Основы DevOps

Администрирование ОС Ubuntu

Файловая структура Ubuntu

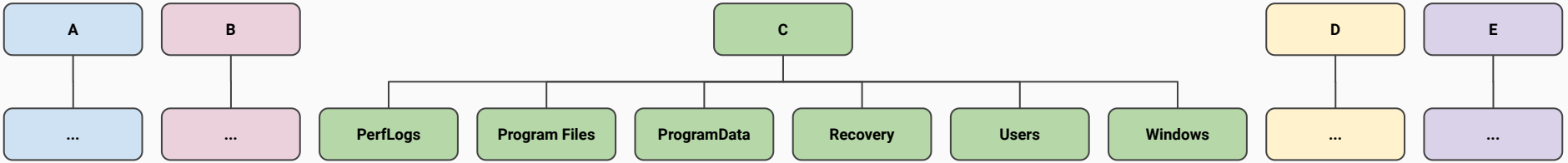
Введение

- Первой особенностью файловой структуры Linux (Ubuntu), которая сразу бросается в глаза, является ее строгая иерархичность. Например, в Windows при подключении диска или USB-устройства автоматически создается независимая система каталогов, которой назначается свое буквенное обозначение (**C**, **D** и т.д.). В свою очередь в Linux (Ubuntu) существует главный корневой каталог (**/**), и уже в него диски и устройства могут монтироваться в качестве подкаталогов (как автоматически, так и вручную).

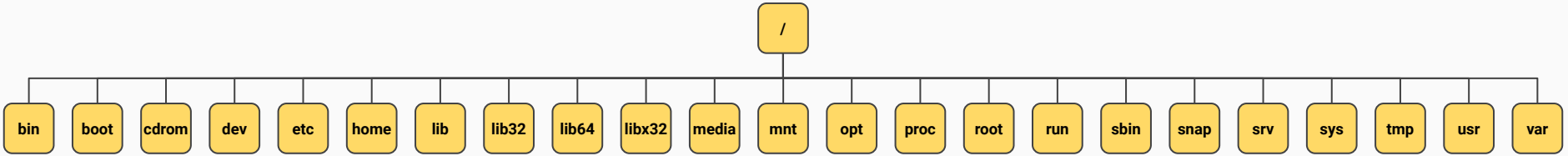
- Кроме того, операционные системы на ядре Linux (Ubuntu) отличается от Windows тем, что использует другой разделитель каталогов - если в Windows это обратный слэш - “****”, например, “**C:\Windows\System32**”, то в Linux (Ubuntu) это простой слэш - “**/**”, например, “**/home/user**”.

- В Windows многие системные настройки могут храниться в реестре (специальной базе данных для хранения конфигураций), что может быть менее прозрачным для пользователей, а в контесте Linux (Ubuntu) практически все системные настройки хранятся в текстовых файлах, что облегчает их настройку и администрирование.

Сравнение файловых структур Windows и Linux



Файловая структура Windows



Файловая структура Linux (Ubuntu)

Файловые системы

- **Файловая система** - это способ, с помощью которого операционная система организует и управляет доступом к данным и ресурсам на диске. Файловая система определяет структуру и формат хранения файлов, каталогов и метаданных. Она также предоставляет интерфейс для создания, чтения, записи и удаления файлов, а также для навигации по файлам и каталогам.

- Основной файловой системой Ubuntu является **EXT4 (Extended File System 4)**. Она поддерживает логирование, что означает, что все изменения в файловой системе фиксируются в журнале перед их записью на диск. Это повышает надежность файловой системы и позволяет восстановить ее в случае сбоев или выключений. Максимальный размер файла составляет 16 терабайт, а максимальный размер файловой системы — 1 эксабайт (миллион терабайт).

- Также Ubuntu поддерживает другие файловые системы, такие как **NTFS** (для совместимости с Windows), **exFAT** (для хранения больших файлов - до 16 эксабайт) и **ReiserFS** (для оптимизированного хранения множества мелких файлов). Таким образом, в одном каталоге может быть примонтирован диск с **EXT4**, а в другом - диск с **NTFS**, при этом они могут спокойно взаимодействовать.

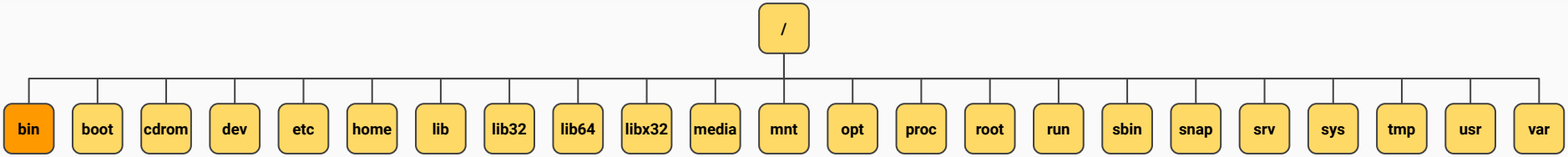
Хранение файлов - индексные дескрипторы (inodes)

- Важной особенностью некоторых файловых систем (в том числе **EXT4**) является ограничение на количество файлов и каталогов. Дело в том, что при создании такой системы также происходит генерация фиксированного количества т.н. индексных дескрипторов (**inodes**). После создания каждого файла или каталога им автоматически присваивается один такой дескриптор, который хранит данные о размере файла, его расположении на диске, его пути внутри файловой структуры, владельце, дате создания, правах и многом другом.

- Необходимо помнить, что при базовом создании файловой системы (например, при помощи команды **mkfs.ext4**) **inodes** будут рассчитаны автоматически. Однако если указать дополнительные настройки (**inode-size** или **bytes-per-inode**), то расчет количества индексных дескрипторов будет опираться уже на них.

- После создания количество **inodes** не может быть изменено. Поэтому может настать момент, когда **inodes** закончатся, и возможность создавать файлы будет потеряна. Чтобы решить эту проблему, надо создать новую файловую систему с увеличенным количеством **inodes**, скопировать в нее файлы старой системы, а затем смонтировать новую систему на место старой.

Каталог **/bin**

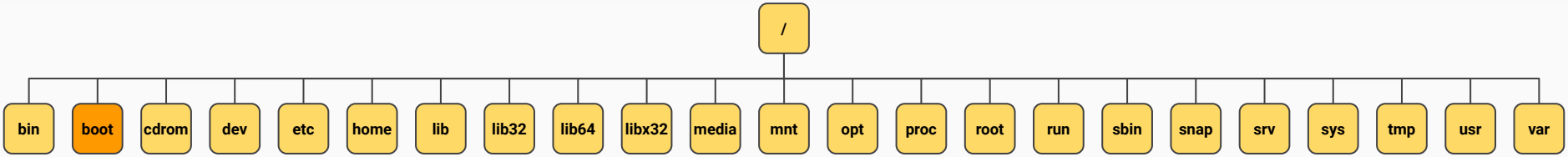


- **/bin** - содержит бинарные (содержащие машинный код - данные и команды в бинарном, т.е. двоичном формате, который понимает компьютер) исполняемые файлы (команды и утилиты), необходимые для работы операционной системы и для работы с этой системой.

- В качестве примера команд и утилит, хранящихся в **/bin**, можно упомянуть **ls** (просмотр содержимого каталогов), **cp** (копирование файлов), **mv** (перемещение файлов), **rm** (удаление файлов), **mkdir** (создание каталогов), **cat** (просмотр содержимого файлов), **echo** (вывод текстовой информации при работе в командной строке) и т.д.

- В наши дни **/bin** является ссылкой на **/usr/bin** - это связано с историческими причинами (утилиты приходилось держать в разных каталогах на разных дисках, т.к. один диск не мог вместить сразу все утилиты из-за ограничений размера).

Каталог **/boot**

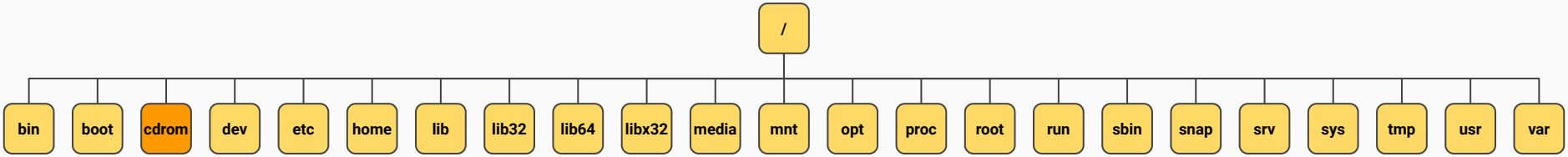


- **/boot** - одна из самых важных частей при старте Ubuntu. Этот каталог содержит файлы, необходимые для загрузки операционной системы. Здесь находится файл ядра Linux (обычно с названием **vmlinuz**), содержащий код самой операционной системы.

- Каталог **/boot** может также содержать дополнительные конфигурационные файлы и сценарии, связанные с загрузкой. Например, **/boot/grub** хранит файлы загрузчика GRUB (Grand Unified Bootloader), который непосредственно управляет процессом загрузки. В свою очередь **/boot/efi** содержит файлы, связанные с загрузкой в контексте технологии UEFI (Unified Extensible Firmware Interface).

- В числе прочего каталог **/boot** может включать в себя разные версии ядра Linux, если операционная система была обновлена. Обычно они хранятся в формате vmlinuz-X.Y.Z, где X, Y и Z - это номера версии ядра.

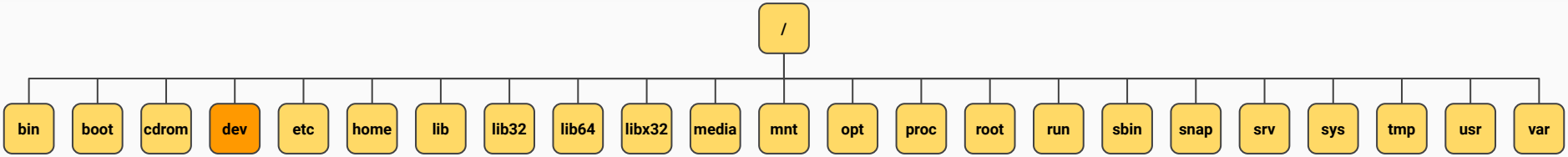
Каталог **/cdrom**



- **/cdrom** - в этом каталоге может располагаться точка монтирования для оптических носителей, таких как CD-ROM и DVD-ROM. Таким образом, при вставке оптического диска в привод CD/DVD, **Ubuntu может** автоматически сделать содержимое диска доступным для чтения и использования в системе.

- На современных версиях операционной системы Ubuntu данный каталог может вообще не использоваться, т.к. основным каталогом для подключения внешних съемных носителей является **/media**. Таким образом, **/cdrom** сохраняется в системе по историческим причинам и для сохранения некой обратной совместимости.

Каталог **/dev**

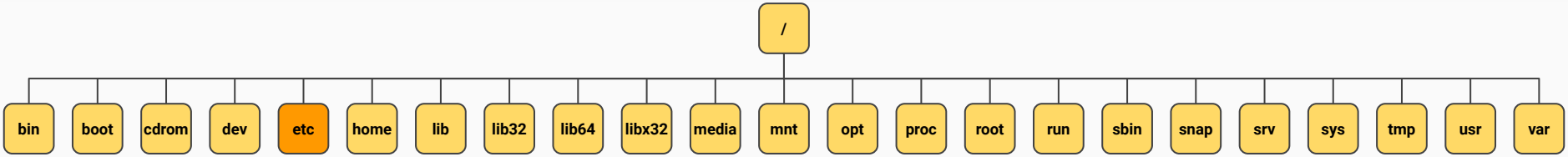


- **/dev** - содержит специальные файлы, представляющие аппаратное оборудование и различные системные ресурсы. Эти файлы позволяют самой операционной системе и пользовательским программам взаимодействовать с аппаратным оборудованием и выполнять различные операции ввода/вывода.

- В качестве примеров файлов в каталоге **/dev** можно упомянуть физические и логические блочные устройства, такие как жесткие диски (**/dev/sda**, **/dev/sdb**, и так далее) и их разделы (**/dev/sda1**, **/dev/sda2**), сетевые интерфейсы (например, **/dev/enp0s3**, **/dev/eth0** или **/dev/wlan0**), CD/DVD-приводы (**/dev/sr0**, **/dev/sr1**), а также клавиатура и мышь (**/dev/input/event0**, **/dev/input/event1**).

- Кроме того, здесь хранятся устройства для генерации случайных чисел (**/dev/random** и **/dev/urandom**), а также файлы терминалов (или консолей), напрямую соединенных с операционной системой (**/dev/tty1**, **/dev/tty2** и т.д.).

Каталог **/etc**

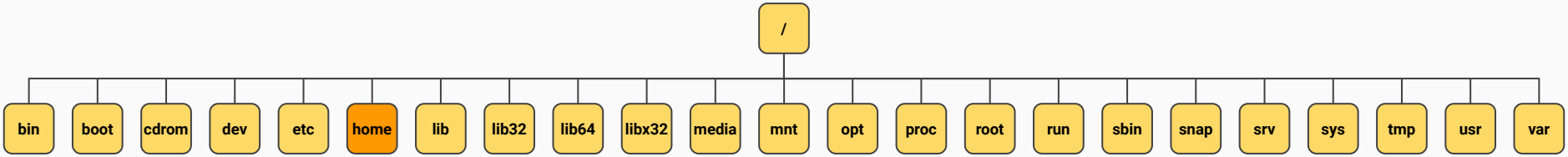


- **/etc** - здесь хранятся конфигурационные файлы для различных системных программ и служб. Эти файлы определяют поведение системы и приложений. Например, файлы **/etc/network/interfaces** управляют настройками сети, а файлы **/etc/apt/sources.list** содержат информацию о репозиториях пакетов.

- Данный каталог также содержит различные подкаталоги, в которых хранятся конфигурационные файлы для дополнительных программ и необязательных компонентов системы. Например, может присутствовать (после установки) каталог **/etc/nginx/**, который содержит конфигурацию веб-сервера Nginx и настройки для его сайтов.

- Поскольку **/etc** содержит конфиденциальную информацию о системе, в большинстве случаев доступ к файлам в нем ограничен правами доступа, и обычные пользователи не могут изменять эти файлы без прав администратора.

Каталог **/home**

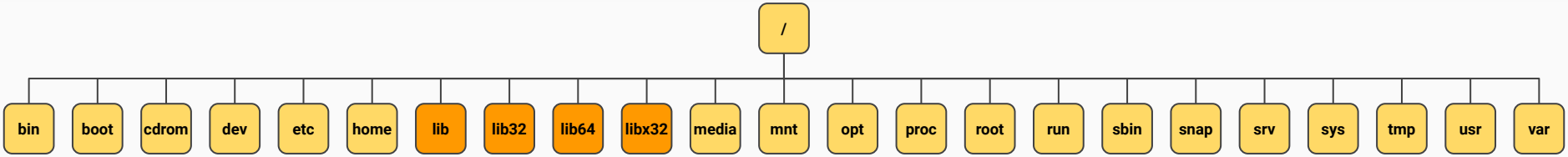


- **/home** - внутри данного каталога создается каталог для каждого зарегистрированного пользователя, и название этого каталога соответствует имени пользователя. Например, если у нас есть пользователь с именем **user**, его домашний каталог будет расположен здесь - **/home/user**.

