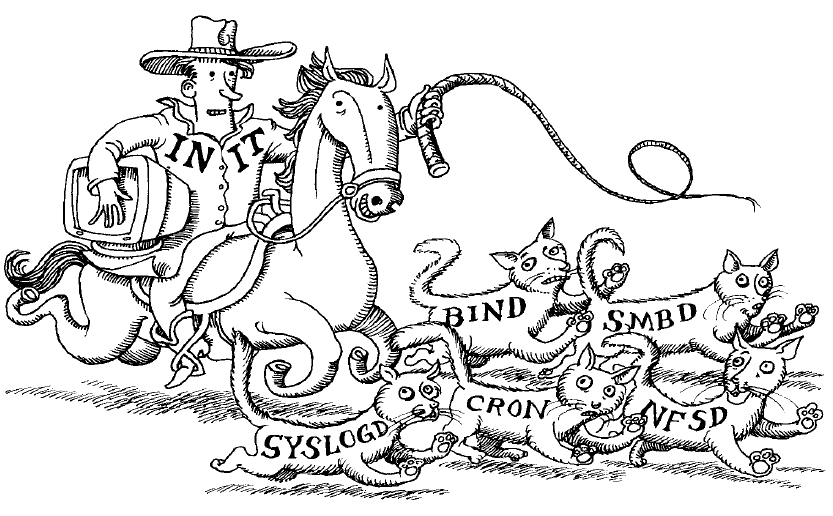
***2 Загрузка и службы управления системой***



"Загрузка" является стандартным термином для "процесса запуска компьютера". Это сокращенная форма слова bootstrapping (самозагрузка), которое следует из того, что компьютер должен "тянуть себя за ушко сапога".

Процесс загрузки состоит из нескольких широко определенных задач:

- Поиск, загрузка и запуск загрузочного кода

- Поиск, загрузка и запуск ядра ОС

- Запуск скриптов запуска и системных демонов

- Поддержание гигиены процесса и управление переходами состояния системы

Действия, включенные в последний пункт, продолжаются до тех пор, пока система остается включенной, поэтому граница между загрузкой и нормальной работой по своей сути немного размыта.

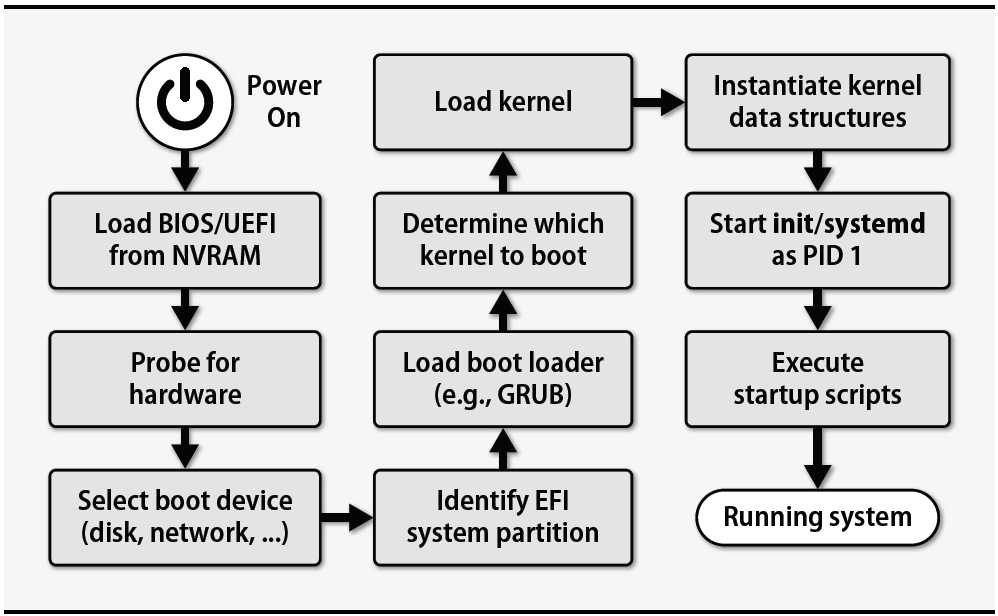
**2.1 обзор процесса загрузки**

Процедуры запуска сильно изменились за последние годы. Появление современной (UEFI) Bios упростило начальные этапы загрузки, по крайней мере, с концептуальной точки зрения. На более поздних этапах большинство дистрибутивов Linux теперь используют демон управления системой под названием **systemd** вместо традиционного **init** в Unix. **systemd** упрощает процесс загрузки, добавляя управление зависимостями, поддержку параллельных процессов запуска и комплексный подход к ведению журнала, среди прочих функций.

Управление загрузкой также изменилось по мере миграции систем в облака. Переход к виртуализации, облачным серверам и контейнеризации уменьшил потребность администраторов в физическом оборудовании. Вместо этого теперь у нас есть управление образами, API и панели управления.

Во время начальной загрузки, ядро загружается в память и начинает выполняться. Выполняются различные задачи инициализации, после чего система становится доступной для пользователей. Общий обзор этого процесса показан на рисунке А.

**Рисунок А - Процесс загрузки Linux и UNIX**



Администраторы имеют мало прямого, интерактивного контроля над большинством шагов, необходимых для загрузки системы. Вместо этого администраторы могут изменять конфигурации начальной загрузки, редактируя файлы конфигурации для скриптов запуска системы или изменяя аргументы, передаваемые загрузчиком ядру.

Перед полной загрузкой системы необходимо проверить и смонтировать файловые системы и запустить системные демоны. Эти процедуры управляются серией shell скриптов (иногда называемых "**init** скриптами") или единичными файлами, которые выполняются последовательно **init** или анализируются **systemd**. Точное расположение скриптов запуска и способ их выполнения различаются в разных системах. Мы рассмотрим их детально позже в этой главе.

**2.2 Встроенные системные микропрограммы**

Когда машина включена, аппаратный процессор выполняет загрузочный код, который хранится в ПЗУ. В виртуальных системах это "ПЗУ" может быть условным, но идея остается прежней.

Микропрограммное обеспечение системы обычно знает обо всех устройствах, подключенных к материнской плате, таких как контроллеры SATA, сетевые интерфейсы, контроллеры USB и датчики питания и температуры1. В дополнение к разрешениям аппаратной конфигурации этих устройств микропрограмма позволяет либо предоставлять их операционной системе, либо отключать и скрывать их.

*1. Виртуальные системы эмулируют этот же набор устройств.*

На физическом (в отличии от виртуального) оборудовании большинство микропрограмм предлагают пользовательский интерфейс. Однако, как правило, он бывает "сырым" и к нему немного сложно получить доступ. Для получения доступа к нему или консоли вам необходимо нажать определенную клавишу сразу после включения системы. К сожалению, "волшебные" клавиши варьируются в зависимости от производителя; посмотрите, можете ли вы заглянуть в скрытую строку инструкций в тот момент, когда система впервые включается2. Также можете пробовать нажимать клавиши Delete, Ctrl, F6, F8, F10 или F11. Чтобы получить б***о***льшие шансы на успех, нажимайте клавишу несколько раз, затем удерживайте ее нажатой.

*2. Может быть полезным временно отключить функции управления питанием дисплея.*

Во время обычной начальной загрузки системное ПО проверяет аппаратное обеспечение и диски, выполняет простой набор проверок работоспособности, а затем ищет следующий этап кода начальной загрузки. Интерфейс системного ПО позволяет указать загрузочное устройство из списка, который имеет приоритет из доступных вариантов (например, "попробуйте загрузиться с DVD-диска, потом с USB диска, затем с жесткого диска").

В большинстве случаев системные диски находятся в списоке вторичного приоритета. Для загрузки с определенного диска необходимо установить его в качестве диска с наивысшим приоритетом и убедиться, что "жесткий диск" доступен в качестве загрузочного носителя.

**BIOS против UEFI**

Традиционно системное ПО в ПК было названо BIOS, Базовая Система Ввода/Вывода (Basic Input/Output System). За последнее десятилетие, однако, BIOS был вытеснен более формализованным и современным стандартом, Унифицированным Расширяемым Программным Интерфейсом (UEFI). Вы часто будете видеть UEFI, называемый "UEFI BIOS", но для ясности, мы будем использовать термин BIOS для устаревшего стандарта в этой главе. Большинство систем, реализующих UEFI, могут вернуться к устаревшей реализации BIOS, если операционная система, которую они загружают, не поддерживает UEFI.

UEFI является текущей реализацией раннего стандарта EFI. Ссылки на имя EFI сохраняются в некоторых старых документах и даже в некоторых стандартных терминах, таких как "системный раздел EFI". Во всех, кроме самых технически точных случаях, вы можете рассматривать эти термины как равнозначные.

Поддержка UEFI в значительной степени более однородна на новом аппаратном обеспечении ПК в наши дни, но множество систем с BIOS все еще остается. Кроме того, виртуальное оборудование часто использует BIOS в качестве основного механизма загрузки, поэтому мир BIOS еще не находится под угрозой исчезновения.

Как бы мы не предпочитали игнорировать BIOS и просто говорить об UEFI, вполне вероятно, что вы столкнетесь с обоими типами систем в ближайшие годы. UEFI также имеет встроенные приспособления для старого режима BIOS, поэтому знание работы BIOS может быть довольно полезным для расшифровки документации UEFI.

**Legacy BIOS**

*Разбиение - это способ деления физических дисков. Более подробное обсуждение см. на стр. 742.*

Традиционный BIOS предполагает, что загрузочное устройство начинается с записи, называемой MBR (Master Boot Record). MBR включает в себя как загрузчик первого уровня ("блок загрузки"), так и примитивную таблицу разделов диска. Объем пространства, доступный для загрузчика, настолько мал (менее 512 байт), что он не способен сделать ничего, кроме загрузки и запуска загрузчика второго уровня.

Ни загрузочный блок, ни BIOS не достаточно сложны, чтобы читать любой тип стандартной файловой системы, поэтому загрузчик второго уровня должен храниться в месте, где его легко можно найти. В типичном сценарии блок загрузки считывает информацию о секционировании из MBR и определяет раздел диска, помеченный как "активный". Затем он считывает и выполняет загрузчик второго уровня с начала этого раздела. Эта схема называется загрузочной записью раздела.

В другом случае, загрузчик второго уровня может находиться в мертвой зоне между MBR и началом первого раздела диска. Исторически так сложилось, что первый раздел не начинается до 64-го блока диска, поэтому эта зона обычно содержит по крайней мере 32 КБ памяти: не много, но достаточно для хранения драйвера файловой системы. Эта схема хранения обычно используется загрузчиком GRUB; см. стр. 35.

Для успешной загрузки все компоненты цепочки загрузки должны быть правильно установлены и совместимы друг с другом. Может быть установлено несколько версий ОС, поскольку загрузочный блок MBR не зависит от операционной системы, и он берет на себя определение расположения загрузчика второго уровня. Загрузчик второго уровня, как правило, осведомлен об операционных и файловых системах (он может поддерживать несколько из них) и обычно имеет собственные параметры конфигурации.

**UEFI**

*Дополнительные сведения о разделах GPT см. на стр. 746.*

Спецификация UEFI включает современную схему разделения диска, известную как GPT (Таблица разделов GUID, где GUID обозначает "глобальный уникальный идентификатор"). UEFI также понимает файловые системы FAT (File Allocation Table), простую, но функциональную разметку, которая появилась в MS-DOS. Эти функции в совокупности определяют концепцию системного раздела EFI (ESP). Во время загрузки, прошивка обращается к таблице разделов GPT, чтобы определить ESP. Затем она считывает настроенное целевое приложение непосредственно из файла в ESP и выполняет его.

