидна: при вызове addtree() передан нулевой указатель.

Пользовательские интерфейсы к GDB

На нижнем уровне определен интерфейс машинного управления GDB, GDB/MI, который позволяет погрузить GDB в более крупную программу с пользовательским интерфейсом и тем самым значительно расширить набор возможностей.

Я расскажу только о тех возможностях, которые полезны в контексте встраиваемой системы.

Терминальный пользовательский интерфейс

Терминальный пользовательский интерфейс (**TUI**) – факультативная часть стандартного пакета GDB. В центре находится окно кода, где показана строка, ожидающая выполнения, а также все точки прерывания. Это, конечно, лучше, чем команда `list` в командном режиме GDB.

TUI хорош тем, что «просто работает», не требуя никаких дополнительных настроек. А поскольку он работает в текстовом режиме, то его можно использовать в терминальном сеансе ssh, когда `gdb` запущен локально в целевой системе. В большинстве перекрестных наборов инструментов сконфигурирован интерфейс TUI к GDB. Достаточно добавить в командную строку флаг `-tui`, как появится такое окно:

```
Terminal
File Edit View Search Terminal Help
   ┌─sort-debug.c─────────────────────────────────────────────────────┐
   │36       * the count, otherwise add a new node */                  │
   │37      static struct tnode *addtree (struct tnode *p, char *w)    │
   │38      {                                                          │
   │39              int cond;                                          │
   │40                                                                 │
B+>│41              p->word = strdup (w);                              │
   │42              p->count = 1;                                      │
   │43              p->left = NULL;                                    │
   │44              p->right = NULL;                                   │
   │45                                                                 │
   │46              cond = strcmp (w, p->word);                        │
   │47                                                                 │
   │48              if (cond == 0)                                     │
   └───────────────────────────────────────────────────────────────────┘
remote Thread 95 In: addtree                           Line: 41   PC: 0x85b4
Breakpoint 1, main (argc=1, argv=0xbefffe24) at sort-debug.c:72
(gdb) break addtree
Breakpoint 2 at 0x85b4: file sort-debug.c, line 41.
(gdb) c
Continuing.

Breakpoint 2, addtree (p=0x0, w=0xbefffc60 "the") at sort-debug.c:41
(gdb) 
```

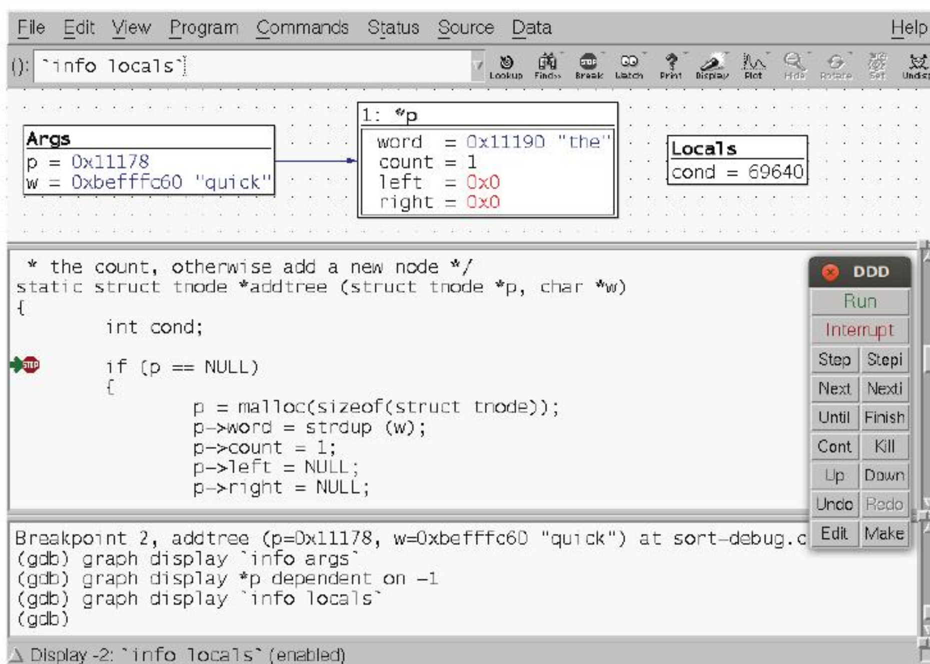
Отладчик DDD

Data display debugger (DDD) – простая автономная программа, которая предоставляет графический интерфейс к GDB с минимальными хлопотами. Хотя элементы управления в пользовательском интерфейсе выглядят старомодными, программа делает все необходимое.

Параметр -debugger сообщает DDD, какую версию GDB из вашего набора инструментов использовать, а параметр -x позволяет указать командные файлы GDB:

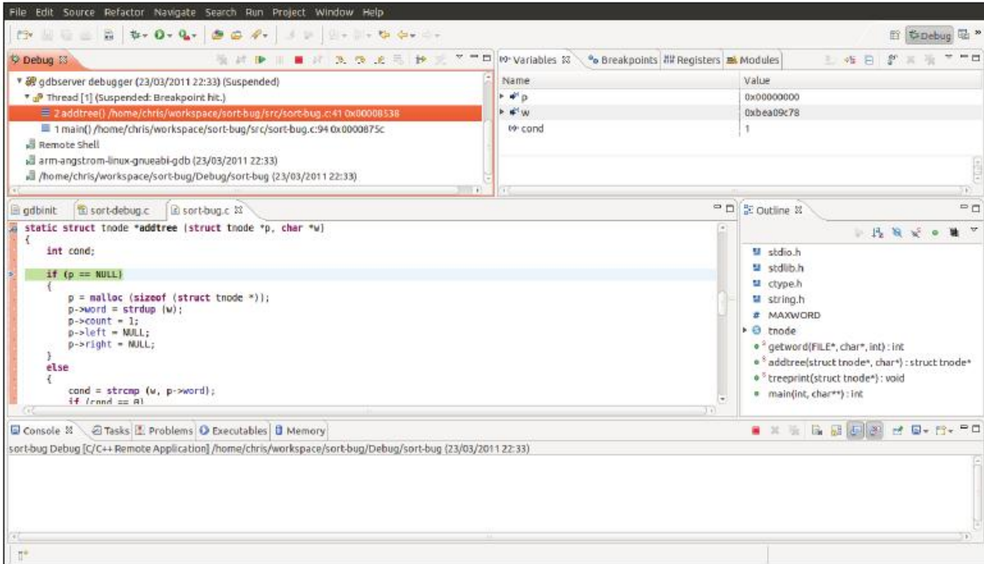
```
$ ddd --debugger arm-poky-linux-gnueabi-gdb -x gdbinit sort-debug
```

На рисунке ниже показана одна из самых удобных функций: окно данных с сеткой элементов, которые можно переупорядочивать по своему желанию. Если дважды щелкнуть по указателю p, то появится новый элемент данных и ссылка на него в виде стрелки:



Eclipse

Eclipse с подключаемым модулем разработки на C (C development toolkit – **CDT**) поддерживает отладку в GDB, в том числе удаленную. Если вы ведете всю разработку в Eclipse, то, конечно, им и стоит пользоваться. Если же вы используете Eclipse от случая к случаю, то настраивать его ради одной лишь этой задачи смысла нет. Мне потребовалась бы целая глава, чтобы объяснить, как сконфигурировать CDT для работы с перекрестным набором инструментов и подключиться к удаленному устройству, поэтому я лучше отошлю вас к ссылкам в конце главы. На рисунке ниже показано отладочное представление CDT. В левом верхнем углу находятся кадры стека для каждого потока, а в правом верхнем – окно наблюдения, в котором отображаются переменные. В центре расположено окно кода, в нем выделена строка, на которой остановился отладчик.



Отладка кода ядра

Отладка кода приложения помогает понять, как работает программа и что происходит, когда она сбоит. То же самое применимо и к ядру, правда, с некоторыми ограничениями.

Для отладки ядра на уровне исходного кода можно воспользоваться программой `kgdb` – примерно так же, как в случае удаленной отладки с помощью `gdbserver`. Существует также автономный отладчик ядра `kdb`, удобный для сравнительно простых задач, например посмотреть, какая команда выполняется, и получить трассировку обратных вызовов. Наконец, сообщения об ошибках (oops) и паниках ядра содержат развернутую информацию о произошедшем в ядре исключении.

Отладка кода ядра в kgdb

Анализируя исходный код ядра в отладчике, вы должны помнить, что ядро – сложная система, иногда требующая работы в реальном масштабе времени. Не надейтесь, что отлаживать его будет так же просто, как приложения. Пошаговое выполнение кода, изменяющего отображение памяти или переключающего контекст, скорее всего, приведет к странным результатам.

`kgdb` – это собирательное название вставок в ядро, предназначенных для поддержки GDB, которые уже много лет назад включены в стержневую версию Linux. В составе документации по ядру DocBook имеется руководство пользователя, а его онлайновая версия находится по адресу https://www.kernel.org/doc/htmldocs/kgdb/index.html.

Чаще всего используется метод подключения к kgdb по последовательному интерфейсу, который обычно разделяется с последовательной консолью, поэтому называется `kgdboc` – «kgdb поверх консоли». Для работы понадобится платформенный драйвер tty, поддерживающий опрос ввода-вывода вместо прерываний, поскольку kgdb вынужден запрещать прерывания при взаимодействии с GDB. На нескольких платформах поддерживается kgdb поверх USB, есть также версии, работающие поверх Ethernet, но, к сожалению, ни одна из них не включена в стержневую версию Linux.

Высказанные ранее замечания об оптимизации и кадрах стека относятся и к ядру – с тем ограничением, что в коде ядра предполагается уровень оптимизации не ниже `O1`. Чтобы переопределить флаги компиляции ядра, установите переменную окружения `KCGLAGS` до запуска `make`.

Ниже перечислены конфигурационные параметры ядра, необходимые для его отладки:

- `CONFIG_DEBUG_INFO` в меню **Kernel hacking → Compile-time checks and compiler options → Compile the kernel with debug info**;
- `CONFIG_FRAME_POINTER` может относиться к вашей архитектуре и находится в меню **Kernel hacking → Compile-time checks and compiler options → Compile the kernel with frame pointers**;
- `CONFIG_KGDB` в меню **Kernel hacking → KGDB: kernel debugger**;
- `CONFIG_KGDB_SERIAL_CONSOLE` в меню **Kernel hacking → KGDB: kernel debugger → KGDB: use kgdb over the serial console**.

Помимо сжатого образа ядра `uImage` или `zImage`, вам понадобится образ ядра в объектном формате ELF, чтобы GDB мог загрузить символы в память. Этот файл называется `vmlinux` и создается в каталоге, где собиралась Linux. В Yocto Project можно потребовать включения копии этого файла в образ целевой системы, это удобно для рассматриваемой и других задач отладки. Файл является частью пакета `kernel-vmlinux`, который можно установить, как любой другой, например добавив его в список `IMAGE_INSTALL_append`. Файл помещается в каталог `boot` и имеет имя вида:

```
boot/vmlinux-3.14.26ltsi-yocto-standard
```

В Buildroot файл `vmlinux` создается в каталоге, где собиралось ядро: `output/build/linux-<строка с номером версии>/vmlinux`.

Пример сеанса отладки

Понять, как все работает, лучше всего на простом примере.

Необходимо сообщить `kgdb`, какой последовательный порт использовать, – либо в командной строке ядра, либо на этапе выполнения через `sysfs`. В первом случае добавьте в командную строку параметр `kgdboc=<tty>,<baud rate>`:

```
kgdboc=ttyO0,115200
```

Во втором случае загрузите устройство и запишите имя терминала в файл `/sys/module/kgdboc/parameters/kgdboc`, например:

```
# echo tty00 > /sys/module/kgdboc/parameters/kgdboc
```

Отметим, что задать скорость передачи данных при таком способе не получится. Используется та же скорость, что для консоли tty, если она уже задана; если нет, воспользуйтесь stty или другой подобной программой.

Теперь можно запустить GDB в исходной системе, указав файл vmlinux, соответствующий работающему ядру:

```
$ arm-poky-linux-gnueabi-gdb ~/linux/vmlinux
```

GDB загружает таблицу символов из файла vmlinux и ждет команды.

Затем закройте эмулятор терминала, присоединенный к консоли: вы собираетесь использовать консоль для GDB, и если обе программы работают одновременно, то отладочные строки могут быть повреждены.

Теперь вернитесь в GDB и попробуйте подключиться к kgdb. Как выясняется, ответ, полученный в этот момент от команды target remote, ничем не помогает:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
Bogus trace status reply from target: qTStatus
```

Проблема в том, что сейчас kgdb не ожидает подключения. Прежде чем начать интерактивный сеанс отладки в GDB, ядро нужно прервать. К сожалению, простого нажатия **Ctrl**+**C** в GDB, как в случае отладки приложения, недостаточно. Необходимо принудительно активировать ловушку в ядре, запустив еще одну оболочку в целевой системе, например через ssh, и записать символ g в файл /proc/sysrq-trigger:

```
# echo g > /proc/sysrq-trigger
```

После этого целевая система намертво зависает. Теперь можно подключиться к kgdb по последовательному интерфейсу:

```
(gdb) set remotebaud 115200
(gdb) target remote /dev/ttyUSB0
Remote debugging using /dev/ttyUSB0
0xc009a59c in arch_kgdb_breakpoint ()
```

Наконец-то GDB получил управление. Можно расставлять точки прерывания, просматривать переменные, трассировать вызовы и т. д. Для примера установим точку прерывания на функции sys_sync:

```
(gdb) break sys_sync
Breakpoint 1 at 0xc0128a88: file fs/sync.c, line 103.
(gdb) c
Continuing.
```

Целевая система ожила. Если ввести в ней команду sync, то будет вызвана функция sys_sync и произойдет останов в точке прерывания:

```
[New Thread 87]
[Switching to Thread 87]
Breakpoint 1, sys_sync () at fs/sync.c:103
```

Если вы закончили сеанс отладки и хотите деактивировать `kgdboc`, то просто запишите в файл `kgdboc` пустую строку:

```
# echo "" > /sys/module/kgdboc/parameters/kgdboc
```

Отладка на ранних стадиях

В примере выше мы рассмотрели случай, когда интересующий нас код выполняется в полностью загруженной системе. Если необходимо начать отладку на более ранней стадии, то можно попросить ядро подождать во время загрузки, добавив в командную строку параметр `kgdbwait` после `kgdboc`:

```
kgdboc=ttyO0,115200 kgdbwait
```

Тогда на этапе загрузки на консоли появится такое сообщение:

```
1.103415] console [ttyO0] enabled
[ 1.108216] kgdb: Registered I/O driver kgdboc.
[ 1.113071] kgdb: Waiting for connection from remote gdb...
```

Теперь можете закрыть консоль и подключиться из GDB, как обычно.

Отладка модулей

Отладка модулей ядра представляет отдельную проблему, потому что код модуля перемещен в памяти на этапе выполнения, так что необходимо узнать, где он находится. Эту информацию дает `sysfs`. Адреса перемещения для каждой секции модуля хранятся в каталоге `/sys/module/<имя модуля>/sections`. Отметим, что поскольку имена секций ELF начинаются с точки, в каталоге они оказываются скрытыми файлами, поэтому, чтобы их увидеть, нужно выполнить команду `ls -a`. Наиболее важны файлы `.text`, `.data` и `.bss`.

В качестве примера рассмотрим модуль `mbx`:

```
# cat /sys/module/mbx/sections/.text
0xbf000000
# cat /sys/module/mbx/sections/.data
0xbf0003e8
# cat /sys/module/mbx/sections/.bss
0xbf0005c0
```

Эти числа можно указать GDB для загрузки таблицы символов модуля, находящегося по этому адресу:

```
(gdb) add-symbol-file /home/chris/mbx-driver/mbx.ko 0xbf000000 \
-s .data 0xbf0003e8 -s .bss 0xbf0005c0
add symbol table from file "/home/chris/mbx-driver/mbx.ko" at
  .text_addr = 0xbf000000
  .data_addr = 0xbf0003e8
  .bss_addr = 0xbf0005c0
```

Теперь все должно работать, как обычно: можно расставлять точки прерывания, просматривать глобальные и локальные переменные модуля точно так же, как для vmlinux:

```
(gdb) break mbx_write
Breakpoint 1 at 0xbf00009c: file /home/chris/mbx-driver/mbx.c, line 93.
(gdb) c
Continuing.
```

Затем заставим драйвер устройства вызвать mbx_write и увидим, как код остановился в точке прерывания:

```
Breakpoint 1, mbx_write (file=0xde7a71c0, buffer=0xadf40 "hello\n\n",
  length=6, offset=0xde73df80)
  at /home/chris/mbx-driver/mbx.c:93
```

Отладка кода ядра в kdb

Отладчик kdb не располагает всеми возможностями kgdb и GDB, но у него есть своя ниша, а поскольку это автономная программа, то можно не беспокоиться о зависимостях. У kdb простой командный интерфейс, пригодный для использования на последовательной консоли. Команды позволяют просматривать содержимое памяти, регистры, списки процессов, сообщения dmesg и даже устанавливать точки прерывания.

Чтобы сконфигурировать kdb для работы на последовательной консоли, разрешите kgdb, как показано выше, и добавьте еще один параметр:

- CONFIG_KGDB_KDB в меню **Kernel hacking → kernel debugger → KGDB_KDB: include kdb frontend for kgdb**.

Если теперь активировать ловушку ядра, то мы не попадем в сеанс GDB, а увидим на консоли приглашение kdb:

```
# echo g > /proc/sysrq-trigger
[ 42.971126] SysRq : DEBUG
Entering kdb (current=0xdf36c080, pid 83) due to Keyboard Entry
kdb>
```

В оболочке kdb можно выполнить ряд операций, полный список которых печатает команда help. Приведем краткий обзор.

Получение информации:

- ps: вывести список активных процессов;
- ps A: вывести список всех процессов;
- lsmod: вывести список модулей;
- dmesg: распечатать буфер журнала ядра.

Точки прерывания:

- bp: установить точку прерывания;
- bl: вывести список точек прерывания;
- bc: удалить точку прерывания;
- bt: вывести обратную трассу вызовов;
- go: продолжить выполнение.

Просмотр памяти и регистров:

- md: вывести содержимое участка памяти;
- rd: вывести содержимое регистров.

Ниже приведен пример установки точки прерывания:

