Linux:

Структура каталога и файловой системы, распространение лицензии, многопользовательский режим, стабильность, командная оболочка, управление из консоли.

Дистрибутивы Linux: отличия, сходства. Командная оболочка Bash: использование в тестировании.

Структура каталогов

Пользователю **Windows**, структура файловой системы **Linux** может показаться чуждой. Диск C:\ и буквы диска исчезли, их заменили каталоги / и каталоги, большинство из которых имеют трехбуквенные имена.
Стандарт иерархии файловой системы (**FHS - Filesystem Hierarchy Standard**) определяет структуру файловых систем в Linux и других UNIX-подобных операционных системах. Однако файловые системы Linux также содержат некоторые каталоги, которые еще не определены стандартом. Структура каталогов, которую мы рассмотрим, применима к большинству дистрибутивов Linux независимо от того, какую файловую систему они используют.

Основные типы контента, хранящегося в файловой системе Linux.

- **Постоянный (Persistent)** - это содержимое, которое должно быть постоянным после перезагрузки, например, параметры конфигурации системы и приложений.
- **Время выполнения (Runtime)** - контент, созданный запущенным процессом, обычно удаляется перезагрузкой
- **Переменный/динамический (Variable/Dynamic)** - это содержимое может быть добавлено или изменено процессами, запущенными в системе Linux.
- **Статический контент (Static)** - остается неизменным до тех пор, пока не будет явно отредактирован или перенастроен.

В ОС Windows жесткие диски называются латинскими буквами (C:, D:, ...), и каждый из дисков представляет собой корневой каталог с собственным **деревом папок.**
Подключение же нового устройства приведет к появлению нового корневого каталога со своей буквой (например, F:). В ОС Linux **файловая система представлена единым корневым каталогом, обозначаемым как слэш (/).** Соответственно, при данной файловой структуре не диски содержат каталоги, а каталог — диски.
Если в Windows программы, зачастую, хранят все данные в одной папке, например в «C:Program FilesProgramName», то в Linux файлы программы разделяются по каталогам в зависимости от типа. Например, исполняемые файлы в /bin, библиотеки в /lib, файлы конфигураций в /etc, логи и кэш в /var.

В таблице 1 приведены несколько примеров того, какие каталоги (точнее, файлы каких каталогов) относятся к каждому из 4 классов, образующихся при разбиении всего множества файлов по этим двум критериям.

Таблица 1. Пример выделения классов файлов

	Разделяемые	Неразделяемые
Статические	*/usr* */opt*	*/etc* */boot*
Изменяемые	*/var/mail* */var/spool/news*	*/var/run* */var/lock*

Выделение этих 4 классов файлов помогает понять те принципы, на которых строится стандартная структура каталогов, предлагаемая стандартом FHS. Ее описание начнем, естественно, в описания структуры корневого каталога, который имеет имя “/”.

Понятие файла

В UNIX-подобных ОС структура каталогов представлена в виде единого дерева. Отдельные «ветви» этого дерева могут располагаться на разных носителях, или в разных файловых системах, причем эти файловые системы могут быть разными по своей внутренней организации – на одном носителе это файловая система ext2fs, на другом – vfat, и так далее. Разработчики стандарта стремились обеспечить оптимальное размещение файлов в разных файловых системах с тем, чтобы оптимизировать процессы загрузки, последующего функционирования и возможного обновления системы.

Любая UNIX-система, в том числе и Linux - система сетевая, и эти файловые системы и соответствующие носители могут физически располагаться даже на разных компьютерах. Поэтому при размещении отдельных файлов в различных частях файловой структуры надо учитывать, что некоторые файлы должны быть доступны с других компьютеров в сети (быть разделяемыми), а к другим файлам доступ по сети необходимо ограничить. Выделение группы разделяемых файлов позволяет экономить общее дисковое пространство. Группа неразделяемых файлов вычленяется как по соображениям безопасности, так и просто потому, что эти файлы определяют локальную конфигурацию системы и поэтому нужны только на данном компьютере. Например, пользовательские каталоги могут (а часто и должны) быть разделяемыми, а файлы настройки процедур загрузки системы должны быть неразделяемыми.

Файлы делятся на статические (неизменяемые) и изменяемые. К числу статических файлов относятся исполняемые файлы, библиотеки, документация и другие файлы, изменять которые может только администратор системы. Для остальных пользователей эти файлы должны быть доступны только по чтению. Изменяемые файлы – это те, которые любой пользователь может менять без привлечения администратора.

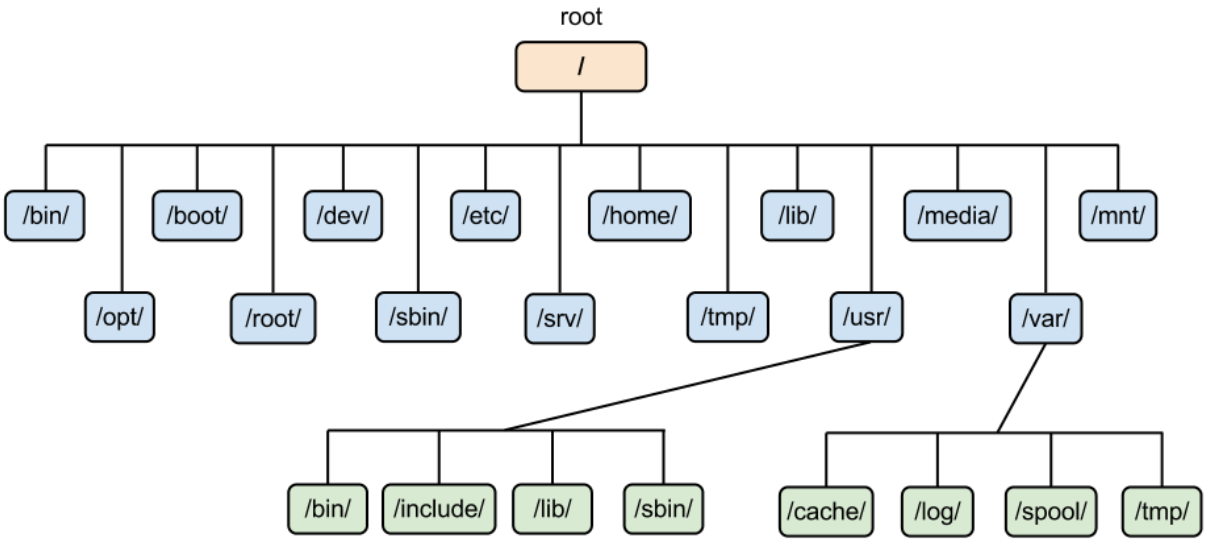
Понятие «файл» в Linux имеет несколько другое значение, нежели в Windows. «Файлом» можно назвать обычный файл, содержащий данные, и интерпретируемый программой. Директория также является «файлом», содержащим в себе ссылки на другие директории или файлы с данными. Файлы устройства указывает на драйвер, благодаря которому система взаимодействует с физическими устройствами. Имеются и многие другие типы файлов.