Поскольку ESP-это просто общая файловая система FAT, она может быть смонтирована, прочитана, записана и поддержана любой операционной системой. Не требуется наличие на диске никаких блоков загрузки "mystery meat"3.

*3. По правде говоря, UEFI поддерживает MBR-совместимую запись в начале каждого диска для облегчения взаимодействия с системами BIOS. Системы BIOS не могут видеть полную таблицу разделов в виде GPT, но они, по крайней мере, распознают диск как отформатированный. Не стоит запускать специальные средства администрирования MBR на GPT-дисках. Они могут думать, что они понимают разметку диска, но на самом деле нет.*

Фактически, никакой загрузчик технически вообще не требуется. Целью загрузки UEFI может быть ядро UNIX или Linux, сконфигурированное для прямой загрузки UEFI, таким образом производя загрузку без загрузчика. На практике, однако, большинство систем по-прежнему используют загрузчик, отчасти потому, что это облегчает поддержание совместимости с устаревшим BIOS.

UEFI сохраняет путь для загрузки из ESP в качестве параметра конфигурации. Если такой конфигурации нет, то он ищет по стандартному пути (обычно **/efi/boot/bootx64.efi**) на современных системах Intel. Более характерный путь на настроенной системе (для примера Ubuntu с загрузчиком GRUB) является **/efi/ubuntu/grubx64.efi**. Другие дистрибутивы следуют аналогичному условию.

UEFI определяет стандартные API для доступа к оборудованию системы. В этом плане UEFI выступает вроде миниатюрной операционной системы со своими порядками. Она даже обеспечивает драйверами дополнительные устройства уровня UEFI, которые написаны на языке, не зависящем от процессора и хранятся в ESP. Операционные системы могут использовать интерфейс UEFI или взять на себя непосредственное управление оборудованием.

Поскольку UEFI имеет формальный API, вы можете просматривать и изменять переменные UEFI (включая пункты меню загрузки) в работающей системе. Например, **efibootmgr -v** показывает следующий отчет по конфигурации загрузки:

$ **efibootmgr -v**

BootCurrent: 0004

BootOrder: 0000,0001,0002,0004,0003

Boot0000\* EFI DVD/CDROM PciRoot(0x0)/Pci(0x1f,0x2)/Sata(1,0,0)

Boot0001\* EFI Hard Drive PciRoot(0x0)/Pci(0x1f,0x2)/Sata(0,0,0)

Boot0002\* EFI Network PciRoot(0x0)/Pci(0x5,0x0)/MAC(001c42fb5baf,0)

Boot0003\* EFI Internal Shell MemoryMapped(11,0x7ed5d000,0x7f0dcfff)/

FvFile(c57ad6b7-0515-40a8-9d21-551652854e37)

Boot0004\* ubuntu HD(1,GPT,020c8d3e-fd8c-4880-9b61-

ef4cffc3d76c,0x800,0x100000)/File(\EFI\ubuntu\shimx64.efi)

**efibootmgr** позволяет изменить порядок загрузки, выбрать следующий настроенный параметр загрузки или даже создавать и удалять загрузочные записи. Например, установим порядок загрузки, чтобы первым загружался системный диск перед сетевой загрузкой, и игнорировались другие параметры загрузки, мы могли бы использовать команду:

$ **sudo efibootmgr -o 0004,0002**

Здесь мы указываем параметры Boot0004 и Boot0002 из данных полученных выше.

Возможность изменять конфигурацию UEFI из пользовательского пространства означает, что информация о конфигурации встроенного ПО доступна для чтения и записи – это и хорошо и плохо. В системах (которые как правило используют **systemd**), которые по умолчанию предоставляют права записи, команда **rm -rf /** может быть достаточной, чтобы окончательно уничтожить систему на уровне встроенного ПО; в дополнение к удалению файлов, **rm** также удаляет переменные и другую информацию UEFI, доступную через **/sys**4. Не пытайтесь это повторять!

*4. См. goo.gl/QMSiSG (ссылка на статью Phoronix) чтобы получить дополнительные детали.*

**2.3 Загрузчики**

Большинство процедур начальной загрузки включает запуск загрузчика, который отличается от кода BIOS/UEFI и ядра ОС. Он также отделен от начального загрузочного блока в системе BIOS.

Основная задача загрузчика - определить и загрузить соответствующее ядро операционной системы. Большинство загрузчиков могут также представлять пользовательский интерфейс во время загрузки, который позволяет выбрать и запустить из нескольких возможных ядер или операционных систем.

Другая задача, которая относится к загрузчику - упорядочивание параметров конфигурации ядра. Ядро не имеет командной строки как таковой, его обработка параметров запуска будет казаться сильно похожей на shell. Например, аргумент **single** или **-s** обычно указывает ядру войти в однопользовательский режим, а не выполнять нормальный процесс загрузки.

Такие параметры могут быть жестко привязаны к конфигурации загрузчика если вы хотите, чтобы они использовались при каждой загрузке, или они могут быть подставлены "на лету" через пользовательский интерфейс загрузчика.

В следующих нескольких разделах мы обсудим GRUB (главный загрузчик в мире Linux) и загрузчики, используемые с FreeBSD.

**2.4 GRUB: GRand Unified Boot loader**

GRUB, разработанный проектом GNU, является загрузчиком по умолчанию в большинстве дистрибутивов Linux. Семейство GRUB имеет два основных направления: оригинальный GRUB, теперь называемый Grub Legacy, и более новый GRUB 2, который сейчас является стандартом. Убедитесь, что вы знаете, с какой GRUB вы имеете дело, так как две версии совершенно разные.

GRUB 2 был загрузочным менеджером по умолчанию для Ubuntu с версии 9.10, и он недавно стал дефолтным для RHEL 7. Все наши примеры дистрибутивов Linux используют его по умолчанию. В этой книге мы обсуждаем только GRUB 2, и мы называем его просто GRUB.

У FreeBSD есть собственный загрузчик (более подробно описан на стр. 39). Однако GRUB отлично справляется с загрузкой FreeBSD. Эта конфигурация может оказаться полезной, если вы планируете загружать несколько операционных систем на одном компьютере. В противном случае загрузчик FreeBSD более чем достаточен.

**Конфигурация GRUB**

*Подробнее о режимах работы см. стр. 49.*

GRUB позволяет указать такие параметры, как загрузочное ядро (указанное в качестве записи меню GRUB) и режим работы для загрузки.

Поскольку эта информация о конфигурации необходима во время загрузки, вы можете себе представить, что она будет храниться где-то в странном месте, например, в NVRAM системы или в дисковых блоках, зарезервированных для загрузчика. На самом деле, GRUB понимает большинство используемых файловых систем и обычно может самостоятельно найти корневую файловую систему. Этот навык позволяет GRUB читать его конфигурацию из обычного текстового файла.

Во время загрузки GRUB может потребоваться доступ к конфигурационному файлу **grub.cfg**, который обычно хранится в **/boot/grub** (**/boot/grub2** в Red Hat и CentOS), наравне с выбором других ресурсов и модулей кода5 в которые GRUB может понадобиться доступ во время загрузки. Изменить конфигурацию загрузки просто – достаточно обновить файл **grub.cfg**.

*5. Если вы знакомы с условностями файловой системы UNIX (см. Главу 5, "Файловая система", начиная со страницы 120), вы можете задаться вопросом, почему* ***/boot/grub*** *не назван более стандартно, например* ***/var/lib/grub*** *или* ***/usr/local/etc/grub****. Причина в том, что драйверы файловой системы, используемые во время загрузки, несколько упрощены. Загрузчики не могут обрабатывать дополнительные функции, такие как точки монтирования, когда они перемещаются в файловой системе. Все, что находится в* ***/boot*** *должно быть простым файлом или каталогом.*

Если Вы не примете меры, чтобы предотвратить это, ваша ручная работа.cfg файл будет затерт.

Хотя вы можете сами создать файл **grub.cfg**, чаще всего его лучше сгенерировать с помощью утилиты **grub-mkconfig**, которая называется **grub2-mkconfig** в Red Hat и CentOS и обернута в **update-grub** в Debian и Ubuntu. Фактически, большинство дистрибутивов предполагают, что **grub.cfg** может быть восстановлен по желанию, и они делают это автоматически после обновлений. Если Вы не примете меры, чтобы это предотвратить, ваш созданный вручную файл **grub.cfg** будет затерт.

Как и везде в Linux, дистрибутивы конфигурируют **grub-mkconfig** различными способами. Чаще всего конфигурация задается в **/etc/default/grub** в виде присвоенных переменных **sh**. В таблице 2.1 показаны некоторые из часто изменяемых опций.

**Таблица 2.1 Общие параметры конфигурации GRUB в /etc/default/grub**

|  |  |
| --- | --- |
| **Названия переменных shell** | **Содержание или функция** |
| GRUB\_BACKGROUND | Фоновое изображениеa |
| GRUB\_CMDLINE\_LINUX | Параметры ядра для добавления в записи меню для Linuxb |
| GRUB\_DEFAULT | Номер или название элемента меню по умолчанию |
| GRUB\_DISABLE\_RECOVERY | Предотвращает создание записей режима восстановления |
| GRUB\_PRELOAD\_MODULES | Список модулей GRUB, которые необходимо загрузить как можно раньше |
| GRUB\_TIMEOUT | Секунды для отображения меню загрузки перед автозагрузкой |

*а. Файл фоновое изображения должен быть в формате* ***.png****,* ***.tga****,* ***.jpg*** *или* ***.jpeg****.*

*b. В таблице 2.3 на стр. 38 перечислены некоторые из доступных опций.*

После редактирования **/etc/default/grub** запустите **update-grub** или **grub2-mkconfig**, чтобы перевести вашу конфигурацию в правильный файл **grub.cfg**. В рамках процесса построения конфигурации эти команды описывают загрузочные ядра системы, поэтому они могут быть полезны для запуска после внесения изменений в ядро, даже если вы явно не изменили конфигурацию GRUB.

Возможно, вам потребуется отредактировать файл **/etc/grub.d/40\_custom**, чтобы изменить порядок, в котором ядра указаны в меню загрузки (например, после создания своего ядра), установить пароль для загрузки или изменить имена пунктов меню загрузки. Как обычно, после внесения изменений запустите **update-grub** или **grub2-mkconfig**.

В качестве примера рассмотрим файл **40\_custom**, который вызывает пользовательское ядро в системе Ubuntu:

#!/bin/sh

exec tail -n +3 $0

# This file provides an easy way to add custom menu entries. Just type

# the menu entries you want to add after this comment. Be careful not to

# change the 'exec tail' line above.

menuentry 'My Awesome Kernel' {

set root='(hd0,msdos1)'

linux /awesome\_kernel root=UUID=*XXX-XXX-XXX* ro quiet

initrd /initrd.img-awesome\_kernel

}

*См. стр. 122 для получения дополнительной информации об установке файловых систем.*

В этом примере GRUB загружает ядро из **/awesome\_kernel**. Пути ядра относятся к загрузочному разделу, который исторически был установлен как **/boot**, но с появлением UEFI теперь, скорее всего, является демонтированным системным разделом EFI. Используйте **gpart show** и **mount** для проверки вашего диска и определения состояния загрузочного раздела.