- Каталог пользователя содержит все личные файлы и документы этого пользователя, включая музыку, изображения, видео, рабочие файлы и т.д. В каталоге пользователя также хранятся конфигурационные файлы и настройки для программ и приложений, которые этот пользователь использует. Например, настройки оболочки командной строки, SSH ключи и многое другое.

- Каталог **/home** защищен правами доступа так, чтобы каждый пользователь имел доступ только к своей собственной домашней директории. Это обеспечивает приватность и безопасность данных пользователей.

Каталоги **/lib**, **/lib32**, **/lib64** и **/libx32**

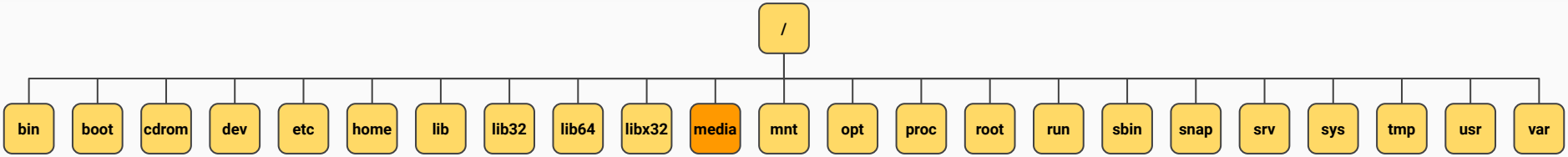


- **/lib, /lib32, /lib64 и /libx32** - содержат необходимые разделяемые библиотеки для запуска программ из других каталогов, таких как **/bin**, **/sbin**, **usr/bin**, **usr/sbin**. Можно упомянуть, что в разделе **/lib/modules** можно помещать модули для ядра Linux (например, драйвер к какому-нибудь необычному устройству).

- Каталог **/lib32** предназначен для хранения 32-битных версий библиотек, (необходимых для поддержки 32-битных программ) каталог **/lib64** - для 64-битных версий библиотек (поддержка 64-битных программ), **/lib** - для библиотек общего пользования, а **/libx32** - для библиотек системы ABI x32 (позволяет использовать 32-битные адреса памяти внутри 64-битных систем).

- Как и в некоторых других случаях, **/lib, /lib32, /lib64 и /libx32** являются ссылками на каталоги **/usr/lib, /usr/lib32, /usr/lib64** и **/usr/libx32** - это опять связано с историческими причинами и ограничением места на одном диске.

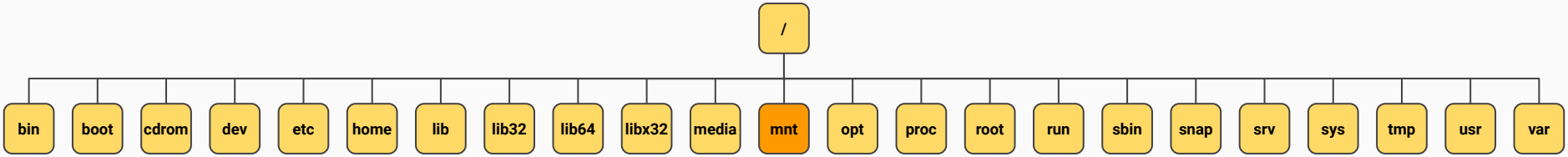
Каталог **/media**



- **/media** - современный (в отличие от **/cdrom**) каталог для автоматического монтирования носителей, таких как USB-флешки и оптические диски (CD/DVD). Обычно настроен так, чтобы обеспечивать доступ к данным для текущего пользователя и других пользователей, имеющих доступ к съемным устройствам.

- Когда вы вставляете съемное устройство в компьютер, Ubuntu обычно автоматически монтирует его как раз в каталог **/media**. Это означает, что содержимое устройства становится доступным для чтения и записи через файловую систему. Например, если вы вставите USB-флешку, она может быть автоматически смонтирована в **/media/имя-пользователя/имя-флешки**.

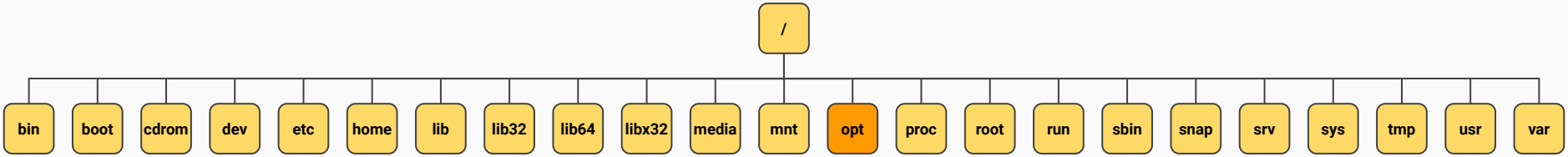
Каталог **/mnt**



- **/mnt** - предназначен для ручного временного монтирования различных файловых систем, таких как жесткие диски, некие сетевые ресурсы (расшаренные сетевые папки) или съемные устройства (например, USB-флешки). Для монтирования обычно используется команда **mount**.

- Каталог **/mnt** обычно пустой по умолчанию после установки операционной системы. Пользователи и администраторы могут создавать подкаталоги внутри **/mnt**, чтобы организовать монтирование различных ресурсов. Например, мы можем создать подкаталог **/mnt/usb** для монтирования USB-флешек или **/mnt/network** для сетевых ресурсов.

Каталог **/opt**

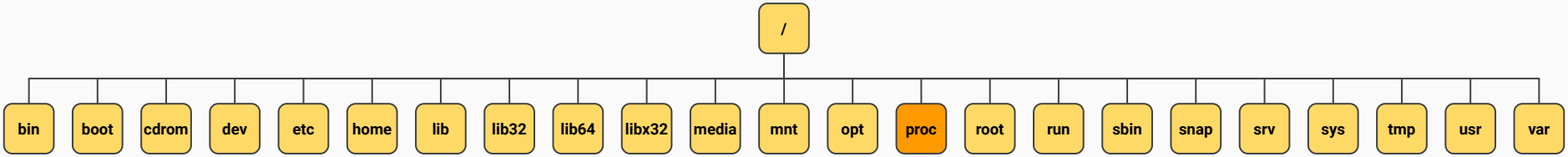


- **/opt** - представляет собой одно из стандартных расположений для установки дополнительных приложений и программ, которые не входят в стандартную поставку Ubuntu, не входят в стандартный репозиторий Ubuntu и не управляются системными пакетными менеджерами, такими как **apt**.

- Внутри **/opt** каждое приложение обычно имеет свой подкаталог, названный именем этого приложения. В этом подкаталоге хранятся файлы, бинарные файлы, библиотеки, конфигурационные файлы и все другие ресурсы, связанные с этим приложением. Таким образом, данные приложения не следуют стандартной схеме (конфигурации - **/etc**, исполняемые файлы - **/bin** или **/usr/bin**).

- Файлы и исполняемые программы, находящиеся в **/opt**, обычно не включаются в системные переменные **PATH** по умолчанию. Это означает, что для запуска программ из **/opt** мы должны явно указывать путь к исполняемому файлу.

Каталог **/proc**

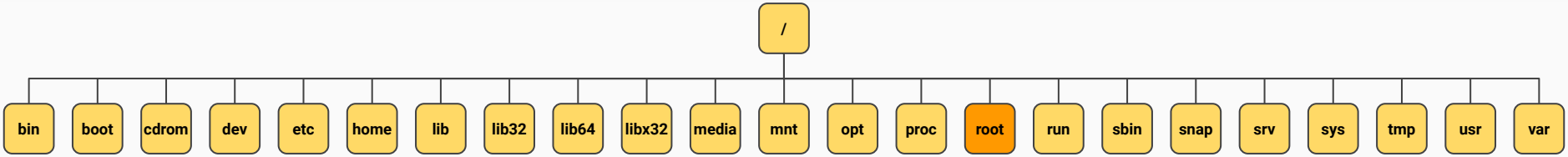


- **/proc** - по сути, виртуальная файловая система **procfs**, которая предоставляет информацию о текущем состоянии системы и процессах. Содержит файлы и подкаталоги, каждый из которых предоставляет информацию о состоянии процессора (**/proc/cpuinfo**), состоянии памяти (**/proc/meminfo**) и т.д.

- Кроме чисто технических вещей, данный каталог позволяет узнать статистическую информацию, такую как время работы системы - как с последней загрузки, так и общее (**/proc/uptime**), текущую нагрузку на систему, включая среднюю загрузку за последние 1, 5 и 15 минут (**/proc/loadavg**) и даже список последних запросов на прерывания процессора (**/proc/interrupts**).

- Также **/proc** хранит подкаталоги, соответствующие идентификаторам запущенных процессов (**PID**). В них можно найти информацию о процессе, такую как использование памяти, открытые файлы и т.д.

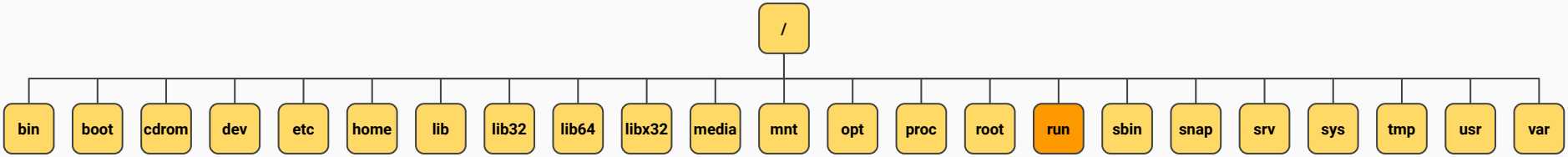
Каталог **/root**



- **/root** - является аналогом домашнего каталога обычных пользователей, таких как **/home/user**, но предназначен для суперпользователя (**root**) - пользователя системы с неограниченными правами. Когда суперпользователь входит в систему, он оказывается в рабочем каталоге **/root**.

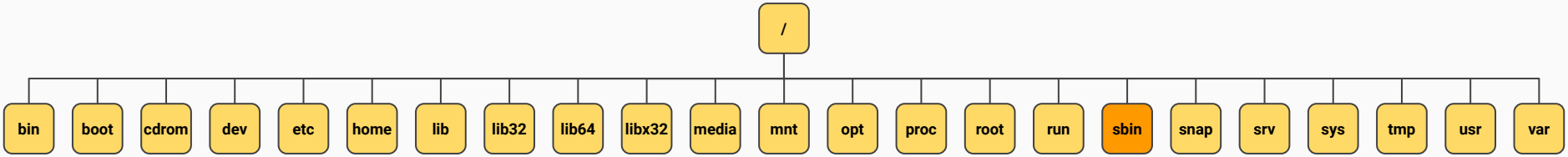
- Каталог суперпользователя **/root** имеет очень строгие права доступа. Обычные пользователи системы не имеют доступа к этому каталогу без использования команды **sudo**, и им нельзя просматривать, изменять или удалять его содержимое. Это делает **/root** крайне защищенным от обычных пользователей и уменьшает поверхность возможной атаки для потенциального злоумышленника.

Каталог **/run**



- **/run** - используется системой для хранения временных файлов, связанных с запущенными процессами. Данные каталога удаляются при выключении компьютера. Это означает, что **/run** является временным и не хранит данные между перезагрузками.

Каталог **/sbin**

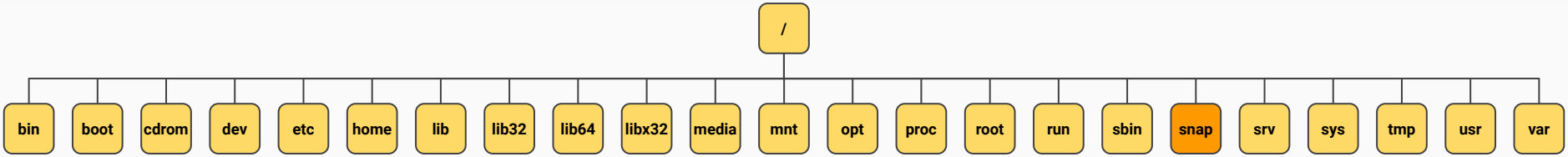


- **/sbin** - также, как и **/bin**, содержит бинарные исполняемые файлы, необходимые для работы операционной системы и приложений. Однако эти файлы требуют повышенных привилегий (права суперпользователя) для выполнения, поскольку они часто управляют самыми важными настройками системы и службами.

- Важнейшими файлами являются **/sbin/init** - является первым процессом, который запускается при загрузке системы (**systemd**), **/sbin/halt** и **/sbin/shutdown** - используются для завершения работы системы, а также **/sbin/mount** и **/sbin/umount** - используются для монтирования и размонтирования файловых систем, позволяя подключать и отключать устройства хранения данных.

- **/sbin** повторяет поведение **/bin** еще в одном аспекте - данный каталог является ссылкой на каталог **/usr/sbin** - как мы помним, это связано с историческими причинами и ограничением размера диска в былые времена.

Каталог **/snap**

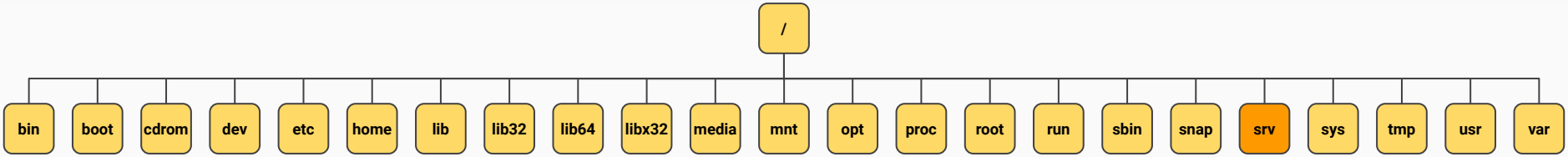


- **/snap** - это место, где хранятся снап-пакеты (snap packages). Snap-пакеты - это формат приложений, который был введен компанией Canonical и предоставляет способ упаковки приложений в изолированные неделимые контейнеры, что, как утверждается, делает их более безопасными (но более медленными).

- Внутри каталога **/snap** создаются специальные символические ссылки (мы поговорим о них позже), позволяющие приложениям, упакованным как снап-пакеты, быть доступными из системного **PATH**. Это означает, что после установки снап-пакета мы можем запустить его, введя его имя в терминале, без необходимости указывать полный путь.

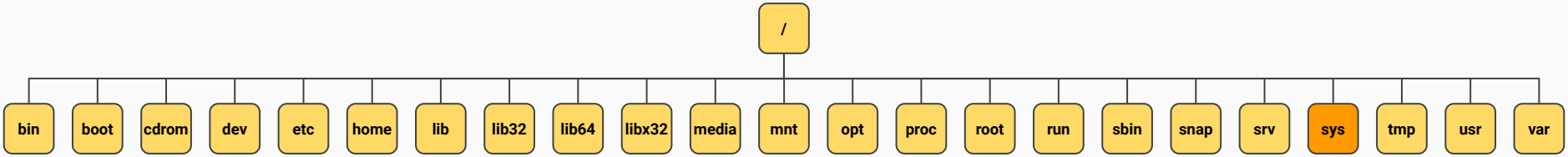
- Снап-пакеты автоматически обновляются в фоновом режиме, что помогает поддерживать приложения в актуальном состоянии. Однако при желании автоматическое обновление можно отключить.