Также стоит отметить чувствительность файловой системы Linux к регистру. Файлы Temp.txt и temp.txt будут интерпретироваться как разные файлы и могут находиться в одной директории, в отличие от ОС Windows, который не различает регистр имен. То же

правило действует и на каталоги — имена в разных регистрах указывают на разные каталоги.

Назначение каждой директории регламентирует «Стандарт иерархии файловой системы» FHS (Filesystem Hierarchy Standard). FHS разработан чтобы представить общую схему для упрощения независимой от дистрибутива разработки программного обеспечения, поскольку так все необходимое располагается одинаково в большинстве дистрибутивов. FHS определяет следующее дерево директорий (взято непосредственно из спецификации):

Ниже опишем основные директории согласно стандарту FHS:



Стандарт иерархии файловой системы

- **/ — root корневой каталог. Содержит в себе всю иерархию системы, Чтение и изменение файлов в этом каталоге доступно только пользователю root.**
- /bin — здесь находятся двоичные исполняемые файлы. Основные общие команды, хранящиеся отдельно от других программ в системе (прим.: pwd, ls, cat, ps);
- /boot — тут расположены файлы, используемые для загрузки системы (образ initrd, ядро vmlinuz);
- /dev — в данной директории располагаются файлы устройств (драйверов). С помощью этих файлов можно взаимодействовать с устройствами. К примеру, если это жесткий диск, можно подключить его к файловой системе. В файл принтера же можно написать напрямую и отправить задание на печать;
- /etc — в этой директории находятся файлы конфигураций программ. Эти файлы позволяют настраивать системы, сервисы, скрипты системных демонов;
- /home — каталог, аналогичный каталогу Users в Windows. Содержит домашние каталоги учетных записей пользователей (кроме root). При создании нового пользователя здесь создается одноименный каталог с аналогичным именем и хранит личные файлы этого пользователя;
- /lib — содержит системные библиотеки, с которыми работают программы и модули ядра;
- /lost+found — содержит файлы, восстановленные после сбоя работы системы. Система проведет проверку после сбоя и найденные файлы можно будет посмотреть в данном каталоге;
- /media — точка монтирования внешних носителей. Например, когда вы вставляете диск в дисковод, он будет автоматически смонтирован в директорию /media/cdrom;
- /mnt — точка временного монтирования. Файловые системы подключаемых устройств обычно монтируются в этот каталог для временного использования;

- /opt — тут расположены дополнительные (необязательные) приложения. Такие программы обычно не подчиняются принятой иерархии и хранят свои файлы в одном подкаталоге (бинарные, библиотеки, конфигурации);
- /proc — содержит файлы, хранящие информацию о запущенных процессах и о состоянии ядра ОС;
- /root — директория, которая содержит файлы и личные настройки суперпользователя;
- /run — содержит файлы состояния приложений. Например, PID-файлы или UNIX-сокеты;
- /sbin — аналогично /bin содержит бинарные файлы. Утилиты нужны для настройки и администрирования системы суперпользователем;
- /srv — содержит файлы сервисов, предоставляемых сервером (прим. FTP или Apache HTTP);
- /sys — содержит данные непосредственно о системе. Тут можно узнать информацию о ядре, драйверах и устройствах;
- /tmp — содержит временные файлы. Данные файлы доступны всем пользователям на чтение и запись. Стоит отметить, что данный каталог очищается при перезагрузке;
- /usr — содержит пользовательские приложения и утилиты второго уровня, используемые пользователями, а не системой. Содержимое доступно только для чтения (кроме root). Каталог имеет вторичную иерархию и похож на корневой;
- /var — содержит переменные файлы. Имеет подкаталоги, отвечающие за отдельные переменные. Например, логи будут храниться в /var/log, кэш в /var/cache, очереди заданий в /var/spool/ и так далее.

Распространение лицензии

Существует множество дистрибутивов GNU/Linux как общего назначения, так и специализированных (для серверов, сетевых устройств, встраиваемых систем). Как правило, дистрибутивы распространяются бесплатно, но в некоторых случаях предполагается оплата технической поддержки.
Создать свой дистрибутив может любой — для этого не надо подписывать NDA, лицензионные соглашения и даже просто спрашивать разрешения. Нужно всего лишь соблюдать лицензии программных модулей, которые помещаются в дистрибутив.

Лицензия GNU GPL
И ядро Linux, и основная масса прикладных программ защищены лицензией GNU GPL. Эта лицензия предоставляет пользователю права копировать, модифицировать и распространять (в т.ч. на коммерческой основе) программы, при условии сохранения вышеперечисленных прав у пользователей всех производных программ.
Условие сохранения прав — ключевое для GPL — было введено для защиты прав разработчиков. Исследования говорят о том, что внутренняя стоимость GNU/Linux составляет несколько миллиардов долларов, и тем, кто занимался разработкой этих продуктов, совершенно не хочется, чтобы полученные результаты «растащили». Таким образом, в случае дистрибутивов GNU/Linux мы имеем дело со свободным ПО — Open Source.