**Командная строка GRUB**

GRUB поддерживает интерфейс командной строки для редактирования записей в файле конфигурации "на лету" во время загрузки. Чтобы войти в режим командной строки, введите "**c**" на экране загрузки GRUB.

В командной строке вы можете загружать операционные системы, которые не указаны в файле **grub.cfg**, отображать системную информацию и выполнять элементарное тестирование файловой системы. Все, что можно сделать через **grub.cfg**, также можно выполнить с помощью командной строки.

Нажмите клавишу <Tab>, чтобы просмотреть список возможных команд. В таблице 2.2 на следующей странице показаны некоторые из наиболее полезных команд.

Для получения подробной информации о GRUB и параметрах командной строки см. официальное руководство по адресу gnu.org/software/grub/manual.

**Таблица 2.2 Команды GRUB**

|  |  |
| --- | --- |
| **Команда** | **Функция** |
| boot | Загружает систему из указанного образа ядра |
| help | Получает интерактивную помощь для команды |
| linux | Загружает ядро Linux |
| reboot | Перезагружает систему |
| search | Ищет устройства по файлу, метке файловой системы или UUID |
| usb | Тестирует поддержку USB |

**Параметры ядра Linux**

*Подробнее о параметрах ядра см. в Главе 11.*

Параметры запуска ядра обычно изменяют значения параметров ядра, предписывают ядру проверять определенные устройства, указывают путь к процессу **init** или **systemd** или назначают определенное корневое устройство. В таблице 2.3 приведено несколько примеров.

**Таблица 2.3 Примеры параметров времени загрузки ядра**

|  |  |
| --- | --- |
| **Параметр** | **Значение** |
| debug | Включает отладку ядра |
| init=/bin/bash | Запускает только оболочку **bash**; полезно для аварийного восстановления |
| root=/dev/foo | Сообщает ядру использовать **/dev/foo** в качестве основного устройства |
| single | Загрузка в однопользовательском режиме |

Если параметры ядра указаны во время загрузки, они не являются постоянными. Отредактируйте соответствующую строку ядра в **/etc/grub.d/40\_custom** или **/etc/defaults/grub** (переменная с именем GRUB\_CMDLINE\_LINUX), чтобы сохранить изменение после перезагрузки.

Патчи безопасности, исправления ошибок и новые функции регулярно добавляются в ядро Linux. Однако, в отличие от других программных пакетов, новые версии ядра обычно не заменяют старые. Вместо этого новые ядра устанавливаются рядом с предыдущими версиями, так что вы можете вернуться к старому ядру в случае возникновения проблем.

Это соглашение помогает администраторам отказаться от обновления, если патч ядра нарушает их систему, хотя это также означает, что загрузочное меню имеет тенденцию загромождаться старыми версиями ядра. Попробуйте выбрать другое ядро, если ваша система не загружается после обновления.

**2.5 Процесс загрузки FreeBSD**

Система загрузки FreeBSD во многом похожа на GRUB, поскольку загрузчик последнего шага (называемый **loader**) использует файл конфигурации на основе файловой системы, поддерживает меню и предлагает интерактивный режим командной строки. **loader** - это последний общий путь для обоих путей загрузки BIOS и UEFI.

**Путь BIOS: boot0**

Как и в случае с GRUB, полная среда **loader** слишком велика, чтобы поместиться в загрузочный блок MBR, таким образом цепочка все более и более сложных предварительных загрузчиков запускает **loader** в системе BIOS.

GRUB объединяет все эти компоненты под общим названием "GRUB", но в FreeBSD ранние загрузчики являются частью отдельной системы под названием boot0, которая используется только в системах BIOS. **boot0** имеет свои собственные опции, в основном потому, что он хранит более поздние этапы цепочки загрузки в загрузочной записи тома (см. *Legacy BIOS* на стр. 33), а не перед первым разделом диска.

По этой причине загрузочная запись MBR нуждается в указателе на раздел, который она должна использовать для продолжения процесса загрузки. Обычно, все это автоматически настраивается для вас как часть процесса установки FreeBSD, но если вам когда-либо понадобится изменить конфигурацию, вы можете сделать это с помощью команды **boot0cfg**.

**Путь UEFI**

В системах UEFI FreeBSD создает системный раздел EFI и устанавливает в нем код загрузки по пути **/boot/bootx64.efi**6. Это путь по умолчанию, который проверяет UEFI во время загрузки (по крайней мере на современных ПК), поэтому не требуется никакая конфигурация на уровне встроенных программ, за исключением случая, когда необходимо убедиться, что приоритеты загрузочных устройств выставлены правильно.

*6. Не путайте каталог* ***/boot*** *системного раздела EFI с корневым каталогом* ***/boot*** *в файловой системе FreeBSD. Они разделены и служат различным целям, хотя, конечно, оба связаны с загрузкой.*

По умолчанию FreeBSD не сохраняет системный раздел EFI после загрузки. Вы можете проверить таблицу разделов с помощью **gpart**, чтобы его выявить:

**$ gpart show**

=> 40 134217648 ada0 GPT (64G)

40 1600 1 efi (800K)

1640 127924664 2 freebsd-ufs (61G)

127926304 6291383 3 freebsd-swap (3.0G)

134217687 1 - free - (512B)

*См. стр. 122 для получения дополнительной информации об установке файловых систем.*

Хотя вы можете смонтировать ESP, если вам интересно посмотреть, что в нем находится (используйте опцию **-t msdos** команды **mount**), вся файловая система на самом деле является просто копией образа, найденного в **/boot/boot1.efifat** на корневом диске. Внутри нет компонентов, полезных для пользователя.

Если раздел ESP поврежден или удален, его можно повторно создать, настроив раздел с помощью **gpart** и затем скопировать в образ файловой системы с помощью **dd**:

**$ sudo dd if=/boot/boot1.efifat of=/dev/ada0p1**

Как только загрузчик UEFI первого уровня находит раздел типа *freebsd-ufs*7, он загружает UEFI версию **loader** из **/boot/loader.efi**. Оттуда загрузка выполняется как в BIOS, с **loader**’ом, определяющим ядро для загрузки, какие параметры ядра установить и так далее.

*7. Начиная с FreeBSD 10.1, можно использовать ZFS в качестве корневого раздела в системе UEFI.*

**Настройка loader**

**loader** на самом деле является скриптовой средой, а язык сценариев – Forth8. Внутри **/boot** хранится куча кода Forth, который управляет операциями **loader**’а, но он разработан быть автономным - вам не нужно изучать Forth.

*8. Этот факт покажется вам замечательным и интересным, если вы историк языков программирования.*

Скрипты Forth запускают **/boot/loader.conf**, чтобы получить значения переменных конфигурации, так что в этом месте вам нужно будет вносить изменения. Не путайте этот файл с **/boot/defaults/loader.conf**, который содержит настройки по умолчанию и не предназначен для изменения. К счастью, присваивания переменных в **loader.conf** синтаксически похожи на стандартные назначения **sh**.

man-страницы для **loader** и **loader.conf** загрязняют все параметры загрузчика и переменные конфигурации, которые их контролируют. Некоторые из наиболее интересных настроек включают опции, которые защищают меню загрузки паролем, меняют экран заставки, отображаемый при загрузке и передают параметры ядра.

**Команды loader**

**loader** понимает различные интерактивные команды. Например, чтобы найти и загрузить альтернативное ядро, вы должны использовать последовательность команд:

Type ‘?’ for a list of commands, ‘help’ for more detailed help.

OK **ls**

/

d .snap

d dev

...

d rescue

l home

...

OK **unload**

OK **load /boot/kernel/kernel.old**

/boot/kernel/kernel.old text=0xf8f898 data=0x124 ... b077]

OK **boot**

Здесь мы перечислили содержимое (по умолчанию) корневой файловой системы, выгрузили ядро по умолчанию (**/boot/kernel/kernel**), загрузили старое ядро (**/boot/kernel/kernel.old**) и затем продолжили процесс загрузки.

*Полную документацию по доступным командам смотрите в* ***man loader****.*

**2.6 Демоны управления системой**

Как только ядро было загружено завершен процесс инициализации, оно создает дополнительные "спонтанные" процессы в пользовательском пространстве. Они называются спонтанными процессами, потому что ядро запускает их автономно - в нормальном ходе событий новые процессы создаются только по воле существующих процессов.

Большинство спонтанных процессов действительно являются частью реализации ядра. Они не обязательно соответствуют программам в файловой системе. Они не настраиваются и не требуют внимания администратора. Вы можете распознать их в списках **ps** (см. стр. 98) по их низким PID и скобкам вокруг их имен (например, [pagedaemon] в FreeBSD или [kdump] в Linux).

Исключением из этого шаблона является демон управления системой. Он имеет ID процесса 1 и обычно работает под именем **init**. Система дает **init** несколько специальных привилегий, но по большей части это просто программа на уровне пользователя, как любой другой демон.

**Обязанности init**

У **init** есть несколько функций, но его главная цель – убедиться в том, что система запускает правильный набор сервисов и демонов в любой момент времени.

Для достижения этой цели **init** хранит режим, в котором система должна работать. Некоторые общепринятые режимы:9

- Однопользовательский режим, в котором монтируется только минимальный набор файловых систем, службы не запущены, а в консоли запущена root оболочка

- Многопользовательский режим, в котором смонтированы все привычные файловые системы и запущены все настроенные сетевые сервисы, а также оконная система и графический менеджер входа в консоль

- Режим сервера, аналогичный многопользовательскому режиму, но без использования GUI, запущена только консоль

*9. Не воспринимайте эти имена или описания режимов слишком буквально; они всего лишь примеры общих режимов работы, которые большинство систем определяют так или иначе.*

Каждый режим связан с определенным набором системных служб, демон инициализации запускает или останавливает службы по мере необходимости, чтобы привести фактическое состояние системы в соответствие с текущим активным режимом. Режимы могут также иметь набор связанных задач, которые запускаются всякий раз, когда начинается или заканчивается режим.

В качестве примера, **init** обычно заботится о многих различных задачах запуска в качестве побочного эффекта перехода от начальной загрузки к многопользовательскому режиму. Эти задачи могут включать:

- Установка имени компьютера

- Установка часового пояса

- Проверка дисков с помощью **fsck**

- Монтирование файловых систем

- Удаление старых файлов из каталога **/tmp**

- Настройка сетевых интерфейсов

- Настройка фильтров пакетов

- Запуск других демонов и сетевых служб

У **init** очень мало встроенных знаний об этих задачах. Обычно запускается набор команд или скриптов, которые были назначены для выполнения в этом конкретном контексте.

**Реализации init**

В настоящее время широко используются три совершенно разных процесса управления системой:

- **init,** стилизованный под **init** от AT&T System V UNIX, который мы называем "традиционный **init**". Этот **init** был преобладающим и использовался в системах Linux до выхода **systemd**.

- Вариант **init**, производный от BSD UNIX, используется в большинстве BSD-систем, включая FreeBSD, OpenBSD и NetBSD. Он так же хорош, как и SysV **init**, и имеет много претензий на то, чтобы его называли "традиционным", но для ясности мы называем его "BSD **init**". Этот вариант достаточно прост по сравнению с **init** в стиле SysV. Мы обсудим его отдельно, начиная со страницы 57.

- Более поздний соперник, названный **systemd**, который стремится стать универсальным средством решения всех проблем, связанных с демонами и состояниями системы. Как следствие, **systemd** решает гораздо больше вопросов, чем любая существующая версия **init**. Это вызывает противоречия, о них мы говорим позже. Тем не менее, все примеры дистрибутивов Linux в этой книге уже используют **systemd**.