Каталог **/srv**



- **/srv** - этот каталог может использоваться для хранения данных, связанных со службами и серверами (FTP сервера, Apache и т.д.). Однако чаще всего **/srv** не используется вообще, т.к. по историческим причинам данные серверов располагаются в каталоге **/var**.

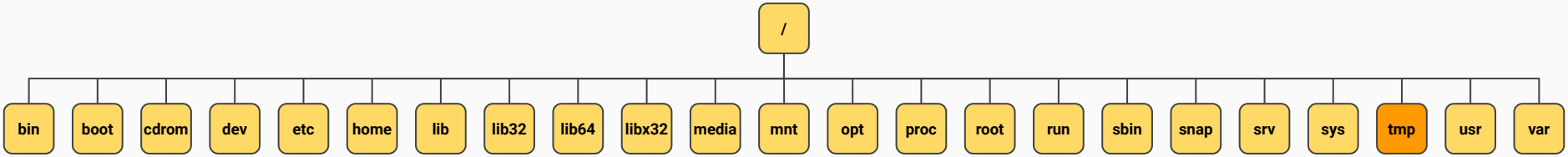
Каталог **/sys**



- **/sys** - предоставляет доступ к различным системным параметрам. Отличается от **/proc** (предназначен для анализа процессов и общего обзора состояния системы) тем, что ориентирован на управление аппаратными устройствами (например, драйверами) и параметрами ядра .

- В числе прочего в **/sys** можно изменять параметры устройств (отключить или включить какой-либо из сетевых интерфейсов, изменить состояние энергосбережения для USB-портов), получить информацию о состоянии батареи (на ноутбуках и устройствах с поддержкой батареи), а также узнать температуру процессора.

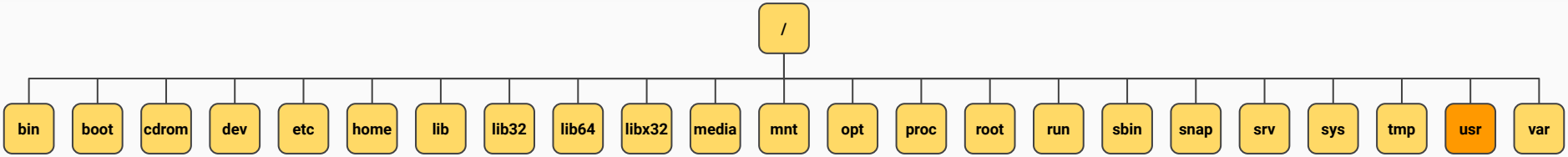
Каталог **/tmp**



- **/tmp** - представляет собой специальный каталог, который используется для временного хранения файлов и данных во время выполнения различных программ. Важно помнить - **/tmp** может очищаться при каждой загрузке компьютера, поэтому важные данные там хранить не надо.

- Каталог **/tmp** обычно доступен для всех пользователей системы. Это означает, что любой пользователь может создавать и читать файлы в этом каталоге. Это может быть полезно, например, для обмена временными данными между разными программами или пользователями.

Каталог **/usr**

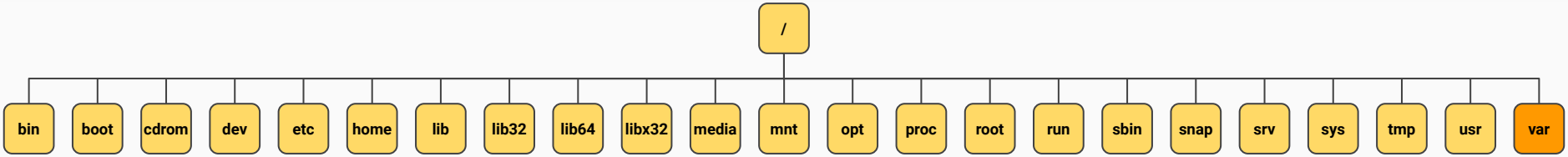


- **/usr -** название данного каталога расшифровывается как Unix System Resources, и, как уже многократно упоминалось, здесь хранятся важнейшие исполняемые утилиты, необходимые для работы операционной системы, а также библиотеки для этих утилит.

- Также не лишним будет вспомнить, что на **/usr/bin** и **usr/sbin** ссылаются корневые каталоги **/bin** и **/sbin**, а на **/usr/lib, /usr/lib32, /usr/lib64** и **/usr/libx32** соответственно ссылаются корневые каталоги **/lib, /lib32, /lib64 и /libx32** (вызвано историческими причинами - попытка решить старое ограничение на размер диска, а также желание поддержать обратную совместимость).

- В каталогах **/usr/share/doc** и **/usr/share/man** хранится документация для установленных, а в **/usr/local** расположены программы, которые не входят в стандартный репозиторий, но могут использовать общие библиотеки.

Каталог **/var**



- **/var** - данный каталог содержит переменные данные, которые могут изменяться в процессе работы системы, но сохранятся после перезагрузки системы. Также здесь иногда размещают файлы, связанные с сайтами веб-серверов - Apache или Nginx (**/var/www**) или файлы серверов баз данных (**/var/lib/mysql**).

- В качестве примера важнейших файлов можно упомянуть логи (**/var/log**). Эти файлы записывают информацию о событиях, происходящих в системе и приложениях, и могут быть полезными для диагностики. Другой пример - каталог **/var/spool**, который используется для хранения файлов, ожидающих обработки, таких как печать (**/var/spool/cups**) или почта (**/var/spool/mail**).

Навигация по файловой структуре

Введение

- В данной подглаве мы впервые начнем работать с командной строкой (также называется терминалом или консолью - когда-то эти термины обозначали разные явления, но сегодня это уже не имеет значения) операционной системы. Открыть командную строку можно двумя основными способами. Во-первых, есть возможность нажать на клавиатуре комбинацию клавиш **Ctrl Alt T**. Во-вторых, можно нажать мышкой на вкладку ***Applications***, найти в предоставленном списке пункт ***Terminal*** и нажать мышкой уже на него.

- Далее мы познакомимся с основами навигации по файловой структуре операционной системы, научимся переключаться между рабочими каталогами, просматривать содержимое каталогов, определять свое местоположение среди каталогов и многое другое. Перед тем, как непосредственно приступить к работе, надо учесть два момента. Во-первых, после открытия командной строки мы оказываемся в каталоге нашего пользователя (если имя нашего пользователя **user**, то каталог будет называться **/home/user** - также для личного каталога есть псевдоним **~**). Во-вторых, для удобства глаз можно уменьшать/увеличивать размер символов на экране. Для увеличения символов поможет **Ctrl Shift +**, а для уменьшения - **Ctrl -**.

Команда **cd** - основа навигации

- С помощью команды **cd** можно осуществлять переход между каталогами - сначала пишется сама команда **cd**, а после нее путь к нужному каталогу. Для примера перейдем в директорию с конфигурациями - **etc**:

```
cd /etc
```

- Если мы ставим перед путем слеш “/”, то путь подразумевается от корневого каталога. Если же слеша нет, то началом подразумевается тот каталог, в котором мы в данный момент находимся (в контексте командной строки). Примем, что мы находимся в каталоге **/etc** - чтобы перейти в его внутренний каталог **default** (полный путь **/etc/default**), мы можем просто написать **cd default** и результат будет успешным:

```
cd default
```

Если бы мы находились в другом каталоге (не **/etc**), то для перехода нам пришлось бы прописывать полный путь (иначе была бы ошибка):

```
cd /etc/default
```

Команда **cd** - использование спецсимволов ~, .., .

- Команда **cd** позволяет использовать ряд специальных символов, которые являются зарезервированными псевдонимами некоторых путей. Например, если мы хотим сразу же оказаться в каталоге того пользователя, под которым мы в данный момент залогинены, то следует использовать символ “**~**”:

```
cd ~
```

- Следующий удобный спецсимвол-псевдоним - двоеточие “**..**”. Он подразумевает, что мы хотим перейти в родительский каталог относительно того каталога, в котором мы сейчас находимся. Предположим, что мы производим работу в каталоге **/etc/default/grub.d** - для того чтобы попасть в родительский каталог (**/etc/default**), нам всего навсего надо написать следующее:

```
cd ..
```

- Последний псевдоним - точка “**.**”. Это ни что иное, как тот каталог, в котором мы в данный момент находимся. Например, если наш каталог - **/etc/default/**, а нам надо попасть в **/etc/default/grub.d**, то для перехода можно использовать точку:

```
cd ./grub.d
```

Команда **cd** - особенности использования спецсимволов

- При переключении каталогов нам позволяется многократное использование спецсимволов. Например, если мы находимся в очень глубокой категории (**/var/lib/systemd/catalog**), то при помощи “**..**” мы можем вернуться не только в родительскую категорию (**/var/lib/systemd**), но и на несколько уровней выше, например, в **/var**:

```
cd ../..
```

- Кроме того, при использовании специальных символов нам вовсе необязательно ограничиваться только ими. Допустим, мы находимся в каталоге **/proc/sys/kernel/keys**, а хотим попасть в **/proc/sys/fs/quota**. Заметим, что первые два каталога у обоих путей одинаковые - следовательно, мы можем “подняться” два раза “наверх” (в общего предка), а затем от него перейти в **fs/quota**:

```
cd ../../fs/quota
```

Команда **ls** - получение файлов и подкаталогов внутри каталога

- Команда **ls** позволяет получить список всех файлов и каталогов внутри того каталога, в котором мы находимся на данный момент. Кроме того, можно посмотреть информацию о владельце файла, время его создания и изменения, права и тип (файл, каталог, ссылка и т.д.). Представим, что мы находимся в корневом каталоге **/** и выполняем команду **ls** - в результате получим список названий всех каталогов, которые обсуждались в предыдущей подглаве:

```
ls
```

- Если мы перейдем в каталог нашего пользователя **/home/user** и выполним команду **ls**, то увидим, что в этом каталоге нет файлов. Тем не менее, там есть файлы, но они являются скрытыми. Дело в том, что в Linux те файлы и директории, которые начинаются с точки, по умолчанию не видны. Однако это можно исправить, если применить к команде **ls** параметр **-a**. В этом случае мы уже увидим название некоторых файлов, например, **.bashrc**:

```
ls -a
```

Команда **ls** - получение детальной информации

- Иногда нам мало только названий вложенных файлов и каталогов - мы желаем получить размер файлов, владельца и т.д. Для этого у команды **ls** есть еще один параметр **-l** (маленькая **L**), который поможет отобразить нужные нам данные:

```
ls -l
```

Ниже расположен пример возможного результата исполнения команды **ls -l**:

```
-rw-rw-rw- user user 4096 Oct 29 12:00 index.html
drwx------ user user  256 Oct 22 14:00 files
lrwxrwxrwx root root    7 Oct 14 16:00 mybin -> usr/bin
```

- У нас есть возможность использовать несколько параметров одновременно - если мы хотим получить детальную информацию о содержимом каталога, притом одновременно отобразить все скрытые файлы, то следует использовать как параметр **l** (маленькая **L**), так и параметр **a**:

```
ls -la
```

Команда **pwd** - путь рабочего каталога

- Если мы забыли полный путь каталога, в котором в данный момент происходит работа, либо хотим передать этот путь в виде строки для какой-либо дальнейшей обработки (об этом мы поговорим позднее), то можно использовать команду **pwd**. Например, если мы находимся в каталоге **/usr/lib** и напишем команду **pwd**, то на экране закономерно отобразится **/usr/lib** в виде строки:

```
pwd
```

- Следует знать, что **pwd** поддерживает дополнительные параметры. Например, если мы хотим увидеть тот путь, по которому мы явно попали в каталог, то следует использовать параметр **-L**, а если мы хотим увидеть каталог, на который ссылается данный каталог (если он ссылается), то применяется параметр **-P**. Попробуем для наглядности использовать второй параметр (**-P**), находясь в каталоге **/bin**. Как мы помним, он ссылается на каталог **/usr/bin**, поэтому на экране мы увидим именно **/usr/bin**:

```
pwd -P
```

Работа с файлами и каталогами: создание, удаление, перемещение

Команда **touch** - создание файла

- Рано или поздно нам надо будет создать какой-либо файл. Наполнение его информацией нас пока не интересует, а вот непосредственно для создания используется команда **touch** после которой надо написать имя нового файла. Важно помнить, что каталог, в котором мы создаем файл, должен существовать:

```
touch document.txt
```

- Если мы применим команду **touch** к уже существующему файлу, то его внутреннее содержимое останется неизменным. Единственное изменение - обновится время обновления этого файла и время последнего доступа к файлу.

- С помощью команды **touch** можно создать несколько файлов одновременно (для этого просто надо перечислить их через пробел), более того, можно создавать файлы с пробелами в имени (для этого имя файла надо написать в кавычках). Ниже создаются файлы **main.java**, **app.py** и **my document.txt**:

```
touch main.java app.py 'my document.txt'
```

Команда **rm** - удаление файла

- Для удаления файлов используется команда **rm**, после которой следует указать путь к удаляемому файлу:

```
rm document.txt
```

- Если нам нужно удалить несколько файлов, то, как и в случае с **touch**, мы можем указать несколько путей через пробел. Кроме того, если в пути некого файла имеется пробел, то мы должны обернуть этот путь в кавычки:

```
rm main.java app.py 'my document.txt'
```

Команда **rm** - удаление файлов, подпадающих под специальный шаблон

- При написании путей для удаления файлов можно использовать т.н. “регулярные выражения”, которые позволяют задавать специальный шаблон. Все файлы, которые соответствуют этому шаблону, будут удалены. Далее представлены специальные символы, которые могут быть использованы для задания шаблона:
  - **?** - любой символ один раз
  - ***** - любой символ сколько угодно раз (даже 0 раз)
  - **{index,app,main}** - любой из вариантов, перечисленных через запятую

- Пример удаления файлов, имеющих расширение **.py**:

```
rm *.py
```

- Пример удаления файлов, имеющих расширение **.java**, но в названии содержащих только 4 символа:

```
rm ????.java
```

- Пример удаления файлов, имеющих любое название, но только с расширением **.py**, **.java** или **.php**:

```
rm *.{py,java,php}
```

Команда **mkdir** - создание каталогов (директорий)

- Закономерно, что кроме создания файлов Ubuntu позволяет создавать своего рода контейнеры для файлов - каталоги (директории). Осуществляется это при помощи команды **mkdir**, после которой следует указывать путь к новому каталогу (если каталог по этому пути существует, система сообщит об ошибке):

```
mkdir documents
```

- Команда **mkdir** позволяет создавать несколько каталогов одновременно, разделяя их пути пробелом. Как и во всех предыдущих случаях, если в названии каталога есть пробелы, то путь этого каталога стоит обернуть в кавычки:

```
mkdir files games 'my new directory'
```

- Иногда нам потребоваться создать каталог с большим уровнем вложенности (по сути несколько вложенных друг в друга каталогов). Команда **mkdir** может нам в этом помочь, но для этого случая следует указывать параметр **-p**:

```
mkdir -p some/deep/directory
```

Команда **rmdir** - удаление пустых каталогов (директорий)