Сообщество разработчиков
Успех Linux обусловлен тем, что с самого начала вокруг этого проекта стало формироваться сообщество разработчиков (Community). Как ни странно, но бесплатная работа над открытым ПО заинтересовала многих — ведь здесь каждый мог найти задачи по душе: кто-то разрабатывал ПО, кто-то прорабатывал пользовательские интерфейсы, кто-то создавал документацию, кто-то переводил все на свой родной язык, а кто-то компоновал дистрибутивы. На сегодняшний день в разработке Linux задействовано больше специалистов, чем может себе позволить нанять любая коммерческая компания.
В Community входят и обычные пользователи, которые повсеместно объединяются в группы (Linux User Group, LUG), охотно оказывающие помощь новичкам.
И, как показывает практика, получить реальную помощь в форуме пользователей либо

обращаясь напрямую к разработчикам можно быстрее, чем при возникновении аналогичных проблем с коммерческим ПО.

Многопользовательский режим

Многопользовательский режим – это обычный **режим** работы системы, который включается после загрузки по умолчанию. В ряде систем (например, в **Linux**) существует большее количество **режимов** работы (часто – семь, с номерами от 0 до 6, в Solaris к ним еще добавляется **режим** s). Среди этих **режимов** выделяют **режим** выключения системы, однопользовательский **режим** и несколько **многопользовательских**, которые отличаются друг от друга тем, какие именно **сетевые** службы запускаются при старте системы.

Unix (и Linux) был изначально ориентирован на то, что одним компьютером могут пользоваться одновременно несколько человек. Но даже если компьютером обычно пользуется только один человек, такой подход все равно помогает разделить пользовательские настройки от системных, т.е. тех, которые относятся ко всем пользователям и к системе в целом. Такое разделение положительно сказывается на устойчивости и безопасности системы. Приложения изначально пишутся с учетом того, что ими может пользоваться несколько пользователей сразу и, как правило, не требуют прав записи в системные каталоги. Все настройки они сохраняют в собственном, т.н. «домашнем» каталоге пользователя. Каждый пользователь может настроить систему в соответствии со своими предпочтениями и это не вызовет проблем у других пользователей. Обычно работа ведется под пользователем, у которого нет прав испортить что-то за пределами своего каталога, а настройка системы производится под суперпользователем по мере необходимости. Многопользовательский режим позволяет производить настройку системы, не прерывая работы пользователей.

Работа в системе под пользователем с ограниченными правами позволяет предотвратить повреждение системы при неаккуратных действиях пользователя, а отсутствие доступа на запись к системным каталогам не приносит неудобств**.**

Стабильность

Linux традиционно считается самой стабильной операционной системой для инженеров и программистов, но уже давно прикладываются реальные усилия, чтобы сделать Linux привлекательнее для остальных людей. Это очень важно, поскольку неустранимые архитектурные проблемы безопасности Windows и закрытая экосистема Apple не позволяют рассматривать их как надёжные варианты для массового использования.

Командная оболочка

Операционные системы семейства Linux, как впрочем, и любые другие ОС, предполагают наличие интерфейса взаимодействия между компонентами компьютерной системы и конечным пользователем, т. е. наличие программного уровня, который обеспечивает ввод команд и параметров для получения желаемых результатов. Такой программный уровень получил название "оболочка" или, на английском языке - shell. Оболочка — это программа, которая предназначена для обеспечения выполнения других программ по желанию пользователя.

В. Linux доступны следующие командные оболочки: **Bash — самая распространённая оболочка. Она ведёт историю команд и предоставляет возможность их редактирования; pdksh — клон korn shell, хорошо известной оболочки в системах; tcsh — улучшенная версия**>. C shell; zsh — новейшая из перечисленных здесь оболочек; реализует улучшенное дополнение и другие удобные функции. Оболочкой по умолчанию является Bash (Bourne Again Shell).

Командная оболочка (shell) обеспечивает взаимотействие между пользователем и средой операционной системы Linux. Она является специализированным программным продуктом, который обеспечивает выполнение команд и получения результатов их выполнения. Примером оболочки может быть, например, интерпретатор команд command.com операционной системы MS DOS, или оболочка bash операционных систем Unix / Linux.

Все оболочки имеют схожие функции и свойства, в соответствием с их основным предназначением - выполнять команды пользователя и отображать результаты их выполнения:

- Интерпретация командной строки.
- Доступ к командам и результатам их выполнения.
- Поддержка переменных , специальных символов и зарезервированных слов.
- Обработка файлов, операций стандартного ввода и вывода.
- Реализация специального языка программирования оболочки.

Для операционных систем семейства Unix / Linux возможно использование нескольких различных оболочек, отличающихся свойствами и методами взаимодействия с системой.

Управление из консоли

Консоль Linux - удобный инструмент, позволяющий управлять всей системой короткими командами. Из нее можно совершать что угодно - от установки программ, до изменения оболочки, и в целом это экономит кучу времени, и это главная причина почему стоит пользоваться консолью.. (В Windows чтоб установить программу необходимо прошёлкать сколько-то окон и поснимать галочки, чтоб лишнее не поставилось, здесь же одна команда и получаешь необходимое, то же самое можно получить и при установке deb пакетов или из центра программ или вообще собрать программу из исходников прямо в системе.)

Запустить консоль Linux можно запустив терминал (Ctrl+Alt+T зависит от сборки), находится он в
Пуск (Меню) → Администрирование → Терминал (Или слева при раскрытии меню, зависит от оболочки..)
Для KDE: Пуск (Kickoff) → Система → Терминал. Или в файловом менеджере Dolphin он всегда под рукой горячая клавиша F4.
Полезной опцией является тот момент, что в консоли можно запустить любое приложение и оно будет туда складывать свой вывод, таким образом можно ловить ошибки приложений.
В Линуксе расширение файла может не быть, исполняемым может быть любой файл у которого стоят разрешения на выполнение (можно выставить правой клавишей в свойствах или **chmod +x ./start-tor-browser**) в Виндовс обычно исполняемые только .exe и ещё некоторые файлы.