Хотя эти реализации и преобладают сегодня, но они далеки от того, чтобы остаться единственными. Например, MacOS от Apple использует систему с именем **launchd**. Пока не был принят **systemd**, Ubuntu использовал другой современный вариант **init** под названием **Upstart**.

В системах Linux теоретически можно заменить **init**, используемый по умолчанию в вашей системе на любой другой вариант, какой предпочтете. Но на практике **init** имеет настолько фундаментальное значение для работы системы, что без него большое количество дополнительного программного обеспечения может сломаться. Если вы не можете терпеть **systemd**, используйте дистрибутив, в котором его нет.

**Традиционный init**

В мире традиционного **init**, системные режимы (например, однопользовательский или многопользовательский) известны как "уровни запуска" (run levels). Большинство уровней запуска обозначаются одной буквой или цифрой.

Традиционный **init** существует с начала 80-х годов, и седые уже представители лагеря анти-**systemd**, часто ссылаются на принцип "Если работает, не трогай". Тем не менее, традиционный **init** имеет ряд заметных недостатков.

Начнем с того, что традиционный **init** сам по себе не настолько силен, чтобы справляться с потребностями современных систем. Большинство систем, которые его используют, на самом деле имеют стандартную и фиксированную конфигурацию **init**, которая никогда не меняется. Эта конфигурация указывает на второй уровень shell-скриптов, которые выполняют фактическую работу по изменению уровней запуска и позволяют администраторам вносить изменения в конфигурацию.

Второй уровень скриптов поддерживает еще третий уровень скриптов, специфичных для демонов и систем, которые перекрестно связаны с каталогами уровней запуска, которые указывают, какие службы должны предположительно работать на каком уровне запуска. Все это выглядит немного банально и некрасиво.

Более конкретно, эта система не имеет общей модели зависимостей между службами, поэтому она требует, чтобы все запуски и остановы выполнялись в числовом порядке, который поддерживается администратором. Последующие действия не могут выполняться до тех пор, пока предыдущие действия не завершатся, поэтому невозможно выполнять действия параллельно и системе требуется много времени для изменения состояний.

**systemd против всего мира**

Немногие проблемы в Linux обсуждались также жарко, как миграция с традиционного **init** на **systemd**. По большей части, жалобы сосредотачиваются на постоянно растущих размерах **systemd**.

**systemd** берет все функции **init**, ранее реализованные "на коленке", ломает shell-скрипты, заставляет попотеть администраторов и оформляет их в единую теорию о том, как настраивать сервисы, получать к ним доступ и управлять ими.

*Дополнительные сведения об управлении пакетами см. в Главе 6 "Установка программного обеспечения и управление им".*

Подобно системе управления пакетами, **systemd** определяет надежную модель зависимостей не только среди сервисов, но и среди "целей" - термин **systemd** для режимов работы, которые традиционный **init** называет уровнями запуска. **systemd** не только управляет процессами параллельно, но также управляет сетевыми подключениями (**networkd**), записями журнала ядра (**journald**) и входами в систему (**logind**).

Лагерь анти-**systemd** приводит аргументы, что философия UNIX заключается в том, чтобы сохранить системные компоненты небольшими, простыми и модульными. Говорят, что компонент, такой же важный как **init**, не должен иметь единого контроля над многими другими подсистемами ОС. **systemd** не только увеличивает сложность, но также создает потенциальные проблемы в безопасности и запутывает отличия между платформой ОС и службами, которые работают поверх нее.

**systemd** также критикуется за введение новых стандартов и функций в ядро Linux, за его качество кода, за предполагаемую невосприимчивость своих разработчиков к сообщениям об ошибках, за функциональный дизайн его основных функций и за то, что они заставляют пользователей смешно выглядеть. Мы не можем справедливо решить эти проблемы в книге, но вы можете найти много "доводов против" в разделе **systemd** на сайте without-systemd.org, главном сайте ненависти к **systemd** в Интернете.

**Cуд и надлежащее наказание для init**

Архитектурные возражения против **systemd**, описанные выше, являются обоснованными. **systemd** действительно отражает большую часть недостатков перегруженного ПО.

Однако, на практике, многие администраторы очень любят **systemd**, и мы тоже непосредственно относимся к этому лагерю. Проигнорируйте разногласия на некоторое время и дайте **systemd** шанс вам понравиться. Как только вы привыкнете к нему, скорее всего окажется, что его многочисленные достоинства перевешивают его недостатки.

В конечном счете имейте в виду, что традиционный **init**, вытесняемый **systemd**, не был национальным достоянием. **systemd** привносит с собой некоторую ценность, просто устраняя несколько ненужных различий между дистрибутивами Linux.

Споры уже не имеют значения, потому что переход на **systemd** уже завершен. Ссора была эффектно улажена, когда Red Hat, Debian и Ubuntu перешли на **systemd**. Многие другие дистрибутивы Linux в настоящее время адаптируют **systemd**, по своему выбору или впихивая ногами и криками через родительские дистрибутивы.

Традиционный **init** по-прежнему играет определенную роль, когда дистрибутив либо нацелен на небольшой размер установки, либо не нуждается в расширенных функциях управления процессами **systemd**. Существует также значительное количество реваншистов, которые не используют **systemd** в принципе, поэтому некоторые дистрибутивы Linux обязательно сохранят традиционный **init** в течение неопределенного времени в качестве формы протеста.

Тем не менее, мы не считаем, что традиционный **init** имеет большое будущее, чтобы заслужить подробное обсуждение в этой книге. Для Linux мы в основном ограничиваемся **systemd**. Мы также обсуждаем, к счастью простую, систему, используемую в FreeBSD, начиная со страницы 57.

**2.7 systemd в деталях**

Конфигурация и управление системными службами — это область, в которой дистрибутивы Linux традиционно очень сильно отличаются друг от друга. **systemd** нацелен на стандартизацию этого аспекта системного администрирования, и для этого он продвигается дальше к нормальной работе системы, чем любая предыдущая альтернатива.

**systemd** — это не один демон, а набор программ, демонов, библиотек, технологий и компонентов ядра. Сообщение в блоге **systemd** на странице 0pointer.de/blog отмечает, что полная сборка проекта генерирует 69 разных **bin-**файлов.

**systemd** сильно зависит от особенностей ядра Linux, поэтому вы не увидите его в BSD или любых других вариантах UNIX в течение следующих пяти лет.

**Юниты и юнит-файлы**

Сущность, которой управляет **systemd**, известна как юнит. Более конкретно, юнит может быть "службой, сокетом, устройством, точкой монтирования, точкой автоматического монтирования, файлом или разделом подкачки, целью запуска, отслеживаемым путем в файловой системе, таймером, контролируемым и управляемым **systemd**, срезом управления ресурсами, группой внешних процессов или червоточиной, ведущей в альтернативную вселенную"10. Итак, мы составили часть альтернативной вселенной, но она все еще охватывает много территории.

*10. В основном цитируется на man-странице* ***systemd.unit***

Внутри **systemd** поведение каждого юнита определяется и настраивается юнит-файлом. Например, в случае службы юнит-файл указывает местоположение исполняемого файла для демона, сообщает **systemd** как запускать и останавливать службу, и идентифицирует любые другие юниты, от которых зависит служба.

*См. стр. 594 для получения дополнительной информации о* ***rsync****.*

Вскоре мы рассмотрим синтаксис юнит-файлов, вот простой пример из системы Ubuntu. Это юнит-файл - **rsync.service**; он обрабатывает запуск демона **rsync**, который зеркалирует файловые системы.

[Unit]

Description=fast remote file copy program daemon

ConditionPathExists=/etc/rsyncd.conf

[Service]

ExecStart=/usr/bin/rsync --daemon --no-detach

[Install]

WantedBy=multi-user.target

Если вы узнали в нем формат файла, используемый **.ini** файлами в MS-DOS, то вы хорошо понимаете как **systemd**, так и страдания ненавистников **systemd**.

Юнит файлы могут находиться в нескольких разных местах. **/usr/lib/systemd/system** - основное место, куда пакеты складывают свои юнит-файлы во время установки; в некоторых системах путь может отличаться - **/lib/systemd/system**. Содержимое этого каталога считается запасным, поэтому изменять его не следует. Ваши локальные юнит-файлы и настройки могут храниться в **/etc/systemd/system**. Также существует директория юнит-файлов **/run/systemd/system**, которая является "перевалочным пунктом" для временных юнит-файлов.

**systemd** поддерживает телескопическое представление всех этих каталогов, поэтому они в значительной степени одинаковы. Если появляется конфликт, то файлы в **/etc** имеют наивысший приоритет.

По соглашению, в названии юнит-файлов присутствует суффикс, который зависит от типа настраиваемого юнита. Например, юниты сервисов имеют суффикс **.service**, а таймеры используют **.timer**. Внутри юнит-файла некоторые разделы (например, [Unit]) применяются в общем случае ко всем типам юнитов, но другие (например, [Service]) могут отображаться только в контексте определенного типа юнита.

**systemctl: управление systemd**

*См. стр. 235 для получения дополнительной информации о Git.*

**systemctl** — это универсальная команда для изучения состояния **systemd** и внесения изменений в его конфигурацию. Как и в случае с Git и несколькими другими сложными пакетами программного обеспечения, первый аргумент **systemctl** обычно представляет собой подкоманду, которая задает план действий, а последующие аргументы являются специфическими для этой конкретной подкоманды. Подкоманды могут быть командами верхнего уровня сами по себе, но для согласованности и ясности они объединены в общий **systemctl**.

Запуск **systemctl** без каких-либо аргументов вызывает подкоманду **list-units** по умолчанию, которая отображает все загруженные и активные службы, сокеты, цели, точки монтирования и устройства. Чтобы показать только загруженные и активные службы, используйте ключ **--type=service**:

**$ systemctl list-units --type=service**

UNIT LOAD ACTIVE SUB DESCRIPTION

accounts-daemon.service loaded active running Accounts Service

...

wpa\_supplicant.service loaded active running WPA supplicant

Также иногда полезно видеть все установленные юнит-файлы, независимо от того, активны они или нет:

**$ systemctl list-unit-files --type=service**

UNIT FILE STATE

...

cron.service enabled

cryptdisks-early.service masked

cryptdisks.service masked

cups-browsed.service enabled

cups.service disabled

...

wpa\_supplicant.service disabled

x11-common.service masked

188 unit files listed.

Для подкоманд, действующих на определенный юнит (например, **systemctl status**), **systemctl** обычно может принимать имя юнита без суффикса (например, **cups** вместо **cups.service**). Тем не менее, тип юнита по умолчанию, с которым объединены простые имена, зависит от подкоманды.

В таблице 2.4 показаны наиболее распространенные и полезные подкоманды **systemctl**. Для получения полного списка см. man-страницу **systemctl**.