```
kdb> bp sys_sync
Instruction(i) BP #0 at 0xc01304ec (sys_sync)
is enabled addr at 00000000c01304ec, hardtype=0 installed=0
kdb> go
```

Ядро возвращается к жизни, и на консоли появляется обычное приглашение bash. Если ввести команду sync, то ядро встретит точку прерывания и снова войдет в kdb:

```
Entering kdb (current=0xdf388a80, pid 88) due to Breakpoint @ 0xc01304ec
```

kdb – не отладчик исходного кода, поэтому вы не увидите исходного кода и не сможете войти в режим пошагового выполнения. Но можно распечатать трассу вызовов командой bt и получить представление о потоке выполнения программы и иерархии вызовов.

Если ядро пытается получить доступ к несуществующей памяти или выполнить недопустимую команду, то в журнал ядра записывается сообщение об ошибке (oops). Самая полезная его часть – трасса вызовов, и я хочу показать, как воспользоваться этой информацией, чтобы найти, в какой строке произошла ошибка. Я также расскажу, как сохранить сообщение об ошибке ядра, если оно привело к краху системы.

Сообщения об ошибках ядра

Сообщение об ошибке ядра (oops) выглядит так:

```
[ 56.225868] Unable to handle kernel NULL pointer dereference at virtual ad-
dress 00000400[ 56.229038] pgd = cb624000[ 56.229454] [00000400] *pgd=6b715-
831, *pte=00000000, *ppte=00000000[ 56.231768] Internal er-
ror: Oops: 817 [#1] SMP ARM[ 56.232443] Modules linked in: mbx(O)
[ 56.233556] CPU: 0 PID: 98 Comm: sh Tainted: G O 4.1.10 #1[ 56.234234] Hardware name:
ARM-Versatile Express[ 56.234810] task: cb709c80 ti: cb71a000 task.ti: cb71a000[ 56.236801]
PC is at mbx_write+0x14/0x98 [mbx][ 56.237303] LR is at __vfs_write+0x20/0xd8[ 5
6.237559] pc : [<bf0000a0>] lr : [<c0307154>] psr: 800f0013[ 56.237559] sp : cb-
71bef8 ip : bf00008c fp : 00000000[ 56.238183] r10: 00000000 r9 : cb71a000 r8 : c02107c4[
56.238485] r7 : cb71bf88 r6 : 000afb98 r5 : 00000006 r4 : 00000000[ 56.238857] r3 : cb71bf
88 r2 : 00000006 r1 : 000afb98 r0 : cb61d600

[ 56.239276] Flags: Nzcv IRQs on FIQs on Mode SVC_32 ISA ARM Segment user[ 56.239685] Con-
trol: 10c5387d Table: 6b624059 DAC: 00000015[ 56.240019] Process sh (pid: 98, stack lim-
it = 0xcb71a220)
```

Строка PC is at mbx_write+0x14/0x98 [mbx] содержит почти все, что нужно знать: последняя выполненная команда находилась в функции mbx_write модуля ядра mbx и отстояла на 0x14 байтов от начала функции, полная длина которой составляет 0x98 байтов.

Теперь взглянем на трассу вызовов:

```
[ 56.240363] Stack: (0xcb71bef8 to 0xcb71c000)[ 56.240745] bee0:
cb71bf88 cb61d600[ 56.241331] bf00: 00000006 c0307154 00000000 c020a308 cb619d88
00000301 00000000 00000042[ 56.241775] bf20: 00000000 cb61d608 cb709c80 cb709c78 cb71bf60
c0250a54 00000000 cb709ee0[ 56.242190] bf40: 00000003 bef4f658 00000000 cb61d600 cb61d600
00000006 000afb98 cb71bf88[ 56.242605] bf60: c02107c4 c030794c 00000000 00000000 cb61d600
cb61d600 00000006 000afb98[ 56.243025] bf80: c02107c4 c0308174 00000000 00000000 00000000
000ada10 00000001 000afb98[ 56.243493] bfa0: 00000004 c0210640 000ada10 00000001 00000001
000afb98 00000006 00000000[ 56.243952] bfc0: 000ada10 00000001 000afb98 00000004 00000001
00000020 000ae274 00000000[ 56.244420] bfe0: 00000000 bef4f49c 0000fcdc b6f1aedc 600f0010
00000001 00000000 00000000[ 56.245653] [<bf0000a0>] (mbx_write [mbx])
from [<c0307154>]
(__vfs_write+0x20/0xd8)[ 56.246368] [<c0307154>]
(__vfs_write) from [<c030794c>]
(vfs_write+0x90/0x164)[ 56.246843] [<c030794c>] (vfs_write) from [<c0308174>]
(SyS_write+0x44/0x9c)[ 56.247265] [<c0308174>] (SyS_write) from [<c0210640>]
(ret_fast_syscall+0x0/0x3c)[ 56.247737] Code: e5904090 e3520b01 23a02b01 e1a05002 (e584240
0)[ 56.248372]
---[ end trace 999c378e4df13d74 ]---
```

В данном случае мы не узнали почти ничего нового, разве что тот факт, что mbx_write вызвана из кода виртуальной файловой системы.

Было бы хорошо узнать, какой строке программы соответствует адрес mbx_write+0x14, для этого можно использовать утилиту objdump. Выполнив команду objdump -S, мы увидим, что mbx_write отстоит на 0x8c байтов от начала mbx.ko, т. е. последняя выполненная команда имела смещение 0x8c + 0x14 = 0xa0. Остается только посмотреть, что находится по этому смещению:

```
$ arm-poky-linux-gnueabi-objdump -S mbx.kostatic ssize_t mbx_
write(struct file *file,const char *buffer, size_t length, loff_t * offset){ 8c: e92d4038 pu
sh {r3, r4, r5, lr} struct mbx_data *m = (struct mbx_data *)file->private_data; 90: e590409
0 ldr r4, [r0, #144] ; 0x90 94: e3520b01 cmp r2, #1024 ; 0x400 98: 23a02b01 movcs r2, #102
4 ; 0x400 if (length > MBX_LEN) length = MBX_LEN; m->mbx_len = length; 9c: e1a05002 mov r5
, r2 a0: e5842400 str r2, [r4, #1024] ; 0x400
```

Вот теперь видно, на какой строке программа остановилась:

```
m->mbx_len = length;
```

Как видим, m имеет тип struct mbx_data *. Вот как определена эта структура:

```
#define MBX_LEN 1024 struct mbx_data { char mbx[MBX_LEN]; int mbx_ len;};
```

Похоже, что в переменной m находится нулевой указатель, что и привело к ошибке.

Сохранение сообщения об ошибке ядра

Чтобы разобраться в сообщении об ошибке, нужно его для начала сохранить. Если крах системы произошел во время загрузки до инициализации консоли или после

приостановки, то сообщение мы не увидим. Существуют способы записать сообщения об ошибках ядра и информационные сообщения в MTD-раздел или в нестираемую память, но я опишу простую технику, которая работает в большинстве случаев и не требует почти никакой предварительной подготовки.

При условии, что содержимое памяти не повреждено во время сброса (а обычно так и бывает), мы можем повторно войти в начальный загрузчик и использовать его для отображения памяти. Необходимо знать адрес буфера журнала ядра, памятуя о том, что это простой кольцевой буфер текстовых сообщений. Ему соответствует символ `__log_buf`. Поищем его в файле `System.map`, содержащем карту памяти ядра:

```
$ grep __log_buf System.mapc0f72428 b __log_buf
```

Затем отобразим этот логический адрес ядра на физический, который понимает U-Boot, для чего вычтем величину `PAGE_OFFSET`, равную `0xc0000000`, и прибавим адрес начала физического ОЗУ, который для платы BeagleBone равен `0x80000000`. Получим `c0f72428 - 0xc0000000 + 0x80000000 = 80f72428`.

Затем распечатаем журнал с помощью команды U-Boot `md`:

```
U-Boot# md 80f7242880f72428: 00000000 00000000 00210034
c6000000    ........4.!.....80f72438: 746f6f42 20676e69 756e694c
6e6f2078    Booting Linux on80f72448: 79687020 61636973 5043206c
78302055     physical CPU 0x80f72458: 00000030 00000000 00000000 00730084 0.............s.
80f72468: a6000000 756e694c 65762078 6f697372   ....
Linux versio80f72478: 2e34206e 30312e31 68632820 40736972   n 4.1.10 (chris@80f72488:
6c697562 29726564 63672820 65762063   builder)
(gcc ve80f72498: 6f697372 2e34206e 20312e39 6f726328   rsion 4.9.1 (cro80f724a8: 6f747373
4e2d6c6f 2e312047 302e3032    sstool-NG
1.20.080f724b8: 20292029 53203123 5720504d 4f206465   ) ) #1 SMP Wed
O80f724c8: 32207463 37312038 3a31353a 47203335    ct 28 17:51:53 G
```

Начиная с версии Linux 3.5, каждой строке в буфере журнала ядра предшествует 15-байтовый двоичный заголовок, в котором закодированы временная метка, уровень протоколирования и другие вещи. Его структура обсуждается в статье «Toward more reliable logging» на сайте Linux Weekly News по адресу https://lwn.net/Articles/492125/.

Дополнительная литература

Ниже перечислены ресурсы, в которых можно найти дополнительные сведения по вопросам, затронутым в данной главе.

- *Matloff N., Salzman P. J.* The Art of Debugging with GDB, DDD, and Eclipse. 1st ed. No Starch Press, 2008. ISBN 978-1593271749.
- *Robbins A.* GDB Pocket Reference. 1st ed. O'Reilly Media, 2005. ISBN 978-0596100278.
- Getting to grips with Eclipse: cross compiling. URL: http://2net.co.uk/tutorial/eclipse-cross-compile.

- Getting to grips with Eclipse: remote access and debugging. URL: http://2net.co.uk/ tutorial/eclipse-rse.

Резюме

Интерактивный отладчик GDB – полезный инструмент в арсенале разработчика встраиваемых систем. Это стабильная, хорошо документированная и широко известная программа. В ней есть возможность удаленной отладки, для чего необходимо поместить агент на целевую платформу: `gdbserver` для приложений или `kgdb` для отладки ядра. К ее командному интерфейсу нужно привыкнуть, но существуют и альтернативные оболочки. Я упомянул три из них: TUI, DDD и Eclipse, –которых хватает в большинстве ситуаций, но есть и другие – пробуйте.

Не менее важная методика отладки – апостериорный анализ дампов памяти. В эту категорию я отнес также сообщения об ошибках ядра.

Однако отладка – лишь один из способов поиска дефектов в программах. В следующей главе мы поговорим о профилировании и трассировке как способах анализа и оптимизации программ.

Глава 13

Профилирование и трассировка

Интерактивная отладка исходного кода, описанная в предыдущей главе, может пролить свет на работу программы, но она ограничена очень небольшим участком кода. В этой главе мы взглянем на систему в целом, чтобы понять, работает ли она, как было задумано.

Известно, что программисты и проектировщики очень плохо угадывают, где возникнут узкие места. Поэтому если возникают проблемы с производительностью, то следует начать с обзора системы в целом, а затем спускаться вниз, применяя более специализированные инструменты. В этой главе мы начнем рассмотрение с широко известной команды `top`, позволяющей получить общее представление о работе системы. Зачастую проблему можно свести к одной программе, а затем проанализировать ее с помощью профилировщика Linux, `perf`. Если проблему не удается локализовать настолько узко и нужно получить более полную картину, то `perf` пригодится и тут. В контексте диагностики проблем в ядре я опишу инструменты трассировки `Ftrace` и `LTTng`, позволяющие собрать детальную информацию.

Мы рассмотрим также инструмент Valgrind, который исполняет программу в песочнице и потому способен наблюдать за ней и сообщать сведения о коде прямо по ходу его выполнения. И завершим главу описанием простого средства трассировки `strace`, которое сообщает обо всех системных вызовах, сделанных программой.

Эффект наблюдателя

Прежде чем с головой погрузиться в изучение инструментов, поговорим о том, что же они показывают. Как и во многих других областях, измерение свойства оказывает влияние на само свойство. Для измерения силы электрического тока необходимо измерить падение напряжения на небольшом сопротивлении. Однако наличие сопротивления влияет на силу тока. То же справедливо и в отношении профилирования: любое наблюдение за системой потребляет процессорное время, а значит, ресурсы расходуются не только на приложение. Измерительные инструменты так-

же изменяют поведение кэширования, «съедают» память и пишут на диск, что еще больше искажает картину. Не существует измерений без побочных эффектов.

Я часто слышал от инженеров, что результаты профилирования только вводят в заблуждение. Обычно так происходит потому, что измерения выполняются в ситуации, лишь аппроксимирующей реальную. Всегда старайтесь проводить измерения непосредственно на целевой платформе, используя выпускные сборки ПО на реальном наборе данных с минимально возможным числом дополнительных сервисов.

Таблицы символов и флаги компиляции

С первой проблемой мы сталкиваемся сразу же. Систему желательно наблюдать в ее естественном состоянии, но инструментам часто требуется дополнительная информация, чтобы можно было интерпретировать происходящие события.

Для работы некоторых инструментов, в частности `perf`, `Ftrace` и `LTTng`, необходимо специальным образом сконфигурировать ядро. Поэтому для проведения испытания придется собрать и установить новое ядро.

Отладочные символы очень полезны, когда надо транслировать адреса в имена функций и номера строк. Наличие отладочных символов в исполняемом файле не изменяет характеристик выполнения кода, но требует, чтобы приложения и ядро были откомпилированы для отладки, по крайней мере те компоненты, которые вы собираетесь профилировать. Некоторые инструменты, к примеру `perf`, работают лучше, если отладочные символы присутствуют. Методы здесь такие же, как при отладке (см. главу 12).

Если требуется, чтобы инструмент строил графы вызовов, то необходимо компилировать с включенными кадрами стека. Чтобы инструмент точно сопоставлял номера строк адресам, нужно уменьшить уровень оптимизации при компиляции.

Наконец, для работы некоторых инструментов следует оснащать программу специальными средствами измерения, которые формируют выборку, поэтому программу придется пересобрать, включив эти компоненты. Это касается профилировщика приложений `gprof` и программ `Ftrace` и `LTTng` для ядра.

Имейте в виду, что чем сильнее вы изменяете наблюдаемую систему, тем труднее соотнести результаты измерения с работой системы в реальных условиях.

Лучше занять выжидательную позицию и вносить изменения только тогда, когда необходимость в них очевидна, памятуя, что любое изменение обязательно изменяет объект измерения.

Приступая к профилированию

Для получения общего представления о системе лучше начать с простого инструмента, например `top`. Он показывает, сколько использовано памяти, какие процессы потребляют время и как загрузка распределена между процессорными ядрами.

Если `top` показывает, что одно приложение «съедает» все время, то можно заняться его профилированием с помощью `perf`.

Если процессор занимают два и более процессов, то высока вероятность, что их что-то связывает, например обмен данными. Если много времени тратится на системные вызовы или обработку прерываний, то, возможно, есть какая-то проблема в конфигурации ядра или в работе драйвера устройства. В таком случае следует снять профиль всей системы, снова прибегнув к услугам perf.

Чтобы получить больше информации о работе ядра и последовательности событий в нем, стоит воспользоваться программами Ftrace или LTTng.

Возможны и другие проблемы, в решении которых top не поможет. Если приложение многопоточное и периодически происходят зависания или имеют место случайные повреждения данных, то может оказаться полезной программа Valgrind с подключаемым модулем Helgrind. К той же категории относятся утечки памяти, их диагностика была описана в главе 11.

Профилирование с помощью top

Программа top – простой инструмент, для работы которого не нужны ни специальная конфигурация ядра, ни таблицы символов. Базовая версия входит в комплект BusyBox, а функционально более полная – в пакет procps, имеющийся как в Yocto Project, так и в Buildroot. Возможно, вас заинтересует также программа htop, аналогичная top, но с более удобным (по мнению некоторых) пользовательским интерфейсом.

Прежде всего обратите внимание на сводную строку – вторую сверху в версии top из BusyBox и третью в версии из пакета procps. Ниже приведен пример для top из BusyBox:

```
Mem: 57044K used, 446172K free, 40K shrd, 3352K buff, 34452K cached
CPU:  58% usr  4% sys   0% nic   0% idle   37% io   0% irq   0% sirq
Load average: 0.24 0.06 0.02 2/51 105
  PID  PPID USER    STAT    VSZ  %VSZ  %CPU  COMMAND
  105   104 root    R     27912    6%   61%  ffmpeg -i track2.wav
  [...]
```

В сводной строке показана процентная доля времени, проведенного в различных состояниях, а именно:

procps	BusyBox	
us	usr	Программы в пользовательском пространстве с подразумеваемым по умолчанию значением nice
sy	sys	Код ядра
ni	nic	Программы в пользовательском пространстве с измененным значением nice
id	idle	Простой
wa	io	Ожидание завершения ввода-вывода
hi	irq	Аппаратные прерывания
si	sirq	Программные прерывания
st	--	«Украденное» время, имеет смысл только в виртуализированной среде

В примере выше почти все время (58%) потрачено в пользовательском режиме и лишь небольшая часть (4%) в режиме ядра, так что для этой системы характерно высокое потребление процессора пользовательскими приложениями. Из первой строки после сводной видно, что ответственность за это несет всего одно приложение: ffmpeg. Усилия по сокращению потребления ЦП должны быть направлены именно на него.

Вот еще пример:

```
Mem: 13128K used, 490088K free, 40K shrd, 0K buff, 2788K cached
CPU:   0% usr   99% sys   0% nic   0% idle   0% io   0% irq   0% sirq
Load average: 0.41 0.11 0.04 2/46 97
  PID  PPID USER     STAT   VSZ %VSZ  %CPU  COMMAND
   92    82 root      R    2152   0%  100%  cat /dev/urandom
  [...]
```

Эта система тратит почти все время в пространстве ядра – из-за того, что cat читает из /dev/urandom. В этом искусственном случае профилирование cat не поможет, но могло бы помочь профилирование функций ядра, к которым обращается cat.

В представлении по умолчанию показаны только процессы, поэтому потребление ЦП складывается из потребления всех потоков, работающих в процессе. Чтобы увидеть информацию для каждого потока, нажмите **H**. Аналогично в потреблении времени учтена сумма по всем имеющимся процессорам. В версии top из пакета procps можно посмотреть распределение времени по отдельным процессорам, нажав клавишу **1**.

Допустим, что большую часть времени потребляет один процесс в пользовательском пространстве, и посмотрим, как организовать его профилирование.

Профилировщик для бедных

Для профилирования приложения можно просто с помощью GDB останавливать его через случайные промежутки времени и смотреть, что происходит. Это профилировщик для бедных. Его легко настроить, и какой-никакой профиль он позволит построить.

Процедура такова:

1. Присоединиться к процессу с помощью gdbserver (в случае удаленной отладки) или gdb (в случае локальной отладки). Процесс приостанавливается.
2. Записать, в какой функции произошел останов. Можно воспользоваться командой backtrace для вывода стека вызовов.
3. Выполнив команду continue, возобновить выполнение.
4. Через некоторое время снова остановить приложение нажатием **Ctrl**+**C** и перейти к шагу 2.

Повторив шаги 2–4 несколько раз, вы поймете, крутится ли приложение в цикле или делает что-то поступательное. А если повторять их достаточно часто, то можно составить представление о том, где приложение проводит больше всего времени.

Этой идее посвящен целый сайт http://poormansprofiler.org, где имеются скрипты, немного упрощающие процедуру. Я сам много раз применял эту технику в разных операционных системах и с разными отладчиками.

Это пример так называемого статистического профилирования, когда строится выборка состояний программы в разные моменты времени. Если объем выборки достаточно велик, то можно оценить статистические характеристики выполняемых функций. Удивительно, что на самом деле нужно совсем немного примеров. Из других статистических профилировщиков назову `perf`, `OProfile` и `gprof`.

Выборка с применением отладчика влияет на поведение программы, потому что выполнение приостанавливается на достаточно длительное время, чтобы получить очередной пример. У других инструментов накладные расходы значительно ниже.

Рассмотрим теперь статистическое профилирование с применением `perf`.

Применение perf

Аббревиатура `perf` означает «**Linux performance event counter subsystem**» (подсистема счетчиков событий в Linux), `perf_events`. Точно так же называется командная утилита для взаимодействия с `perf_events`. То и другое включено в ядро, начиная с версии Linux 2.6.31. В файле `tools/perf/Documentation` в исходном коде ядра, а также на сайте https://perf.wiki.kernel.org приведено много информации по этому поводу.

Побудительным мотивом разработки `perf` было желание предоставить унифицированный доступ к регистрам блока измерения производительности (performance measurement unit – **PMU**), являющегося составной частью большинства современных процессорных ядер. После того как API был определен и включен в Linux, появилась естественная идея распространить его и на другие типы счетчиков производительности.

На внутреннем уровне `perf` представляет собой набор счетчиков событий вместе с правилами, описывающими, когда в них собираются данные. По-разному задавая правила, можно организовать сбор данных обо всей системе, или только о ядре, или только об одном процессе и его потомках, причем включить все процессоры или только один. Механизм очень гибкий. Располагая одним лишь этим инструментом, можно начать с обзора всей системы, а затем сконцентрироваться на одном подозрительном драйвере, или на медленно работающем приложении, или на библиотечных функциях, которые работают дольше, чем ожидалось.

Код командного инструмента `perf` частично находится в ядре – в каталоге `tools/perf`. Сам инструмент и соответствующая ему подсистема ядра разрабатываются синхронно, поэтому `perf` может работать только в паре со «своей» версией ядра. Функциональность `perf` весьма обширна. В этой главе я рассмотрю лишь использование его в качестве профилировщика. Полное описание возможностей имеется в странице руководства по `perf`, а также в вышеупомянутой документации.

Конфигурирование ядра для работы с perf

Вам понадобятся ядро, в котором сконфигурирована подсистема `perf_events`, и команда `perf`, кросс-компилированная для целевой платформы. Конфигурационный параметр ядра называется `CONFIG_PERF_EVENTS` и находится в меню **General setup → Kernel Performance Events And Counters**.

Если вы хотите использовать для профилирования точки трассировки (подробнее о них ниже), то нужно будет также сконфигурировать параметры, описанные в разделе, посвященном Ftrace. Заодно уж имеет смысл включить параметр CONFIG_DEBUG_INFO.

У команды perf много зависимостей, которые заметно усложняют кросс-компиляцию. Однако и в Yocto Project, и в Buildroot есть соответствующие пакеты для целевых платформ.

Понадобятся также отладочные символы в целевой системе для двоичного кода, который вы собираетесь профилировать, иначе perf не сможет сопоставить осмысленные символы адресам. В идеале хорошо бы иметь отладочные символы для всей системы, включая ядро. Напомню, что отладочные символы для ядра находятся в файле vmlinux.

Сборка perf в Yocto Project

В стандартное ядро linux-yocto подсистема perf_events уже включена, поэтому делать ничего не надо.

Для сборки команды perf необходимо явно добавить ее в состав зависимостей для образа целевой системы или добавить зависимость tools-profile, которая тянет за собой также gprof. Как уже было сказано, понадобятся также отладочные символы для целевой системы и образ ядра vmlinux. Короче говоря, в файл conf/local.conf нужно добавить следующие строки:

```
EXTRA_IMAGE_FEATURES = "debug-tweaks dbg-pkgs tools-profile"
IMAGE_INSTALL_append = " kernel-vmlinux"
```

Сборка perf в Buildroot

Во многие конфигурации ядра в Buildroot подсистема perf_events не входит, поэтому для начала нужно проверить, включены ли параметры, упомянутые в предыдущем разделе.

Для кросс-компиляции perf запустите menuconfig из Buildroot menuconfig и выберите параметр:

- BR2_LINUX_KERNEL_TOOL_PERF в меню **Kernel → Linux Kernel Tools**.

Чтобы собирать пакеты с отладочными символами, не вырезая их при установке на целевую платформу, задайте еще два параметра:

- BR2_ENABLE_DEBUG в меню **Build options → build packages with debugging symbols**;
- BR2_STRIP = none в меню **Build options → strip command for binaries on target**.

Затем выполните команды make clean и make.

После того как все будет собрано, нужно вручную скопировать файл vmlinux в образ целевой системы.

Профилирование с помощью perf

Мы можем использовать perf для построения выборки состояний программы в терминах какого-то счетчика событий за период времени. Это еще один пример

статистического профилирования. По умолчанию подразумевается счетчик событий `cycles` – обобщенный аппаратный счетчик, отображаемый на регистр PMU, который представляет число тактов процессорного ядра.

Процесс создания профиля с применением `perf` состоит из двух этапов. На первом этапе команда `perf record` строит выборку и записывает ее в файл `perf.data` (по умолчанию), а на втором – `perf report` – анализирует результаты. Обе команды запускаются в целевой системе. Из выборки оставляются только примеры для процесса, соответствующего указанной вами команде, и его потомков. Ниже приведен пример профилирования скрипта оболочки, который ищет строку «linux»:

```
# perf record sh -c "find /usr/share | xargs grep linux > /dev/null"
[ perf record: Woken up 2 times to write data ]
[ perf record: Captured and wrote 0.368 MB perf.data (~16057 samples) ]
# ls -l perf.data
-rw------- 1 root root 387360 Aug 25 2015 perf.data
```

Теперь можно показать собранные в `perf.data` результаты с помощью команды `perf report`. Для нее имеются три пользовательских интерфейса, в командной строке можно указать любой из них:

- `--stdio`: текстовый интерфейс без возможности взаимодействия с пользователем;
- `--tui`: простой интерактивный текстовой интерфейс с возможностью перелистывания страниц;
- `--gtk`: графический интерфейс, который работает так же, `--tui`.

По умолчанию подразумевается интерфейс TUI, показанный на рисунке ниже.

```
Samples: 9K of event 'cycles', Event count (approx.): 2006177260
 11.29%  grep  libc-2.20.so        [.] re_search_internal
  8.80%  grep  busybox.nosuid      [.] bb_get_chunk_from_file
  5.55%  grep  libc-2.20.so        [.] _int_malloc
  5.40%  grep  libc-2.20.so        [.] _int_free
  3.74%  grep  libc-2.20.so        [.] realloc
  2.59%  grep  libc-2.20.so        [.] malloc
  2.51%  grep  libc-2.20.so        [.] regexec@@GLIBC_2.4
  1.64%  grep  busybox.nosuid      [.] grep_file
  1.57%  grep  libc-2.20.so        [.] malloc_consolidate
  1.33%  grep  libc-2.20.so        [.] strlen
  1.33%  grep  libc-2.20.so        [.] memset
  1.26%  grep  [kernel.kallsyms]   [k] __copy_to_user_std
  1.20%  grep  libc-2.20.so        [.] free
  1.10%  grep  libc-2.20.so        [.] _int_realloc
  0.95%  grep  libc-2.20.so        [.] re_string_reconstruct
  0.79%  grep  busybox.nosuid      [.] xrealloc
  0.75%  grep  [kernel.kallsyms]   [k] __do_softirq
  0.72%  grep  [kernel.kallsyms]   [k] preempt_count_sub
  0.68%  find  [kernel.kallsyms]   [k] __do_softirq
  0.53%  grep  [kernel.kallsyms]   [k] __dev_queue_xmit
  0.52%  grep  [kernel.kallsyms]   [k] preempt_count_add
  0.47%  grep  [kernel.kallsyms]   [k] finish_task_switch.isra.85
Press '?' for help on key bindings
```

`perf` может протоколировать время работы функций ядра, выполняемых от имени процессов, поскольку отбирает примеры и в пространстве ядра тоже.

Список упорядочен по времени, проведенному в функции, в порядке убывания. В данном случае все примеры, кроме одного, отобраны, когда работала программа grep. Одни примеры попали в библиотеку libc-2.20, другие – в программу busybox.nosuid, третьи – в ядро. Показаны имена символов, соответствующие функциям из программы и библиотеки, поскольку все двоичные исполняемые файлы в целевой системе содержат отладочную информацию, а символы ядра читаются из файла /boot/vmlinux. Если файл vmlinux находится в другом месте, добавьте флаг -k <path> в командную строку perf report. Выборка может храниться не в файле perf.data, а в любом другом файле, нужно только указать его имя в команде perf record -o <имя файла> и в команде perf report -i <имя файла>.

По умолчанию perf record отбирает примеры с частотой 1000 Гц, используя счетчик тактов.

Частота отбора 1000 Гц, возможно, выше, чем вам необходимо, и может стать причиной эффекта наблюдателя. Попробуйте уменьшить частоту: по моему опыту в большинстве случаев достаточно 100 Гц. Частота задается с помощью флага -F.

Графы вызовов

Но мы еще не решили всех проблем: первые функции в списке – по большей части низкоуровневые операции с памятью, и с большой долей уверенности можно утверждать, что они уже оптимизированы. Хорошо бы посмотреть, откуда эти функции вызываются. Это можно сделать, если запоминать для каждого отобранного примера трассу вызовов, что позволяет сделать флаг -g команды perf record.

Теперь perf report отображает знак + там, где функция является частью цепочки вызовов. Трассу можно раскрыть и посмотреть функции под ней:

```
Samples: 10K of event 'cycles', Event count (approx.): 2256721655
-   9.95%  grep  libc-2.20.so        [.] re_search_internal
   - re_search_internal
        95.96% 0
        3.50% 0x208
+   8.19%  grep  busybox.nosuid      [.] bb_get_chunk_from_file
+   5.07%  grep  libc-2.20.so        [.] _int_free
+   4.76%  grep  libc-2.20.so        [.] _int_malloc
+   3.75%  grep  libc-2.20.so        [.] realloc
+   2.63%  grep  libc-2.20.so        [.] malloc
+   2.04%  grep  libc-2.20.so        [.] regexec@@GLIBC_2.4
+   1.43%  grep  busybox.nosuid      [.] grep_file
+   1.37%  grep  libc-2.20.so        [.] memset
+   1.29%  grep  libc-2.20.so        [.] malloc_consolidate
+   1.22%  grep  libc-2.20.so        [.] _int_realloc
+   1.15%  grep  libc-2.20.so        [.] free
+   1.01%  grep  [kernel.kallsyms]   [k] __copy_to_user_std
+   0.98%  grep  libc-2.20.so        [.] strlen
+   0.89%  grep  libc-2.20.so        [.] re_string_reconstruct
+   0.73%  grep  [kernel.kallsyms]   [k] preempt_count_sub
+   0.68%  grep  [kernel.kallsyms]   [k] finish_task_switch.isra.85
+   0.62%  grep  busybox.nosuid      [.] xrealloc
+   0.57%  grep  [kernel.kallsyms]   [k] __do_softirq
Press '?' for help on key bindings
```

Для генерации графа вызовов необходимы кадры вызова в стеке – как и для генерации трассы вызовов в GDB. Данные, необходимые для раскрутки стека, закодированы в составе отладочной информации в исполняемом файле, но имеются не в любой комбинации архитектуры и набора инструментов.

perf annotate

Зная, на какие функции обращать внимание, было бы хорошо зайти внутрь и получить счетчики выполнения каждой команды. Именно это и делает команда `perf annotate`, которая вызывает программу `objdump`, установленную в целевой системе. Нужно только выполнить `perf annotate` вместо `perf report`.

Для работы `perf annotate` необходимы таблицы символов для исполняемых файлов и `vmlinux`. Ниже показан пример аннотированной функции:

```
re_search_internal  /lib/libc-2.20.so
       │          cmp    r1,
       │          beq    c362c <gai_strerror+0xcaf8>
       │          str    r3, [fp, #-40]         ; 0x28
       │          b      c3684 <gai_strerror+0xcb50>
  0.65 │          ldr    ip, [fp, #-256]        ; 0x100
  0.16 │          ldr    r0, [fp, #-268]        ; 0x10c
  2.44 │          add    r3,
  4.15 │          cmp    r0,
  3.91 │          strle  r3, [fp, #-40]         ; 0x28
       │          ble    c3684 <gai_strerror+0xcb50>
  4.72 │          ldrb   r1, [r2, #1]!
 10.26 │          ldrb   r1, [ip, r1]
  6.68 │          cmp    r1,
       │          beq    c3660 <gai_strerror+0xcb2c>
  0.90 │          str    r3, [fp, #-40]         ; 0x28
  2.12 │          ldr    r3, [fp, #-40]         ; 0x28
  0.08 │          ldr    r2, [fp, #-268]        ; 0x10c
  0.33 │          cmp    r2,
       │          bne    c3804 <gai_strerror+0xccd0>
  0.08 │          mov    r3,
       │          ldr    r2, [fp, #-280]        ; 0x118
  0.08 │          cmp    r3,
Press 'h' for help on key bindings
```

Чтобы видеть не только ассемблерный, но и исходный код, нужно скопировать соответствующие файлы на целевое устройство. Если вы пользуетесь Yocto Project и собирали образ с опцией `dbg-pkgs` или установили отдельный пакет `-dbg`, то исходный код будет находиться в каталоге `/usr/src/debug`. В противном случае можете узнать ожидаемое местоположение исходного кода, прочитав отладочную информацию:

```
$ arm-buildroot-linux-gnueabi-objdump --dwarf lib/libc-2.19.so | grep DW_AT_comp_dir
  <3f> DW_AT_comp_dir : /home/chris/buildroot/output/build/host-gcc-initial-4.8.3/build/
arm-buildroot-linux-gnueabi/libgcc
```

Путь в целевой системе должен быть точно таким, как указано в `DW_AT_comp_dir`. Ниже показан пример аннотации с исходным и ассемблерным кодом:

```
re_search_internal  /lib/libc-2.20.so
                          ++match_first;
                        goto forward_match_found_start_or_reached_end;

                      case 6:
                        /* Fastmap without translation, match forward.  */
                        while (BE (match_first < right_lim, 1)
  4.15          cmp    r0,
  3.91          strle  r3, [fp, #-40]        ; 0x28
                ble    c3684 <gai_strerror+0xcb50>
                               && !fastmap[(unsigned char) string[match_first]])
  4.72          ldrb   r1, [r2, #1]!
 10.26          ldrb   r1, [ip, r1]
  6.68          cmp    r1,
                beq    c3660 <gai_strerror+0xcb2c>
  0.90          str    r3, [fp, #-40]        ; 0x28
                          ++match_first;

                      forward_match_found_start_or_reached_end:
                        if (BE (match_first == right_lim, 0))
  2.12          ldr    r3, [fp, #-40]        ; 0x28
  0.08          ldr    r2, [fp, #-268]        ; 0x10c
  0.33          cmp    r2,
Press 'h' for help on key bindings
```

Другие профилировщики: OProfile и gprof

Эти два статистических профилировщика – предшественники `perf`. Тот и другой реализуют лишь подмножество функциональности `perf`, но все еще весьма популярны. Я опишу их, не вдаваясь в подробности.

OProfile – профилировщик ядра, его разработка началась в 2002 году. Первоначально он включал собственный код отбора примеров из ядра, но в последних версиях для этого используется инфраструктура `perf_events`. Дополнительная информация имеется на сайте http://oprofile.sourceforge.net. OProfile состоит из компоненты, работающей в пространстве ядра, демона в пользовательском пространстве и команд анализа.

Для работы OProfile нужно включить два конфигурационных параметра ядра:

- `CONFIG_PROFILING` в меню **General setup → Profiling support**;
- `CONFIG_OPROFILE` в меню **General setup → OProfile system profiling**.

При сборке с помощью Yocto Project компоненты, работающие в пользовательском пространстве, устанавливаются, если в образ включен пакет `tools-profile`. При работе с Buildroot пакет включается, если задан параметр `BR2_PACKAGE_OPROFILE`.

Для отбора примеров служит команда