Управление консолью Linux

Часто понадобится запрашивать привилегии суперпользователя **root** для многих действий, связанных с установкой, изменением файлов итп. Для этого предусмотрен механизм предоставления необходимого количества привилегий, но не более - **sudo**. Эта команда следует перед другими и вводится пароль пользователя для разрешения на запуск, в Линуксе везде, где производятся существенные изменения - запрашивается пароль.

Важно: в целях безопасности **консоль Линукс** не отображает вводимые пароли, но по факту они вводятся.

В редких случаях и это не советуется, можно запросить права суперпользователя и что-то сделать под ним, не вводя каждый раз **sudo**, если поставить Debian или разные серверные дистрибутивы, в консоли необходимо прописать **su**, но поскольку в сборках пароль суперпользователя не задан, то можно получить доступ к суперпользователю написав: **sudo su.**

Если ввести любую команду, напрмиер, **apt**, то можно увидеть описание или **apt** и 2 раза **TAB** и увидеть только список функций. Если написать **man apt** попадаем в описание (мануал) к текущей функции. Ещё помощь можно получить в утилитах, к примеру если ввести **ls --help**..

При работе в консоли (терминале) в текстовогом редакторе **nano**, чтоб сохранить изменения нажмите закрыть **Ctrl+X** и подтвердите сохранение, отмена **Ctrl+C** (во многих случаях прервать любое действие эта команды **Ctrl+C** или **q**).

Ctrl+D отключиться (отлогиниться) от текущего пользователя, сервера. Повторное нажатие закрывает консоль.

Для копирования используется сочетание **Ctrl+Shift+C**, для вставки **Ctrl+Shift+V** (альтернатива: правая кнопка мышки или Ctrl+Insert с Shift+Insert).

В любой ситуации, обычно, есть подсказки на экране.

Многообразие дистрибутивов Linux, в чем их отличия и сходства.

Старинная айтишная мудрость гласит: «Лучший Линукс тот, с которым умеет обращаться ваш сисадмин». Действительно, универсального ответа на вечный вопрос «какой дистрибутив Linux выбрать» быть не может. Ведь каждый из них имеет свои сильные и слабые стороны.
Операционные системы Linux — одна из самых востребованных и активно развивающихся технологий современности. Существует множество различных ОС семейства Linux, которые позволяют настроить персонализацию под любые задачи и предпочтения пользователей.
Составить топ дистрибутивов Linux, учитывая их многообразие, крайне сложно. Эти продукты распространяются различными компаниями и не имеют единой системы отчётности. Любая статистика (включая, запросы в поисковиках и рейтинги на тематических ресурсах, вроде Distrowatch) будет далека от объективности.

Лучшие дистрибутивы Linux в 2022

Представленный ниже рейтинг дистрибутивов Linux отражает разумный компромисс между сложившимися годами предпочтениями опенсорс-сообщества и новыми тенденциями в развитии этой технологии.
Ubuntu
Основа: Debian.
Архитектура: aarch64, armel, armhf, i386, i686, mips, mipsel, ppc64el, s390x, x86_64.
Преимущества:

- простота установки, есть LiveCD;
- большое и развитое сообщество;
- обширная база документации, в том числе на русском;
- лёгкое использование PPA;
- большое число дополнительных репозиториев.

Debian
Основа: Linux kernel.
Архитектура: armhf, ppc64el, riscv, s390x, x86_64.
Преимущества:

- легковесность;
- возможность выбора окружения рабочего стола при установке;
- большое и развитое сообщество;
- высокая надёжность и стабильность релизов;
- поддержка 32- и 64-битных архитектур.

MX Linux
Основа: Debian (Stable), antiX.
Архитектура: armhf, i686, x86_64.
Преимущества:

- средние габариты;
- элегантный и эффективный десктоп;
- высокая стабильность;
- простота настройки;
- документация на русском.

Manjaro
Основа: Arch Linux.
Архитектура: aarch64, x86_64.
Преимущества:

- простота установки;
- автоматическое определение оборудования;
- возможность установки нескольких ядер;
- специальные сценарии Bash для управления графическими драйверами;
- развитое сообщество.

EndeavourOS
Основа: Arch Linux.
Архитектура: aarch64, x86_64.
Преимущества:

- вариативность установки (2 режима);
- большое число доступных десктопов;
- удобная настройка системы;
- развитое сообщество.

Дистрибутивы Linux для начинающих

Nitrux
Основа: Debian (Unstable), Ubuntu (LTS).
Архитектура: x86_64.
Преимущества:

- широкие возможности настройки;
- совместимость с форматом AppImages;
- гибкость;
- легковесность.

Linux Lite
Основа: Debian, Ubuntu (LTS).
Архитектура: i686, x86_64.
Преимущества:

- нетребовательность к ресурсам;
- простота установки и использования;
- легковесность;
- удобный интерфейс.

deepin
Основа: Debian.
Архитектура: x86_64.
Преимущества:

- простота установки и использования;
- удобный интерфейс;
- безопасность;
- надёжность.

Дистрибутивы Linux для программистов и продвинутых пользователей

Arch Linux
Основа: Linux kernel.
Архитектура: x86_64.
Преимущества:

- модель обновлений rolling release;
- возможность установки несвободного ПО;
- контроль зависимостей пакетов;
- гибкость настройки.

Gentoo
Основа: Linux kernel, FreeBSD.
Архитектура: i486, i586, i686, x86_64, alpha, arm, hppa, ia64, mips, powerpc, ppc64, sparc64.
Преимущества:

- много альтернативных способов установки;
- продвинутое управление пакетами;
- аппаратная многоплатформенность;
- низкое потребление оперативной памяти;
- гибкость настройки модулей.

Slackware Linux
Основа: Linux kernel.
Архитектура: arm, i586, s390, x86_64.
Преимущества:

- простота устройства;
- стабильность работы;
- развитое сообщество;

Дистрибутивы Linux для работы с безопасностью

Kali Linux
Основа: Debian (Testing)
Архитектура: mhf, i686, x86_64.
Преимущества:

- более 600 инструментов;
- меню приложений с разбивкой по категориям;
- поддержка нескольких платформ;
- обширная документация.