**Таблица 2.4 Обычно используемые подкоманды systemctl**

|  |  |
| --- | --- |
| **Подкоманда** | **Функция** |
| **list-unit-files [** *шаблон* **]** | Показывает установленные юниты; шаблон необязателен |
| **enable** *unit* | Активирует юнит при загрузке |
| **disable** *unit* | Деактивирует юнит при загрузке |
| **isolate** *target* | Изменяет режим работы на *target* |
| **start** *unit* | Немедленно запускает юнит |
| **stop** *unit* | Немедленно останавливает юнит |
| **restart** *unit* | Перезапускает юнит (или запускает, если он остановлен). |
| **status** *unit* | Показывает статус юнита и последние записи лога |
| **kill** *шаблон* | Отправляет сигнал юниту, соответствующему шаблону |
| **reboot** | Перезагружает компьютер |
| **daemon-reload** | Перезагружает юнит-файлы и конфигурацию **systemd** |

**Статусы юнитов**

В выводе **systemctl list-unit-files** выше мы видим, что **cups.service** отключен. Мы можем использовать **systemctl status** для получения более подробной информации:

**$ sudo systemctl status -l cups**

cups.service - CUPS Scheduler

Loaded: loaded (/lib/systemd/system/cups.service; disabled; vendor

preset: enabled)

Active: inactive (dead) since Sat 2016-12-12 00:51:40 MST; 4s ago

Docs: man:cupsd(8)

Main PID: 10081 (code=exited, status=0/SUCCESS)

Dec 12 00:44:39 ulsah systemd[1]: Started CUPS Scheduler.

Dec 12 00:44:45 ulsah systemd[1]: Started CUPS Scheduler.

Dec 12 00:51:40 ulsah systemd[1]: Stopping CUPS Scheduler...

Dec 12 00:51:40 ulsah systemd[1]: Stopped CUPS Scheduler.

Здесь **systemctl** показывает нам, что служба в настоящее время неактивна (dead) и сообщает нам, когда процесс умер (всего несколько секунд назад; мы отключили его для примера.) Он также показывает (в разделе с надписью Loaded), что служба по умолчанию включена при запуске, но теперь она отключена.

Последние четыре строки — это последние записи лога. По умолчанию записи журнала сжимаются так, что каждая запись занимает только одну строку. Это сжатие часто делает записи нечитаемыми, поэтому мы включили опцию **-l** для запроса на получение полных записей. В этом случае нет никакой разницы, но это будет полезной привычкой.

В таблице 2.5 представлены статусы, с которыми вы чаще всего будете сталкиваться при проверке юнитов.

**Таблица 2.5 Статусы юнит-файлов**

|  |  |
| --- | --- |
| **Подкоманда** | **Функция** |
| bad | Некоторая проблема в **systemd**; обычно плохой юнит-файл |
| disabled | Существует, но не настроен для автоматического запуска |
| enabled | Установлен и готов к запуску; запустится автоматически |
| indirect | Отключен, но имеет равные узлы в *Also*, которые могут быть включены |
| linked | Юнит-файл, доступен через символическую ссылку |
| masked | Изгнан из мира **systemd** с логической точки зрения |
| static | Зависит от другого юнита, нет требований к установке |

Состояния "enabled" и "disabled" применяются только к юнит-файлам, которые находятся в одном из системных каталогов **systemd** (то есть, они не связаны символьной ссылкой) и имеют раздел [Install] в своих юнит-файлах. Юниты "enabled", по-видимому, должны считаться "installed" (установленными), что означает, что директивы в разделе [Install] были выполнены и что юнит подключен к его обычным триггерам активации. В большинстве случаев это состояние заставляет юнит активироваться автоматически после загрузки системы.

Аналогично, состояние "disabled" имеет "неправильное" название, потому что единственное, что фактически отключено — это нормальный путь активации. Вы можете вручную активировать "disabled" юнит, запустив **systemctl start**, **systemd** жаловаться не будет.

У многих юнитов нет процедуры установки, поэтому нельзя сказать, что они "disabled" или "enabled", они просто доступны. Статус таких юнитов указан как статический. Они становятся активными только при активации вручную (**systemctl start**) или называются зависимыми от других активных единиц.

Связанные ссылкой юнит-файлы были созданы с помощью **systemctl link**. Эта команда создает символическую ссылку из одного из каталогов **systemd** **system** на юнит-файл, который находится в другом месте файловой системы. Такие юнит-файлы могут адресоваться командами или называться зависимостями, но они не являются полноправными представителями экосистемы и имеют некоторые заметные причуды. Например, при запуске **systemctl disable**, эта команда удаляет ссылки и отношения в юнит-файле.

К сожалению, точное поведение связанных юнит-файлов недостаточно хорошо документировано. Хотя идея хранить локальные юнит-файлы в отдельном репозитории и связывать их с **systemd** имеет определенную привлекательность, но это не лучший подход на данный момент. Просто делайте копии.

Статус "masked" означает "административно заблокирован". **systemd** знает о юните, но ему запрещено активировать его или действовать по любой из его конфигурационных директив с помощью маски **systemctl**. Как правило, отключайте юниты со статусом "enabled" или "linked" с **systemctl disable** и резервируйте **systemctl mask** для юнитов "static".

Возвращаясь к нашему исследованию службы **cups**, мы могли бы использовать следующие команды для повторного ее включения и запуска:

**$ sudo systemctl enable cups**

Synchronizing state of cups.service with SysV init with /lib/systemd/

systemd-sysv-install...

Executing /lib/systemd/systemd-sysv-install enable cups

insserv: warning: current start runlevel(s) (empty) of script `cups’

overrides LSB defaults (2 3 4 5).

insserv: warning: current stop runlevel(s) (1 2 3 4 5) of script `cups’

overrides LSB defaults (1).

Created symlink from /etc/systemd/system/sockets.target.wants/cups.socket

to /lib/systemd/system/cups.socket.

Created symlink from /etc/systemd/system/multi-user.target.wants/cups.

path to /lib/systemd/system/cups.path.

**$ sudo systemctl start cups**

**Цели**

Юнит-файлы могут объявлять свои связи с другими юнитами различными способами. Например, на странице 45 предложение *WantedBy* говорит, что если в системе есть юнит **multisuser.target** - он должен зависеть от юнита - **rsync.service**, когда он находится в состоянии "enabled".

Поскольку юниты непосредственно поддерживают управление зависимостями, то для реализации похожих уровней запуска **init** не требуется никаких дополнительных механизмов. Для ясности **systemd** определяет отдельный класс юнитов (типа **.target**), чтобы действовать как хорошо известные маркеры для общих режимов работы. Однако цели не имеют никаких сверхспособностей, кроме управления зависимостями, которое доступно любому другому юниту.

Традиционный **init** определяет как минимум семь числовых уровней запуска, но многие из них на самом деле не используются. Возможно, необдуманным жестом в сторону исторической преемственности, **systemd** определяет цели, которые являются прямыми аналогами уровней запуска **init** (**runlevel0.target** и т.д.). Он также определяет мнемонические цели для повседневного использования, такие как **poweroff.target** и **graphical.target**. В таблице 2.6 сопоставляются уровнями запуска **init** и цели **systemd**.

**Таблица 2.6 Сопоставление уровней запуска init целям systemd**

|  |  |  |
| --- | --- | --- |
| **Уровень запуска** | **Цель** | **Описание** |
| 0 | **poweroff.target** | Остановка системы |
| emergency | **emergency.target** | Голая оболочка для восстановления системы |
| 1, s, single | **rescue.target** | Однопользовательский режим |
| 2 | **multi-user.targeta** | Многопользовательский режим (командная строка) |
| 3 | **multi-user.targeta** | Многопользовательский режим с сетью |
| 4 | **multi-user.targeta** | Обычно не используется **init** |
| 5 | **graphical.target** | Многопользовательский режим с сетью и графическим интерфейсом |
| 6 | **reboot.target** | Перезагрузка системы |

*а - По умолчанию* ***multi-user.target*** *сопоставляется* ***runlevel3.target****, многопользовательскому режиму с активной сетью*

Единственные цели, которые действительно нужно знать, являются **multi-user.target** и **graphical.target** для повседневного использования и **rescue.target** для доступа к однопользовательскому режиму. Чтобы изменить текущий режим работы системы, используйте команду **systemctl isolate**:

**$ sudo systemctl isolate multi-user.target**

Подкоманда **isolate** имеет такое название, потому что она активирует указанную цель и ее зависимости, но дезактивирует все остальные юниты.

В традиционном **init** вы используете команду **telinit** для изменения уровней запуска после загрузки системы. Некоторые дистрибутивы теперь определяют **telinit** как символическую ссылку на команду **systemctl**, которая распознает, как она вызывается и ведет себя соответствующим образом.

Чтобы увидеть цель, в которую система загружается по умолчанию, запустите подкоманду **get-default**:

**$ systemctl get-default**

graphical.target

Большинство дистрибутивов Linux загружаются по умолчанию в режиме **graphical.target**, который не подходит для серверов, потому что им не нужен графический интерфейс. Но это легко изменить:

**$ sudo systemctl set-default multi-user.target**

Чтобы увидеть все доступные цели системы, запустите **systemctl list-units**:

**$ systemctl list-units --type=target**

**Зависимости между юнитами**

Пакеты программного обеспечения Linux обычно поставляются со своими собственными юнит-файлами, поэтому администраторы не нуждаются в подробном знании всего языка конфигурации. Тем не менее, им необходимы рабочие знания системы зависимостей **systemd** для диагностики и устранения проблем с зависимостями.

Начнем с того, что не все зависимости явные. **systemd** берет на себя функции старого **inetd** и расширяет эту идею в области межпроцессной системы связи D-Bus. Другими словами, **systemd** знает, на каких сетевых портах или точках соединений IPC будет размещаться указанный сервис, и он может прослушивать запросы по этим каналам, не запуская сам сервис. Если появляется клиент, **systemd** просто запускает необходимую службу и разрывает соединение. Служба работает если она используется на самом деле, в противном случае она становится бездействующей.

Во-вторых, **systemd** делает некоторые предположения о нормальном поведении большинства видов юнитов. Точные предположения варьируются в зависимости от типа юнита. Например, **systemd** предполагает, что средняя услуга является надстройкой, которая не должна запускаться на ранних этапах инициализации системы. Отдельные юниты могут отключать эти допущения с записью в разделе [Unit] своего юнит-файла;

*DefaultDependencies=false*

значение по умолчанию - true. См. man-страницу для **systemd.***тип-юнита*, чтобы увидеть точные предположения, которые применяются к каждому типу юнита (например, **man systemd.service**).

Третий класс зависимостей — это те, которые явно объявлены в разделе [Unit] юнит-файлов. В таблице 2.7 показаны доступные параметры.

**Таблица 2.7 Явные зависимости в разделе [Unit] юнит-файлов**

|  |  |
| --- | --- |
| **Параметр** | **Значение** |
| Wants | Юниты, которые должны быть совместно активированы, если это возможно, но не требуются |
| Requires | Строгие зависимости; сбой в каких-либо предварительных условиях останавливает эту услугу |
| Requisite | Подобно Requires, но уже должен быть активным |
| BindsTo | Подобно Requires, но еще более тесно связаны |
| PartOf | Подобно Requires, но влияет только на запуск и остановку |
| Conflicts | Отрицательные зависимости; не может взаимодействовать с этими юнитами |

За исключением Conflicts, все параметры в таблице 2.7 выражают основную идею о том, что настроенный юнит зависит от некоторого набора других юнитов. Точные различия между этими параметрами едва различимы и в первую очередь интересны для разработчиков услуг. Наименее ограниченный вариант, Wants, является предпочтительным, когда это возможно.