```
# operf <program>
```

Дождитесь естественного завершения приложения или нажмите **Ctrl+C**, чтобы прекратить профилирование. Собранные данные хранятся в файле `<текущий-каталог>/oprofile_data/samples/current`.

Для генерации профиля используется команда `opreport`. Ее флаги документированы в руководстве по OProfile.

gprof – часть набора инструментов GNU, один из самых первых профилировщиков с открытым исходным кодом. Сочетает в себе оснащение на этапе компиляции и технику отбора примеров с частотой 100 Гц. К преимуществам можно отнести тот факт, что программа не нуждается в поддержке со стороны ядра.

Для подготовки к профилированию программу следует откомпилировать и скомпоновать с флагом -pg, в результате чего в преамбулу каждой функции вставляется код сбора информации о дереве вызовов. Во время работы программы отобранные примеры сохраняются в буфере, который записывается в файл gmon.out при завершении программы.

Для чтения примеров из gmon.out и отладочной информации из файла программы используется команда gprof.

Так, чтобы произвести профилирование аплета grep из комплекта BusyBox grep, нужно пересобрать BusyBox с флагом -pg, выполнить команду и убедиться, что выходной файл сформирован:

```
# busybox grep "linux" *
# ls -l gmon.out
-rw-r--r-- 1 root root 473 Nov 24 14:07 gmon.out
```

Затем выборку можно проанализировать в целевой или исходной системе:

```
# gprof busybox
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

% cumulative   self              self     total
time  seconds  seconds   calls  Ts/call  Ts/call  name
0.00    0.00     0.00     688    0.00     0.00    xrealloc
0.00    0.00     0.00     345    0.00     0.00    bb_get_chunk_from_file
0.00    0.00     0.00     345    0.00     0.00    xmalloc_fgetline
0.00    0.00     0.00       6    0.00     0.00    fclose_if_not_stdin
0.00    0.00     0.00       6    0.00     0.00    fopen_for_read
0.00    0.00     0.00       6    0.00     0.00    grep_file
[...]
  Call graph

granularity: each sample hit covers 2 byte(s) no time propagated

index  % time   self   children   called     name
                0.00   0.00       688/688    bb_get_chunk_from_file [2]
[1]      0.0    0.00   0.00       688        xrealloc [1]
----------------------------------------------------
                0.00   0.00       345/345    xmalloc_fgetline [3]
[2]      0.0    0.00   0.00       345        bb_get_chunk_from_file [2]
                0.00   0.00       688/688    xrealloc [1]
---------------------------------------------------
                0.00   0.00       345/345    grep_file [6]
[3]      0.0    0.00   0.00       345        xmalloc_fgetline [3]
```

```
                0.00   0.00       345/345     bb_get_chunk_from_file [2]
-------------------------------------------------------
                0.00   0.00       6/6         grep_main [12]
[4]     0.0     0.00   0.00       6           fclose_if_not_stdin [4]
[...]
```

Отметим, что время выполнения во всех случаях равно 0, потому что большая его часть приходится на системные вызовы, а их gprof не трассирует.

gprof умеет отбирать примеры только из главного потока многопоточной программы и не заходит в пространство ядра. То и другое ограничивает полезность программы.

Трассировка событий

До сих пор мы рассматривали лишь статистические выборки. Но часто хочется знать больше о порядке возникновения событий, чтобы можно было их просмотреть и соотнести друг с другом. Для трассировки функций необходимо оснастить код точками трассировки, в которых собирается информация о событиях, например:

- временная метка;
- контекст, скажем текущий PID;
- параметры и возвращенное значение функции;
- стек вызовов.

Эта техника в большей степени влияет на работу программы, чем статистическое профилирование, и в результате может порождаться большой объем данных. Последнюю проблему можно смягчить, применив фильтры на этапе сбора данных и позже, при просмотре трассы.

Я расскажу о двух инструментах трассировки функций ядра: Ftrace и LTTng.

Введение в Ftrace

Трассировщик функций ядра, Ftrace, уходит корнями в работу Стивена Ростедта (Steven Rostedt) и многих других, начатую в поисках причин больших задержек. Ftrace вошел в версию Linux 2.6.27 и с тех пор активно развивается. В каталоге Documentation/trace в исходном коде ядра есть ряд документов, описывающих трассировку ядра.

Ftrace состоит из нескольких трассировщиков, каждый из которых умеет протоколировать различные виды операций. Я расскажу о трассировщиках function и function_graph, а также о точках трассировки событий. В главе 14 мы вернемся к Ftrace и воспользуемся им для показа задержек реального времени.

Трассировщик function оснащает каждую функцию ядра, так чтобы можно было запомнить все вызовы, снабдив их временными метками. Кстати говоря, для оснащения средствами измерения ядро компилируется с флагом -pg, но на этом сходство с gprof и заканчивается. Трассировщик function_graph идет дальше и запоминает точки входа и выхода из функций, чтобы можно было построить граф

вызовов. При включении аппарата точек трассировки запоминаются также параметры каждого вызова.

У `Ftrace` очень удобный для встраиваемых систем пользовательский интерфейс, полностью реализованный посредством виртуальных файлов в файловой системе `debugfs`. Это означает, что на целевое устройство не нужно устанавливать никаких дополнительных инструментов. Но для желающих есть и другие интерфейсы. Командный инструмент `trace-cmd`, который запоминает и показывает трассы, доступен как в Buildroot (`BR2_PACKAGE_TRACE_CMD`), так и в Yocto Project (`trace-cmd`). Существует также графическое средство просмотра трасс KernelShark, доступное в виде пакета для Yocto Project.

Подготовка к работе с Ftrace

Различные возможности `Ftrace` задаются с помощью конфигурационных параметров ядра. Как минимум, нужно включить параметр

- `CONFIG_FUNCTION_TRACER` в меню **Kernel hacking → Tracers → Kernel Function Tracer**.

По причинам, которые станут ясны позже, рекомендуется включить также параметры:

- `CONFIG_FUNCTION_GRAPH_TRACER` в меню **Kernel hacking → Tracers → Kernel Function Graph Tracer**;
- `CONFIG_DYNAMIC_FTRACE` в меню **Kernel hacking → Tracers → enable/disable function tracing dynamically**.

Поскольку все происходит внутри ядра, конфигурация в пользовательском пространстве отсутствует.

Прежде чем запускать `Ftrace`, необходимо смонтировать файловую систему `debugfs` – по соглашению, на каталог `/sys/kernel/debug`:

```
# mount -t debugfs none /sys/kernel/debug
```

Все элементы управления `Ftrace` находятся в каталоге `/sys/kernel/debug/tracing`; там даже есть мини-`HOWTO` в файле `README`.

Вот список трассировщиков, имеющихся в ядре:

```
# cat /sys/kernel/debug/tracing/available_tracers
blk function_graph function nop
```

Имя активного трассировщика находится в файле `current_tracer`; в начальный момент это ничего не делающий трассировщик с именем `nop`.

Чтобы получить трассу, выберите трассировщик, записав имя одного из перечисленных в `available_tracers` в файл `current_tracer`, а затем некоторое время – но недолго – что-то поделайте:

```
# echo function > /sys/kernel/debug/tracing/current_tracer
# echo 1 > /sys/kernel/debug/tracing/tracing_on
# sleep 1
# echo 0 > /sys/kernel/debug/tracing/tracing_on
```

За прошедшую секунду в буфер трассировки будут записаны подробные сведения о каждой вызывавшейся функции ядра. Формат буфера текстовый, он описан в файле Documentation/trace/ftrace.txt. Прочитать содержимое буфера можно из файла trace:

```
# cat /sys/kernel/debug/tracing/trace
# tracer: function
#
# entries-in-buffer/entries-written: 40051/40051 #P:1
#
#                          _-----=> irqs-off
#                         / _----=> need-resched
#                        | / _---=> hardirq/softirq
#                        || / _--=> preempt-depth
#                        ||| /     delay
#      TASK-PID    CPU#  ||||   TIMESTAMP FUNCTION
#         | |        |   ||||      |         |
         sh-361    [000] ...1  992.990646: mutex_unlock <- rb_simple_write
         sh-361    [000] ...1  992.990658: __fsnotify_parent <-vfs_write
         sh-361    [000] ...1  992.990661: fsnotify <- vfs_write
         sh-361    [000] ...1  992.990663: __srcu_read_lock <-fsnotify
         sh-361    [000] ...1  992.990666: preempt_count_add <- __srcu_read_lock
         sh-361    [000] ...2  992.990668: preempt_count_sub <- __srcu_read_lock
         sh-361    [000] ...1  992.990670: __srcu_read_unlock <-fsnotify
         sh-361    [000] ...1  992.990672: __sb_end_write <- vfs_write
         sh-361    [000] ...1  992.990674: preempt_count_add <-__sb_end_write
[...]
```

За одну секунду можно собрать очень много данных.

Как и в случае профилировщиков, извлечь что-то полезное из такого плоского списка вызовов функций трудно. Если выбрать трассировщик function_graph, то Ftrace построит такие графы:

```
# tracer: function_graph
#
# CPU  DURATION                FUNCTION CALLS
#|     |   |                   |   |   |   |
 0) + 63.167 us   |                 } /* cpdma_ctlr_int_ctrl */
 0) + 73.417 us   |               } /* cpsw_intr_disable */
 0)               |               disable_irq_nosync() {
 0)               |                 __disable_irq_nosync() {
 0)               |                   __irq_get_desc_lock() {
 0)   0.541 us    |                     irq_to_desc();
 0)   0.500 us    |                     preempt_count_add();
 0) + 16.000 us   |                   }
 0)               |                   __disable_irq() {
 0)   0.500 us    |                     irq_disable();
 0)   8.208 us    |                   }
 0)               |                   __irq_put_desc_unlock() {
```

```
 0)   0.459 us    |                preempt_count_sub();
 0)   8.000 us    |              }
 0) + 55.625 us    |            }
 0) + 63.375 us    |          }
```

Теперь видна вложенность вызовов функций, обозначенная фигурными скобками. Напротив закрывающей скобки указано время, проведенное в функции, аннотированное знаком +, если оно больше 10 мкс, и восклицательным знаком, если больше 100 мкс.

Часто нас интересуют только действия ядра, вызванные одним процессом или потоком. В таком случае трассу можно ограничить, записав в файл `set_ftrace_pid` идентификатор потока.

Динамический режим Ftrace и фильтры трассировки

Включение параметра `CONFIG_DYNAMIC_FTRACE` позволяет Ftrace модифицировать места вызова функции `trace` во время выполнения, что дает два преимущества. Во-первых, на этапе сборки команды вызова `trace` подвергаются дополнительной обработке, что позволяет подсистеме Ftrace находить их на этапе загрузки и заменять командами NOP, сводя накладные расходы на код трассировки почти к нулевым. Затем мы можем включить Ftrace в производственном или почти производственном ядре, не оказав ни малейшего влияния не производительность.

Второе преимущество заключается в том, что можно избирательно включать некоторые места вызова `trace`, а не трассировать все подряд. Список функций хранится в файле `available_filter_functions`; всего их несколько десятков тысяч. Чтобы включить трассировку отдельной функции, нужно скопировать ее имя из `available_filter_functions` в `set_ftrace_filter`, а когда в трассировке отпадет надобность, записать имя функции в файл `set_ftrace_notrace`. Можно использовать метасимволы и дописывать имена в конец списка. Допустим, нас интересует обработка `tcp`:

```
# cd /sys/kernel/debug/tracing
# echo "tcp*" > set_ftrace_filter
# echo function > current_tracer
# echo 1 > tracing_on
```

Прогоним несколько тестов, а затем посмотрим на трассу:

```
# cat trace
# tracer: function
#
# entries-in-buffer/entries-written: 590/590 #P:1
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#         TASK-PID     CPU# ||||    TIMESTAMP FUNCTION
```

```
#             | |         |  ||||       |         |
  dropbear-375    [000] ...1 48545.022235: tcp_poll <-sock_poll
  dropbear-375    [000] ...1 48545.022372: tcp_poll <-sock_poll
  dropbear-375    [000] ...1 48545.022393: tcp_sendmsg <- inet_sendmsg
  dropbear-375    [000] ...1 48545.022398: tcp_send_mss <- tcp_sendmsg
  dropbear-375    [000] ...1 48545.022400: tcp_current_mss <- tcp_send_mss
[...]
```

Файл set_ftrace_filter может содержать также команды, например, для запуска и остановки трассировки при выполнении определенных функций. Здесь мы не можем вдаваться в такие детали, но желающие могут почитать раздел **Filter commands** в файле Documentation/trace/ftrace.txt.

События трассировки

Трассировщики function и function_graph, описанные в предыдущем разделе, запоминают только время, когда была вызвана функция. В событиях трассировки запоминаются также параметры вызова, что повышает удобочитаемость и информативность трассы. Так, вместо фиксации самого факта вызова kmalloc в событии трассировки будут запомнены количество запрошенных байтов и возвращенный указатель. События трассировки используются не только в Ftrace, но и в perf и LTTng, но разработка этой подсистемы началась с проекта LTTng.

Разработчикам ядра приходится прилагать усилия для создания событий трассировки, потому что все они разные. Их определения хранятся в исходном коде в виде макросов TRACE_EVENT; на данный момент их больше тысячи. На этапе выполнения список событий доступен в файле /sys/kernel/debug/tracing/available_events. Имена событий имеют вид подсистема:функция, например kmem:kmalloc. Каждое событие представлено также подкаталогом tracing/events/[подсистема]/[функция], например:

```
# ls events/kmem/kmalloc
enable   filter   format   id   trigger
```

Файлы имеют следующий смысл:

- enable: в этот файл записывается 1, чтобы разрешить событие;
- filter: если содержащееся в файле выражение равно true, то событие трассируется;
- format: формат события и параметров;
- id: числовой идентификатор;
- trigger: команда, которая исполняется при возникновении события, записывается с помощью синтаксиса, определенного в разделе **Filter commands** файла Documentation/trace/ftrace.txt. Я приведу простой пример, включающий функции kmalloc и kfree.

Трассировка событий не зависит от трассировщиков функций, поэтому для начала выберем трассировщик nop:

```
# echo nop > current_tracer
```

Затем укажем, какие события трассировать, каждое по отдельности:

```
# echo 1 > events/kmem/kmalloc/enable
# echo 1 > events/kmem/kfree/enable
```

Можно также записать имена событий в файл `set_event`:

```
# echo "kmem:kmalloc kmem:kfree" > set_event
```

Если теперь прочитать файл `trace`, то мы увидим функции вместе с параметрами:

```
# tracer: nop
#
# entries-in-buffer/entries-written: 359/359 #P:1
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#       TASK-PID    CPU#  ||||   TIMESTAMP FUNCTION
#          | |       |    ||||      |         |
         cat-382    [000] ...1 2935.586706: kmalloc:
call_site=c0554644 ptr=de515a00 bytes_req=384 bytes_alloc=512
gfp_flags=GFP_ATOMIC|GFP_NOWARN|GFP_NOMEMALLOC
         cat-382 [000] ...1 2935.586718: kfree:
call_site=c059c2d8 ptr= (null)
```

Точно те же события трассировки видны в perf как события точек трассировки.

Использование LTTng

Проект Linux Trace Toolkit, начатый Каримом Ягмуром (Karim Yaghmour), преследовал цель трассировать действия ядра, это был один из первых общедоступных инструментов трассировки для ядра Linux. Впоследствии Матье Денуайе (Mathieu Desnoyers) подхватил идею и заново реализовал ее в трассировщике следующего поколения LTTng. Затем проект был распространен также на трассировку в пользовательском пространстве. На сайте проекта по адресу http://lttng.org/ имеется подробное руководство пользователя.

LTTng состоит из трех компонентов:

- базовый диспетчер сеансов;
- трассировщик ядра, реализованный в виде группы модулей ядра;
- трассировщик пользовательского пространства, реализованный в виде библиотеки.

Кроме того, понадобится средство просмотра трасс, например Babeltrace (http://www.efficios.com/babeltrace) или подключаемый к Eclipse модуль `Trace Compaas`, чтобы просматривать и фильтровать собранные данные в исходной или целевой системе.

Для работы LTTng ядро должно быть сконфигурировано с параметром CONFIG_TRACEPOINTS, который представлен пунктом меню **Kernel hacking → Tracers → Kernel Function Tracer**.

Приведенное ниже описание относится к версии LTTng 2.5; другие версии могут отличаться.

LTTng и Yocto Project

В состав зависимостей для целевой системы, например в файл conf/local.conf, нужно добавить следующие пакеты:

```
IMAGE_INSTALL_append = " lttng-tools lttng-modules lttng-ust"
```

Чтобы запускать в целевой системе Babeltrace, добавьте также пакет babeltrace.

LTTng и Buildroot

Необходимо включить следующие параметры:

- BR2_PACKAGE_LTTNG_MODULES в меню **Target packages → Debugging, profiling and benchmark → lttng-modules**;
- BR2_PACKAGE_LTTNG_TOOLS в меню **Target packages → Debugging, profiling and benchmark → lttng-tools**.

Для трассировки в пользовательском пространстве включите еще параметр

- BR2_PACKAGE_LTTNG_LIBUST в меню **Target packages → Libraries → Other → lttng-libust**.

Существует пакет lttng-babletrace для целевой платформы. Buildroot собирает babeltrace для исходной платформы автоматически и помещает его в файл output/host/usr/bin/babeltrace.

Применение LTTng для трассировки ядра

LTTng может использовать описанные выше события ftrace как потенциальные точки трассировки. В начальный момент они выключены.

Управляющим интерфейсом для LTTng служит команда lttng. Чтобы перечислить точки трассировки в ядре, выполните команду:

```
# lttng list --kernel
Kernel events:
-------------
     writeback_nothread (loglevel: TRACE_EMERG (0)) (type: tracepoint)
     writeback_queue (loglevel: TRACE_EMERG (0)) (type: tracepoint)
     writeback_exec (loglevel: TRACE_EMERG (0)) (type: tracepoint)
[...]
```

Трассировочные данные собираются в контексте сеанса, который в примере ниже называется test:

```
# lttng create test
Session test created.
Traces will be written in /home/root/lttng-traces/test-20150824-140942
# lttng list
```

```
Available tracing sessions:
  1) test (/home/root/lttng-traces/test-20150824-140942) [inactive]
```

Теперь разрешим несколько событий в текущем сеансе. Можно разрешить все точки трассировки ядра, указав флаг `-all`, но помните об опасности собрать слишком много данных. Начнем с двух событий трассировки, относящихся к планировщику:

```
# lttng enable-event --kernel sched_switch,sched_process_fork
```

Проверим, что все настроено правильно:

```
# lttng list test
Tracing session test: [inactive]
    Trace path: /home/root/lttng-traces/test-20150824-140942
    Live timer interval (usec): 0

=== Domain: Kernel ===

Channels:
-------------
- channel0: [enabled]

   Attributes:
     overwrite mode: 0
     subbufers size: 26214
     number of subbufers: 4
     switch timer interval: 0
     read timer interval: 200000
     trace file count: 0
     trace file size (bytes): 0
     output: splice()

   Events:
     sched_process_fork (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
     sched_switch (loglevel: TRACE_EMERG (0)) (type: tracepoint) [enabled]
```

Теперь начнем трассировку:

```
# lttng start
```

Прогоним тестовую нагрузку и остановим трассировку:

```
# lttng stop
```

Трассы, сгенерированные в сеансе, записываются в каталог сеанса `lttng-traces/<session>/ kernel`.

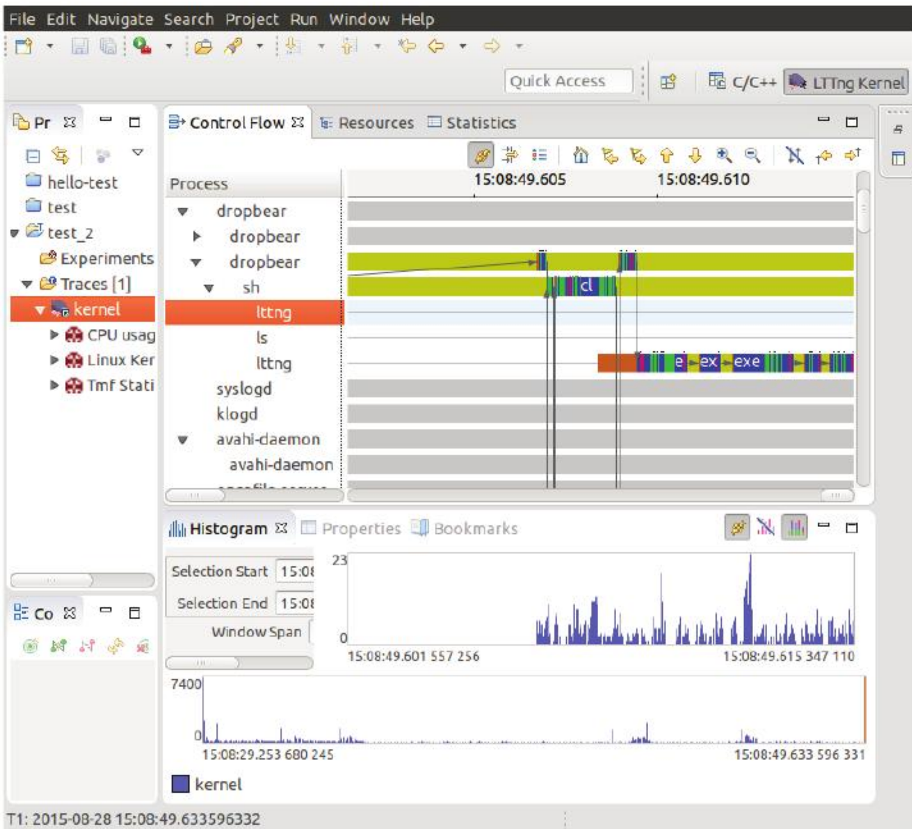
Мы можем воспользоваться средством просмотра Babeltrace, чтобы вывести собранные данные в текстовом формате. В данном случае я запустил его на исходном компьютере:

```
$ babeltrace lttng-traces/test-20150824-140942/kernel
```

Данных слишком много, на одной странице они не поместятся, поэтому оставлю это в качестве упражнения для читателя. Текст, который выводит Babeltrace, хорош тем, что в нем легко искать строки с помощью `grep` и других подобных команд.

Неплохой графической альтернативой является подключаемый к Eclipse модуль Trace Compass, входящий в состав пакета Eclipse IDE для разработчиков на C/C++. Импортировать трассировочные данные в Eclipse довольно канительно. Вот вкратце перечень необходимых шагов.

1. Откройте перспективу трассировки.
2. Создайте новый проект, выполнив команду **File → New → Tracing project**.
3. Введите имя проекта и нажмите кнопку **Finish**.
4. Щелкните правой кнопкой мыши по пункту **New Project** в меню **Project Explorer** и выберите команду **Import**.
5. Раскройте список **Tracing** и выберите **Trace Import**.
6. Перейдите в каталог, содержащий трассы (например, `test-20150824-140942`), отметьте флажками интересующие вас подкаталоги (среди них может быть kernel) и нажмите кнопку **Finish**.
7. Раскройте проект, а внутри него ветвь **Traces[1]** и в ней дважды щелкните по элементу **kernel**.
8. Экран должен выглядеть так, как показано на рисунке ниже.



На этом рисунке я приблизил представление «Control Flow», чтобы показать переходы состояний между dropbear и оболочкой, а также активность демона lttng.

Использование Valgrind для профилирования приложений

В главе 11 я представил Valgrind как средство для выявления ошибок работы с памятью – для этого использовался инструмент memcheck. Но в Valgrind есть также полезные инструменты для профилирования приложений. Здесь я рассмотрю два из них: **Callgrind** и **Helgrind**. Поскольку Valgrind исполняет код в песочнице, она может наблюдать за его поведением и формировать такие отчеты, которые недоступны обычным трассировщикам и профилировщикам.

Callgrind

Callgrind – это профилировщик, который умеет строить графы вызовов, а также собирает информацию о частоте попаданий в кэш процессора и о прогнозировании ветвлений. Callgrind полезен только в том случае, когда узким местом является процессор. Он ничем не поможет, если программа ограничена скоростью ввода-вывода или включает несколько процессов.

Для Valgrind не требуется конфигурировать ядро, но нужны отладочные символы. Соответствующий пакет имеется как в Yocto Project, так и в Buildroot (BR2_PACKAGE_VALGRIND).

Для запуска инструмента Callgrind на целевом устройстве выполните команду:

```
# valgrind --tool=callgrind <program>
```

В результате будет создан файл callgrind.out.<PID>, который можно скопировать на исходный компьютер и проанализировать с помощью команды callgrind_annotate.

По умолчанию данные обо всех потоках помещаются в один файл. Если указать флаг --separate-threads=yes на этапе сбора данных, то будут созданы профили каждого потока в файлах вида callgrind.out.<PID>-<thread id>, например: callgrind.out.122-01, callgrind.out.122-02 и т. д.

Callgrind может эмулировать кэш процессора уровня L1/L2 и формировать отчет о непопаданиях в кэш. Для этого нужно задать флаг --simulate-cache=yes. Непопадание в кэш L2 обходится гораздо дороже, чем непопадание в кэш L1, поэтому обращайте особое внимание на большие значения счетчиков D2mr и D2mw.

Helgrind

Это детектор ошибок синхронизации потоков в программах на C, C++ и Fortran, в которых используются потоки POSIX.

Helgrind способен обнаруживать ошибки трех типов. Во-первых, некорректное использование API. Например, освобождение уже освобожденного мьютекса, освобождение мьютекса, захваченного в другом потоке, отсутствие проверки значения, возвращенного некоторыми Pthread-функциями. Во-вторых, Helgrind следит за порядком захвата мьютексов в потоках и, следовательно, обнаруживает потенциальные взаимоблокировки, возникающие из-за образования циклов захвата. Наконец, обнаруживаются состояния гонки за данные, которые могут возникать, когда два потока обращаются к разделяемой памяти, не выполнив надлежащей синхронизации, которая гарантировала бы доступ только из одного потока.

Для использования Helgrind нужно лишь выполнить такую команду:

```
# valgrind --tool=helgrind <program>
```

Она выведет отчет о реальных и потенциальных проблемах. Отчет можно направить в файл, добавив флаг `--log-file=<имя файла>`.

Использование strace для показа системных вызовов

Я начал эту главу с описания простого и вездесущего инструмента `top`, а закончу рассказом о не менее популярной программе `strace`. Это очень простой трассировщик, который показывает, какие системные вызовы имели место в программе и, возможно, в порожденных ей процессах. Он удобен для следующих целей:

- чтобы узнать, какие системные вызовы выполняет программа;
- чтобы найти неудачные системные вызовы и возвращенные ими коды ошибки. Это бывает полезно, когда программа не запускается и ничего не печатает или печатает слишком общее сообщение об ошибке; `strace` покажет, какой системный вызов завершился с ошибкой;
- чтобы узнать, какие файлы открывает программа;
- чтобы узнать, какие системные вызовы совершает работающая программа, например понять, не зациклилась ли она.

В сети есть много примеров, поищите фразу «strace tips and tricks». У каждого найдется своя любимая история, смотрите, например, статью по адресу http://chadfowler.com/blog/2014/01/26/the-magic-of-strace.

В `strace` используется функция `ptrace(2)`, которая перехватывает вызовы ядра из пользовательского пространства. Чтобы узнать, как работает `ptrace`, загляните в страницу руководства, она написана подробно и на удивление понятно.

Простейший способ получить трассу – запустить программу из-под `strace`, как показано ниже (для большей понятности листинг слегка отредактирован):

```
# strace ./helloworld
execve("./helloworld", ["./helloworld"], [/* 14 vars */]) = 0
brk(0)                                  = 0x11000
uname({sys="Linux", node="beaglebone", ...}) = 0
```

```
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, - 1, 0)
= 0xb6f40000
access("/etc/ld.so.preload", R_OK)     = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=8100, ...}) = 0
mmap2(NULL, 8100, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb6f3e000
close(3)                               = 0
open("/lib/tls/v7l/neon/vfp/libc.so.6", O_RDONLY|O_CLOEXEC) = -1
ENOENT (No such file or directory)
[...]
open("/lib/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0$`\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1291884, ...}) = 0
mmap2(NULL, 1328520, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) =
0xb6df9000
mprotect(0xb6f30000, 32768, PROT_NONE) = 0
mmap2(0xb6f38000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENY-
WRITE, 3, 0x137000) = 0xb6f38000
mmap2(0xb6f3b000, 9608, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONY-
MOUS, -1, 0) = 0xb6f3b000
close(3)
[...]
write(1, "Hello, world!\n", 14Hello, world!
)          = 14
exit_group(0)                          = ?
+++ exited with 0 +++
```

Большая часть трассы связана с созданием среды выполнения. В частности, мы видим, как загрузчик ищет библиотеку `libc.so.6` и в конце концов находит ее в каталоге `/lib`. Затем программа доходит до функции `main()`, которая печатает сообщение и завершает программу.

Чтобы `strace` прослеживала дочерние процессы или потоки, созданные исходным процессом, добавьте флаг `-f`.

Если вы используете `strace` для трассировки программы, создающей потоки, то почти наверняка захотите добавить флаг `-f`. А еще лучше – укажите флаги `-ff` и `o <filename>`, чтобы результаты для каждого дочернего процесса или потока записывались в отдельный файл с именем вида `<filename>.<PID | TID>`.

Очень часто `strace` используется для того, чтобы узнать, какие файлы программа открывает сразу после запуска. Можно ограничить набор трассируемых системных вызовов, указав флаг `-e`, и выводить трассу не на `stdout`, а в файл, задав флаг `-o`:

```
# strace -e open -o ssh-strace.txt ssh localhost
```

В данном случае будет показано, какие библиотеки и конфигурационные файлы открывает программа `ssh`, когда устанавливает соединение.

Можно даже использовать strace в качестве простого средства профилирования: при запуске с флагом -c она измеряет время, проведенное внутри системных вызовов, и выводит его в следующем виде:

```
# strace -c grep linux /usr/lib/* > /dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------
 78.68    0.012825           1     11098        18 read
 11.03    0.001798           1      3551           write
 10.02    0.001634           8       216        15 open
  0.26    0.000043           0       202           fstat64
  0.00    0.000000           0       201           close
  0.00    0.000000           0         1           execve
  0.00    0.000000           0         1         1 access
  0.00    0.000000           0         3           brk
  0.00    0.000000           0       199           munmap
  0.00    0.000000           0         1           uname
  0.00    0.000000           0         5           mprotect
  0.00    0.000000           0       207           mmap2
  0.00    0.000000           0        15        15 stat64
  0.00    0.000000           0         1           getuid32
  0.00    0.000000           0         1           set_tls
------ ----------- ----------- --------- --------- -----------
100.00 0.016300 15702 49 total
```

Резюме

Грех жаловаться на недостаток средств профилирования и трассировки в Linux. В этой главе мы познакомились с наиболее употребительными.

Столкнувшись с системой, которая работает хуже, чем хотелось бы, начните с top и попытайтесь идентифицировать проблему. Если виновато одно приложение, то можно провести его профилирование с помощью команд perf record/report, не забыв о необходимости сконфигурировать ядро для работы perf и подготовить отладочные символы для исполняемых файлов и ядра. Альтернативой perf record является программа OProfile, которая может сообщить примерно ту же информацию. Что касается gprof, то она, откровенно говоря, устарела, но зато не требует поддержки со стороны ядра. Если локализовать проблему не удается, попробуйте применить perf (или OProfile) для получения профиля системы в целом.

Ftrace выходит на передний план, когда нужно получить ответы на конкретные вопросы о поведении ядра. Трассировщики function и function_graph дают детальное представление о последовательности системных вызовов и связях между ними. Трассировщики событий позволяют получить еще больше информации о вызовах функций, в том числе о параметрах и возвращенных значениях. LTTng играет похожую роль, она пользуется механизмом трассировки событий и добавляет высокоскоростные кольцевые буферы для хранения больших объемов данных о работе ядра. Достоинством Valgrind является тот факт, что она исполняет

программу в песочнице, а значит, может сообщить о проблемах, которые трудно обнаружить другими способами.

Благодаря инструменту Callgrind эта программа может строить графы вызовов и генерировать отчет об использовании процессорного кэша, а Helgrind обнаруживает проблемы, связанные с потоками. Не забывайте и о программе `strace`. Это удобное средство, когда нужно понять, какие системные вызовы совершает программа: от перечисления обращений к функциям открытия файла с указанием полных путей до проверки входящих сигналов.

Никогда не забывайте об эффекте наблюдателя и старайтесь избегать его: удостоверьтесь, что результаты всех измерений сохраняют силу для производственной системы. В следующей главе я продолжу эту тему, рассмотрев трассировщики задержек, которые помогают количественно оценить производительность целевой системы в режиме реального времени.

Глава 14

Программирование в режиме реального времени

Значительная часть взаимодействий между вычислительной системой и реальным миром происходит в режиме реального времени, так что эта тема весьма важна для разработчиков встраиваемых систем. Я уже несколько раз касался программирования в режиме реального времени: в главе 10 рассматривался вопрос о политиках планирования и инверсии приоритетов, а в главе 11 – проблема страничных отказов и необходимость блокировки памяти. Теперь пришла пора собрать отрывочные сведения воедино и глубже изучить эту тему.

Мы начнем обсуждение с характеристик систем реального времени, а затем рассмотрим, как они отражаются на дизайне системы как на уровне приложений, так и на уровне ядра. Я опишу заплаты ядра для режима реального времени, `PREEMPT_RT`, и покажу, как их получить и применить к стержневой версии. В последних разделах рассказано о том, как оценить системные задержки с помощью инструментов `cyclictest` и `Ftrace`.

Существуют и другие способы добиться от встраиваемой Linux-системы поведения, характерного для режима реального времени, например воспользоваться специализированным микроконтроллером или установить отдельное ядро реального времени рядом с ядром Linux, как сделано в проектах Xenomai и RTAI. Я не буду их рассматривать, потому что книга посвящена использованию Linux как основы встраиваемых систем.

Что такое реальное время?

Природа программирования в режиме реального времени – одна из тем, которые программисты любят обсуждать подробно, зачастую оперируя противоречивыми определениями. Начну с описания того, что лично я считаю важным в понятии реального времени.

Говорят, что имеется задача реального времени, если она должна быть завершена не позднее определенного момента, который называют критическим сроком обслуживания. Разницу между обычной задачей и задачей реального времени можно проиллюстрировать на примере воспроизведения аудиопотока на компьютере, занятом в это время компиляцией ядра Linux.

Первая задача должна решаться в реальном времени, потому что драйверу поступает постоянный поток данных, и звуковые отсчеты необходимо записывать в аудиоинтерфейс со скоростью воспроизведения. А компиляция не является задачей реального времени, потому что критический срок обслуживания для нее не определен. Мы просто хотели бы, чтобы она завершилась поскорее, но от того, потребуется для этого 10 секунд или 10 минут, качество ядра не зависит.

Важно также рассмотреть последствия пропуска критического срока. Диапазон широк: от досадной, но мелкой неприятности до сбоя и выхода из строя системы. Приведем несколько примеров.

- **Воспроизведение аудиопотока.** Критический срок обслуживания составляет порядка десятков миллисекунд. Если буфер окажется пустым, мы услышим щелчок – неприятно, но можно пережить.
- **Перемещение и щелчки мышью.** Критический срок обслуживания также составляет десятки миллисекунд. В случае пропуска курсор мыши двигается хаотически, а щелчки игнорируются. Если проблема не исчезает, то работать с системой становится невозможно.
- **Печать на листе бумаги.** Критический срок подачи бумаги – несколько миллисекунд. Если он пропущен, может произойти замятие, и кому-то придется идти и вытаскивать замятый лист. С редкими замятиями можно смириться, но никто не станет покупать принтер, который зажевывает бумагу постоянно.
- **Печать срока хранения на бутылках, движущихся по производственной линии.** Если дата не нанесена всего лишь на одну бутылку, приходится останавливать всю линию, изымать бутылку и перезапускать линию – это дорого.
- **Выпекание торта.** Критический срок обслуживания – порядка 30 минут. Если опоздать на несколько минут, то торт сгорит. Если задержаться подольше, сгорит весь дом.
- **Система обнаружения скачков напряжения.** После обнаружения скачка размыкатель цепи должен сработать в течение 2 миллисекунд. В противном случае возможны повреждение оборудования и травма или смерть персонала.

Как видим, последствия пропуска критического срока обслуживания разнообразны. Часто выделяют три категории.

- **Мягкий режим реального времени.** Выдерживание критического срока обслуживания желательно, но иногда его можно пропустить без катастрофических последствий для системы. К этой категории относятся первые два примера.

- **Жесткий режим реального времени.** В этом случае пропуск критического срока обслуживания влечет серьезные последствия. Системы жесткого реального времени можно далее подразделить на ответственные – важные для выполнения задачи организации, когда пропуск критического срока влечет только затраты, и важные для безопасности, когда опасность угрожает жизни и здоровью, как в последних двух примерах. Я привел пример с тортом, чтобы показать, что не в любой системе жесткого реального времени критический срок измеряется в микросекундах.

Программное обеспечение систем, важных для безопасности, должно удовлетворить различным стандартам, цель которых – обеспечить надежность. Очень трудно удовлетворить этим требованиям с помощью такой сложной операционной системы, какой является Linux.

Если же речь идет об ответственных системах, то Linux вполне возможно использовать в самых разных управляющих системах. Требования к программному обеспечению зависят от комбинации критического срока обслуживания и степени надежности, которая обычно устанавливается в результате всесторонних испытаний.

Поэтому, чтобы назвать продукт системой реального времени, мы должны измерить время отклика при максимальной ожидаемой нагрузке и продемонстрировать, что критический срок обслуживания выдерживается с оговоренной частотой. В первом приближении можно сказать, что правильно настроенная Linux-система со стержневым ядром пригодна для работы в мягком режиме реального времени с критическим сроком обслуживания не менее десятков миллисекунд, а ядро с заплатами `PREEMPT_RT` – для мягкого реального времени и ответственных систем жесткого реального времени с критическим сроком обслуживания порядка нескольких сотен микросекунд.

Ключ к созданию системы реального времени – уменьшить непостоянство времени отклика и тем самым повысить уверенность в том, что критический срок не будет пропущен; иными словами, система должна быть более детерминированной. Часто для достижения этой цели приносят в жертву производительность. Например, кэш ускоряет работу системы, сокращая среднее время доступа к данным, но в случае непопадания в кэш максимальное время может увеличиться. Кэширование делает систему более быстрой, но менее детерминированной, а это как раз то, чего мы хотим избежать.

Это миф, что система реального времени обязательно быстрая. Напротив, чем более детерминирована система, тем ниже ее пропускная способность.

Далее в этой главе мы займемся определением причин задержек и способами их уменьшения.

Определение источников недетерминированности

По большому счету смысл программирования в режиме реального времени – гарантировать своевременное планирование потоков, отвечающих за действия си-

стемы, чтобы они могли завершить работу до истечения критического срока обслуживания. Все, что этому мешает, следует считать проблемой. Вот несколько потенциальных источников проблем.

- **Планирование.** Потоки реального времени необходимо планировать раньше прочих, т. е. для них обязательно задавать политику реального времени: `SCHED_FIFO` или `SCHED_RR`. Кроме того, согласно теории частотно-монотонного анализа, упомянутой в главе 10, им следует назначать приоритеты в порядке убывания, начиная с потока с самым коротким критическим сроком.
- **Задержка планирования.** Ядро должно перепланировать потоки, как только произойдет прерывание или сработает таймер, неограниченные задержки недопустимы. Уменьшение задержек планирования – основная тема этой главы.
- **Инверсия приоритетов.** Это следствие планирования на основе приоритетов, которое приводит к неопределенно долгим задержкам, если высокоприоритетный поток блокирован в ожидании мьютекса, удерживаемого низкоприоритетным потоком (см. главу 10). В пользовательском пространстве используются мьютексы с наследованием приоритета и предельным приоритетом; в пространстве ядра – rt-мьютексы, реализующие протокол наследования приоритета; о них я расскажу ниже, в разделе, посвященном ядру реального времени.
- **Высокоточные таймеры.** Если необходим критический срок обслуживания порядка нескольких миллисекунд или даже микросекунд, то и таймеры должны быть соответствующие. Без таймеров высокого разрешения не обойтись, но, к счастью, их можно сконфигурировать почти во всех ядрах.
- **Страничные отказы.** Страничный отказ во время выполнения критического участка кода сбивает все оценки времени. Этого можно избежать с помощью блокировки памяти, о чем я расскажу ниже.
- **Прерывания.** Прерывания происходят в непредсказуемые моменты времени и могут привести к неожиданной перегрузке, если поступают быстро и в большом количестве. Избежать этого можно двумя способами. Первый – реализовать обработчики прерываний в виде потоков ядра, второй, применимый в многоядерных устройствах, – оградить одно или несколько ядер от обработки прерываний. Я рассмотрю обе возможности.
- **Процессорные кэши.** Это буфер между процессором и оперативной памятью, и, как все кэши, он является источником недетерминированности. К сожалению, рассмотрение этой темы выходит за рамки книги, но в конце главы есть ссылки.
- **Конкуренция за шину памяти.** Если периферийные устройства обращаются к памяти напрямую через канал DMA, то они забирают себе часть пропускной способности шины памяти, что замедляет доступ со стороны процессорного ядра (или нескольких ядер), а стало быть, увеличивает степень недетерминированности программы. Но это проблема аппаратная, к данной книге не имеющая отношения.

В следующих разделах мы обсудим поднятые вопросы и посмотрим, что тут можно делать.

В списке не отражено управление энергосбережением. Потребности реального времени и энергосбережения прямо противоположны. Стремление управлять энергосбережением часто приводит к большим задержкам при переключении состояний сна, поскольку для настройки регуляторов электропитания и пробуждения процессоров требуется время, как и для изменения частоты тактового генератора (поскольку выход в новый стационарный режим происходит не мгновенно). Но, разумеется, нельзя ожидать, что приостановленное устройство мгновенно ответит на прерывание. Я-то точно знаю, что по утрам ни на что не способен, пока не выпью хотя бы одну чашку кофе.

Задержки планирования

Потоки реального времени должны планироваться, как только у них появится работа. Но даже если нет ни одного потока с таким же или более высоким приоритетом, между событием – прерыванием или срабатыванием системного таймера – и моментом, когда поток приступит к работе, обязательно проходит какое-то время. Это время и называется задержкой планирования. В задержке можно выделить несколько составных частей, как показано на рисунке ниже.

Прежде всего имеется аппаратная задержка прерывания – время от момента изменения уровня сигнала на линии прерывания до начала работы обработчика прерывания (interrupt service routine – **ISR**). Малая часть этой задержки приходится на само оборудование, но основная проблема – запрет прерывания в программе. Очень важно минимизировать время нахождения программы в состоянии с запрещенными прерываниями.

Далее идет задержка прерывания, т. е. время обработки прерывания до момента пробуждения потоков, ждущих соответствующего события. Ее величина зависит главным образом от того, как написан обработчик, и обычно составляет несколько микросекунд.

И наконец, задержка вытеснения, т. е. время от момента, когда ядро получило уведомление о готовности потока, до момента, когда планировщик запускает этот

поток. Оно зависит от того, может ядро вытеснить текущий поток или нет. Если исполняется критическая секция кода, то планировщику придется подождать. Величина задержки определяется конфигурацией вытеснения в ядре.

Вытеснение в ядре

Задержка вытеснения возникает из-за того, что не всегда безопасно или желательно вытеснять текущий поток выполнения и вызывать планировщик. В стержневой версии Linux есть три конфигурационных параметра вытеснения в меню **Kernel Features → Preemption Model**:

- `CONFIG_PREEMPT_NONE`: без вытеснения;
- `CONFIG_PREEMPT_VOLUNTARY`: производятся дополнительные проверки запросов о вытеснении;
- `CONFIG_PREEMPT`: разрешено вытеснение в ядре.

Если установлен режим без вытеснения, то ядро продолжит работу без перепланирования до тех пор, пока не вернется из системного вызова в пользовательское пространство, где вытеснение всегда разрешено, или не встретит состояние сна, в котором текущий поток останавливается. Поскольку в таком режиме уменьшается количество переходов между ядром и пользовательским пространством и, возможно, общее число контекстных переключений, он обеспечивает более высокую пропускную способность ценой увеличения задержки вытеснения. Этот режим подразумевается по умолчанию на серверах и в некоторых ядрах для ПК, где пропускная способность считается важнее скорости реакции.

Во втором режиме имеются явные точки вытеснения, в которых вызывается планировщик, если поднят флаг `need_resched`. Это уменьшает задержки вытеснения в худшем случае ценой некоторого снижения пропускной способности. Такой режим установлен в некоторых дистрибутивах для ПК.

В третьем режиме ядро допускает вытеснение, т. е. прерывание может привести к немедленному перепланированию, если только ядро не выполняется в атомарном контексте, который будет описан ниже. Это уменьшает задержку вытеснения в худшем случае, а значит, и общую задержку планирования до нескольких миллисекунд на типичном встраиваемом оборудовании. Часто такой режим называют режимом мягкого реального времени, и большинство ядер для встраиваемых систем так и конфигурируется. Разумеется, общая пропускная способность немного падает, но обычно это не так важно, как гарантии более детерминированного планирования.

Ядро Linux реального времени (PREEMPT_RT)

Уже давно предпринимаются усилия еще больше сократить задержки, соответствующий проект называется по имени конфигурационного параметра – `PREEMPT_RT`. У истоков проекта стояли Инго Молнар (Ingo Molnar), Томас Глейкснер (Thomas Gleixner) и Стивен Ростедт (Steven Rostedt), при содействии многих других разработчиков. Заплаты для ядра находятся по адресу https://www.kernel.org/

pub/linux/kernel/projects/rt, а по адресу https://rt.wiki.kernel.org организован вики-сайт, включающий в том числе и FAQ (быть может, немного устаревший).

С годами в стержневую версию Linux были включены многие части проекта, в том числе таймеры высокого разрешения, ядерные мьютексы и потоковые обработчики прерываний. Однако основные заплаты остаются вне стержневой версии, потому что они затрагивают довольно много частей кода и (как утверждают некоторые) приносят пользу лишь небольшому подмножеству пользователей Linux. Быть может, когда-нибудь в ядро войдет весь набор заплат.

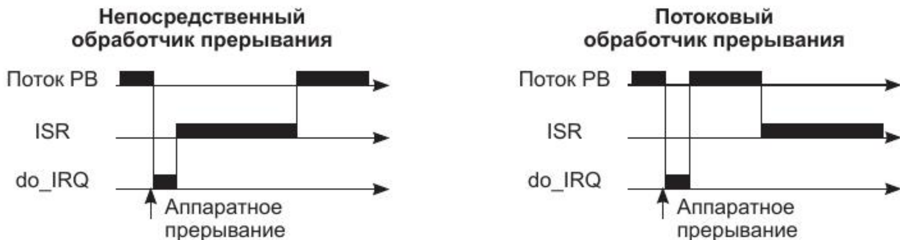
Суть плана состоит в том, чтобы уменьшить время нахождения ядра в атомарном контексте, в котором небезопасно вызывать планировщик и переключаться на другой поток. Перечислим типичные атомарные контексты:

- ядро исполняет обработчик прерывания или ловушки;
- ядро удерживает спин-блокировку или находится в критической секции RCU (read-copy update – чтение и обновление копированием). Спин-блокировки (spin lock) и RCU – это примитивы блокировки в ядре, детали которых нам сейчас не важны;
- ядро выполняет код между вызовами `preempt_disable()` и `preempt_enable()`;
- запрещены аппаратные прерывания.

Входящие в состав проекта `PREEMPT_RT` изменения распадаются на две части: одни уменьшают влияние обработчиков прерываний путем преобразования их в потоки ядра, а другие делают блокировки вытесняемыми, так что поток может заснуть, не освобождая блокировку. Очевидно, что в обоих случаях накладные расходы велики, поэтому обработка прерывания в среднем оказывается медленнее, зато намного более детерминирована, а именно это нам и нужно.

Потоковые обработчики прерываний

Не все прерывания запускают задачи реального времени, но любое прерывание крадет процессорное время у задачи реального времени. Механизм потоковых обработчиков прерываний позволяет назначить прерыванию приоритет и запланировать его обработку в удобное время, как показано на рисунке ниже.



Если обработчик прерывания выполняется как поток ядра, то нет причин, мешающих его вытеснению высокоприоритетным потоком, работающим в пользова-

тельском пространстве, поэтому обработчик прерывания не дает вклада в задержку планирования пользовательского потока. Потоковые обработчики прерываний включены в стержневую версию Linux 2.6.30. Чтобы обработчик конкретного прерывания стал потоковым, его следует зарегистрировать функцией request_threaded_irq() вместо request_irq(). Можно сделать потоковую обработку прерываний режимом по умолчанию, присвоив конфигурационному параметру ядра CONFIG_IRQ_FORCED_THREADING значение y, тогда все обработчики становятся потоковыми, если не отменить это явно, подняв флаг IRQF_NO_THREAD. После наложения заплат PREEMPT_RT обработчики прерываний по умолчанию делаются потоковыми. Ниже показано, как это может выглядеть.

```
# ps -Leo pid,tid,class,rtprio,stat,comm,wchan | grep FF
PID   TID   CLS   RTPRIO  STAT  COMMAND            WCHAN
3     3     FF    1       S     ksoftirqd/0       smpboot_th
7     7     FF    99      S     posixcputmr/0     posix_cpu_
19    19    FF    50      S     irq/28-edma       irq_thread
20    20    FF    50      S     irq/30-edma_err   irq_thread
42    42    FF    50      S     irq/91-rtc0       irq_thread
43    43    FF    50      S     irq/92-rtc0       irq_thread
44    44    FF    50      S     irq/80-mmc0       irq_thread
45    45    FF    50      S     irq/150-mmc0      irq_thread
47    47    FF    50      S     irq/44-mmc1       irq_thread
52    52    FF    50      S     irq/86-44e0b000   irq_thread
59    59    FF    50      S     irq/52-tilcdc     irq_thread
65    65    FF    50      S     irq/56-4a100000   irq_thread
66    66    FF    50      S     irq/57-4a100000   irq_thread
67    67    FF    50      S     irq/58-4a100000   irq_thread
68    68    FF    50      S     irq/59-4a100000   irq_thread
76    76    FF    50      S     irq/88-OMAP UAR   irq_thread
```

В данном случае, относящемся к плате BeagleBone с ядром linux-yocto-rt, только обработчик прерывания от таймера gp_timer является не потоковым, а непосредственным, так и должно быть.

Отметим, что всем потокам обработки прерываний назначены политика планирования по умолчанию SCHED_FIFO и приоритет 50. Но не имеет особого смысла оставлять значения по умолчанию, вы можете расставить приоритеты в соответствии со сравнительной важностью прерываний и потоков реального времени в пользовательском пространстве.

Ниже приведен рекомендуемый порядок убывания приоритетов потоков.

- Поток POSIX-таймеров, posixcputmr, всегда должен иметь наивысший приоритет.
- Аппаратные прерывания, ассоциированные с самым высокоприоритетным потоком реального времени в пользовательском пространстве.
- Самый высокоприоритетный поток реального времени в пользовательском пространстве.

- Аппаратные прерывания для потоков реального времени с уменьшающимися приоритетами, за каждым из которых следует сам поток.
- Аппаратные прерывания для интерфейсов, не требующих обработки в режиме реального времени.
- Программный демон обработки прерываний `ksoftirqd`, который в ядрах реального времени отвечает за выполнение отложенных обработчиков прерываний и до выхода версии Linux 3.6 отвечал еще и за выполнение сетевого стека, уровня блочного ввода-вывода и другие вещи. Можете поэкспериментировать с различным назначением приоритетов и посмотреть, когда получается наилучший результат.

Приоритеты можно изменить командой `chrt`, вызываемой из скрипта загрузки:

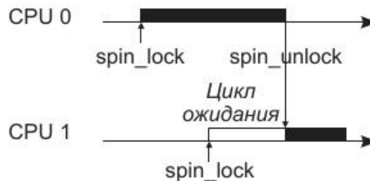
```
# chrt -f -p 90 `pgrep irq/28-edma`
```

Команда `pgrep` входит в пакет `procps`.

Вытесняемые блокировки ядра

Преобразование большинства блокировок ядра в вытесняемые – самое нелокальное изменение, из-за которого `PREEMPT_RT` остается вне стержневой версии.

Проблема связана со спин-блокировками, которые применяются в большинстве мест, где нужны блокировки. Спин-блокировка – это мьютекс, крутящийся в цикле активного ожидания, он не требует контекстного переключения при наличии конкуренции и потому очень эффективен, если только блокировку нужно захватить ненадолго. В идеале время захвата должно быть меньше, чем потребовалось бы для двух операций планирования. На рисунке ниже показаны потоки, работающие на двух процессорах, конкурирующих за одну спин-блокировку. Процессор **CPU0** получает блокировку первым, а **CPU1** вынужден крутиться в цикле, ожидая ее освобождения.



Поток, удерживающий спин-блокировку, нельзя вытеснять, потому что иначе другой поток мог бы войти в тот же участок кода и навеки застрять там, пытаясь захватить ту же блокировку. Поэтому в стержневой версии Linux захват спин-блокировки запрещает вытеснение ядра, создавая атомарный контекст. Это означает, что низкоприоритетный поток, удерживающий спин-блокировку, может препятствовать планированию высокоприоритетного потока.

В PREEMPT_RT эта проблема решается заменой почти всех спин-блокировок rt-мьютексами. Мьютекс медленнее спин-блокировки, зато допускает вытеснение без ограничений. К тому же rt-мьютексы реализуют протокол наследования приоритета и потому не так чувствительны к инверсии приоритетов.

Получение заплат PREEMPT_RT

Разработчики PREEMPT_RT не создают наборов заплат для каждой версии ядра, потому что это потребовало бы чрезмерных усилий. В среднем заплаты создаются для каждой второй версии. Ниже перечислены последние версии ядра, которые поддерживались на момент написания книги:

- 4.1-rt;
- 4.0-rt;
- 3.18-rt;
- 3.14-rt;
- 3.12-rt;
- 3.10-rt.

Заплаты можно скачать по адресу https://www.kernel.org/pub/linux/kernel/ projects/rt.

Если вы работаете с Yocto Project, то версия ядра с заплатами реального времени уже есть. В противном случае может статься, что на ядро, которое вы откуда-то скачали, уже были наложены заплаты PREEMPT_RT. Если же это не так, то наложить их придется самостоятельно. Сначала убедитесь, что версия заплат соответствует версии ядра, иначе наложить их не получится. Потом действуйте, как обычно:

```
$ cd linux-4.1.10
$ zcat patch-4.1.10-rt11.patch.gz | patch -p1
```

После этого можно будет сконфигурировать ядро с параметром CONFIG_PREEMPT_RT_FULL.

Но тут есть одна проблема. Наложить заплаты реального времени можно, только если используется совместимое стержневое ядро. Но очень может быть, что это не так, из-за самой природы ядер для встраиваемых Linux-систем, и тогда придется потратить какое-то время, чтобы понять, почему заплата не встала, исправить ошибки, а затем проанализировать пакет поддержки вашей целевой платформы и добавить недостающую поддержку реального времени. Но эти детали уже выходят за рамки книги. Если вы не понимаете, что делать, задайте вопросы разработчикам используемого ядра и на форуме разработчиков ядра.

Yocto Project и PREEMPT_RT

В Yocto Project предлагаются два стандартных рецепта ядра: linux-yocto и linux-yoco-rt, причем в последнем случае заплаты реального времени уже наложены. В предположении, что эти ядра поддерживают целевую платформу, нужно только выбрать linux-yocto-rt в качестве предпочтительного ядра и объявить, что машина полностью совместима, например добавив такие строки в файл conf/local.conf:

```
PREFERRED_PROVIDER_virtual/kernel = "linux-yocto-rt"
COMPATIBLE_MACHINE_beaglebone = "beaglebone"
```

Таймеры высокого разрешения

Разрешение таймеров важно, если предъявляются требования к точности хронометража – для приложений реального времени это типично. По умолчанию в Linux в качестве таймера используются часы с настраиваемой частотой, обычно 100 Гц для встраиваемых систем и 250 Гц для серверов и ПК. Интервал между двумя тактами таймера называется мигом (jiffy), в примерах выше миг равен 10 миллисекундам во встраиваемой SoC-системе и 4 миллисекундам на сервере.

Благодаря проекту ядра реального времен Linux получила более точные таймеры в версии 2.6.18, и теперь они доступны на всех платформах при условии, что имеется источник таймера высокого разрешения и драйвер для него – почти всегда так оно и есть. При конфигурировании ядра необходимо задать параметр `CONFIG_HIGH_RES_TIMERS=y`.

Если все указанное в наличии, то все таймеры в ядре и в пользовательском пространстве будут отсчитывать время с точностью, обеспечиваемой оборудованием. Определить, какова эта точность, трудно. Очевидный ответ – значение, возвращенное функцией `clock_getres(2)`, но она всегда утверждает, что разрешение равно одной наносекунде. Описанная ниже утилита `cyclictest` позволяет проанализировать время, сообщаемое таймером, и высказать гипотезу о разрешающей способности оборудования:

```
# cyclictest -R
# /dev/cpu_dma_latency set to 0us
WARN: reported clock resolution: 1 nsec
WARN: measured clock resolution approximately: 708 nsec
You can also look at the kernel log messages for strings like this:
# dmesg | grep clock
OMAP clockevent source: timer2 at 24000000 Hz
sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every 178956969942ns
OMAP clocksource: timer1 at 24000000 Hz
Switched to clocksource timer1
```

Два метода дают разные величины. У меня нет объяснения этому феномену, но поскольку в любом случае разрешение меньше одной микросекунды, то я доволен.

Предотвращение страничных отказов в приложении реального времени

Страничный отказ происходит, когда приложение читает или записывает по адресу, для которого еще не выделена физическая память. Невозможно (или очень трудно) предсказать, когда произойдет страничный отказ, поэтому это еще один источник недетерминированности в вычислительной системе.

По счастью, существует функция, которая позволяет передать процессу всю заказанную им память и запретить ему страничный обмен, так что в дальнейшем страничных отказов не будет. Она называется `mlockall(2)` и принимает один или два битовых флага:

- `MCL_CURRENT`: заблокировать все уже отображенные страницы;
- `MCL_FUTURE`: заблокировать все страницы, которые будут переданы процессу в будущем.

Обычно `mlockall(2)` вызывается на этапе инициализации приложения с обоими поднятыми флагами, чтобы заблокировать все текущие и будущие страницы.

Отметим, что во флаге `MCL_FUTURE` нет ничего волшебного: при выделении или освобождении памяти из кучи с помощью функций `malloc()/free()` или `mmap()` все равно возможна недетерминированная задержка. Такие операции лучше выполнять на этапе инициализации, а не в главном управляющем цикле.

С памятью, выделяемой из стека, дело обстоит сложнее, потому что это делается автоматически, и если в результате вызова функции стек становится глубже, чем он был до того, то возникнет задержка управления памятью. Предотвратить это просто – в самом начале выделите под стек память, превышающую его ожидаемый рост в ходе работы программы. Это можно сделать так:

```
#define MAX_STACK (512*1024)
static void stack_grow (void)
{
  char dummy[MAX_STACK];
  memset(dummy, 0, MAX_STACK);
  return;
}

int main(int argc, char* argv[])
{
  [...]
  stack_grow ();
  mlockall(MCL_CURRENT | MCL_FUTURE);
  [...]
```

Функция `stack_grow()` выделяет в стеке память для большой переменной, а затем обнуляет ее, чтобы заставить ОС передать процессу физическую память под эти страницы.

Экранирование прерываний

Потоковые обработчики прерываний позволяют уменьшить накладные расходы путем назначения некоторым потокам более высокого приоритета, чем у обработчиков, не оказывающих влияния на работу задач реального времени. При наличии многоядерного процессора можно применить другой подход и полностью экрани-

ровать одно или несколько ядер от обработки прерывания, отдав их исключительно под выполнение задач реального времени. Это работает как в стандартном ядре Linux, так и в ядре с заплатами `PREEMPT_RT`.

Чтобы сделать это, нужно закрепить потоки реального времени на одном процессоре, а обработчики прерываний – на другом. Для этого нужно задать привязку процессора к потоку или процессу с помощью командной утилиты `taskset` или воспользовавшись функциями `sched_setaffinity(2)` и `pthread_setaffinity_np(3)`.

Чтобы задать привязку обработчика прерывания, заметим, что каждому прерыванию соответствует каталог `/proc/irq/<номер прерывания>`. В нем находятся все управляющие файлы, в том числе `smp_affinity`, содержащий маску процессоров. Запишите в него битовую маску, в которой поднят бит для каждого процессора, имеющего право обрабатывать это прерывание.

Измерение задержек планирования

Как бы ни конфигурировать и ни оптимизировать систему, все будет бесполезно, если невозможно удостовериться, что устройство удовлетворяет требованиям к критическому сроку обслуживания. Критерии окончательного тестирования устанавливаете вы сами, а я лишь опишу два важных измерительных инструмента: `cyclictest` и `Ftrace`.

cyclictest

Первоначальную версию программы `cyclictest` написал Томас Глейкснер, а теперь она имеется для большинства платформ и входит в состав пакета `rt-tests`. Если вы используете Yocto Project, то можете создать образ целевой системы, включающий `rt-tests`, собрав его по такому рецепту:

```
$ bitbake core-image-rt
```

При работе с Buildroot нужно будет добавить пакет `BR2_PACKAGE_RT_TESTS` в меню **Target packages → Debugging, profiling and benchmark → rt-tests**.

Для измерения задержек планирования `cyclictest` сравнивает фактическое время, потраченное на сон, с запрошенным. Если бы никакой задержки не было, то обе величины совпали бы, и программа сообщила бы, что задержка равна 0. `cyclictest` предполагает, что разрешение таймера меньше одной микросекунды.

У команды много параметров. Для начала попробуйте запустить ее на целевом устройстве от имени root:

```
# cyclictest -l 100000 -m -n -p 99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 1.14 1.06 1.00 1/49 320

T: 0 ( 320) P:99 I:1000 C: 100000 Min: 9 Act: 13 Avg: 15 Max: 134
```

Параметры интерпретируются следующим образом:

- -l N: повторять цикл N раз, по умолчанию бесконечно;
- -m: заблокировать память, вызвав mlockall;
- -n: использовать clock_nanosleep(2) вместо nanosleep(2);
- -p N: задать приоритет реального времени, равный N.

Теперь опишем, что означает напечатанный результат (слева направо).

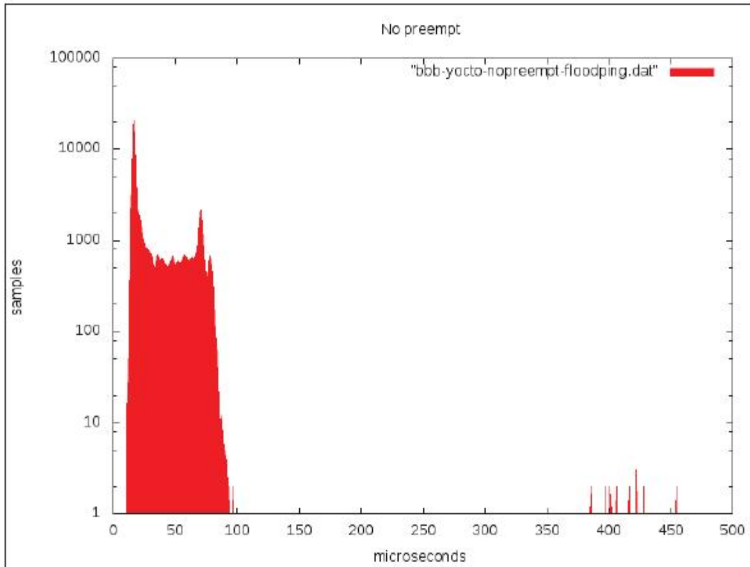
- T: 0: это поток 0, единственный в программе. Количество потоков можно задать с помощью параметра -t.
- (320): PID процесса равен 320.
- P:99: был установлен приоритет 99.
- I:1000: интервал между циклами равен 1000 микросекунд. Его можно задать с помощью параметра -i N.
- C:100000: номер последней итерации цикла был равен 100 000.
- Min: 9: минимальная задержка составила 9 микросекунд.
- Act:13: фактическая задержка составила 13 микросекунд. Фактической считается последняя измеренная задержка, эта величина имеет смысл, только если вы наблюдаете за выполнением cyclictest.
- Avg:15: средняя задержка составила 15 микросекунд.
- Max:134: максимальная задержка составила 134 микросекунды.

Эти результаты были получены на простаивающей системе с немодифицированным ядром linux-yocto просто ради демонстрации. Реальную пользу программа принесет, только если запустить тест на 24 часа или даже больше при загрузке системы, близкой к максимально ожидаемой.

Из всех чисел, которые вычисляет cyclictest, наибольший интерес представляет максимальная задержка, но полезно было бы также получить представление о разбросе. Для этого нужно задать флаг -h <N> – будет создана гистограмма за последние N микросекунд. Пользуясь этой техникой, я получил три графика для одной и той же целевой платы с тремя разными ядрами: без вытеснения, со стандартным вытеснением и с вытеснением реального времени. В качестве нагрузки во всех случаях выступал трафик Ethernet, создаваемый ping-флудом. Выполнялась следующая команда:

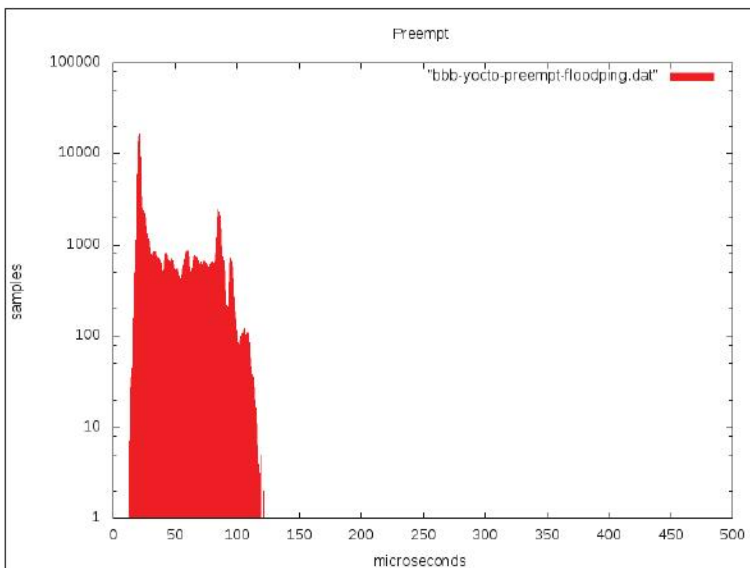
```
# cyclictest -p 99 -m -n -l 100000 -q -h 500 > cyclictest.data
```

Для ядра без вытеснения была получена такая гистограмма:



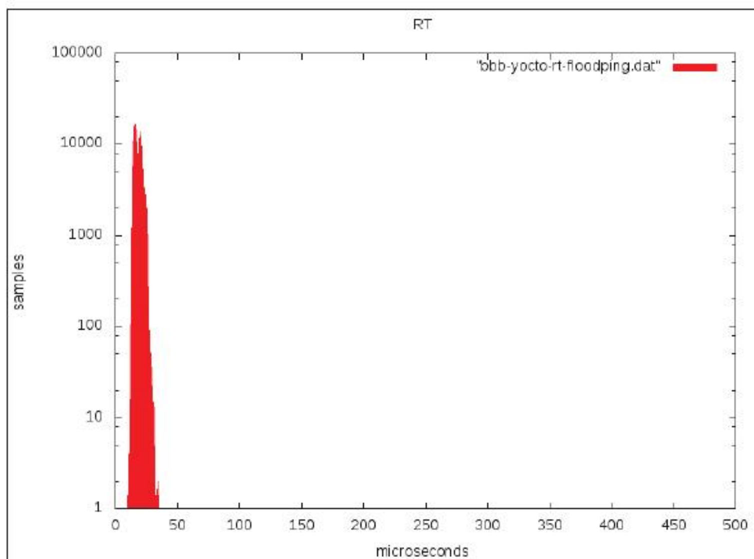
В этом случае большинство примеров отстает от критического обслуживания срока обслуживания не более чем на 100 микросекунд, но есть несколько, отстающих аж на 500 микросекунд. В общем, этого и следовало ожидать.

Для ядра со стандартным вытеснением была получена такая гистограмма:



Теперь примеры разбросаны по более широкому интервалу, но ни один не отстает больше чем на 120 микросекунд.

А вот диаграмма для ядра с вытеснением реального времени.



Ядро реального времени – очевидный победитель, потому что все примеры сосредоточены в районе отметки 20 микросекунд и ни один не отстает больше, чем на 35 микросекунд.

Таким образом, `cyclictest` – стандартный измеритель задержек планирования. Однако он не поможет найти и разрешить специфические проблемы, связанные с задержками в ядре. Для этого понадобится `Ftrace`.

Ftrace

В этой программе есть трассировщики для отслеживания задержек в ядре – в конце концов, именно для этого она и была написана. Трассировщики запоминают трассу худшей задержки, обнаруженной за время прогона, и показывают все функции, приведшие к этой задержке. Ниже перечислены интересные нам трассировщики и соответствующие конфигурационные параметры ядра.

- `irqsoff`: `CONFIG_IRQSOFF_TRACER`, трассирует код, исполняемый с запрещенными прерываниями, запоминает худший случай.
- `preemptoff`: `CONFIG_PREEMPT_TRACER`, аналогичен `irqsoff`, но трассирует самый длительный период времени, в течение которого было запрещено вытеснение в ядре (доступен только для ядер с вытеснением).
- `preemptirqsoff`: объединяет обе указанные выше трассы, т. е. трассирует самый длительный период времени, в течение которого были запрещены прерывания и (или) вытеснение в ядре.

- wakeup: трассирует и запоминает максимальное время, прошедшее от момента пробуждения самой высокоприоритетной задачи до ее планирования;
- wakeup_rt: то же, что wakeup, но только для потоков реального времени с политиками планирования SCHED_FIFO, SCHED_RR или SCHED_DEADLINE.
- wakeup_dl: то же, но только для потоков с политикой планирования SCHED_DEADLINE.

Имейте в виду, что работа Ftrace заметно увеличивает задержку – на величину порядка десятков миллисекунд при каждом запоминании нового максимума. Сама Ftrace может вычесть эту задержку, но она искажает результаты трассировщиков, работающих в пользовательском пространстве, в частности cyclictest. Иными словами, не следует обращать внимания на результаты cyclictest, если одновременно с ней работают трассировщики Ftrace.

Порядок выбора трассировщика такой же, как для трассировщиков функций, рассмотренных в главе 13. Ниже приведен пример запоминания трассы с максимальным временем работы с запрещенным вытеснением на протяжении 60 секунд:

```
# echo preemptoff > /sys/kernel/debug/tracing/current_tracer
# echo 0 > /sys/kernel/debug/tracing/tracing_max_latency
# echo 1 > /sys/kernel/debug/tracing/tracing_on
# sleep 60
# echo 0 > /sys/kernel/debug/tracing/tracing_on
```

Результат (сильно отредактированный) выглядит так:

```
# cat /sys/kernel/debug/tracing/trace
# tracer: preemptoff
#
# preemptoff latency trace v1.1.5 on 3.14.19-yocto-standard
# --------------------------------------------------------------------
# latency: 1160 us, #384/384, CPU#0 | (M:preempt VP:0, KP:0, SP:0 HP:0)
#    -----------------
#    | task: init-1 (uid:0 nice:0 policy:0 rt_prio:0)
#    -----------------
# => started at: ip_finish_output
# => ended at:   __local_bh_enable_ip
#
#
#                 _------=> CPU#
#                / _-----=> irqs-off
#               | / _----=> need-resched
#               || / _---=> hardirq/softirq
#               ||| / _--=> preempt-depth
#               |||| /     delay
# cmd    pid    ||||| time   |   caller
#    \ /        ||||| \      | /
init-1          0..s.     1us+: ip_finish_output
init-1          0d.s2    27us+: preempt_count_add <-cpdma_chan_submit
init-1          0d.s3    30us+: preempt_count_add <-cpdma_chan_submit
```

```
init-1         0d.s4    37us+: preempt_count_sub <-cpdma_chan_submit

[...]

init-1         0d.s2  1152us+: preempt_count_sub <-__local_bh_enable
init-1         0d..2  1155us+: preempt_count_sub <-__local_bh_enable_ip
init-1         0d..1  1158us+: __local_bh_enable_ip
init-1         0d..1  1162us!: trace_preempt_on <-__local_bh_enable_ip
init-1         0d..1  1340us : <stack trace>
```

Здесь видно, что самый длинный период, в течение которого вытеснение было запрещено, составляет 1160 микросекунд. Об этом-то факте можно узнать, просто прочитав файл /sys/kernel/debug/tracing/tracing_max_latency, но трасса содержит также последовательность вызовов функций ядра, которые привели к такой задержке. В столбце delay показано, в какой момент вызывалась функция, причем последний вызов – функции trace_preempt_on() – произошел спустя 1162 мкс с начала трассы, после чего вытеснение было снова разрешено. Располагая этой информацией, можно проследить всю цепочку вызовов и, хочется надеяться, разобраться, есть здесь проблема или нет.

Остальные трассировщики работают аналогично.

Комбинирование cyclictest и Ftrace

Если cyclictest сообщает о неожиданно больших задержках, то можно воспользоваться параметром breaktrace, чтобы завершить программу и запустить Ftrace для получения дополнительной информации.

Для этого задайте в командной строке параметр -b<N> или --breaktrace=<N>, где N – длительность задержки в микросекундах, при достижении которой запускается трассировка. Тип трассировщика задается с помощью параметра -T[имя трассировщика] или одного из следующих флагов:

- -C: переключение контекста;
- -E: событие;
- -f: функция;
- -w: wakeup;
- -W: wakeup_rt.

Например, следующая команда запустит трассировщик функций Ftrace, если измеренная задержка превышает 100 микросекунд.