Parrot Security
Основа: Debian (Testing)
Архитектура: armel, armhf, i686, x86_64.
Преимущества:

- большой выбор инструментов защиты данных;
- меню приложений с разбивкой по категориям;
- сквозное шифрование.

Kodachi
Основа: Debian, Xubuntu
Архитектура: x86_64.
Преимущества:
- маршрутизация соединения через VPN и Tor;
- поддержка DNS шифрования (DNSCrypt);
- большой выбор инструментов защиты данных;
- возможность загрузки в Live-режиме.

Лучшие дистрибутивы Linux на домашний компьютер

Linux Mint
Основа: Debian, Ubuntu (LTS).
Архитектура: i686, x86_64.
Преимущества:
- легковесность;
- возможность выбора окружения рабочего стола при установке;
- мультимедиа кодеки доступны «из коробки»;
- обилие дополнительного ПО в центре приложений;
- совместимость с репозиториями Ubuntu.

Fedora
Основа: Red Hat Linux.
Архитектура: aarch64, armhfp, x86_64.
Преимущества:
- модульность;
- большой выбор сред рабочего стола (Fedora spins);
- быстрая интеграция новых технологий;
- развитое сообщество.

Solus
Основа: Linux kernel.
Архитектура: x86_64.
Преимущества:
- широкие возможности оптимизации;
- удобство использования;
- большая база инструментов для разработчика.

Лучшие серверные дистрибутивы Linux

Red Hat Enterprise Linux (RHEL)
Основа: Fedora
Архитектура: aarch64, i386, ia64, IBM Z, ppc, ppc64el, s390, s390x, x86_64.
Преимущества:
- высокая стабильность и безопасность;
- мультиплатформенность;
- оптимизация под облачную инфраструктуру;
- высокое качество и срок коммерческой поддержки;
- наличие полноценной бесплатной версии.

SUSE Linux Enterprise Server (SLES)
Основа: openSUSE
Архитектура: armv7, armv8, IBM Z, Power, x86_64.
Преимущества:
- высокая адаптивность;
- возможность обновления в атомарном режиме;
- обширная документация;
- интеграция с технологиями виртуализации;
- продвинутая техподдержка.

Oracle Linux
Основа: Red Hat.
Архитектура: aarch64, x86_64.
Преимущества:

- высокая стабильность и безопасность;
- двоичная совместимости с RHEL;
- 2 ядра на выбор (Unbreakable Enterprise Kernel (UEK) и Red Hat Compatible Kernel (RHCK);
- есть доступная по цене коммерческая версия с техподдержкой.

Ubuntu Server
Основа: Debian.
Архитектура: x86-64, ARM v7, ARM64, POWER8, POWER9, IBM s390x (LinuxONE).
Преимущества:

- простота установки;
- долгосрочная коммерческая поддержка;
- обширная база документации, в том числе на русском;
- производительность;
- гибкость настройки.

Дистрибутивы Linux для пользователей Windows и MacOS

Xubuntu
Основа: Debian, Ubuntu.
Архитектура: i686, x86_64.
Преимущества:

- лёгкость установки и настойки;
- элегантное окружение рабочего стола;
- легковесность;
- высокая быстрота и стабильность.

elementary OS (eOS)
Основа: Debian, Ubuntu.
Архитектура: x86_64.
Преимущества:

- лёгкая установка (графическая утилита);
- легковестность и нетребовательность к ресурсам;
- высокая скорость работы;
- полная совместимость с репозиториями и пакетами Ubuntu.

Zorin OS
Основа: Debian, Ubuntu (LTS).
Архитектура: x86_64.
Преимущества:

- адаптивный рабочий стол;
- возможность запускать программы для Windows и macOS;
- легковесность;
- высокая скорость работы.

Лучшие дистрибутивы Linux для игр

Pop!_OS
Основа: Debian, Ubuntu.
Архитектура: x86_64.
Преимущества:

- есть отдельные ISO под видеокарты Nvidia и AMD;
- простота в использовании;
- легкий доступ к Steam и Lutris;
- широкие настройки для ноутбуков и гибридного игрового оборудования.
- Garuda Linux

Основа: Arch Linux.
Архитектура: x86_64.
Преимущества:

- широкие возможности настройки;
- легкость установки приложений;
- разнообразие графических интерфейсов;
- обновление по модели плавающих релизов;
- высокая производительность.

Lakka OS
Основа: LibreELEC.
Архитектура: aarch64, armhf, i386, x86_64.
Преимущества:
Легковесные дистрибутивы Linux
Peppermint OS
Основа: Debian, Lubuntu (LTS).
Архитектура: i686, x86_64.
Преимущества:

- простота установки;
- скорость работы;
- интеграция с облачной инфраструктурой;
- элегантный и удобный интерфейс.

Lubuntu
Основа: Debian, Ubuntu.
Архитектура: i686, x86_64.
Преимущества:

- высокая производительность;
- скорость работы;
- поддержка ПО и репозиториев Ubuntu;
- энергоэффективность.

Puppy Linux
Основа: Linux kernel.
Архитектура: i686, x86_64.
Преимущества:

- расширенная функциональность;
- скорость работы;
- лёгкая настраиваемость;
- простота использования.

Описание командной оболочки Bash, как ее можно использовать в тестировании.

Bash — это **командная оболочка для UNIX-подобных операционных систем** (UNIX, GNU/Linux, MacOS). Она дает пользователю систему команд для работы с файлами и папками, поиском, настройкой окружения и позволяет управлять ОС прямо из командной строки. Логотип bash. Слово bash читается как «баш» и расшифровывается как Bourne-Again Shell.