Вы можете расширить группы юнитов Wants или Requires, создав папку *unit-file***.wants** или *unit-file***.requires** в **/etc/systemd/system** и добавив туда символические ссылки на другие юнит-файлы. Будет даже лучше если дадите **systemctl** сделать это за вас. Например, команда:

**$ sudo systemctl add-wants multi-user.target my.local.service**

добавляет зависимость от **my.local.service** к стандартному многопользовательскому режиму, гарантируя, что служба будет запущена всякий раз, когда система переходит в многопользовательский режим.

В большинстве случаев такие специальные зависимости автоматически заботятся о вас, благодаря разделам [Install] в юнит-файлах. Этот раздел содержит опции WantedBy и RequiredBy, которые считываются, только если юнит включен командой **systemctl enable** или отключен **systemctl disable**. При включении они заставляют **systemctl** выполнять эквивалент **add-wants** для каждого WantedBy или **add-require** для каждого RequiredBy.

Классы [Install] сами по себе не имеют никакого эффекта при нормальной работе, поэтому, если юнит, кажется, не запускается, когда должен, удостоверьтесь, что он был должным образом включен и на него была создана символическая ссылка.

**Порядок запуска**

Вы можете резонно предположить, что если юнит A требует юнит B, то юнит B будет запущен или настроен перед юнитом A. Но на самом деле это не так. В **systemd** порядок, в котором запускаются юниты (или деактивируются), является совершенно отдельным вопросом, который определяет какие юниты следует активировать.

Когда система переходит в новое состояние, **systemd** сначала отслеживает различные источники зависимой информации, изложенные в предыдущем разделе, для определения юнитов, которые будут затронуты. Затем он использует пункты "Before" и "After" из юнит-файлов, чтобы отсортировать рабочий список соответствующим образом. До тех пор, пока юниты не имеют ограничений "Before" или "After", они могут свободно настраиваться параллельно.

Хотя это и потенциально непредсказуемо, это действительно достойная внимания конструктивная особенность. Одной из основных целей проектирования **systemd** было содействие параллелизму, поэтому имеет смысл, чтобы юниты не приобретали зависимости сериализации, если они явно не запрашивают их.

На практике, условие "After" обычно используются чаще, чем "Wants" или "Requires". Определения целей (и, в частности, обратные зависимости, закодированные в пунктах "WantedBy" и "RequiredBy") устанавливают общие черты служб, запущенных в каждом рабочем режиме, а отдельные пакеты беспокоятся только о своих непосредственных и прямых зависимостях.

**Пример более сложного юнит-файла**

Теперь внимательно рассмотрим несколько директив, используемых в юнит-файлах. Ниже приведен пример юнит-файла веб-сервера NGINX, **nginx.service**:

[Unit]

Description=The nginx HTTP and reverse proxy server

After=network.target remote-fs.target nss-lookup.target

[Service]

Type=forking

PIDFile=/run/nginx.pid

ExecStartPre=/usr/bin/rm -f /run/nginx.pid

ExecStartPre=/usr/sbin/nginx -t

ExecStart=/usr/sbin/nginx

ExecReload=/bin/kill -s HUP $MAINPID

KillMode=process

KillSignal=SIGQUIT

TimeoutStopSec=5

PrivateTmp=true

[Install]

WantedBy=multi-user.target

Эта служба имеет тип "forking", что означает, что команда запуска должна завершиться, даже если фактический демон продолжает работать в фоновом режиме. Так как **systemd** не будет непосредственно запускать демон, демон записывает свой идентификатор PID (ID процесса) в указанном "PIDFile", чтобы **systemd** мог определить, какой процесс является основным экземпляром демона.

Строки "Exec" — это команды для запуска при различных условиях. Команды "ExecStartPre" выполняются перед запуском сервиса; показанные здесь команды проверяют синтаксис конфигурационного файла Nginx и гарантируют, что любой существующий PID файл будет удален.

"ExecStart" — это команда, которая фактически запускает службу. "ExecReload" сообщает **systemd**, как заставить службу перечитать ее файл конфигурации. (**systemd** автоматически устанавливает переменную среды "MAINPID" в соответствующее значение.)

*Дополнительную информацию о сигналах см. на стр. 94.*

Завершение этой службы обрабатывается через "KillMode" и "KillSignal", которые сообщают **systemd**, что демон службы интерпретирует сигнал "QUIT" как инструкцию для очистки и выхода. Запись

ExecStop=/bin/kill -s HUP $MAINPID

будет иметь, по существу, тот же эффект. Если демон не завершается в течение секунд, определенных в "TimeoutStopSec", **systemd** принудительно решит этот вопрос забрасывая демона сигналами "TERM", а затем посылая неперехватываемый сигнал "KILL".

Параметр "PrivateTmp" - попытка повысить безопасность. Он помещает временный каталог службы **/tmp** где угодно, но не в фактическом **/tmp**, который используется всеми процессами и пользователями системы.

**Локальные службы и настройки**

Как вы могли видеть в предыдущих примерах, довольно просто создать юнит-файл для "доморощенного" сервиса. Просмотрите примеры в **/usr/lib/systemd/system** и адаптируйте любой юнит-файл, который лучше подходит под ваши нужды. См. man-страницу **systemd.service** для получения полного списка параметров конфигурации для служб. Информация о параметрах, общих для всех типов юнитов, см. руководство **systemd.unit**.

Поместите свой новый юнит-файл в **/etc/systemd/system**. Затем вы можете запустить команду для активации зависимостей, перечисленных в разделе [Install] в файле **.service**.

**$ sudo systemctl enable custom.service**

Как правило, вы никогда не должны редактировать юнит-файл, который Вы не писали. Вместо этого создайте каталог конфигураций в **/etc/systemd/system/***unit-file***.d** и добавьте один или несколько файлов конфигурации, которые называются *xxx***.conf**. Часть *xxx* не имеет значения; просто убедитесь, что файл имеет суффикс **.conf** и находится в нужном месте. Например, стандартное имя - **override.conf**.

**.conf** файлы имеют тот же формат, что и юнит-файлы, и на самом деле **systemd** смешивает их вместе с исходным юнит-файлом. Однако переопределенные файлы имеют приоритет над исходным юнит-файлом, если они оба попытаются установить значение одного и того же параметра.

Следует иметь в виду, что многие параметры **systemd** могут использоваться более одного раза в юнит-файле. В этих случаях множественные значения образуют список и активны одновременно. Если вы присваиваете значение в вашем файле **override.conf**, это значение присоединяется к списку, но не заменяет существующие записи. Это может быть или не быть тем, что вы хотите. Чтобы удалить существующие записи из списка, просто назначьте параметру пустое значение перед добавлением своего.

Давайте рассмотрим пример. Предположим, что ваша система хранит файл конфигурации NGINX в нестандартном месте, например, **/usr/local/www/nginx.conf**. Вам нужно запустить демон **nginx** с параметром **-c /usr/local/www/nginx.conf**, чтобы он мог найти нужный файл конфигурации.

Вы не можете просто добавить эту опцию в **/usr/lib/systemd/system/nginx.service**, потому что этот файл будет заменен всякий раз, когда обновляется пакет NGINX. Вместо этого вы можете использовать следующую последовательность команд:

**$ sudo mkdir /etc/systemd/system/nginx.service.d**

**$ sudo cat > !$/override.conf11**

[Service]

ExecStart=

ExecStart=/usr/sbin/nginx -c /usr/local/www/nginx.conf

<Control-D>

**$ sudo systemctl daemon-reload**

**$ sudo systemctl restart nginx.service**

*11. > и !$ - метасимволы оболочки. > Переадресовывает вывод в файл, а значение !$ расширяется до последнего компонента предыдущей команды, так что вам не нужно его повторно вводить. Все оболочки понимают эти обозначения. См. Раздел Основы Shell, для получения доп. информации.*

Первый "ExecStart=" удаляет текущую запись, а второй устанавливает альтернативную команду запуска. **systemctl daemon-reload** заставляет **systemd** перечитать юнит-файлы. Тем не менее, он не перезапускает демоны автоматически, поэтому вам также понадобится явный перезапуск **systemctl**, чтобы изменения вступили в силу немедленно.

Эта последовательность команд является такой распространенной идиомой, что **systemctl** теперь реализует ее напрямую:

**$ sudo systemctl edit nginx.service**

<edit the override file in the editor>

**$ sudo systemctl restart nginx.service**

Как показано выше, вы все равно должны сделать **restart** вручную.

Последнее, что нужно знать о переопределении файлов, заключается в том, что они не могут изменить раздел [Install] юнит-файла. Любые сделанные вами изменения молча игнорируются. Просто добавьте зависимости напрямую с помощью **systemctl add-needs** или **systemctl add-require**.

**Подводные камни служб и контроля запуска**

**systemd** имеет большое значение в архитектуре системы и его внедрение является непростой задачей для команд, которые создают дистрибутивы Linux. Текущие релизы больше похожи на Франкенштейна, они используют большую часть **systemd**, но также сохраняют несколько отсылок на прошлое. Иногда пережитки просто еще не были полностью преобразованы. В других случаях для облегчения совместимости разработчики оставляют некоторые из форм этих пережитков.

Хотя **systemctl** можно и нужно использовать для управления службами и демонами, не удивляйтесь, когда запустятся традиционные скрипты **init** или связанные с ними вспомогательные команды. Если вы попытаетесь использовать **systemctl** для отключения сети в системе CentOS или Red Hat, вы можете получить следующий вывод:

**$ sudo systemctl disable network**

network.service is not a native service, redirecting to /sbin/chkconfig.

Executing /sbin/chkconfig network off

*Дополнительную информацию об Apache см. на стр. 696.*

Традиционные **init** скрипты часто продолжают функционировать в системе **systemd**. Например, **init** скрипт **/etc/rc.d/init.d/my-old-service** может быть автоматически сопоставлен с юнит-файлом, таким как **my-old-service.service**, во время инициализации системы или при запуске **systemctl daemon-reload**. Apache 2 на Ubuntu 17.04 является одним из таких примеров: попытка отключить **apache2.service** приводит к следующему выводу:

**$ sudo systemctl disable apache2**

Synchronizing state of apache2.service with SysV service script with

/lib/systemd/systemd-sysv-install.

Executing: /lib/systemd/systemd-sysv-install disable apache2

Конечный результат — то, что вы хотели, но он проходит довольно долгий путь.

Red Hat, и в дополнении CentOS, все еще запускают скрипт **/etc/rc.d/rc.local** во время загрузки, если он настроен как исполняемый12. Теоретически вы можете использовать этот скрипт для изменения настроек системы или задач, выполняемых после загрузки системы, если это необходимо. На данный момент, однако, вам нужно пропустить этот хак и настроить систему как это делается в **system** - создать соответствующий набор юнит-файлов.

*12.* ***sudo chmod + x /etc/rc.d/rc.local*** *гарантирует, что файл будет исполняемым.*

Некоторые задачи загрузки Red Hat и CentOS продолжают использовать конфигурационные файлы, найденные в каталоге **/etc/sysconfig**. В таблице 2.8 суммируются эти данные.