- Для удаления ненужных пустых каталогов в операционных системах на ядре Linux используется команда **rmdir**, после которой нужно указать путь удаляемого каталога:

```
rmdir documents
```

- Если нам надо удалить несколько пустых каталогов, то мы можем указать их после команды **rmdir** через пробел (как и всегда, если какой-либо путь содержит пробелы, то его следует оборачивать в кавычки):

```
rmdir games 'my new directory'
```

- Как и **rm**, команда **rmdir** поддерживает шаблоны на основе подъязыка “регулярных” выражений и специальные символы ?, * и {}. Например, если мы хотим удалить любую пустую директорию, в название которой входит пять (5) символов (например, ***files***), то следует применить следующую конструкцию:

```
rmdir ?????
```

Команда **rm -r** - удаление любых каталогов (директорий)

- Если мы попытаемся удалить каталог ***some*** при помощи команды **rmdir**, то мы увидим ошибку, которая гласит, что в каталоге что-то есть (другой каталог - ***deep***) и удалить его нельзя. В данном случае нам спешит на помощь старая добрая команда **rm**, которую следует выполнить с дополнительным параметром **-r**, а затем после нее уже написать название удаляемого каталога. В данном случае последовательно (рекурсивно) будет удалено содержимое всех внутренних каталогов, а затем уже сам каталог ***some***:

```
rm -r some
```

- Иногда можно столкнуться с советом использовать дополнительный параметр **-f** (*force*) при удалении каталогов. Этот параметр означает, что если удаляемого каталога не будет, то ошибки не произойдет, а также у нас не будет появляться сообщение о подтверждении удаления определенных файлов:

```
rm -rf some
```

Команда **cp** - копирование файлов (простейшие примеры)

- Для копирования файлов используется команда **cp** - в самом простом случае сразу после нее следует указать путь к копируемому файлу, а затем полный путь к тому целевому месту, куда файл будет скопирован. Если по этому пути уже расположен другой файл, то этот файл будет перезаписан, если же по этому пути расположен каталог, то файл будет скопирован в этот каталог с оригинальным названием. Далее пример простого копирования данных файла ***file.txt*** в файл ***copy.txt***:

```
cp path/to/file.txt path/to/copy.txt
```

Пример простого копирования файла ***file.txt*** в директорию ***/home/user***:

```
cp path/to/file.txt /home/user
```

- Для копирования одновременно нескольких файлов в другой каталог после команды **cp** через пробел следует указать копируемые файлы, а последним аргументом - путь к каталогу, в который мы хотим копировать файлы:

```
cp main.java index.php app.py some/target/directory
```

Команда **cp** - копирование файлов (использование регулярных выражений)

- Как и во всех предыдущих случаях, для взаимодействия (в данном случае копирования) с файлами можно использовать подъязык регулярных выражений. Однако крайне важно знать следующее - регулярные выражения принципиально игнорируют скрытые файлы и каталоги (их название начинается с точки), поэтому при копировании можно не досчитаться некоторых важных данных. Далее пример того, как скопировать все (кроме скрытых файлов и каталогов) из каталога **/home/user/first/*** в каталог ***/home/user/second***:

```
cp /home/user/first/* /home/user/second
```

Команда **cp** - копирование каталогов (директорий)

- Для копирования каталогов и их содержимого нам необходимо использовать ту же самую команду **cp**, но с дополнительным параметром **-r**. После команды и параметра следует указать тот каталог (или несколько каталогов через пробел), который мы хотим скопировать, а затем, в самом конце, тот каталог, в который мы собираемся копировать. Далее пример копирования каталога **/home/user/programs** в каталог **/home/user/archive** (в результате в появится каталог **programs** со всем содержимым):

```
cp -r /home/user/programs /home/user/archive
```

- Если же надо скопировать только файлы, но не сам каталог, то в самом конце пути копируемого каталога следует добавить слеш и точку - “**/.**”:

```
cp -r /home/user/programs/. /home/user/archive
```

Команда **mv** - перемещение (переименование) файлов

- Для перемещения (что по сути можно трактовать как переименование) файлов мы должны использовать команду **mv**. Сначала рассмотрим простейший пример - предположим, что у нас есть файл ***test.java***, а мы хотим переименовать его в ***example.java***. Для осуществления нашей задумки сначала пишем команду **mv**, затем путь к файлу, который должен быть переименован, а затем путь к новому имени этого файла (если такой файл есть, он будет перезаписан):

```
mv test.java example.java
```

- Далее предположим, что у нас есть файл ***index.php***, а мы хотим переместить его в уже существующий каталог ***/home/user/php*** - если нам надо переместить файл под своим именем, то в качестве последнего аргумента мы можем просто указать целевой каталог (без имени файла в конце):

```
mv index.php /home/user/php
```

- Для перемещения нескольких файлов в другой каталог мы будет использовать подход из предыдущего примера, но названий файлов будет несколько:

```
mv app.py utils.py lib.py /home/user/python
```

Команда **mv** - перемещение (переименование) файлов (шаблоны)

- Для перемещения файлов нам опять позволяется использовать шаблоны с использованием регулярных выражений. Например, воспользуемся звездочкой “*”, чтобы скопировать все файлы (кроме скрытых) из нашего рабочего каталога на данный момент (он может быть любым) в каталог ***/home/user/data*** (надо его создать перед перемещением, иначе будет ошибка):

```
mv * /home/user/data
```

Команда **mv** - перемещение (переименование) каталогов (директорий)

- Каталоги в Ubuntu перемещаются схожим с файлами образом - после команды mv указываем название того каталога (или нескольких каталогов через пробел), а в самом конце указываем каталог, в которых хотим переместить все указанное до этого. Далее пример простого переименования каталога ***java_src*** в ***java_src_archive***:

```
mv java_src java_src_archive
```

- Теперь попробуем переместить несколько каталогов в другой каталог (при этом сохраняя их оригинальные названия) - например, переместим каталоги ***php***, ***java*** и ***python*** в каталог ***home/user/programming***:

```
mv php java python /home/user/programming
```

- Наконец, переместим каталог ***javascript*** в каталог ***/home/user/programming*** и одновременно переименуем этот каталог ***javascript*** в ***js***:

```
mv javascript /home/user/programming/js
```

Запись данных в файл

Команда **echo** - простейший способ записи в файл

- Для записи в файл можно использовать команду **echo**, которая по умолчанию предназначена для вывода информации в консоль. Сначала воспользуемся ей по прямому назначению и просто напишем в консоли слово ***Hello***:

```
echo Hello
```

Самое удивительное, что текст, который мы увидели на экране, можно перенаправить напрямую в файл - для этого после текста нам надо сразу поставить знак **>** и написать название файла (даже если файла нет - не беда, он будет создан автоматически):

```
echo Hello > text.txt
```

- Важно помнить, что простое перенаправление **>** каждый раз перезаписывает файл. В том случае, если нам надо не перезаписать, а дополнить текст в файле, следует применить конструкцию **>>** (заметим, что мы обернули текст в кавычки - это сделано для того, чтобы сохранить пробел в начале текста):

```
echo ' World' >> text.txt
```

Перенаправление потока **stdout** в файл

- В мире современных операционных систем у каждой процесса (программы, которая выполняется на компьютере) есть три так называемых потока (***streams***) ввода/вывода (не путать со стандартными потоками - ***threads***), ответственных за ввод данных (**stdin**), вывод данных (**stdout**) и вывод ошибок (**stderr**) (по сути, это виртуальные файловые дескрипторы). Когда мы выполняем команды в консоли, то мы запускаем процессы, которые могут писать данные в **stdout** (или **stderr**, но об этом позже), затем это “читает” консоль и отображает на экране. Однако у нас есть возможность не отображать данные, а перехватить их с помощью операторов **>** и **>>** и записать в файл. Покажем это на примере команды **ls -la**, которую будем использовать для получения содержимого каталога ***/home/user***:

```
ls -la /home/user > users_info.txt
```

Далее покажем пример вставки в файл данных из директории ***/home*** - сделаем это с помощью оператора **>>**, который не перезаписывает файл, а дополняет его:

```
ls -la /home >> users_info.txt
```

Использование текстового редактора **nano** для записи данных в файл

- Для работы с содержимым текстовых файлов в командной строке можно использовать настоящие текстовые редакторы. Конечно, по функционалу они уступают таким мощным графическим программам как **Microsoft Word**, однако они дают определенное удобство и ускоряют работу. Одним из самых лучших редакторов (по мнению автора этих строк) считается **nano**. Для работы с ним (если он уже установлен) в командной строке следует написать слово **nano**, а затем путь к тому файлу, в который мы хотим записать данные (так можно сделать даже в том случае, если файл не существует):

```
nano document.txt
```

- В открывшемся окне у вас будет возможность вводить текст, редактировать его, производить поиск, копировать данные и вставлять в нужное место. Все возможные опции будут перечислены внизу экрана (небольшое пояснение - в инструкциях знак **^** означает кнопку **Ctrl**, а знак **M** - кнопку **Alt**). Для того, чтобы закончить работу, следует нажать комбинацию клавиш **Ctrl X**. Если не было изменений - на этом **nano** закроется. Если же изменения были, и мы хотим их сохранить, надо нажать кнопку **Y**, если же передумали и не хотим - кнопку **N**, а после этого нажать кнопку **Enter**, чтобы подтвердить свой выбор.

Использование текстового редактора **vi** для записи данных в файл

- Иногда мы можем столкнуться с очень старым текстовым редактором **vi** или его более новой версией **vim**. Для современных пользователей он покажется не очень удобным, но и сегодня существуют разработчики, которые его ценят и уважают. Перед тем, как работать с **vi**, перейдем в каталог нашего пользователя и настроим **vi** так, чтобы он перестал поддерживать некие очень древние принципы, которые будут сильно нам мешать при работе с файлами:

```
cd ~
echo 'set nocompatible' > .vimrc
```

- Для работы с файлом надо написать команду **vi**, а потом указать путь:

```
vi document.txt
```

- Редактор **vi** работает в двух режимах - вставки данных и команд. Для перехода в режим вставки данных следует нажать кнопку **I** - мы получим возможность писать и редактировать текст. Для перехода в режим команд следует нажать кнопку **Esc** - в этом режиме основное - сохранить текст и закрыть документ. Для сохранения с закрытием нажимаем кнопку **:**, затем пишем **wq** и нажимаем **Enter**. Для простого закрытия - нажимаем кнопку **:**, затем пишем **q!** и нажимаем **Enter**.

Просмотр содержимого файла

Просмотр содержимого файла при помощи редакторов **nano** и **vi**

- Самый простой и удобный вариант для просмотра содержимого (а также для редактирования этого содержимого) - использование текстовых редакторов nano и vi. Для начала заходим в файл с помощью команды **nano** или **vi**, а затем пишем путь к файлу. Пример для **nano**:

```
nano /path/to/file.txt
```

  Пример для **vi**:

```
vi /path/to/file.txt
```

- Находясь внутри открытого файла, в обоих редакторах мы можем перемещаться по тексту при помощи кнопок на клавиатуре (вверх, вниз, направо, налево), а также при помощи кнопок **PageUp** и **PageDown**, которые позволяют пролистывать вверх и вниз целые страницы текста.

- Надо заметить, что редакторы нам не всегда будут доступны, кроме того, иногда нам будет нужно не просто посмотреть содержимое или отредактировать его, а передать данные другой программе, хитро обработать и т.д. Поэтому далее мы рассмотрим дополнительные возможности получения данных из файла.

Команды **cat** и **tac**

- Если нам надо просто вывести в консоль все строки файла, то можно применить команду **cat**, после которой следует указать путь к файлу или к путям нескольких файлов (в этом случае строки файлов будут показаны в той последовательности, в какой были указаны их пути). Далее пример отображения строк одного файла:

```
cat /path/to/file.txt
```

Пример отображения строк нескольких файлов:

```
cat /path/to/file.txt /path/to/another/file.txt
```

- Иногда может потребоваться строки не в оригинальном порядке, а наоборот - от первой к последней. Для этого следует использовать команду **tac** - в качестве аргумента она может принимать как путь к одному файлу, так и к нескольким. Далее пример отображения строк файла в обратном порядке:

```
tac /path/to/file.txt
```

Пример отображения строк нескольких файлов:

```
tac /path/to/file.txt /path/to/another/file.txt
```

Команды **head** и **tail**

- Порой на надо получить не все содержимое файла, а только первые несколько строк. Для этого идеально подходит команда **head**, которой нужно в качестве первого параметра передать тире и количество необходимых строк, а затем вписать путь к файлу или несколько путей к файлу (тогда количество строк будет отображено для каждого файла). Далее пример получения первых трех строк в одном файле:

```
head -3 /path/to/file.txt
```

- Для обратной операции - получения нескольких последних строк - следует использовать команду **tail**. Как и в случае с командой **head**, для **tail** нужно в качестве первого параметра передать тире и количество необходимых строк, а затем вписать путь к файлу или несколько путей к файлу. Далее пример получения последних пяти строк в одном файле:

```
tail -5 /path/to/file.txt
```

Команды **more** и **less**

- Часто в процессе разработки нам приходится просматривать большие файлы, размер которых не умещается на экране. Чтобы разбить файл на условные “страницы”, а затем переключаться между ними, можно использовать команду **more**, после которой следует писать название файла. Для переключение на уже просмотренные страницы используется кнопка “вверх” или **PageUp**, а для навигации вперед - кнопку “вниз”, **PageDown** или пробел. Важная особенность - при пролистывании до конца происходит автоматический выход из многостраничного чтения. Далее пример открытия файла для чтения при помощи команды **more**:

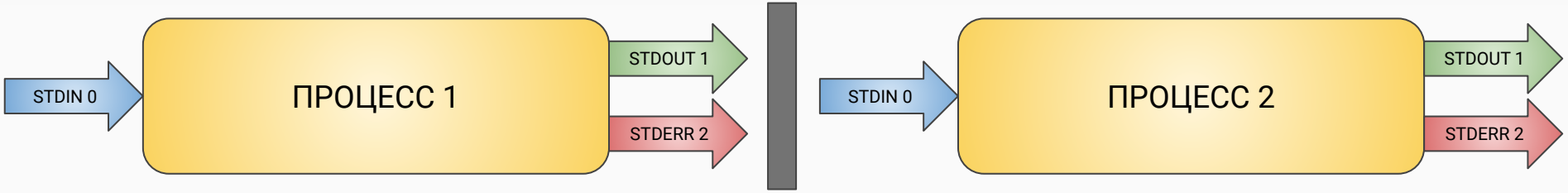
```
more /path/to/file.txt
```

- Также для многостраничного чтения существует еще одна команда - **less**. Она превосходит **more** по всем параметрам, например, при достижении конца файла не происходит выхода из программы, кроме того - доступен поиск (следует написать слеш “/”, затем интересующее нас слово, а потом нажать Enter). Для выхода из этой программы следует нажать кнопку **Q**. Далее пример открытия файла для многостраничного чтения при помощи команды **less**:

```
less /path/to/file.txt
```

Перенаправление потоков ввода и вывода, использование каналов (pipes)

Введение



- Мы уже знаем, у каждого процесса, который запускается в контексте операционной системы, есть три потока (**streams**) - ввода (**stdin**), вывода (**stdout**) и ошибок (**stderr**). Однако из всех троих мы работали только с потоком вывода (**stdout**), который перенаправляли либо для полной перезаписи какого-либо файла (оператор **>**), либо для дополнения файла (оператор **>>**). В этой подглаве мы подробно рассмотрим работу уже со всеми потоками, рассмотрим их взаимодействие и объединение.