Что делает bash
Оболочка принимает команды, которые пользователь ввел в командную строку, и интерпретирует их, то есть переводит в машинный код. Операционная система получает код в качестве инструкций и выполняет их.
Есть и другой способ использования: создание bash- или shell-скриптов, которые сохранены в файле. При каждом запуске файла будет выполняться набор указанных в нем команд.
Зачем использовать bash

Те же самые действия с файлами, папками и поиском можно выполнить с помощью графического интерфейса ОС. Но это дольше, неудобнее и сложнее. Программисты пользуются bash или shell, чтобы упростить и ускорить работу с системой.
Например, чтобы скопировать файл с помощью графического интерфейса, нужно открыть папку, где он расположен, кликнуть на файл правой кнопкой мыши, вызвать контекстное меню и выбрать «Скопировать». А если использовать командную строку и bash — потребуется ввести одну команду.
cp <файл1> <файл2>
Команда *cp* означает «скопировать». *<файл1>* — это путь к исходному файлу, например /home/file.txt. Это значит, что file.txt лежит в папке /home.
<файл2> — путь к копии файла, которая создастся при выполнении команды. Например, /home/usr/file2.txt, где file2.txt — название копии. Она будет находиться в папке /usr внутри директории /home.
В некоторых системах графический интерфейс практически не используется. Все действия нужно выполнять с помощью командной строки. Здесь bash/shell незаменим. Как запустить bash
Оболочка-интерпретатор встроена в операционную систему и включается автоматически. Достаточно открыть окно терминала и начать вводить команды. Чтобы запустить bash-скрипт, записанный в файл, нужно выдать права на его исполнение.
Работа с файлами
Со всем, что есть в системе, можно работать. Для этого bash предоставляет огромное множество команд. Вот некоторые из них:

- mkdir <имя_папки> — создает в текущем каталоге новую папку. Если папку нужно создать в другом месте, понадобится команда mkdir -p <путь>;
- touch <файл> — создает файл. Можно прописать к нему путь, а можно создать его в текущем каталоге. Тогда потребуется указать только имя файла;
- cp <путь1><путь2> — копирует файл из пути1 в путь2;
- mv <путь1><путь2> — перемещает файл из пути1 в путь2;
- rm — удаляет файл, rm -r — удаляет каталог;
- wget <ссылка> — скачивает файл с сайта по ссылке и размещает в текущем каталоге;
- echo <текст или переменная> — выводит аргумент в консоль.

Инструменты Bash-тестирования для тестирования исполняемых файлов в среде оболочки.

Bash Test Tools задуман как простая в использовании платформа для тестирования исполняемых файлов в среде оболочки. Платформа позволяет извлекать и утверждать операции с такими параметрами, как **стандартный вывод** , **стандартная ошибка** , **код выхода** , **время выполнения** , **файловая система** и **сетевые службы** .

Инструменты тестирования Bash подходят для выполнения высокоуровневых тестов исполняемых файлов, т . е. **системных тестов** , которые рассматривают исполняемый файл как черный ящик, проверяя только вывод и состояние исполняемого файла и его окружения. Типичными областями использования могут быть:

- проверить полные **варианты использования**
- выявлять простые, но критические сбои, также известные как **дымовое тестирование**
- проверьте правильность задокументированного поведения и **--help**
- собирать **показатели производительности** , такие как **время выполнения**

Предпосылки
Средство выполнения тестов использует `strace` для отслеживания сигнальных выходов или завершений исполняемых файлов.

Рабочий процесс
Рабочий процесс для реализации тестов выглядит следующим образом:

1. создать bash-скрипт
2. источник bash_test_toolsфайла
3. определить функцию, называемуюsetup
4. определить функцию, называемуюteardown
5. реализовать серию тестовых функций, имена всех должны начинаться с test_(например , test_foo).
    - **должен** содержать вызов запуска, напримерrun "foo --some-opts args"
    - за которым следует хотя бы один вызов assert, напримерassert_success

1. выполнить testrunner функцию (она волшебным образом запустит все тесты, которые были определены)

Когда сценарий выполняется, каждому тесту будет предшествовать вызов, setupза которым следует teardownвызов. Иногда это неэффективно, но гарантирует, что все тесты выполняются изолированно друг от друга. Если какой-либо один оператор assert в тестовой функции терпит неудачу, весь тест не будет выполнен.

Создание скрипта
В демонстрационных целях мы собираемся создать тест для инструмента командной строки UNIX **find** , инструмента, который помогает вам перечислять и искать файлы и каталоги. Сначала мы должны получить исходный код и исходный `bash_test_tools` файл, который находится в корне каталога исходного кода.

```
$ # get the source code
$ git clone https://github.com/thorsteinssonh/bash_test_tools.git
$ cd bash_test_tools
$ # start editing a new test in you favorite editor (here we use nano)
$ nano test_find.sh
```

В верхней части тестового сценария добавьте типичный **шебанг** и источник `bash_test_tools` файла.

```
#! /usr/bin/env bash
# -*- coding: utf-8 -*-
source bash_test_tools
```

Перед реализацией наших тестов мы должны сначала определить функции `setup` и `teardown` , которые заботятся о настройке и удалении среды для тестов. Обычно `setup` функция создает рабочий каталог и `cd` в него. Они `setup` также могут запускать необходимые службы и/или предоставлять тестовые файлы для работы. Вот пример функции настройки, которая создает рабочий каталог и добавляет пустой тестовый файл:

```
function setup
{
  mkdir -p work
  cd work
  touch some_file.txt
}
```

Функция разрыва просто очищается после выполнения теста и обычно может выглядеть следующим образом:

```
function teardown
{
```

```
  cd ..
  rm -rf work
}
```

Теперь мы можем определить наш первый тест. В этом примере давайте найдем файлы в локальном каталоге, а затем подтвердим, что операция прошла успешно и завершилась корректно. Конечно, мы также утверждаем, что **find** обнаружил наш тестовый файл, как и ожидалось,

```
function test_find_local_directory
{
  # Run
  run "find ./"
  # Assert
  assert_success
  assert_output_contains "some_file.txt"
}
```

В конце необходимо выполнить `testrunner`, без него никакие тесты не обработаются. Добавьте следующую строку внизу скрипта,

```
testrunner
```

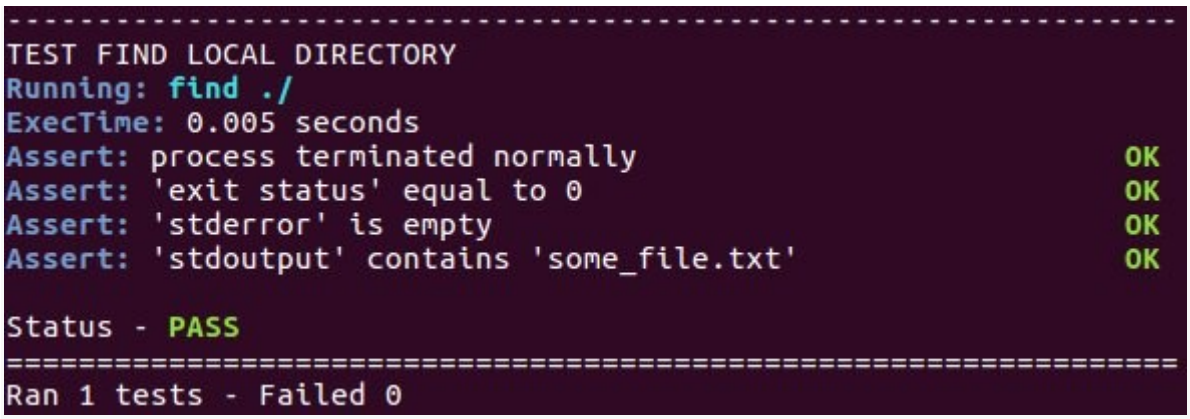
Весь скрипт теперь выглядит следующим образом:

```
#! /usr/bin/env bash
# -*- coding: utf-8 -*-
source bash_test_tools

function setup
{
  mkdir -p work
  cd work
  touch some_file.txt
}

function teardown
{
  cd ..
  rm -rf work
}

function test_find_local_directory
{
  # Run
  run "find ./"
  # Assert
  assert_success
  assert_output_contains "some_file.txt"
}

testrunner
```

Выполнить скрипт

```
chmod u+x test_find.sh
./test_find.sh
```

и вывод должен выглядеть следующим образом,



Обзор

Давайте посмотрим подробнее, что сделал наш тест. Определенный тест делал три вещи:

- Сначала в функцию `"find ./"` был передан вызов выполнения . `bash_test_tools` `run` Обратите внимание на кавычки `"find ./"`, они необходимы. `Run` будет собирать различные метрики в глобальные переменные оболочки с именами `output`, `error`, `exectime` и `returnval` — `strace` они будут подробно рассмотрены позже.
- Второй вызов функции `assert_success` — это общее утверждение для успешного завершения работоспособной программы. На самом деле один `assert_success` вызов состоит из серии более детальных утверждений, называемых `assert_terminated_normally`, `assert_exit_success` и `assert_no_error`.
    - `terminated normally` проверяет, завершился ли исполняемый файл нормально (т . е. **без аварийных** сигналов, таких как SIGENV).
    - `exit success` проверяет, что статус выхода равен 0 (УСПЕХ).
    - `no error` проверит, что ничего не было напечатано со стандартной ошибкой.
- Третий вызов функции `assert_output_contains` просто проверяет `find` правильность сообщения о `standard output` том, что тестовый файл `some_file.txt` найден.

Параметры сценария

Фреймворк автоматически встраивает параметры в тестовый скрипт. Справка будет напечатана с необязательным аргументом `-h`.

```
$ ./test_find.sh -h

test_find.sh - tests built on bash_test_tools

Usage:  test_find.sh [OPTIONS]...

  -l              list all available tests
  -t [TESTNAME]      run only tests ending in TESTNAME
  -o [TAP FILE]      write test results to TAP (test anything) file
  -x              disable teardown (for debugging)
  -h              print this help
```

Например, изучите тестовый сценарий, находящийся в каталоге **примеров .**

```
$ cd examples
$ ./test_find.sh -l
test_find_delete
test_find_local_directory
test_find_txt_files
```

```
test_has_unix_conventions
test_invalid_file_or_directory
test_invalid_option
test_new_feature
```

Мы можем специально запускать только тесты, заканчивающиеся на имя «_directory».

```
$ ./test_find.sh -t _directory
------------------------------------------------------------
TEST FIND LOCAL DIRECTORY
Running: find ./
ExecTime: 0.004 seconds
Assert: process terminated normally                    OK
Assert: 'exit status' equal to 0                       OK
Assert: 'stderror' is empty                            OK
Assert: 'stdoutput' contains 'some_file.txt'           OK

Status - PASS
------------------------------------------------------------
TEST INVALID FILE OR DIRECTORY
Running: find ./non_existing_path
ExecTime: 0.004 seconds
Assert: process terminated normally                    OK
Assert: 'exit status' not equal to 0                   OK
Assert: 'stderror' not empty                           OK
Assert: 'stderror' contains 'No such file or directory'  OK

Status - PASS
============================================================
Ran 2 tests - Failed 0
```

и мы можем выводить результаты тестов в переносимом формате, используя Test Anything Protocol ,

```
$ ./test_find.sh -o result.tap
$ cat result.tap
1..7
ok 1 - test_find_delete
ok 2 - test_find_local_directory
ok 3 - test_find_txt_files
ok 4 - test_has_unix_conventions
ok 5 - test_invalid_file_or_directory
ok 6 - test_invalid_option
not ok 7 - test_new_feature
```

Общие тесты

`bash_test_tools` поставляется с несколькими **общими** тестами, которые подходят для тестирования общих функций. Две очень распространенные особенности в средах UNIX заключаются в том, что исполняемые файлы обычно принимают `--version` и `--help` аргументы. Чтобы протестировать исполняемый файл с помощью общего теста для `--version` и `--help` параметров, добавьте следующие две строки в свой сценарий:

```
generic has_unix_version "find"
generic has_unix_help "find"
```

Это автоматически создаст тесты для исполняемого файла **find** , которые проверят, принимает ли исполняемый файл параметры версии и справки. Они будут утверждать,

что программа завершается нормально, и если она действительно печатает что-то, что нужно удалить.

```
------------------------------------------------
TEST HAS UNIX HELP
Running: find --help
ExecTime: 0.004 seconds
Assert: process terminated normally          OK
Assert: 'exit status' equal to 0             OK
Assert: 'stderror' is empty                  OK
Assert: 'stdout' not empty                   OK
Assert: 'help' contains '--help'             OK

Status - PASS
------------------------------------------------
TEST HAS UNIX VERSION
Running: find --version
ExecTime: 0.005 seconds
Assert: process terminated normally          OK
Assert: 'exit status' equal to 0             OK
Assert: 'stderror' is empty                  OK
Assert: 'stdout' not empty                   OK
Running: find --help
ExecTime: 0.004 seconds
Assert: 'help' contains '--version'          OK

Status - PASS
------------------------------------------------
```

Обратите внимание, что `has_unix_version` также проверяется, было ли оно задокументировано в `--help`. Существует третий общий тест, который вызывает эти два теста, он называется `has_unix_convention`, поэтому вы можете заменить два вышеуказанных теста одной строкой:

```
generic has_unix_convention "find"
```

Не все инструменты командной строки принимают эти параметры, мы можем посмотреть, почему этот тест не работает, когда такой параметр недоступен. **strace** — это инструмент, который этого не делает, вот как он терпит неудачу,

```
------------------------------------------------------
TEST HAS UNIX VERSION
Running: strace --version
ExecTime: 0.003 seconds
Assert: process terminated normally          OK
Assert: 'exit status' equal to 0             FAIL
Assert: 'stderror' is empty                  FAIL
Assert: 'stdout' not empty                   FAIL
Running: strace --help
ExecTime: 0.002 seconds
Assert: 'help' contains '--version'          FAIL

Status - FAIL
======================================================
Ran 1 tests - Failed 1
```

Оказывается, **strace изящно завершается, но через статус выхода** и **стандартную ошибку** указывает , что вызов не поддерживается. Написание тестов, которые терпят

неудачу таким образом, на самом деле является одним из основных моментов тестирования программного обеспечения, чтобы отслеживать через *разработку через тестирование* , были ли реализованы запланированные функции программного обеспечения или нет.

По мере развития `bash_test_tools` кодовой базы мы ожидаем добавления более универсальных тестов, которые помогут выявить общепринятые соглашения. Тесты для протоколов POSIX, Single UNIX и GNU должны быть пригодны для повторного использования и идеально подходят для написания универсальных тестов. Пожалуйста, внесите свой вклад! Найдите в файле **«function generic_»** `bash_test_tools` , чтобы увидеть, как разрабатывается общий тест.

Внешние утверждения

Среда оболочки не идеальна для выполнения сложных тестов текстовых и двоичных файлов. Функция `assert` позволяет вам вызывать любой исполняемый файл, будь то скомпилированный, Python или Ruby, для выполнения более детальных утверждений. Требование к исполняемому файлу внешнего утверждения состоит в том, что он возвращает код выхода 0, если утверждение успешно, и !=0, если утверждение терпит неудачу. В демонстрационных целях мы определим исполняемый файл assert в python, `is_foobar.py`. Он возвращает успех, если аргумент равен `"foobar"`, в противном случае он возвращает отказ.

```
import sys
argument = sys.argv[1]
if argument == "foobar":
   sys.exit(0)
else:
   sys.exit(1)
```

В нашем гипотетическом тесте теперь мы можем добавить утверждение, которое вызывает этот пользовательский тест Python,

```
assert "is_foobar.py foobar"
```

Выходной тест теперь должен включать этот шаг утверждения,

```
Assert: arg_is_foobar.py foobar                    OK
```

Обычно эти пользовательские функции assert будут выполнять более полезные и более подробные задачи, чем показано здесь. Полезно называть утверждения информативно, как указано выше. Например,

```
assert "is_jpeg_image.py some_image.file"
# or
assert "is_json_text.py some_text_file"
```

Это поможет сделать вывод теста более читабельным.

Утвердить во время выполнения

До сих пор мы имели дело только с утверждением условий после завершения исполняемого файла. Однако иногда нам нужно проверять условия во время выполнения. Некоторые исполняемые файлы, например, предназначены для работы в качестве **служб** или **демонов** . В таких ситуациях нам может потребоваться выполнить операторы assert, пока исполняемый файл работает в фоновом режиме. `bash_test_tools` позволяют вам сделать это, добавив набор утверждений в очередь background_assert. Следующий оператор добавляет в очередь проверку службы tcp на порту 1234,

```
add_background_assert assert_service_on_port 1234
```

Затем эти утверждения будут выполняться во время выполнения оператора в фоновом режиме. Здесь мы запускаем **netcat** , инструмент сетевой диагностики, и прослушиваем порт 1234 в течение 2 секунд, прежде чем завершить процесс сигнальным сигналом SIGTERM.

```
run "nc -l 1234" background 2 SIGTERM
```

Поставленные в очередь операторы assert выполняются после 2-секундного ожидания, за которым следует завершение процесса. Некоторое количество сна перед выполнением утверждений необходимо, чтобы позволить процессу или службе загрузиться или инициализироваться. Конечно, необходимая продолжительность сна будет зависеть от тестируемого программного обеспечения и условий. Вся тестовая функция, которую мы описали, выглядит так,

```
function test_nc_listen_on_port
{
  add_background_assert assert_service_on_port 1234
  #run
  run "nc -l 1234" background 2 SIGTERM
  #assert
  assert_terminated_normally
  assert_no_error
}
```

Мы добавили еще пару утверждений «после выполнения» для проверки правильности завершения работы программного обеспечения. Когда мы выполняем тест, мы получаем следующий вывод,

```
------------------------------------------------------------------
TEST NC LISTEN ON PORT
Running: nc -l 1234 (background 2 secs)
Assert: service on port 1234                    OK
ExecTime: 2.016 seconds
Assert: process terminated normally                 OK
Assert: 'stderror' is empty                   OK

Status - PASS
==================================================================
Ran 1 tests - Failed 0
```