**Таблица 2.8 Файлы и подкаталоги каталога Red Hat /etc/sysconfig**

|  |  |
| --- | --- |
| **Файл или папка** | **Содержимое** |
| **console/** | Каталог, который исторически допускает настраиваемое сочетание клавиш |
| **crond** | Аргументы для передачи демону **crond** |
| **init** | Конфигурация для обработки сообщений скриптов запуска |
| **iptables-config** | Загружает дополнительные модули iptables, такие как NAT-помощники |
| **network-scripts/** | Вспомогательные скрипты и файлы сетевой конфигурации |
| **nfs** | Опциональные RPC и аргументы NFS |
| **ntpd** | Параметры командной строки для **ntpd** |
| **selinux** | Символическая ссылка на **/etc/selinux/configa** |

*а. Устанавливает аргументы для SELinux и позволяет полностью его отключить; см. стр. 85.*

Пара пунктов таблицы 2.8 заслуживает дополнительного объяснения:

- Каталог **network-scripts** содержит дополнительные материалы, относящиеся к сетевой конфигурации. Единственное, что вам может понадобиться изменить здесь, это файлы с именем **ifcfg-***interface*. Например, **network-scripts/ifcfg-eth0** содержит параметры конфигурации для интерфейса eth0. Он устанавливает IP-адрес и сетевые параметры интерфейса. Дополнительную информацию о настройке сетевых интерфейсов см. на стр. 419.

- Файл **iptables-config** фактически не позволяет вам изменять сами правила **iptables** (firewall). Это всего лишь способ загрузки дополнительных модулей, таких как трансляция сетевых адресов (NAT) - позволяет пересылать пакеты или использовать систему в качестве маршрутизатора. Дополнительную информацию о настройке **iptables** см. на стр. 440.

**Журналирование systemd**

Захват логов, созданных ядром, всегда был сложной задачей. Она стала еще более важной с появлением виртуальных и облачных систем, поскольку невозможно просто стоять перед консолями этих систем и следить за тем, что происходит. Часто важнейшая диагностическая информация терялась в большом потоке логов.

**systemd** устраняет эту проблему с помощью универсальной структуры ведения журнала, которая включает все сообщения ядра и служб от начальной загрузки до окончательного завершения. Этот объект, называемый журналом, управляется демоном **journald**.

Системные сообщения, захваченные **journald**, хранятся в каталоге **/run**. **rsyslog** может обрабатывать эти сообщения и хранить их в традиционных файлах журналов или пересылать их на удаленный **syslog**-сервер. Вы также можете напрямую обращаться к журналам с помощью команды **journalctl**.

Без аргументов **journalctl** отображает все записи журнала (сначала самые старые):

**$ journalctl**

-- Logs begin at Fri 2016-02-26 15:01:25 UTC, end at Fri 2016-02-26

15:05:16 UTC. --

Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using

4.6M (max allowed 37.0M, t

Feb 26 15:01:25 ubuntu systemd-journal[285]: Runtime journal is using

4.6M (max allowed 37.0M, t

Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpuset

Feb 26 15:01:25 ubuntu kernel: Initializing cgroup subsys cpu

Feb 26 15:01:25 ubuntu kernel: Linux version 3.19.0-43-generic (buildd@

lcy01-02) (gcc version 4.

Feb 26 15:01:25 ubuntu kernel: Command line: BOOT\_IMAGE=/boot/vmlinuz-

3.19.0-43-generic root=UUI

Feb 26 15:01:25 ubuntu kernel: KERNEL supported cpus:

Feb 26 15:01:25 ubuntu kernel: Intel GenuineIntel

...

Вы можете настроить **journald** для хранения сообщений из предыдущих загрузок. Для этого отредактируйте файл **/etc/systemd/journald.conf** и настройте атрибут "Storage":

[Journal]

Storage=persistent

После того, как вы настроили **journald**, вы можете получить список предыдущих загрузок:

**$ journalctl --list-boots**

-1 a73415fade0e4e7f4bea60913883d180dc Fri 2016-02-26 15:01:25 UTC

Fri 2016-02-26 15:05:16 UTC

0 0c563fa3830047ecaa2d2b053d4e661d Fri 2016-02-26 15:11:03 UTC Fri

2016-02-26 15:12:28 UTC

Затем вы можете получить доступ к сообщениям из предыдущей загрузки, ссылаясь на их индекс или по длинному ID:

**$ journalctl -b -1**

**$ journalctl -b a73415fade0e4e7f4bea60913883d180dc**

Чтобы ограничить вывод журналов, связанными с определенным юнитом, используйте флаг **-u**:

**$ journalctl -u ntp**

-- Logs begin at Fri 2016-02-26 15:11:03 UTC, end at Fri 2016-02-26

15:26:07 UTC. --

Feb 26 15:11:07 ub-test-1 systemd[1]: Stopped LSB: Start NTP daemon.

Feb 26 15:11:08 ub-test-1 systemd[1]: Starting LSB: Start NTP daemon...

Feb 26 15:11:08 ub-test-1 ntp[761]: \* Starting NTP server ntpd

...

*Системное журналирование описано более подробно в Главе 10 "Журналирование".*

**2.8 Скрипты init и запуска в FreeBSD**

*Дополнительную информацию о написании скриптов см. в Главе 7.*

FreeBSD использует **init** в стиле BSD, которая не поддерживает концепцию уровней запуска. Чтобы привести систему в полностью загруженное состояние, **init** в FreeBSD запускает **/etc/rc**. Эта программа является shell-скриптом, но ее нельзя изменять напрямую. Вместо этого система **rc** реализует несколько стандартных способов для администраторов и пакетов программного обеспечения для расширения запуска системы и изменения конфигурации.

**/etc/rc** — это, прежде всего, оболочка, которая запускает другие скрипты запуска, большинство из которых находится в **/usr/local/etc/rc.d.** и **/etc/rc.d.** Однако, прежде чем запускать любой из этих скриптов, **rc** запускает три файла, которые содержат информацию о конфигурации для системы:

**- /etc/defaults/config**

**- /etc/rc.conf**

**- /etc/rc.conf.local**

Эти файлы сами являются скриптами, но обычно они содержат только определения для значений переменных shell. Затем скрипты запуска проверяют эти переменные, чтобы определить, как поступить дальше (**/etc/rc** использует некоторую магию shell, чтобы гарантировать, что переменные, определенные в этих файлах, видны везде).

**/etc/defaults/rc.conf** содержит все параметры конфигурации и их настройки по умолчанию. Никогда не редактируйте этот файл, чтобы страшный скрипт запуска не выследил вас и не перезаписал ваши изменения при следующем обновлении системы. Вместо этого просто переопределите значения по умолчанию, установив их снова в **/etc/rc.conf** или **/etc/rc.conf.local**. Страница руководства **rc.conf** содержит обширный список переменных, которые вы можете использовать.

Теоретически, файлы **rc.conf** могут также указывать другие каталоги, в которых стоит искать скрипты запуска, путем изменения значения переменной **local\_startup**. Значение по умолчанию - **/usr/local/etc/rc.d**, и мы рекомендуем оставить его таким13.

*13. Для локальных настроек вы можете либо создать стандартные скрипты rc.d, которые находятся в* ***/usr/local/etc/rc.d****, либо редактировать общесистемный скрипт* ***/etc/rc.local****. Первый вариант предпочтительнее.*

Вы можете найти в **/etc/rc.d** множество различных скриптов запуска - более 150 при стандартной установке. **/etc/rc** запускает эти скрипты в порядке, вычисленном командой **rcorder**, которая считывает скрипты и ищет информацию о зависимостях, которая была закодирована стандартным образом.

Скрипты запуска FreeBSD для обычных сервисов довольно просты. Например, верхняя часть скрипта запуска **sshd** выглядит следующим образом:

#!/bin/sh

# PROVIDE: sshd

# REQUIRE: LOGIN FILESYSTEMS

# KEYWORD: shutdown

. /etc/rc.subr

name="sshd"

rcvar="sshd\_enable"

command="/usr/sbin/${name}"

...

Переменная **rcvar** содержит имя переменной, которая должна быть определена в одном из скриптов **rc.conf**, в данном случае **sshd\_enable**. Если вы хотите, чтобы **sshd** (реальный демон, а не скрипт запуска, оба называются **sshd**) автоматически запускался во время загрузки, поместите строку в **/etc/rc.conf**.

sshd\_enable="YES"

Если эта переменная установлена в "NO" или закомментирована, скрипт **sshd** не запустит демон или не проверит, следует ли его остановить, когда система выключена.

Команда **service** предоставляет интерфейс реального времени в системе **rc.d** FreeBSD14. Например, чтобы остановить службу **sshd** вручную, вы можете запустить команду

**$ sudo service sshd stop**

*14. Версия* ***service****, которую использует FreeBSD, происходит от команды Linux* ***service****, которая управляет традиционными службами* ***init****.*

Обратите внимание, что этот метод работает только в том случае, если служба включена в файлах **/etc/rc.conf**. Если это не так, используйте подкоманду **onestop**, **onestart** или **onerestart**, в зависимости от того, что вы хотите сделать (**service**, как правило, прощает ошибки и напомнит вам о них, если необходимо).

**2.9 Процедуры перезагрузки и выключения**

Исторически сложилось так, что машины UNIX и Linux были чувствительны к тому, как они выключались. Современные системы стали менее чувствительными к этому, особенно при использовании надежных файловых систем, но всегда полезно деликатно выключать машину, когда это возможно.

Пользовательские операционные системы прошлых лет обучали многих системных администраторов перезагрузке системы как первого шага в решении любой проблемы. Тогда это была обычным делом, но в наши дни такой подход чаще тратит время и прерывает работу сервисов. Сосредоточьтесь на выявлении основной причины проблемы, и вы, вероятно, поймаете себя на мысли, что начали перезагружать серверы реже.

Тем не менее, рекомендуется перезагружаться после изменения скриптов запуска или внесения значительных изменений в конфигурацию. Эта проверка необходима для того, чтобы убедиться в успешной загрузке системы. Если вы нашли проблему, но не выяснили ее происхождение по прошествии нескольких недель, вряд ли вы вспомните что и где меняли в настройках системы.

**Выключение физических систем**

Команда **halt** выполняет основные обязанности, необходимые для выключения системы. **halt** регистрирует выключение, останавливает второстепенные процессы, записывает кешированные блоки файловой системы на диск и затем останавливает ядро. В большинстве систем **halt -p** выключает не только систему, но и физический сервер.

**reboot** по существу идентична **halt**, но она заставляет машину перезагружаться, а не останавливаться.

Команда **shutdown** – слой над **halt** и **reboot**, который обеспечивает запланированные выключения и зловещие предупреждения для залогиненных пользователей. Она относится к дням разделения по времени использования систем и в настоящее время в значительной степени устарела. **shutdown** не делает ничего более технически значимого, чем **halt** или **reboot**, поэтому смело игнорируйте его, если вы не используете многопользовательские системы.

**Выключение облачных систем**

Вы можете остановить или перезапустить облачную систему либо с сервера (с помощью **halt** или **reboot**, как описано в предыдущем разделе), либо с помощью веб-консоли поставщика облачных вычислений (или его эквивалента API).

Вообще говоря, отключение из облачной консоли сродни отключению питания. Лучше, если виртуальный сервер управляет собственным выключением, но не стесняйтесь выключать виртуальный сервер из консоли если он перестает отвечать на запросы. Что еще вы можете сделать в таком случае?

В любом случае, убедитесь, что вы понимаете, что означает выключение с точки зрения облачного поставщика. Будет очень позорно уничтожить свою систему, когда все, что вы хотели сделать, это перезагрузить ее.

Во вселенной AWS операции "Stop" и "Reboot" делают то, что вы от них ожидаете. "Terminate" выводит виртуальную машину (инстанс) из эксплуатации и удаляет его из вашего аккаунта. Если базовому устройству хранения установлено значение "удалить при завершении" (delete on termination), будет уничтожен не только ваш инстанс, но и данные на его корневом диске. Это прекрасно, если это то, чего вы ожидаете. Вы можете включить "защиту от завершения" (termination protection), если считаете, что это плохо.

**2.10 Хитрости для незагружающихся систем**

Множество проблем может помешать загрузке системы, начиная от неисправных устройств и заканчивая неправильным обновлением ядра. Существует три основных подхода к этой ситуации, перечисленные здесь в предпочитаемом порядке:

- Не исправляйте баги; просто верните систему к предыдущему рабочему состоянию.

- Приведите систему до уровня, достаточного для запуска оболочки, и отлаживайте ее в интерактивном режиме.

- Загрузите отдельный образ системы, подключите поврежденную файловую систему и исследуйте оттуда.

Первый вариант наиболее часто используется в облаках, но он может быть полезен и для физических серверов, если у вас есть последний образ всего загрузочного диска. Если ваша система делает резервные копии файловой системы, восстановление всей системы может принести больше проблем, чем ценности от ее восстановления. Мы обсуждаем вариант восстановления всей системы в части *Восстановление облачных систем*, которая начинается на стр. 62.

Остальные два подхода сосредоточены на предоставлении вам способа доступа к системе, определении основной проблемы и внесении необходимых исправлений. Загрузка поврежденной системы в shell — это, безусловно, предпочтительный вариант, но проблемы, которые возникают очень рано в процессе загрузки, могут помешать этому подходу.

Режим "загрузка в shell" обычно называется однопользовательским режимом или режимом восстановления. Системы, которые используют **systemd**, имеют еще более примитивный вариант, доступный в форме аварийного режима; он концептуально похож на однопользовательский режим, но делает абсолютный минимум подготовки перед запуском оболочки.

Поскольку однопользовательский, аварийные и режим восстановления не настраивают сеть или не запускают сетевые службы, для их использования обычно требуется физический доступ к консоли. В результате однопользовательский режим обычно недоступен для облачных систем. Мы рассмотрим некоторые варианты восстановления сломанных облачных образов, начиная со страницы 62.

**Однопользовательский режим**

В однопользовательском режиме, также известном как **rescue.target** для систем, использующих **systemd**, запускается только минимальный набор процессов, демонов и сервисов. Корневая файловая система смонтирована, (как **/usr**, в большинстве случаев), но сеть остается неинициализированной.

Во время загрузки вы запрашиваете однопользовательский режим, передавая аргумент ядру, обычно **single** или **-s**. Вы можете сделать это через интерфейс командной строки загрузчика. В некоторых случаях он может быть настроен автоматически в качестве опции меню загрузки.

Если система уже запущена, ее можно перевести в однопользовательский режим с помощью команд **shutdown** (FreeBSD), **telinit** (традиционная **init**) или **systemctl** (**systemd**).

*Дополнительные сведения об учетной записи root см. в Главе 3.*

Нормальные системы запрашивают пароль root перед запуском однопользовательской root оболочки. К сожалению, это означает, что практически невозможно сбросить забытый пароль root в однопользовательском режиме. Если вам нужно сбросить пароль, вам придется получить доступ к диску с помощью отдельного загрузочного носителя.

*Дополнительную информацию о файловых системах и монтировании смотрите в Главе 5.*

Из однопользовательской оболочки вы можете выполнять команды так же, как при входе в полностью загруженную систему. Однако иногда монтируется только корневой раздел; вы должны примонтировать другие файловые системы вручную, чтобы использовать программы, которые не находятся в **/bin**, **/sbin** или **/etc**.

Вы часто можете найти указатели на доступные файловые системы, просмотрев **/etc/fstab**. В Linux вы можете запустить **fdisk -l** (вариант нижнего регистра L), чтобы просмотреть список разделов диска локальной системы. Аналогичная процедура для FreeBSD заключается в том, чтобы запустить **camcontrol devlist** для идентификации дисковых устройств, а затем запустить **fdisk -s** *device* для каждого диска.

Во многих однопользовательских средах корневой каталог файловой системы монтируется только для чтения. Если **/etc** является частью корневой файловой системы (обычный случай), то становится невозможно редактировать многие важные файлы конфигурации. Чтобы устранить эту проблему, необходимо начать однопользовательский сеанс с подключения "**/**" в режиме чтение/запись. В Linux следующая команда обычно является хитростью:

**# mount -o rw,remount /**

В системах FreeBSD опция повторного подключения неявна когда вы заново подключаете файловую систему. Для этого вам нужно явно указать исходное устройство. Например

**# mount -o rw /dev/gpt/rootfs /**

Однопользовательский режим в Red Hat и CentOS немного более агрессивнее обычного. Эти системы попытаются подключить все локальные файловые системы к тому моменту, когда вы окажетесь в командной строке. Хотя это поведение обычно является полезным, но может быть проблематичным если ваша файловая система повреждена. В этом случае вы можете загрузиться в аварийном режиме, добавив **systemd.unit=emergency.target** к аргументам ядра из загрузчика (обычно GRUB). В этом режиме локальные файловые системы не монтируются и запускаются всего несколько основных сервисов.

Команда **fsck** запускается во время обычной загрузки для проверки и восстановления файловых систем. В зависимости от того, какую файловую систему вы используете для корневого каталога, вам может потребоваться запустить **fsck** вручную, при запуске системы в однопользовательском или аварийном режиме. См. стр. 766 для получения более подробной информации о **fsck**.

Однопользовательский режим — это всего лишь точка на обычном пути загрузки, поэтому вы можете завершить работу однопользовательской оболочки с помощью **exit** или <Control-D>, чтобы продолжить загрузку. Вы также можете нажать <Control-D> в приглашении ввода пароля, чтобы полностью обойти однопользовательский режим.



**Однопользовательский режим в FreeBSD**

FreeBSD включает однопользовательский режим в своем меню загрузки:

1. Boot Multi User [Enter]

2. Boot Single User

3. Escape to loader prompt

4. Reboot

Options

5. Kernel: default/kernel (1 of 2)

6. Configure Boot Options...

Одна приятная особенность однопользовательского режима FreeBSD заключается в том, что он спрашивает, какую программу использовать в качестве оболочки. Просто нажмите <Enter> для **/bin/sh**.

Если вы выберете вариант 3, "Escape to loader prompt", вы попадете в среду командной строки на уровне загрузки, реализованную загрузчиком-последнего уровня FreeBSD, **loader**.

**Однопользовательский режим с GRUB**

В системах, использующих **systemd**, вы можете загрузиться в режим восстановления, добавив **systemd.unit=rescue.target** в конец существующей записи ядра Linux. На заставке GRUB выберите нужное ядро и нажмите клавишу "e", чтобы изменить параметры загрузки. Аналогично, для аварийного режима используйте **systemd.unit=emergency.target**.

Вот пример типичной конфигурации:

linux16 /vmlinuz-3.10.0-229.el7.x86\_64 root=/dev/mapper/rhel\_rhel-root

ro crashkernel=auto rd.lvm.lv=rhel\_rhel/swap rd.lvm.lv=rhel\_rhel/root

rhgb quiet LANG=en\_US.UTF-8 systemd.unit=rescue.target

Нажмите <Control-X>, чтобы запустить систему после внесения изменений.

**Восстановление облачных систем**

*См. Главу 9 для более глубокого знакомства с облачными вычислениями.*

По природе облачных систем Вы не можете подключить к ним монитор или USB-накопитель при возникновении проблем с загрузкой. Облачные провайдеры делают все возможное, чтобы облегчить решение проблем, но основные ограничения все еще остаются.

Резервные копии важны для всех систем, но в облачных системах особенно легко сделать моментальный снимок. Провайдеры взимают дополнительную плату за резервные копии, но они стоят недорого. Свободно используйте снимки системы и у вас всегда будет актуальный системный образ. Вы легко сможете вернуться к работоспособному состоянию системы в короткие сроки.

С философской точки зрения вы, вероятно, делаете что-то не так, если вашим облачным серверам требуется отладка во время загрузки. "Pets" и физические серверы получают необходимую помощь, когда они "болеют", но "cattle" просто усыпляют. Ваши облачные серверы – это "cattle"; просто замените их рабочими копиями если они "плохо себя ведут". Использование этого подхода помогает не только избежать критических сбоев, но и облегчает масштабирование и миграцию системы.

Тем не менее, вам неизбежно придется попытаться восстановить облачные системы или диски, поэтому мы кратко обсудим этот процесс ниже.

В AWS однопользовательский и аварийный режимы недоступны. Однако файловые системы EC2 могут быть подключены к другим виртуальным серверам, если они поддерживаются устройствами Elastic Block Storage (EBS). Это значение по умолчанию для большинства инстансов EC2, поэтому, вероятно, этот метод можно использовать при необходимости. Концептуально это похоже на загрузку с USB-накопителя, так что вы можете покопаться в загрузочном диске физической системы.

Вот что необходимо делать:

1. Запустите новый инстанс в той же зоне доступности, что и инстанс, с которым возникли проблемы. В идеале этот инстанс восстановления должен запускаться с одного и того же базового образа и должен использовать тот же тип инстанса, что и "больная" система.
2. Остановите проблемный инстанс. Но будьте осторожны, чтобы не сделать "terminate", эта операция удаляет образ загрузочного диска.
3. С помощью веб-консоли AWS или CLI отсоедините раздел от проблемной системы и присоедините его к инстансу восстановления.
4. Войдите в систему восстановления. Создайте точку монтирования и подключите том, а затем сделайте все, что необходимо для устранения проблемы. Затем отключите раздел (не хочет отключаться? Убедитесь, что вы в это время не в нем).
5. В консоли AWS отсоедините том от инстанса восстановления и подключите его к проблемному инстансу. Запустите проблемный инстанс и надейтесь на лучшее.

Дроплеты в DigitalOcean предлагают консоль с поддержкой VNC, к которой вы можете получить доступ через Интернет, хотя поведение веб-приложения несколько ненадежно в некоторых браузерах. DigitalOcean не предоставляет возможности отсоединить устройства хранения и перенести их в систему восстановления, как это делает Amazon. Вместо этого большинство системных образов позволяют загружаться из альтернативного ядра восстановления15.

*15. Ядро восстановления недоступно во всех современных дистрибутивах. Если вы используете последний релиз и вкладка восстановления сообщает вам, что "Ядро для этого дроплета управляется внутренне и не может быть изменено в панели управления", вам нужно будет обратиться в поддержку DigitalOcean, чтобы связать ваш экземпляр с ISO-образом восстановления, что позволит вам продолжить восстановление.*

Чтобы получить доступ к ядру восстановления, сначала отключите дроплет, а затем примонтируйте ядро восстановления и перезагрузитесь. Если все пойдет хорошо, виртуальный терминал предоставит вам доступ к однопользовательскому режиму. Более подробные инструкции для этого процесса доступны на digitalocean.com.

Проблемы с загрузкой инстанса Google Compute Engine сначала должны быть исследованы путем изучения информации о последовательном порте инстанса:

**$ gcloud compute instances get-serial-port-output** *instance*

Та же информация доступна через веб-консоль GCP.

Процесс перемещения дисков, аналогичный описанному выше для облака Amazon, также доступен в Google Compute Engine. Вы можете использовать CLI для удаления диска из нерабочего инстанса и загрузить новый, который примонтирует диск в качестве дополнительной файловой системы. Затем вы можете запускать проверки файловой системы, изменять параметры загрузки и при необходимости выбирать новое ядро. Этот процесс подробно описан в документации Google по адресу cloud.google.com/compute/docs/troubleshooting.