- Еще одним любопытным моментом ввода/вывода, которого мы коснемся несколько позже, является возможность перенаправлять поток вывода/ошибок одного процесса в поток ввода другого процесса и повторять этот алгоритм сколько потребуется. Это позволяет нам строить довольно эффективные конвейеры по обработке данных, что было бы немыслимо без этого механизма.

Поток ввода (**stdin**)

- Поток ввода (**stdin**) нужен для передачи данных процессу. Оператором для перенаправления является **<** или **<0**. В стандартном виде **stdin** перенаправляется нечасто, так как чаще команды просто принимают данные через аргументы (как путь к файлу или как строку). Но существуют команды, которые могут вести себя не так. Допустим, что у нас есть файл ***lines.txt***. Мы заранее не знаем, сколько строк он содержит, но можем это сделать с помощью команды **wc -l**:

```
wc -l lines.txt
```

  В ответ мы получим слишком многоречивый ответ, содержащий имя файла:

```
10 lines.txt
```

- Мы можем пойти другим путем - не будем предоставлять команде путь к файлу, а перенаправим ей в поток **stdin** данные из файла - в этом случае команде ничего не останется, как только вернуть 10, без всякого упоминания файла:

```
wc -l < lines.txt
```

```
10
```

Поток вывода (**stdout**)

- С потоков вывода (**stdout**) мы уже хорошо знакомы - для его перенаправления используется оператор **>** (полностью перезаписывает файл, если он уже существует) или **>>** (добавляет данные потока в конец файла). Также можно использовать более полные версии этих операторов **1>** и **1>>**. Приведет пример перенаправления потока вывода. Предположим, что изначально у нас есть файл ***document.txt*** и нам нужна его копия с именем ***copy.txt*** - конечно, можно использовать команду cp, но для практики применим именно перенаправление. Для этого прочитаем файл ***document.txt*** при помощи команды **cat**, а потом направим прочитанное (**stdout**) в файл ***copy.txt***:

```
cat document.txt > copy.txt
```

- Если мы хотим добавить в существующую копию (а не полностью ее перезаписать, если она существует) данные какого-либо другого файла (например, document-2.txt), нам следует применить оператор **>>**:

```
cat document-2.txt >> copy.txt
```

Поток ошибок (**stderr**)

- Если при использовании какой-либо команды произошел сбой (или хотя бы нехватка прав что-либо сделать), то команда может записать информацию об этой ошибке не в поток вывода, а в поток ошибок (**stderr**). Важно помнить - если мы работаем в командной строке, то когда она отображает данные, ей не важно, из какого потока они пришли - вывода или ошибок - она читает оба. Однако если мы перенаправляем данные в файл, то тип потока становится важным. Для явного доступа к потоку ошибок (и его перенаправления) используется оператор **2>** или **2>>**. Попробуем использовать (будучи пользователем *user*) команду **ls**, чтобы прочитать содержимое двух директорий ***/home/user*** и ***/root***, притом поток вывода будем направлять в файл ***success.txt***, а поток ошибок в файл ***error.txt***:

```
ls /home/user /root > success.txt 2> error.txt
```

Мы увидим, что содержимое каталога /home/user записалось в файл success.txt, а в error.txt будет информация об ошибке: ***ls: cannot open directory '/root': Permission denied***. Дело в том, что пользователь *user* может просматривать свой каталог и эта часть команды запишется в **stdout** (***success.txt***), но *user* не имеет доступа к каталогу ***/root***, поэтому эта часть команды запишется в **stderr** (***error.txt***).

Перенаправление потоков **stderr** и **stdout** друг в друга

- Ранее было сказано, что консоль умеет перехватывать одновременно и поток вывода и поток ошибок (вернее было бы сказать, консоль перенаправляет **stderr** в **stdout**). Этот функционал доступен не только для консоли, но и для нас. Для подтверждения этого представим ситуацию, что нам надо записать данные **stdout** и **stderr** одновременно в один файл, например, **common.txt**. Безусловно, можно это сделать с помощью уже имеющихся знаний:

```
ls /home/user /root >> common.txt 2>> common.txt
```

Однако более универсальным подходом является конструкция **2>&1** - она обозначает то, что все данные потока **stderr** будут направлены в поток **stdout** (обратная конструкция **1>&2** тоже возможна, но она делает наоборот - перенаправляет поток вывода в поток ошибок):

```
ls /home/user /root > common.txt 2>&1
```

- Некоторые версии консоли поддерживают еще один вариант перенаправления **stderr** в **stdout** (**&>**), но надо помнить, что такой подход может работать не везде:

```
ls /home/user /root &> common.txt
```

Использование каналов (**pipe**) для передачи данных между командами

- Иногда нам может понадобиться получить данные (из файла или с помощью какой-либо команды), потом обработать их каким-либо хитрым образом (отсортировать строки), затем взять первые 10 строк и после этого записать результат в файл. Такой конвейер обработки данных осуществляется с помощью каналов (|), которые позволяют перенаправлять поток **stdout** одной команды в поток **stdin** другой команды. Реализуем с помощью каналов описанный выше пример:

```
cat some-text-file.txt | sort | head -10 > result.txt
```

- Время от времени нам может понадобиться передать данные между командами не только из потока **stdout**, но и из потока **stderr** - это тоже возможное при помощи знакомого нам оператора **2>&1** (перенаправим поток ошибок в поток вывода). Для примера посчитаем количество слов в сообщении об ошибке при попытке прочитать содержимое каталога **/root** (не будучи администратором) - в конце воспользуемся командой **awk**, которая возьмет из потока ввода только второй столбец (по умолчанию **wc** возвращает три столбца - кол-во строк, кол-во слов и кол-во символов):

```
ls /root 2>&1 | wc | awk '{print $2}'
```

Поиск файлов (по названию и содержимому)

Команда **find** - основы поиска файлов по их названию

- Самый мощным инструментом, который позволяет находить файлы по их названию, а также сразу же производить операции над найденными файлами, является команда команда **find**. Базовая схема ее использования представлена ниже (операция над найденным не является обязательным элементом):

```
find КАТАЛОГ КРИТЕРИИ_ПОИСКА [ОПЕРАЦИЯ_НАД_НАД_НАЙДЕННЫМ]
```

- Приведем пример поиска файла (критерий **-type f**) с названием ***sources.list*** (критерий **-name "sources.list"**) в каталоге ***/etc*** (поиск будет произведен как в самом каталоге, так и во вложенных каталогах):

```
find /etc -type f -name "sources.list"
```

- Далее пример поиска каталога (критерий **-type d**) с названием ***core*** (критерий **-name "core"**) в каталоге **/proc**:

```
find /proc -type d -name "core"
```

Команда **find** - поиск файлов по их названию через регулярные выражения

- Важно знать, что команда **find** поддерживает поиск не только по названию, но и по какой-то его части - для этого используются уже знакомые нам базовые регулярные выражения. Напомним их: **?** - любой символ один раз, ***** - любой символ сколько угодно раз (даже 0 раз). Однако в отличие от предыдущих случаев шаблон вида **{index,app,main}** (один из предложенных вариантов) не поддерживается. Для примера найдем все файлы в каталоге ***/usr***, имеющие расширение **.py** (таким образом, перед расширением может быть любой символ сколь угодно раз - т.е. идеальным кандидатом является звездочка):

```
find /usr -type f -name "*.py"
```

Далее приведем пример поиска в каталоге ***/home*** - предположим нам надо найти все каталоги, название которых состоит из четырех любых символов:

```
find /home -type d -name "????"
```

Команда **find** - использование расширенных регулярных выражений (теория)

- Если для **find** нам не доступны некоторые инструменты базовых регулярных выражений, то с другой стороны нам доступны еще более мощный синтаксис расширенных регулярных выражений:
  - **[0-9]** - любая цифра от 1 до 9 (доступен любой интервал)
  - **[a-z]** - любая маленькая латинская буква от a до z (любой интервал)
  - **[А-Я]** - любая большая русская буква от а до я (любой интервал)
  - **[a-zA-z]** или **[A-z]** - любая латинская буква (любой интервал)
  - **\w** - любая буква или любая цифра
  - **.** - любой символ
  - ***** - повторение предыдущего символа сколь угодно раз (хоть ноль)
  - **+** - повторение предыдущего символа как минимум один раз
  - **?** - повторение предыдущего символа ноль или один раз
  - **{5}** - повторение предыдущего символа ровно пять раз
  - **{7,}** - повторение предыдущего символа минимум семь раз
  - **^** - знак начала пути (т.е. символы после него должны идти в начале пути)
  - **$** - знак конца пути (т.е. символы перед ним должны идти в конце пути)
  - **[^c-k]** - запрещенный интервал буков
  - **[^2-7]** - запрещенный интервал цифр
  - **[^A-z0-9]** - одновременно запрещенный интервал цифр и буков
  - **(user|cookie)** - один из предложенных вариантов (user или cookie)

Команда **find** - использование расширенных регулярных выражений (примеры)

- Перед началом использования расширенных выражений надо помнить три правила. Во-вервых, их надо включить опцией **-regextype posix-egrep**, во-вторых при поиске надо использовать критерий **-regex**, и, наконец, в-третьих, поиск учитывает не только имя файла или директории, а весь путь (начиная от корня).
- Найдем все файлы, которые начинаются со слова ***app***, затем идут любые 5 цифр, а затем расширение либо **.c** либо **.py**:

```
find / -type f -regextype posix-egrep -regex ".*/app[0-9]{5}\.(c|py)$"
```

Далее приведем пример поиска только тех каталогов, у который в пути встречается каталог с названием **cpu**:

```
find / -type d -regextype posix-egrep -regex ".*/cpu/.*"
```

Попытаемся найти все файлы, у которых в названии (но не в пути) нет буквы **z**:

```
find / -type f -regextype posix-egrep -regex ".*/[^z/]+$"
```

Наконец, найдем все корневые директории, путь которых начинается с буквы **s**:

```
find / -type d -regextype posix-egrep -regex "^/s[^/]*"
```

Команда **find** - дополнительные операции над найденными файлами

- После того, как find находит файл или каталог, он должен произвести с ним какое либо действие или операцию. Операцией по умолчанию является **-print**, которая отображает полный путь к файлу или каталогу. Однако нам позволяется совершать и другие действия. Во-первых, это **-delete** (уничтожает найденные файлы):

```
find / -type f -name "some-file-name.txt" -delete
```

Во-вторых, нам доступна операция **-ls**, которая отображает содержимое найденных каталогов:

```
find / -type d -name "some-directory-name" -ls
```

В-третьих, операция **-exec**, которая позволяет производить дополнительную обработку найденных файлов или каталогов с помощью знакомы нам команд. Единственное, что стоить запомнить - доступ к найденному файлу можно получить с помощью **{}**, а сами команды завершаются символами **\;**. Допустим, будем отображать содержимое всех найденных файлов:

```
find / -type f -name "some-file-name.txt" -exec cat {}\;
```

Команда **find** - дополнительные критерии поиска

- Искать файлы можно не только по имени или части имени, но и по другим параметрам. В эти параметры входят:
  - имя владельца файла:

```
find / -user user
```

  - время последнего доступа к файлу (в примере - за последние 10 минут):

```
find / -amin -10
```

  - время последнего изменения файла (в примере - за последние 5 минут):

```
find / -mmin -5
```

  - размер файла (в примере - более 1024 мегабайт)

```
find / -size +1024M
```

Команда **grep** - поиск по содержимому файлов

- Далее мы познакомимся с командой **grep**, которая умеет искать строки в файлах. Прежде всего ознакомиться с базовым применением. Представим, что у нас есть файл ***text.txt***, и нам надо понять, есть ли в этом файле слово ***admin***. Для этого применяется следующая конструкция, которая вернет все строки, где такое слово существует (в начале, конце или середине строки - неважно):

```
grep "admin" text.txt
```

Также мы можем искать нужное значение в нескольких файлах одновременно:

```
grep "admin" text.txt another-text.txt
```

Поиск можно сделать регистронезависимым (при помощи параметра **-i**):

```
grep -i "admin" text.txt another-text.txt
```

Мы можем показать номера строк, в которых были найдены нужные слова (**-n**):

```
grep -n "admin" text.txt another-text.txt
```

Можно даже инвертировать логику - показать строки, где слово не найдено (**-v**):

```
grep -v "admin" text.txt another-text.txt
```

Команда **grep** - поиск файлов по их содержимому

- Команда **grep** обладает еще одной опцией - если мы будем использовать параметр **-r** и вместо пути к файлу подставим путь к некому каталогу, то в результате получим все строки и название всех файлов (из представленного каталога и его вложенных каталогов), где встречается искомое слово. Таким образом можно осуществлять поиск файлов по их содержимому.

- Приведем пример поиска - попытаемся найти все файлы и строки этих файлов в каталоге ***/etc***, где встречается слово ***config***:

```
grep -r "config" /etc
```

- Предыдущая команда вернет нам очень много текста, т.к. кроме названий файлов мы получим те строки, где встречается слово config. Чтобы вернуть только названия файлов, мы должны применить параметр **-l**:

```
grep -rl "config" /etc
```

Команда **grep** - использование расширенных регулярных выражений

- При поиске с использованием команды grep (как для поиска файлов по строкам, так и для поиска строк в файлах) мы можем использовать регулярные выражения, притом сразу самый мощный их вариант - расширенный. Для этого следуем применить параметр **-E**.

- Важно понимать, что поиск при помощи регулярных выражений производится для строк внутри файла, поэтому, например, шаблоны начала (**^**) и окончания (**$**) не будут применяться не для всего файла, а для каждой конкретной строки - если хотя бы одна подойдет, то файл и найденная строка попадет в результат.

- Приведем пример поиска совпадения со строкой, которая может содержать следующие слова - ***admin***, ***root*** или ***main***. Поиск будет производится в каталоге ***/etc***:

```
grep -rE ".*(admin|root|main).*" /etc
```

Права, пользователи и группы

Введение

- В операционных системах на ядре Linux есть возможность создавать различных пользователей, которые обладают уникальным именем и своим личным каталогом (хотя каталога может и не быть), а также имеют различные уровни доступа к разным компонентам и функциям системы - эти уровни доступа принято называть правами.

- Первым является право на чтение (**read**) - в контексте файла оно означает, что пользователь может увидеть его содержимое, для каталога - показать список вложенных файлов и каталогов. Вторым правом является право на запись (**write**), т.е. сможет ли пользователь записать данные в файл (**write** в контексте файла) или создать/удалить файл или каталог (**write** в контесте каталога). Право на исполнение (**execute**) означает, что файл можно запускать как программу (**execute** в контексте файла) или каталог можно посещать при помощи команды **cd** (**execute** в контексте каталога).

- Также существует такой концепт как группа - каждый пользователь входит в одну или несколько групп. Группы нужны для того, чтобы не выдавать права по отдельности. Можно назначить группу файлу, затем выдать права группе, и они будет распространены на всех пользователей, принадлежащих этой группе.

Схема прав

- При использовании команды **ls -l** мы можем увидеть информацию о файлах и каталогах - в числе прочего будет указана информация о правах для трех сущностей - владельца, группы (владелец может не входит в группу) и остальных - тех кто не является владельцем и не входит в группу. Соответствующая буква означает наличие права, а прочерк на месте права - отсутствие этого права.

Права суперпользователя

- В контексте прав, групп и пользователей есть одно исключение - в системе всегда существует один суперпользователь по имени **root**, который может делать абсолютно все - вне зависимости от того, кому принадлежит файл или каталог, и какие на него установлены права. Самое интересное, что воспользоваться правами суперпользователя могут все, кто входит в группу **sudo**.

- Приведем пример использования прав суперпользователя - представим, что существует файл ***secret.txt***, который следует удалить, но на на который у нас нет прав. Однако мы принадлежим к группе **sudo**, поэтому можем воспользоваться правами суперпользователя. Для этого перед выполнением любых команд нужно ставить команду **sudo** (будет запрошен пароль - вводим пароль нашего пользователя - после подтверждения команда начнет свое выполнение):

```
sudo rm secret.txt
```

Деятельность в качестве суперпользователя

- После того, как мы использовали команду **sudo**, нам 15 минут не будет требоваться пароль, чтобы использовать эту команду вновь. Однако если мы хотим зайти в профиль суперпользователя и действовать от его имени постоянно, нам доступны ровно два варианта:
    - Если мы не принадлежим к группе **sudo**, но знаем пароль суперпользователя, мы можем выполнить команду **su -**. Затем следует ввести упомянутый выше пароль - в результате мы окажемся в командной строке в качестве **root** пользователя, который может все:

```
su -
```

    - Если же мы принадлежим к группе **sudo**, то мы можем выполнить команду **sudo -i** - после этого вводим наш пароль и опять автоматически переходим в командную строку как **root**:

```
sudo -i
```

    - Для выхода из профиля **root** надо ввести в консоли команду **exit**.

Команда **adduser** - создание пользователя в интерактивном режиме

- Если нам надо быстро создать пользователя, и мы хотим удобств - чтобы в процессе создания нам было предложено присвоить пароль и автоматически создалась домашняя директория, то следует использовать команду **adduser**, после которой нам необходимо указать имя нового пользователя. Затем мы пройдем через несколько ответов на различные вопросы, что практически исключит возможность ошибки. Также важно знать, что создавать пользователей может только суперпользователь. Теперь перейдем к практике и попытаемся создать пользователя **visitor**:

```
sudo adduser visitor
```

  В результате будет создан пользователь **visitor**, который сможет заходить в систему под тем паролем, который был указан в процессе создания. Также для пользователя будет создана группа **visitor** и домашняя директория **/home/visitor**.

- При создании типом консоли будет браться тип по умолчанию. Если мы хотим указать тип явно, следует применить параметр **--shell**:

```
sudo adduser --shell=/bin/bash visitor
```

Команда **useradd** - создание пользователя в ручном режиме

- Иногда нам не будет доступен интерактивный режим и мы не сможем отвечать на вопросы команды **adduser**. В данном случае нам спешит на помощь команда **useradd**, которая позволяет указать параметры пользователя явно, без вопросов. Она более сложная, т.к. она не создает пароль и по умолчанию не создает директорию пользователя по умолчанию (автоматически создастся только группа с именем пользователя) - все это мы должны сделать сами. Базовый функционал создание пользователя (допустим с именем **sasha**) следующий:

```
sudo useradd sasha
```

- Для создания директории по умолчанию следовало использовать параметр **-m**:

```
sudo useradd -m sasha
```

- Для указания типа консоли применяется параметр **-s**:

```
sudo useradd -s /bin/bash sasha
```

- Для добавления (или изменения) пароля используется команда **passwd**):

```
sudo passwd sasha
```

Создание, добавление и удаление групп

- Для создания дополнительной группы следует использовать команду **groupadd**:

```
sudo groupadd additional
```

- Для добавления новой группы пользователю используется команда **usermod** и два параметра - во-первых, **-a** (добавление новой группы) и, во-вторых, **-G** (сохранить уже существующие группы). Затем идет имя новой группы, а в самом конце - имя пользователя. Добавим группу **additional** пользователю **sasha**:

```
sudo usermod -aG additional sasha
```

- Для удаления группы у конкретного пользователя используется команда **deluser**, после которой следует имя пользователя, а в конце имя удаляемой группы. Удалим группу **additional** для пользователя **sasha**:

```
sudo deluser sasha additional
```

- Для полного удаления группы используется команда **groupdel**:

```
sudo groupdel additional
```

Удаление пользователей (команды **userdel** и **deluser**)

- Для удаления пользователей следует применять команду **userdel** или команду **deluser**. Далее пример удаления пользователя с помощью команды **userdel**:

```
sudo userdel sasha
```

Чтобы вместе с пользователем удалять его домашний каталог, следует использовать параметр **-r**:

```
sudo userdel -r sasha
```

Чтобы удалить пользователя, который в данный момент залогинен, применяется параметр **-f**:

```
sudo userdel -f sasha
```

- Далее пример удаления пользователя с помощью команды **deluser**:

```
sudo deluser sasha
```

Теперь попросим систему удалить каталог пользователя при его удалении:

```
sudo deluser --remove-home sasha
```

Изменение прав для файлов и каталогов (буквенный вариант)

- Для изменения прав файлов и каталогов используется команда **chmod**, после которой идет назначение прав, а в самом конце те файлы или каталоги, которым права назначаются. Для рекурсивного изменения используется параметр **-R**.

- Если мы меняем права с помощью буквенного варианта, то нам следует запомнить три простых правила. Во-первых, право за чтение обозначается буквой **r**, на запись - **w**, а на исполнение - **x**. Во-вторых, владелец обозначается буквой **u**, группа - **g**, остальные - **o**, все сразу - **a**. В третьих, отсутствие прав обозначается знаком минус (**-**), присутствие - знаком плюс (**+**).

- Пример предоставления права на запись (**w**) для группы (**g**) на файл **example.txt**:

```
chmod g+w example.txt
```

Отъем права на чтения (**r**) и исполнения (**x**) для владельца (**u**) каталога **dir**:

```
chmod u-rx dir
```

Одновременная установка прав для остальных (**o**) в контексте файла **app.py**:

```
chmod o=r-x app.py
```

Изменение прав для файлов и каталогов (числовой вариант)

- Существует еще один способ назначения прав, но уже не через буквы, а числа. Он не столь удобен, но по-прежнему широко применяется. Для рекурсивного изменения также используется параметр **-R**.

- Надо запомнить, что в этом контексте право на чтение равняется числу **4**, на запись - **2**, а на исполнение - **1**. Для каждого вида пользователей права определяются комбинацией сумм этих трех чисел. Например, если назначено число 6, то это означает можно читать (**4**) и писать (**2**), если же назначено число **7**, то это означает возможность читать (**4**), писать(**2**) и исполнять (**1**). Сначала идет ноль (это не всегда так, но пока ограничимся этим правило), затем назначается число для владельца, потом для группы, потом для всех остальных.

- Приведем пример, когда владелец может все, группа читать и писать, а остальные - только читать каталог **numeric**:

```
chmod 0764 numeric
```

Установка права только исполнять файл **index.php** для всех пользователей:

```
chmod 0111 index.php
```

Смена группы (**chgrp**) и владельца (**chown**) файлов и каталогов

- Для смены группы файла или каталога применяется команда **chgrp**, после которой следует написать новую группу, а в самом конце путь к файлам или каталогам. Для примера установим для файла ***Main.java*** группу **developers**:

```
sudo chgrp developers Main.java
```

- Смена владельца файла или каталога происходит схожим образом, только вместо команды **chgrp** используется команда **chown**, после которой следует указать имя владельца, а затем пути к файлам/каталогам. Изменим владельца файла **script.js** на **visitor**:

```
sudo chown visitor script.js
```

- Как и в случае со сменой прав, для рекурсивного изменения группы и владельца используется параметр **-R**:

```
sudo chgrp -R developers some-dir
```

```
sudo chown -R visitor some-other-dir
```

Установка внешних пакетов

Команда **dpkg** - базовый установщик пакетов

- Самым простым установщиком пакетов является утилита **dpkg**. Для установки пакета надо указать параметр **-i**, а в конце путь к установщику. Для примера воспользуемся пакетом **calcurse**, файл которого можно загрузить ссылке ниже: **http://ftp.de.debian.org/debian/pool/main/c/calcurse/calcurse_4.6.0-2_amd64.deb**

```
sudo dpkg -i calcurse_4.6.0-2_amd64.deb
```

- Для просмотра всех файлов установленного пакета применяется параметр **-L**:

```
sudo dpkg -L calcurse
```

- Подробное описание пакета можно увидеть применив параметр **-s**:

```
sudo dpkg -s calcurse
```

- Чтобы удалить пакет (но оставить его конфигурации), применяется параметр **-r**:

```
sudo dpkg -r calcurse
```

- Для полного удаления пакета и всех его файлов нам понадобится параметр **-P**:

```
sudo dpkg -P calcurse
```

Команда **apt** - усовершенствованный установщик пакетов (основы)

- **dpkg** обладает одним недостатком - она не умеет подгружать зависимости для устанавливаемых приложений. Однако этого недостатка лишена команда **apt**, которая сама решает проблемы с зависимостями, зависимостями зависимостей и т.д. Более того, для **apt** даже не надо загружать установочный файл - она имеет свой список репозиториев **/etc/apt/sources.list**, где в режиме онлайн доступны практически все пакеты и их версии. Поэтому при инсталляции нам нужно будет указать только название пакета и больше ничего.

- У **apt** есть более старая версия - **apt-get**. Для пользователя системы среднего уровня особой разницы между ними нет, просто запомним, что **apt** показывает более информативные данные при установке и удалении, а также строже относится к безопасности пакетов.

- Команда **apt** умеет обновлять информацию о репозиториях в системе (узнать последние версии пакетов и т.д.) - для это надо применить параметр **update**:

```
sudo apt update
```

Команда **apt** - установка (**install**) и обновление (**upgrade**) пакетов

- Для установки нам следует использовать параметр **install**, после которого надо указать один или несколько названий пакетов для установки. Для примера установим пакет **curl**, который позволяет делать сетевые запросы:

```
sudo apt install curl
```

- В процессе установки система может предложить подтвердить установку некоторых пакетов - чтобы заранее дать согласие и не отвлекаться во время процесса, подставим параметр **-y**. Приведем пример с установкой пакетов **nodejs** и **npm**:

```
sudo apt install -y nodejs npm
```

- Если мы хотим установить последние исправления и обновления для всех наших пакетов, то нам следует применить параметр **upgrade** (также можно дополнить параметром **-y** для заблаговременного подтверждения всех изменений):

```
sudo apt upgrade -y
```

Команда **apt** - удаление (**remove, purge, autoremove**) пакетов

- Для удаления пакетов нам следует применить параметр **remove**, после которого написать название пакетов. Для примера удалим пакет **npm**:

```
sudo apt remove npm
```

- После удаления пакета с помощью предыдущего параметра у нас могут остаться конфигурационные файлы пакетов и данные, появившиеся во время работы этих пакетов. Для тотального удаления пакета и его данных применяется атрибут **purge**. Применим его, чтобы зачистить систему от всех данных пакета **nodejs**:

```
sudo apt purge nodejs
```

- Команды remove и purge поддерживают неточное указание названия удаляемых пакетов - например, если нам надо удалить пакеты, название которых начинаются со слова **mysql**, то после этого слова можно указать звездочку:

```
sudo apt purge mysql*
```

- После удаления пакетов у нас в системе могут остаться старые зависимости, которые нигде не используются. Для зачистки этих зависимостей следует использовать параметр **autoremove**:

```
sudo apt autoremove
```

Команда **add-apt-repository** - добавление новых репозиториев

- Иногда при при попытке установить очередной пакет (например, язык PHP версии **php8.2-fpm**) система не сможет его найти. Чаще всего это связано с тем, что репозиторий этого пакета не перечислен в файле **/etc/apt/sources.list**, и поэтому нам надо добавить этот репозиторий самим - с помощью параметра **add-apt-repository**. После поиска в Интернете мы выясним, что за PHP отвечает репозиторий ***ppa:ondrej/php*** - именного его мы и добавим:

```
sudo add-apt-repository ppa:ondrej/php
```

После этого у нас появиться возможность установить PHP, что мы и сделаем:

```
sudo apt install -y php8.2-fpm
```

Работа с процессами

Введение

- При запуске любой команды или программы операционная система создает т.н. **процесс**, который содержит в себе код программы и ее зависимостей, данные, необходимые для работы, а также состояние процесса в каждый конкретный момент времени. Все это хранится в памяти, которая принадлежит процессу - другие процессы не имеют прямого доступа к этой памяти.

- Все процессы порождаются другими процессами, кроме первого - процесса инициализации системы, который известен под названием **systemd** - он запускается ядром Linux. Основными идентифицирующими параметрами процесса являются уникальный идентификатор (**PID**), идентификатор родительского процесса (**PPID**) и уже упомянутое состояние (о разных типах состояния мы поговорим немного позднее).

- Всего можно выделить три типа процессов - во-первых, **процессы ядра** (запускаются и контролируются ядром), **пользовательские процессы** (запускаются после команд пользователя) и **процессы-демоны** (работают в фоновом режиме).

Процессы-демоны и службы

- Процессы-демоны часто обслуживают т.н. службы (**services**), которые должны запускаться при старте операционной системы (за эти следит **systemd**) и работать в фоновом режиме. Одним из представителей таких служб является веб-сервер **nginx**, который можно установить следующей командой:

```
sudo apt install -y nginx
```

- Управлять службами можно с помощью утилит **systemctl** и **service**. Более современной является **systemctl**, но делают они примерно то же самое - просят **systemd** повлиять на службу. Ниже представлен их основной функционал:

```
# systemctl
# запуск службы
sudo systemctl start nginx

# остановка службы
sudo systemctl stop nginx

# перезапуск службы
sudo systemctl restart nginx

# получение состояния службы
sudo systemctl status nginx

# перезагрузка конфигурации
sudo systemctl reload nginx
```

```
# service
# запуск службы
sudo service nginx start

# остановка службы
sudo service nginx stop

# перезапуск службы
sudo service nginx restart

# получение состояния службы
sudo service nginx status

# перезагрузка конфигурации
sudo service nginx reload
```

Команда **ps** - базовый мониторинг процессов

- Самый простой способ для отслеживания процессов - использование команды **ps**. Если мы вызовем ее без всяких параметров, то увидим только те процессы, которые связаны с командной оболочкой - **bash** - и процесс, связанный с **ps**. Поэтому почти всегда используют специальные параметры, такие как **a** (все параметры), **u** (представление в понятном человеку формате) и **x** (все процессы).

```
ps aux
```

  В результате вызова команды **ps** мы получим таблицу со столбцами: ***USER*** (пользователь, запустивший процесс), ***PID*** (идентификатор процесса), ***%CPU*** (используемые ресурсы процессора), ***%MEM*** (процент занимаемой RAM), ***VSZ*** (используемая виртуальная память в KB), ***RSZ*** (реальная память в KB), ***TTY*** (терминал, с которым ассоциирован процесс), **STAT** (состояние процесса), ***START*** (время начала процесса), ***COMMAND*** (команда, запустившая процесс).

- Основные состояния процесса - **R** (выполняется процессором), **S** (ожидание, внешние сигналы обрабатываются), **D** (ожидание, внешние сигналы не обрабатываются), **I** (ожидание какой-либо задачи на фоне), **T** (остановлен), **Z** (зомби - завершил выполнение, но его **PID** еще существует).

Команда **top** - мониторинг процессов в реальном времени

- Если **ps** предлагает нам слепок состояния процессов на какой-то определенный момент, то команда **top** показывает работу процессов в реальном времени - каждые несколько секунд обновляя внешний вид командной строки. Чтобы прекратить обновление, нам следует нажать кнопку **Q** на клавиатуре:

```
top
```

- У команды **top** не так много параметров - из них самым полезным является **-u**, после которого следует указать имя какого-нибудь пользователя системы. В результате будут показаны процессы, запущенные только этим пользователем:

```
top -u user
```

- Столбцы таблицы **top** схожи с теми, которые были в **ps**, но с некоторыми добавлениями: ***PID*** - идентификатор процесса, **USER** - пользовтель, запустивший процесс, **PR** - приоритет процесса, **NI** - относительный приоритет, **VIRT** - используемая виртуальная память, **RES** - используемая физическая RAM память, **SHR** - память, используемая совместно с другими процессами, **S** - состояние, ***%CPU*** (используемые ресурсы процессора), ***%MEM*** (процент занимаемой RAM), **TIME+** (время работы процесса) и **COMMAND** (команда, запустившая процесс).

Команда **kill** - остановка процессов

- Иногда процессы могут использовать слишком много ресурсов, родительские процессы по ошибке могут забыть зачистить процесс зомби, кроме того - некоторые процессы могут нести в себе угрозу для системы. Для ручного прекращения выполнения таких процессов существует команда **kill**, после которой следует подставить идентификатор (**PID**) приговоренного процесса:

```
sudo kill 9999
```

- Команда **kill**, вызванная без дополнительных параметров, посылает процессу т.н. сигнал **SIGTERM**, который позволяет процессу самому завершиться, сначала освободив все ресурсы. Полный список всех возможных сигналов можно получить при помощи параметра **-l**:

```
kill -l
```

Из полученной таблицы мы узнаем, что **SIGTERM** имеет номер 15. Если же мы хотим уничтожить процесс немедленно, нам нужен сигнал **SIGKILL** под номером 9. Для его отправки используется следующий синтаксис:

```
sudo kill -9 9999
```

Создание и распаковка архивов

Введение

- Современный мир компьютерных технологий немыслим без архивирования. С одной стороны это позволяет уменьшить размер пересылаемых файлов, а с другой стороны - гораздо удобнее пересылать и обрабатывать один архив, чем взаимодействовать с целой россыпью файлов.

- Классическим встроенным способом взаимодействия с архивами в системах на ядре Linux являются команды **tar** (соединяет файлы в один архив) и **gzip** (используется для сжатия). Они создают и распаковывают архивы с расширением **tar** и **gz** соответственно (хотя расширение и не является обязательным условием).

- Для работы с архивами в формате zip в Ubuntu есть две встроенные команды - **zip** (создает архив) и **unzip** (распаковывает архив).

- Также довольно популярными являются архивы в формате rar. К сожалению в Ubuntu нет встроенных средств для работы с ними, однако можно установить пакет **rar**, который будет полезен для генерации и вскрытия подобных архивов.

Команды **tar** и **gzip**

- Для базового создания архива tar мы должны применить команду **tar** с параметрами **-c** (создать) и **-f** (файл), потом мы должны указать имя архива, а затем имена тех директорий и файлов, которые мы хотим включить в архив:

```
tar -cf archive.tar some-file some-dir some-other-file
```

Теперь нам нужно сжать наш архив при помощи команды **gzip** - в результате получится файл ***archive.tar.gz***. По умолчанию промежуточный файл ***archive.tar*** будет удален, если же мы хотим оставить его, то следует применить параметр **-k**:

```
gzip -k archive.tar
```

- Чтобы распаковать архив сначала следует убрать сжатие и получить файл ***archive.tar***. Для этого используется команда **gzip** с ключем **-d** (можно добавить **-k**, чтобы сохранить раскрываемый архив):

```
gzip -kd archive.tar.gz
```

В завершение нам надо распаковать оригинальные файлы - для этого воспользуемся командой **tar** с ключами **-x** (распаковать) и **-f** (файлы):

```
tar -xf archive.tar
```

Команды **tar** - создание и распаковка сжатых архивов без **gzip**

- Важно знать, что команда **tar** умеет создавать и распаковывать уже сжатые архивы без привлечения команды **gzip** - для этого надо добавить при создании ключ **-z**:

```
tar -czf archive.tar.gz some-file some-dir some-other-file
```

- Для быстрой распаковки оригинальных файлов без участия **gzip** мы опять должны применить ключ **-z**, но уже вместе с ключем **-x**:

```
tar -xzf archive.tar.gz
```

- Иногда нам надо распаковать оригинальные файлы не в ту директорию, а некую другую. Для этого мы должны дополнить команду распаковки ключем **-C** после которой нужно указать целевую директорию (она должна реально существовать, иначе будет ошибка):

```
tar -xzf archive.tar.gz -C some-target-dir
```

Команды **zip** и **unzip**

- Для создания архива в формате zip мы должны использовать команду **zip**, после которой следует указать имя архива, а затем названия тех файлов и директорий, которые мы хотим упаковать. Если мы хотим, чтоб были упакованы не только директории, но и их содержимое, следует применить ключ **-r**:

```
zip -r archive.zip some-file some-dir some-other-file
```

- Для распаковки файлов применяется команда **unzip**, после которой следует написать путь к архиву:

```
unzip archive.zip
```

- Для распаковки файлов не в ту директорию, в которой находится архив, а в какую-либо другую, применяется параметр **-d**, после которого следует указать путь к нужной директории:

```
unzip -d some-target-dir archive.zip
```

Команда **rar**

- По умолчанию нам недоступна работа с rar архивами, поэтому сначала мы должны установить соответствующий пакет:

```
sudo apt install -y rar
```

- Чтобы создать rar архив, надо применить команду **rar**, после которой следует подставить параметр **a**, затем указать название архива (пусть будет **archive.rar**), а в конце перечислить все необходимые нам файлы и директории:

```
rar a archive.rar some-file some-dir some-other-file
```

- Для распаковки используется та же самая команда **rar**, но уже с ключем **x**, после которого надо указать путь к архиву:

```
rar x archive.rar
```

- Если необходимо распаковать файлы в конкретную директорию, то следует повторить предыдущую команду, но в конце добавить путь к нужной директории:

```
rar x archive.rar some-target-dir
```

Командная оболочка и переменные окружения

Введение в командные оболочки

- Запуская консоль, мы тем самым начинаем выполнять программу - командную оболочку, которая находится в каталоге **/bin**. Если мы ничего не меняли после установки системы, то этой программой будет **bash**, однако надо помнить, что могут применяться иные утилиты, например **sh**. Для изменения типа командной оболочки используется команда **chsh** (команда приведена просто для примера):

```
chsh -s /bin/sh
```

- Кроме запуска непосредственно текстовой командной строки, **bash** в самом начале запускает определенные файлы в пользовательской директории (всегда - ***~/.bashrc***, при интерактивной аутентификации - например, при входе через ***ssh*** - вначале грузится ***.profile***). Файлы настраивают внешний вид командной строки, подгружают автодополнение и настраивают возможные псевдонимы команд.

- При старте оболочки также подгружаются т.н. переменные окружения. Они уведомляют команды о системных настройках пользователя, установленном языке, могут инициализировать определенный режим для некоторый программ, кроме того, эти переменные в принципе позволяют запускаться командам. Просмотреть переменные можно при помощи команды **printenv**:

```
printenv
```

Создание новых переменных окружения

- Часто нам может понадобиться добавить переменную, которая не генерируется при запуске оболочки. Например, когда мы инсталлируем Java, нам важно установить переменную ***JAVA_HOME***, которую могут использовать приложения, которые от Java зависят. Теоретически мы можем сделать так:

```
export JAVA_HOME=/path/to/java/bin/folder
```

- Переменная установится правильно, и к ней можно получить доступ, однако при следующей загрузке ***JAVA_HOME*** пропадет. Нам надо добиться того, чтобы переменная устанавливалась сама при открытии командной строки. Как мы помним, при загрузке у нас запускаются определенные файлы, которые могут помочь осуществить начальную настройку. Из всех этих файлов всегда запускается ***~/.bashrc*** - поэтому разумнее всего записать приведенный выше пример именно в этот файл.

- Когда мы зайдем в командную строку в следующий раз, переменная ***JAVA_HOME*** будет доступна сразу же - мы можем проверить это при помощи команды **echo**:

```
echo $JAVA_HOME
```

Переопределение системных переменных

- Самой важной системной переменной вне всяких сомнений является **PATH** - по сути это набор путей (разделенных знаком двоеточия **:**) к тем директориям, где оболочка ищет файлы, которые связаны с запускаемыми командами (**cp**, **mv**, **rm**, **ls** и т.д.). Если надо увидеть содержимое этой переменной, то это делается так:

```
echo $PATH
```

- Иногда нам может понадобится добавить наши собственные директории, где могут быть размещены дополнительные исполняемые файлы для команд. Самый правильный способ - переопределить переменную **PATH**, чтобы **bash** загружала эти файлы автоматически. Опять же, самым лучшим местом для переопределения является файл ***~/.bashrc***, где можно через двоеточие (**:**) добавить к оригинальному пути наши дополнительные пути:

```
export PATH=/path/to/new/folder:/one/more/folder:$PATH
```

Переменные окружения и локальные переменные

- В примерах при объявлении переменной часто можно увидеть следующую конструкцию (название переменной и значение отображены схематически):

```
VARIABLE_NAME=VARIABLE_VALUE
```

Мы должны запомнить раз и навсегда - это не переменная окружения, это локальная переменная, которая нужна только для какой-то вспомогательной задачи именно в этом конкретном месте (например, сохранить промежуточный результат неких вычислений). Куда бы мы ни поместили такое объявление переменной (хотя бы даже в ***~/.bashrc***), оно не будет работать глобально.

Переменная не отобразится в списке, который возвращает команда **printenv**, кроме того - процессы, запускаемые командной оболочкой, не будут иметь доступа к значению этой переменной. Поэтому если нам нужна действительно переменная окружения, не забываем применять команду **export**.

Жесткие и символические ссылки

Введение

- Пользователи Windows уже знакомы с концепцией ярлыков - файлов, которые сами по себе ничего не делают, но, например, позволяют запускать другие исполняемые файлы, находящиеся в глубине файловой структуры. Это сделано главным образом для удобства, ведь главный исполняемый файл у каждой программы один, а ярлыков у него может быть много. Схожие механизмы присутствуют в системах на ядре Linux - они называются ссылками.

- Один из самых ярких примеров использования ссылок в Ubuntu - работа с web-сервером **nginx**. Настройки всех сайтов хранятся в ***/etc/nginx/sites-available***, а настройки тех сайтов, которые запущены, находятся в ***/etc/nginx/sites-available*** и являются ссылками на файлы в ***/etc/nginx/sites-enabled***. Это дает нам два преимущества - во-первых, если нам надо отключить сайт, то мы просто удалим его ссылку из ***sites-enabled***, а изначальные настройки останутся нетронутыми. Во-вторых, если мы изменим изначальные настройки, то они станут доступны уже работающему сайту.

- В отличие от Windows, в операционных системах на ядре Linux существуют ссылки двух типов - жесткие (**hard**) и символические (**symbolic**). Далее мы подробно рассмотрим каждый из этих типов.

Команда **ln** - работа с жесткими ссылками

- Жесткая ссылка - несколько более сложный концепт, чем просто ярлык. Такие ссылки могут ссылаться только на файлы (на директории - нельзя) и только в контексте одной файловой системы (**ext4**, **ntfs** и т.д.). Но самое главное - ссылка имеет такой же файловый дескриптор (**inode**), как и оригинальный файл. Таким образом, при удалении оригинального файла его данные не уничтожаются - они будут доступны пока не ликвидируют последнюю связанную с этими данными жесткую ссылку.

- Создание жестких ссылок производится при помощи команды **ln**, после которой сначала необходимо указать путь к целевому файлу, а потом путь к новой жесткой ссылке. Представим, что у нас есть файл ***/home/user/secret.conf***, а нам нужно создать жесткую ссылку на него в каталоге ***/etc*** под названием system.d. Для осуществления этой задачи следует выполнить следующий код:

```
sudo ln /home/user/secret.conf /etc/system.d
```

  В результате все данные из ***secret.conf*** будут доступны при чтении ***system.d***, более того - эти данные будут доступны даже после удаления оригинального **secret.conf**.

Команда **ln** - работа с символическими ссылками

- Символические ссылки по своему смыслу уже гораздо ближе к ярлыкам, чем жесткие ссылки. Это файлы со своим уникальным дескриптором (**inode**), они способны ссылаться как на файлы, так и на директории, кроме того - они способны ссылаться на другие файловые системы. Можно понять, что такой тип ссылок имеет много преимуществ перед своим “жестким” подвидом, но есть и один огромный минус. Если удаляется оригинальный файл или директория, то все данные будут потеряны - даже несмотря на наличие символических ссылок.

- Для создания символических ссылок используется уже знакомая нам команда **ln**, но дополненная параметром **-s**, после которого следует путь к оригинальному файлу или директории, а замет путь к самой новой ссылке. Предположим, что в директории ***/home/user*** создана директория ***executables***. Мы хотим, что в директории ***/var*** была бы создана ссылка ***user-executables***, которая должна ссылаться на первую директорию. Далее представлена команда для этого:

```
sudo ln -s /home/user/executables /var/user-executables
```

Получение всех ссылок для изначального файла/каталога

- Иногда нам может потребоваться получить все ссылки для какого-то файла, чтобы узнать, кто же на нас ссылается. Для получения данных о жестких ссылках мы можем использовать команду find с параметром **-samefile**, который следует дополнить путем к оригинальному файлу. Найдем все жесткие ссылки для файла ***/home/user/secret.conf*** :

```
find / -samefile /home/user/secret.conf
```

- Поиск символических ссылок также производится с помощью команды **find**, но дополнить ее следует параметром **-lname**, после которого нужно поставить путь к оригинальному файлу/каталогу. Найдем все символические ссылки для каталога ***/home/user/executables***:

```
find / -lname /home/user/executables
```

Отображение системных параметров

Команды **df** и **du** - отображение свободного и занятого места на диске

- Если нам надо узнать заполняемость диска, то следует воспользоваться командой **df**, которая показывает свободное и занятое место для всех дисков (в килобайтах), подключенных к системе. Если в качестве аргумента подать название диска, информация будет показана только для него. Также интересен параметр **-h**, отображающий данные в удобном для человека формате:

```
df -h
```

- В том случае если необходимо получить размер только одной конкретной директории, мы можем воспользоваться командой **du**, после которой следует указать путь к директории. По умолчанию будут также показаны размеры всех вложенных директорий - если мы хотим получить только один финальный результат, то надо применить параметр **-s**. Кроме того, размер по умолчанию показан в килобайтах - если хотим получить более приятный формат, следует применить параметр **-h**:

```
du -sh /path/to/directory
```

Отображение информации о процессоре и его загрузке

- Сначала упомянем команду, которая позволяет узнать всю информацию о процессоре - это **lscpu**:

```
lscpu
```

- Для отображения информации о нагрузке процессора у нас есть целых два варианта. Во-первых, мы можем прочитать файл **/proc/loadavg** - в результате нам вернутся целых пять параметров - средняя нагрузка всех ядер за последнюю минуту, за последние пять минуть и за последние пятнадцать минут, отношение количества активных процессов к общему количеству задач в системе, а также идентификатор последнего запущенного процесса:

```
cat /proc/loadavg
```

- Во-вторых, нам доступна команда **uptime** - она показывает общее время работы системы, работу системы с момента последнего включение, количество активных в данный момент пользователей, а затем среднее значения для загрузки всех ядер процессора за одну, пять и пятнадцать минут:

```
uptime
```

Отображение информации о свободной и занятой оперативной памяти

- В некоторых случаях нам может потребоваться информация об оперативной памяти. С этой задачей прекрасно справляется команда **free**, которая может быть дополнена параметром **-h** для отображения результата в удобном для чтения формате:

```
free -h
```

После выполнения данной команды нам вернется:

1. общий объем оперативной памяти (**total**)
2. память, используемая в данный момент (**used**)
3. количество свободной памяти (**free**)
4. память, разделяемая между процессами (**shared**)
5. память для буферизации и кэширования (**buff/cache**)
6. оценка памяти, учитывая буферы и кэши (**available**)

Кроме того, мы увидим информацию о файле подкачки (**Swap**), который будет использоваться, если оперативной памяти будет не хватать.

Автоматизация при помощи **bash** скриптов

Введение

- Мы уже знаем огромное количество концепций, умеем перенаправлять потоки,и даже используем каналы для создание командных конвейеров. Но есть еще один очень способ управления операционной системой. Это создание т.н. bash скриптов, - файлов, которые содержат целые сценарии сложной логики.

- Как правило, bash скрипты имеют расширение .sh, но это не принципиально. Для примера создадим файл ***hello.sh***:

```
touch hello.sh
```

- Скрипты должны быть исполняемыми (т.е. у нас должна быть возможность их запускать как программу) - сделаем это при помощи команды **chmod**:

```
chmod a+x hello.sh
```

Запуск простейшего скрипта

- Чтобы скрипт ***hello.sh*** запустился, уже внутри файла (в самом его начале - на первой строке) должен располагаться шебанг - указатель на то, какой интерпретатор используется для запуска скрипта. Шебанг начинается с решетки (**#**), потом идет восклицательный знак (**!**), а затем - путь к интерпретатору. Мы будем использовать **/bin/bash**:

```
#!/bin/bash
```

- После указания интерпретатора (через шебанг) мы уже можем писать команды скрипта (нам доступны любые команды из тех, которые использовались до этого - **rm**, **cd** и т.д.). Для начала просто выведем приветствие “Hello, World!”:

```
#!/bin/bash

echo "Hello, World!"
```

- Для запуска нашего скрипта нам следует просто указать путь к файлу - либо полный, либо через точку. После этого мы увидим наше приветствие:

```
./hello.sh
```

Объявление переменных и обращение к ним

- Часто для вычислений или просто для удобства нам могут потребоваться переменные. Их объявление происходит при помощи знака **=** - запомним, что не надо ставить вокруг этого знака пробелы. Кроме того, если у нас несколько слов в значении переменной, то значение следует обернуть в кавычки:

```
name=Alex
age=42
greeting='Hello, World'
```

- Перед обращением к переменным нам обязательно надо указать перед их именем знак доллара (**$**) - если мы, например, хотим отобразить значение переменной ***name***, то это делается следующим образом:

```
echo $name
```

Объявление переменных и обращение к ним (продолжение)

- Может случиться, что текстовая переменная при своем объявлении потребует значения других переменных. В этом случае мы можем обратиться к этим переменным, но надо помнить, что следует обернуть все значение в двойные кавычки (в одинарных кавычках подстановка переменных не происходит - данные отображаются точь в точь как они прописаны):

```
firstname=John
lastname=Smith
fullname="$firstname $lastname"
```

- У нас есть возможность объявить переменную не через явное присвоение, а с помощью данных, предоставленных пользователем скрипта. Это значит, что после запуска скрипта мы можем потребовать у клиента ввести что-нибудь с клавиатуры и только потом создать переменную с этим значением. Это осуществляется при помощи команды **read** (можно дополнить ее параметром **-p** и пояснением после него, чтобы дать понять клиенту, что мы от него хотим):

```
read -p "Please, insert your name: " name
echo $name
```

Встроенные переменные

- Существует определенный набор переменных, которые нам не надо создавать - они доступны изначально. Во-первых, это название файла, который запущен - **$0**:

```
echo $0
```

Во-вторых, это количество аргументов, переданных скрипту при его запуске - **$#**:

```
echo $#
```

В-третьих, это сами значения тех аргументов, которые были переданы скрипту. Значение первого аргумента будет записано в **$1**, второго - в **$2**, третьего - в **$3** и т.д. Допустим у нас есть файл ***args.sh*** и он запущен следующим образом:

```
./args.sh My name is Vasya
```

Получается, что в **$1** будет записано слово *My*, в **$2** - *name*, в **$3** - *is*, а в **$4** - *Vasya*.

Запись в переменную результата операций

- Иногда нам необходимо записать в переменную результат исполнения некой команды, чтобы использовать этот результат в дальнейших вычислениях. Это можно сделать двумя способами. Во-первых, обернуть команду в обратные кавычки ``, во-вторых, в качестве обертки воспользоваться конструкцией **$()**. Приведем пример с командой **pwd**, которая показывает путь к рабочему каталогу на данный момент (этот путь мы и запишем в переменную):

```
workdir_example=`pwd`
another_workdir_example=$(pwd)
```

- Также дополнительной обработки требует получение результата математических вычислений. Во-первых, можно обернуть вычисления в конструкцию **$((...))**. Во-вторых, можно воспользоваться директивой **let**. Перед примерами запомним одну особенность **bash** - здесь не поддерживаются операции с дробными числами, для этого следует использовать сторонние решения, например, команду **awk**. Теперь приведем пример с вычислениями:

```
calc_res=$((2 + 2 * 2))
let another_calc_res="2 + 2 * 2"
```

Пример полезной программы с использованием bash скрипта

- Бывает ситуация, когда требуется записать что-то в файл, название которого для нас не принципиально, но важно то, что это название должно быть уникальным. Это особенно полезно для логов и временных файлов для каких-либо хитрых задач. Напишем скрипт, который принимает в качестве аргумента любой текст и записывает этот текст в файл, название которого состоит из числа и времени на данный момент (для получение этих параметров воспользуемся командой **date**). В самом конце будем отображать имя нового файла:

```
#!/bin/bash

# генерация имени файла
filename=$(date +"%Y_%m_%d_%H_%M_%S.txt")

# запись в файл значения первого аргумента
echo $1 > $filename

# отображение имени файла
echo $filename
```

Условные конструкции (введение)

- Во время выполнение перед нами может возникнуть альтернатива исполнения того или иного блока кода - что делать если, например, запрошенный файл отсутствует или в системе не хватает места для совершения того или иного действия. Чтобы разрешить эту проблему используется механизм ветвления, который также называется условными конструкциями (**if … elif … else … fi**).

- Предположим, что у нас есть простейший скрипт, который должен принимать два аргумента (записываются в **$1** и в **$2**) - целых числа - и поделить первое на второе. Мы знаем, что делить на нуль нельзя, поэтому проверим второе число и, если оно равно нулю - сообщим об ошибке:

```
#!/bin/bash

if [[ "$2" == 0 ]]; then
    echo "Division by zero error!"
else
    # отображение результата деления при помощи команды awk
    awk -v a="$1" -v b="$2" 'BEGIN{print a / b}'
fi
```

Списки операторов сравнения для условных конструкций

Сравнение чисел и строк:

<var> -eq (==) <var>	равно
<var> -ne (!=) <var>	не равно
<var> -lt (<) <var>	меньше
<var> -le <var>	меньше или равно
<var> -gt (>) <var>	больше
<var> -ge <var>	больше или равно

Условия сравнения для файлов:

-e <path>	путь существует
-f <path>	файл существует
-d <path>	каталог существует
-s <path>	файл не пустой
-x <path>	файл исполняемый

Логические условия сравнения:

! <expr>	отрицание выражения
<expr> && <expr>	логическое <<И>>
<expr> || <expr>	логическое <<ИЛИ>>

Условия сравнения для строк:

-z <string>	строка пуста
-n <string>	строка не пуста

Условные конструкции (множественное ветвление)

- Бывают случаи, когда у нас есть не одна, а несколько альтернатив - например, нам мало проверки на ноль - кроме этого, мы должны проверить, что количество переданных аргументов равно двум, что первый аргумент - число, и что второй аргумент - тоже число. На любую из ошибок мы должны выдавать правильное сообщение. Для этого всего используется множественное ветвление при помощи оператора **elif**:

```
#!/bin/bash

if [[ "$#" != 2 ]]; then
    echo "Wrong number of arguments! Must be two!"
elif [[ !("$1" =~ ^[0-9]+) ]]; then
    echo "Wrong first argument! Must be number!"
elif [[ !("$2" =~ ^[0-9]+) ]]; then
    echo "Wrong second argument! Must be number!"
elif [[ "$2" == 0 ]]; then
    echo "Division by zero error!"
else
    awk -v a="$1" -v b="$2" 'BEGIN{print a / b}'
fi
```

Условные конструкции (case)

- Иногда нам не нужно сложных сравнений - мы просто хотим проверить, соответствует ли какая-либо переменная некому значению - если соответствует то производим действие. В **bash** существует такая упрощенная конструкция - это **case**. Для примера возьмем случай, когда наш скрипт принимает два аргумента - первый означает название, второй - действие (создание файла или директории). Будем совершать действие в соответствии со вторым аргументом:

```
#!/bin/bash

case $2 in
"file")
    touch "$1"
    ;;
"dir")
    mkdir -p "$1"
    ;;
*)
    echo "Wrong action!"
esac
```

Циклы (подвид **while**)

- При разработке сценариев нам может понадобиться повторить одну и ту же логику много раз со слегка измененными результатами - например создать файлы, у которых в конце будут идти цифры 1, 2, 3 и т.д. Для этого следует применять т.н. циклы, которые позволяют повторять блоки кода определенное количество раз.

- Самым простым циклом является **while**, который исполняется до тех пор, пока его условие (которое мы определяем сами) истинно. Например вот так можно вывести все числа от 1 до 100:

```
#!/bin/bash

num = 1

while [[ "$num" -le 100 ]]; do
    echo "$num"
    num=$(($num + 1))
done
```

Циклы (подвид **for**) - простейшие примеры

- Более сложным, но одновременно более универсальным является цикл **for**, который предназначен для “прокручивания“ неких наборов данных - слов в строке, строк в тексте, файлов в директории, числовых последовательностей и т. д. Например, если нам надо обойти слова в строке, то код будет следующим (для разделения используется пробел):

```
#!/bin/bash

string="Счастье для всех, даром, и пусть никто не уйдет обиженным!"
for word in $string; do
    echo "$word"
done
```

- Далее представим случай, когда у нас есть некий набор предустановленных значений, и мы должны обойти каждое из этих значений:

```
#!/bin/bash

seasons=("Autumn" "Winter" "Spring" "Summer")
for season in "${seasons[@]}"; do
    echo "$season"
done
```

Циклы (подвид **for**) - обход директорий

- Если мы хотим один за другим получить пути в некой директории, то в качестве источника последовательности нам следует поставить путь к этой директории:

```
#!/bin/bash

path=/path/to/directory/*
for file in $path; do
    echo "$file"
done
```

Циклы (подвид **for**) - использование счетчика

- В числе прочего **for** поддерживает итерации со счетчиком. В этом случае контрольная структура обрамляется в двойные скобки **((…))** и делится на три части, разделенные точкой с запятой **(;)**. Первая часть отрабатывает перед началом всех итераций (там можно установить начальное значение счетчика), вторая часть проверяет, не пора ли циклу прекратиться (исполняется перед каждой итерацией), а последняя часть исполняется после каждой итерации (там можно последовательно увеличивать значение счетчика). Покажем все числа от 1 до 100 при помощи цикла **for**:

```
#!/bin/bash

for ((i=1; i<=100; i++)); do
    echo "$i"
done
```

Циклы - управляющие конструкции **break** и **continue**

- Иногда мы хотим не обходить весь предоставленный набор значений полностью, а прерваться в каком-то случае. Например, мы обходим директорию, хотим найти некий определенный файл, вывести его содержимое, а затем прерваться. Для прерывания следует применить конструкцию **break**:

```
#!/bin/bash

for file in /path/to/directory/*; do
    if [[ "$file" == "filename.txt" ]]; then
        cat "$file"
        break
    fi
done
```

- Также существует конструкция **continue**, которая немедленно прекращает данную итерацию (игнорируя весь код после) и пытается начать следующую. Далее показан пример предыдущего кода, переделанный под **continue**:

```
#!/bin/bash

for file in /path/to/directory/*; do
    if [[ "$file" != "filename.txt" ]]; then
        continue
    fi
    cat "$file"
done
```

Функции (основы)

- Повторение логических блоков может потребоваться не только в одном месте (как в циклах), а в совершенно разных местах кода. Для предотвращения повторения следует использовать так называемые функции, внутри которых могут быть любые конструкции (переменные, циклы, команды). Важный момент - если в функцию передаются некоторые аргументы, то внутри они будут доступны как **$1**, **$2** и т.д. (т.е. происходит перезапись оригинальных аргументов скрипта). Создадим функцию **sum**, которая складывает два числа и печатает результат. Затем вызовем функцию несколько раз:

```
#!/bin/bash

function sum() {
    echo $(($1 + $2))
}

# получим 4
sum 2 2

# получим 77
sum 55 22
```

Функции (возвращение значения)

- Нам не всегда будет нужно отображать результат функции визуально - часто будет необходимость записать результат в переменную, чтобы использовать в более сложных вычислениях. Решение для этого есть - это оператор **return**, который, к сожалению, может работать только с числовыми значениями, притом возвращать можно число не более 255. Для получения результата используется встроенная переменная **$?**. Напишем функцию, которая принимает два числа и возвращает их произведение с помощью **return**:

```
#!/bin/bash

function multiply() {
    result=$(($1 * $2))
    return $result
}

multiply 11 12

# будет выведено 132
echo $?
```

Функции (локальные и глобальные переменные)

- После рассказа о возвращении значения могло сложиться впечатление, что получить произвольные данные из функции невозможно. Однако такая возможность есть. Дело в том, что все переменные, которые мы объявляем внутри функции, являются глобальными. Если же мы хотим ограничить область видимости переменной, то перед ней надо ставить модификатор **local**. Покажем это на примере - создадим функцию ***concat***, которая объединяет две строки в одну. Для интереса запишем переданные аргументы в локальные переменные:

```
#!/bin/bash

function concat() {
    local first=$1
    local second=$2
    concat_result="$first $second"
}

concat "John" "Smith"

# будет выведено John Smith
echo $concat_result
```