```
# cyclictest -a -t -n -p99 -f -b100
```

Дополнительная литература

В указанных ниже ресурсах можно найти дополнительные сведения по вопросам, затронутым в данной главе.

- *Buttazzo G.* Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Giorgio, Springer, 2011.
- *Gove D.* Multicore Application Programming. Addison Wesley, 2011.

Резюме

Термин «реальное время» не имеет смысла без указания критического срока обслуживания и допустимой частоты его пропуска. Зная это, вы сможете определить, подходит ли Linux на роль операционной системы, и, если да, приступить к настраиванию системы с учетом предъявляемых требований. Настройка Linux и приложения для работы в режиме реального времени означает стремление к большей детерминированности с целью получить гарантии своевременной обработки данных. Обычно за детерминированность приходится расплачиваться уменьшением общей пропускной способности, поэтому система реального времени не сможет обработать такой же объем данных, как система, не претендующая на это звание.

Невозможно математически строго доказать, что такая сложная операционная система, как Linux, всегда укладывается в критический срок обслуживания, поэтому остается только организовать как можно более полное тестирование с помощью таких инструментов, как `cyclictest` и `Ftrace`, и – самое главное – разработайте для своего приложения набор критериев пригодности.

Для повышения детерминированности необходимо обратить внимание как на приложение, так и на ядро. При написании приложений реального времени следуйте приведенным в этой главе рекомендациям касательно планирования и блокировки памяти.

Ядро оказывает огромное влияние на детерминированность системы в целом. К счастью, за долгие годы в этом направлении проделана большая работа. В качестве первого шага стоит включить режим вытеснения в ядре. Если после этого все равно оказывается, что пропуск критического срока обслуживания случается чаще допустимого, то стоит подумать о наложении на ядро заплат `PREEMPT_RT`. Они, конечно, уменьшат задержку, но, поскольку не включены в стержневую версию, могут возникнуть проблемы при попытке интеграции с ядром для конкретной платы, предоставленным поставщиком. Можно вместо этого – или в дополнение – попробовать найти причину задержек с помощью `Ftrace` и других подобных инструментов.

И на этом мы заканчиваем обсуждение встраиваемых Linux-систем. Разработка встраиваемых систем требует весьма широкого набора знаний и умений: от знакомства с оборудованием на низком уровне, понимания работы начального загрузчика и принципов его взаимодействия с ядром до навыков классного системного инженера, умеющего конфигурировать и настраивать пользовательские приложения для достижения максимальной производительности. И все это – для оборудования, которое почти всегда рассчитано на решение какой-то узкой задачи. Все это можно суммировать одной фразой: «Инженерное искусство – это способность за 1 миллион долларов сделать то, что каждый дурак может сделать за 5 миллионов». Надеюсь, что информация, изложенная в этой книге, поможет вам приблизиться к этой цели.

Предметный указатель

A

Ångström Distribution, 149
Autotools, 54
  библиотека SQLite, 56
Axis, компания, 25

B

Babeltrace, 326
Barebox, 82
BeagleBone, плата, 33
binutils, 37
BSD (Berkeley Software Distribution), 30
Buildroot
  добавление пакета, 147
  история, 142
  конфигурирование, 143
  настройка, 290
  руководство пользователя, 162
  скрипты инициализации, 231
  соответствие лицензионным требованиям, 148
  стабильные версии и поддержка, 142
  установка, 142
BusyBox, 115
  альтернатива, ToyBox, 117
  программа init, 229
  сборка, 116

C

C development toolkit (CDT), 300
CMake, 53
core-файлы, 296
  использование GDB для анализа, 298
crosstool-NG, 43
cyclictest, 347

D

data display debugger (DDD), 299

E

Eclipse, 300
EmbToolkit, 37
eMMC (embedded MMC), 169

F

Flashbench, 187
Flash-Friendly File System (F2FS), 190
Ftrace
  CONFIG_DYNAMIC_FTRACE, 322
  CONFIG_FUNCTION_GRAPH_TRACER, 322
  CONFIG_FUNCTION_TRACER, 322
  введение, 321
  динамический режим, 324
  подготовка к работе, 322
  события трассировки, 325
  фильтры трассировки, 324

G

gdbserver, 290
  использование для удаленной отладки, 289
GDB, отладчик GNU, 287
  командные файлы, 292
  команды, 292
  подключение к gdbserver, 290
GNU Autoconf, 54
GNU Automake, 54
GNU Libtool, 54
GNU набор инструментов, 37
GNU проект, 36
GPIO, 210
  обработка прерывания, 212
gprof, 320
GRUB 2, 65
gummiboot, 65

I

init, программа, 126, 229
Itsy PacKage, формат пакета, 141

J

JFFS2, файловая система, 178
  маркер очистки, 180
  сводные узлы, 179
  создание, 180

K

Kconfig, 79

L

LGPL (Lesser General Public License), 31
Linaro, 42
Linux
  автоматизация загрузки с помощью скриптов U-Boot, 78
  портирование на новую плату, 104
    с деревом устройств, 104
    без деревоа устройств, 105
Linux Weekly News, 308
LLVM, проект, 36
LTTng, 326
  трассировка ядра, 327

M

mmap
  использование для выделения частной памяти, 276
  использование для доступа к памяти устройства, 277
  использование для разделения памяти, 276
  отображение памяти, 276
MultiMediaCard (MMC), 168

N

NAND-память, 166
Native POSIX Thread Library (NPTL), 256
NFS, монтирование файловой системы, 134

O

OEM (производители комплектного оборудования), 28
OOB-область, 166
open NAND flash interface (ONFi), 168
OpenWrt, 142
OProfile, 319

P

PCMCIA (Personal Computer Memory Card International Association), 169
perf, 314
  графы вызовов, 317
  команда annotate, 318
  конфигурирование ядра, 314
  профилирование, 315
  сборка в Buildroot, 315
  сборка в Yocto Project, 315
PREEMPT_RT, заплаты, 343
procrank, 281
proc, файловая система, 120
ps_mem, 281

Q

QEMU, 34

R

RPM (Red Hat Package Manager), 141

S

Scratchbox2, 39
smem, 280
SoC-системы, поставщики, 27
strace, трассировка системных вызовов, 331
sysfs, файловая система, 120
sysroot, задание, 291
systemd, 237
  добавление службы, 240
  загрузка системы, 239
  применение во встраиваемых Linux-системах, 242
  сборка в Buildroot, 237
  сборка в Yocto Project, 237
  цели, службы и агрегаты, 238
System V init, 231
  inittab, 233
  добавление нового демона, 235
  запуск и остановка служб, 236
  скрипты init.d, 235

T

TFTP, протокол, 77
  загрузка ядра, 137
TiVo, 25

U

UBIFS, 185
U-Boot, 72, 74, 79
  загрузка Linux, 78
  загрузка образа, 76
  переменные окружения, 75
  портирование на новую плату, 78
  режим Сапсан, 82
  сборка, 72, 82
  тестирование, 82
  установка, 73
  формат загрузочного образа, 75
uClibc, 41
UEFI, стандарт прошивки, 64
UFS, универсальные флэш-накопители, 203
User Mode Linux (UML), 34

V

Valgrind, 282
  профилирование приложений, 330

Y

YAFFS2, файловая система, 181
Yocto Project, 149
  выполнение, 153
  и PREEMPT_RT, 344
  компоненты, 150
  контроль лицензий, 162
  конфигурирование, 151
  настройка образов, 159
  рецепт создания образа, 159
  сборка, 152
  слои, 154
  создание SDK, 160
  стабильные версии и поддержка, 150
  установка, 151

А

Абсолютно справедливый планировщик (CFS), 263
Архитектура процессора, 39

Б

Библиотека C
  eglibc, 41
  glibc, 41
  musl libc, 41
  uClibc, 41
  компоненты, 49
Библиотеки, компоновка, 50
Блока измерения производительности (PMU), 314
Блок управления памятью (MMU), 32, 268
Блочные устройства, 202

В

Виртуальная память, 268
вторичный загрузчик (SPL), 79
выполнение на месте (XIP), 165

Д

Двоичный интерфейс приложений (ABI), 39
Демон, запуск, 129
Деревья устройств, 66
  включаемые файлы, 69
  компиляция, 71, 99
  прерывания, 68
  свойство reg, 67
  указатели на описатели, 68
Динамическая компоновка, 50
Драйверы устройств
  GPIO, 210
  I2C, 214
  SPI, 216
  /sys/block, 209
  /sys/class, 208
  /sys/devices, 207
  sysfs, определение, 207
  анатомия, 218
  блочных, 200
  в пользовательском пространстве, 210
  загрузка, 221
  компиляция, 221
  определение, 199
  поиск подходящего, 209
  получение информации на этапе выполнения, 205
  светодиоды, 213
  сетевых, 200
  символьных, 200

З

Загрузочный ram-диск
  автономный, 123
  в формате cpio, встраивание в образ ядра, 125
  загрузка, 124
  монтирование proc, 124
  построение ядра с ram-диском, 125
  создание, 123
  формат initrd, 126
Задача, 336
Задержка планирования, 339
  cyclictest, 347
    в комбинации с Ftrace, 352
  Ftrace, 350
  измерение, 347
Запасная область, 166

К

Карты Secure Digital (SD), 168
Каталог технологической подготовки, 111
  права доступа к файлам, 114
Конфигурационные файлы-заголовки, 81
Конфигурация оборудования
  деревья устройств, 223
  определение, 222
  связывание оборудования с драйверами, 224

Копирование при записи, 247
Корневая файловая система, 29
  BusyBox, 115
  ram-диск, 122
  библиотеки, 118
  каталог технологической подготовки, 111
  компоненты, 110
  копирование на карту SD, 134
  монтирование по NFS, 134
  неизменяемая, 193
  оболочка, 115
  образ диска, 122
  определение, 109
  перенос на целевое устройство, 122
  программа init, 114
  программы, 114
  структура каталогов, 111
  тестирование в эмуляторе QEMU, 135
  тестирование с платой BeagleBone Black, 136
  уменьшение размера путем удаления таблицы символов, 119
  утилиты, 115
Кросс-компиляция
  Autotools, 54
  make-файлы, 54
  определение, 53
  проблемы, 58
  утилиты конфигурирования пакета, 57

Л

Линеаризованное дерево устройств (FDT), 60
Лицензии на программы с открытым исходным кодом, 30
Логические стираемые блоки (LEB), 182

М

Межпроцессное взаимодействие, 251
  на основе передачи сообщений, 252
    Unix-сокеты, 252
    именованные каналы, 252
    очереди сообщений POSIX, 253
  на основе разделяемой памяти, 254
Модули, компиляция, 99
Мьютексы
  взаимоблокировка, 259
  инверсия приоритетов, 259
  низкая производительность, 259

Н

Набор инструментов, 29, 36
  crosstool-NG, установка, 43
  sysroot, 48
  анатомия, 46
  библиотека, 48
  выбор, 44
  получение, 42
  получение информации о кросс-компиляторе, 46
  программы в составе, 48
  файлы-заголовки, 48
Наложение, 147
Начальный загрузчик
  выбор, 71
  назначение, 29
  определение, 60
  переход к ядру, 65
Недетерминированность, источники, 337
Номера машин ARM, 106

О

Обновление в месте эксплуатации, 194
Обработчик прерывания, 339
Образ с несортированными блоками (UBI), 182
Образы файловой системы, создание, 133

О

Одноуровневые ячейки (SLC), 166
Операционная система, выбор, 26
Операционные системы реального времени (ОСРВ), 244
Открытый исходный код, 30
  сообщество, 27

П

Пакеты
  pkg-config, утилита конфигурирования, 57
  менеджеры, 141
  форматы, 141
Планирование, 262
  выбор политики, 265
  выбор политики с разделением времени, 263
  выбор приоритета реального времени, 266
  политики реального времени, 264
  справедливость и детерминированность, 262
Платформенный набор инструментов, 38

Подкачка, 275
  выгрузка в сжатую память, 275
Политики реального времени, 264
  SCHED_FIFO, 264
  SCHED_RR, 264
Пользовательское пространство, 102
  структура памяти, 272
Последовательность начальной загрузки, 61
  вторичный загрузчик (SPL), 63
  код в ПЗУ, 62
  третичный загрузчик (TPL), 63
Последовательный периферийный интерфейс (SPI), 62, 168
Постоянное запоминающее устройство (ПЗУ), 164
Потоки, 256
  завершение, 258
  идентификатор (TID), 257
  компиляция многопоточной программы, 258
  межпроцессное взаимодействие, 258
  мьютексы, 259
  определение, 245
  создание, 256
  условные переменные, 259
Потоковые обработчики прерываний, 341
Поточно-локальная память, 258
Права доступа к файлам, проблемы, 136
Пропорциональный размер набора (Pss), 279
Профилирование
  для бедных, 313
  определение, 311
  с помощью top, 312
Процесс, 246
  выполнение другой программы, 249
  демон, 251
  завершение, 247
  карта памяти, 274
  межпроцессное взаимодействие, 251
  определение, 244
  потребление памяти, 278
  создание, 246

Р

Разделяемая память, 254
  в POSIX, 254
Разделяемые библиотеки, 51
  Buildroot, 295
  Yocto Project, 294
  нумерация, версия, 52
  отладка, 294
  преимущества, 52
Расширенный двоичный интерфейс приложений (EABI), 39

С

Самопальная файловая система, 109
Своевременная отладка, 295
Сеанс отладки, начало, 290
Сетевые устройства, 203
Сеть
  конфигурирование, 131
  сетевые компоненты для glibc, 132
Символьные устройства, 200
Системные вызовы, трассировка с помощью strace, 331
Системный раздел EFI (ESP), 65
Системы сборки, назначение, 139
Слои, Yocto Project, 154
  BitBake и рецепты, 156
События, трассировка, 321
Специальный BSP, 145
Стандарт иерархии файловой системы (FHS), 111, 138
Стандартный интерфейс флэш-памяти (CFI), 166
Старый двоичный интерфейс приложений (OABI), 39
Статическая компоновка, 50
Статические библиотеки, 50
Страничные отказы, предотвращение в режиме реального времени, 345

Т

Таблицы устройств, 133
Таймеры высокого разрешения, 345
Терминальный пользовательский интерфейс (TUI), 299
Третичный загрузчик (TPL), 79
Трехуровневые ячейки (TLC), 166

У

Узлы устройств, 119, 129
  пример использования devtmpfs, 130
  пример использования mdev, 130
  статические, 131
Управляемая флэш-память, 168, 169
  eMMC, 169
  MultiMediaCard (MMC), 168
  Secure Digital (SD), 168
Уровень флэш-преобразования, 177
Устройства на основе технологии памяти (MTD), 171

блочное устройство mtdblock, 175
драйверы устройств, 174
запись сообщений об ошибках ядра, 176
разделы, 171
символьное устройство mtd, 174
эмуляция NAND-памяти, 176
Утечки памяти, 281
mtrace, 281
Valgrind, 282
нехватка памяти, 283
Учетные записи пользователей
добавление в корневую файловую систему, 129
конфигурирование, 127

Ф

Файловые системы
временные, 192
выбор, 194
для флэш-памяти, 177
монтирование, 121
неизменяемые сжатые, 191
уровень флэш-преобразования, 177
Файловые системы для управляемой флэш-памяти, 186
Discard и TRIM, 188
ext4, 189
F2FS, 190
FAT16/32, 190
Flashbench, 187
Файловые системы для флэш-памяти типа NOR и NAND, 178
JFFS2, 178
UBI, 182
UBIFS, 182
YAFFS2, 181
Файлы, специфичные для платы, 80
Физические стираемые блоки (PEB), 182
Флэш-память
доступ из Linux, 170
доступ из начального загрузчика, 169

Э

Экранирование прерываний, 346
Элементы встраиваемой Linux-системы, 29
Эффект наблюдателя, 310

Я

Ядро, 29, 86
BeagleBone Black, 100
QEMU, 100
выбор, 88
загрузка по протоколу TFTP, 137
командная строка, 103
компиляция образа, 96
конфигурирование, 91
идентификация, 95
лицензирование, 90
модули, 96, 122, 222
отладка кода, 301
в kdb, 305
в kgdb, 301
на ранних стадиях, 304
пример сеанса, 302
отладка модулей, 304
ошибки (oops), 306
сохранение, 307
поддержка со стороны производителя, 90
получение исходного кода, 90
сборка, 90
сообщения, 102
сообщения о загрузке, 144
стабильные и долгосрочные версии, 89
структура памяти, 269
удаление артефактов сборки, 99
цикл разработки, 88
Ядро, отладчик ядра, 301

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Крис Симмондс

Встраиваемые системы на основе Linux

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Слинкин А. А.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Формат 70×100 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 22,5. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф