Лабораторная работа №1. Linux: базовое взаимодействие.

Содержание

Содержание

1 Цели лабораторной работы

- Изучить базовое устройство Linux;
- Научиться взаимодействовать с ОС Linux;
- Научиться писать bash-скрипты.

2 Задачи к лабораторной работе

1. Изучить методические материалы к лабораторной работе;
2. Написать bash-скрипт для удаления файлов по заданному имени из указанной директории, а также предусмотреть флаг -p для удаления по шаблону.

3 Методические материалы

3.1 Что такое Linux?

Linux - это компьютерная операционная система, разработанная на основе модели открытого исходного кода.

Что такое ядро Linux? Ядро Linux является центральным элементом компьютерной операционной системы Linux. **Ядро** - это базовый интерфейс между аппаратными средствами компьютера и его процессами, который гарантирует наличие свободной памяти для запуска приложений Linux, оптимизирует работу процессоров и отслеживает соблюдение системных требований приложениями.

Ядро Linux обеспечивает взаимодействие между аппаратными средствами и приложениями посредством прерываний. Когда аппаратным средствам требуется связаться с приложением и системой в целом, процессор и ядро выдают прерывание, чтобы эффективно управлять ресурсами.

Ядро Linux выполняет следующие важные функции:

- **Управление памятью** - отслеживание того, каким образом и сколько памяти используется;
- **Управление процессами** - определение того, какие процессы и как могут использовать центральный процессор (ЦП);
- **Драйверы устройств** - действуют в качестве посредников между оборудованием и приложениями/процессами.
- **Системные вызовы и безопасность** - получение запросов на обслуживание от процессов приложений.

Где используется Linux? Операционная система Linux используется в самых разных системах и средах:

- **Суперкомпьютеры.** Все 500 машин из списка самых мощных суперкомпьютеров в мире работают под управлением Linux;

- **Веб-серверы.** Linux используется практически на всех серверах в Сети. Согласно опросам агентства W3Techs, 82% веб-сайтов развёрнуты на машинах с установленным Linux- и Unix-системами;
- **Смартфоны.** Операционная система Android, которая установлена на большинстве смартфонов в мире, работает на ядре Linux;
- **Роутеры.** Операционные системы, установленные в Wi-Fi-маршрутизаторах, также основаны на ядре Linux;
- **Умный дом.** Linux используют многие устройства в умных домах: холодильники, телевизоры, стиральные машины;
- **Авиация и транспорт.** Linux установлена на бортовых компьютерах автомобилей Tesla и в машинах с автопилотом от Google. Ещё Linux используется в авиации: большинство американских систем для отслеживания трафика самолётов разработано на базе Linux.

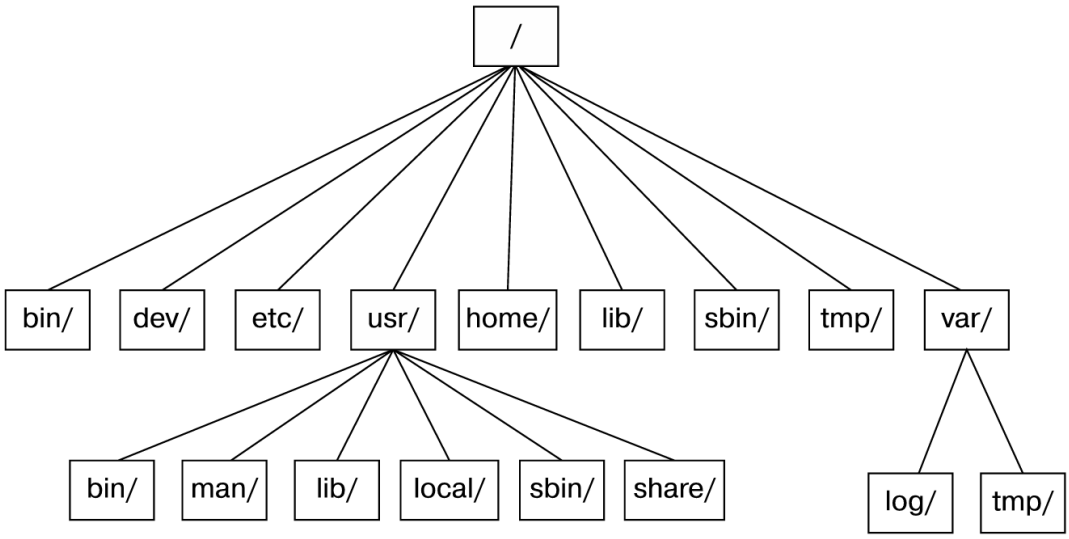
3.2 Файловая система Linux

Файловая система - это некий упорядоченный способ учета, хранения и извлечения данных на различных носителях. Например, жестких дисках, SSD, USB-флешках или в облачных сервисах.

Структурирование файловой системы охватывает процесс форматирования носителя и определения логики размещения файлов и папок на нем. В итоге она должна обеспечивать удобное размещение данных, их быстрый поиск и высокий уровень надежности.

Контроль за файлами и папками осуществляется операционной системой, что требует их взаимной совместимости и корректной работы в рамках данной ОС.

Файловая система Linux организована в форме иерархической структуры, которую образно можно представить в виде дерева. Она содержит в себе каталоги и подкаталоги, образуя таким образом вложенную структуру. Все файлы и каталоги начинаются от корневого каталога, который обозначается символом “/”, а далее распределяются по отдельным ветвям и листьям этого дерева. Таким образом создается логическая и удобная организация данных. На рисунке ниже представлена схематичная структура каталогов Linux.



Иерархия каталогов в системе Linux

Linux придерживается стандартов иерархии файловой системы (Filesystem Hierarchy Standard, FHS). Эти стандарты определяют основные правила организации и содержания каталогов в системах, аналогичных UNIX. Они обеспечивают единообразие в структуре файловой системы, делая работу с разными дистрибутивами более предсказуемой и понятной для пользователей и разработчиков.

Наиболее важные подкаталоги:

- `/bin` - содержит готовые к запуску программы (также известные как исполняемые файлы), включая большинство основных команд Unix, таких как ls и cp;
- `/dev` - содержит файлы устройств;
- `/etc` - центральный каталог конфигурации системы. Содержит пароль пользователя, загрузочные файлы, файлы устройств, сетевые настройки и др;
- `/home` - содержит домашние (личные) каталоги для обычных пользователей;
- `/lib` - в этом каталоге находятся файлы библиотек с кодом, который могут использовать исполняемые файлы. Существует два типа библиотек: статические и разделяемые. Каталог /lib должен содержать только разделяемые библиотеки, но другие каталоги lib, такие как /usr/lib, включают обе разновидности, а также другие вспомогательные файлы;

- `/proc` - предоставляет системную статистику через доступный для просмотра интерфейс каталогов и файлов. Каталог /proc содержит информацию о запущенных в данный момент процессах, а также некоторые параметры ядра;

- `/run` - содержит данные времени выполнения, относящиеся к системе, включая определенные идентификаторы процессов, файлы сокетов, записи состояния и во многих случаях системный журнал;

- `/sys` - похож на каталог /proc тем, что он предоставляет интерфейс устройствам и системе;

- `/sbin` - место для системных исполняемых файлов. Программы в каталогах /sbin связаны с управлением системой, поэтому простые пользователи обычно не имеют компонентов /sbin в своих путях команд. Многие из утилит в этом каталоге работают, только если запущены от имени суперпользователя;

- `/tmp` - место для хранения небольших, временных, не особо важных файлов. Любой пользователь может читать из каталога /tmp и записывать в него, но у пользователя может не быть доступа к файлам другого пользователя. Многие программы задействуют этот каталог в качестве рабочей области. Если какой-то файл важен, не помещайте его в каталог /tmp, потому что большинство дистрибутивов очищают его при загрузке, а некоторые даже периодически удаляют старые файлы;

- `/usr` - сокращение от user (пользователь), однако в этом подкаталоге нет пользовательских файлов. Вместо этого он содержит большую иерархию каталогов, включая основную часть системы Linux. Многие имена каталогов в /usr совпадают с именами в корневом каталоге (например, /usr/bin и /usr/lib), и они содержат файлы одного типа;

- `/var` - подкаталог переменных, куда программы записывают информацию, которая может изменяться с течением времени. Здесь находятся системные журналы, отслеживание активности пользователей, кэши и другие файлы, создаваемые системными программами и управляемые ими. (Здесь также есть каталог /var/tmp, но система не стирает его при загрузке.)

3.3 CLI и оболочка

Прежде, чем перейти к описанию команд, разберёмся с основными понятиями.

CLI (интерфейс командной строки) - это программный механизм, используемый для взаимодействия с операционной системой с помощью текстовых команд. При этом необходимо следовать специальному синтаксису таких команд. CLI существует во всех операционных системах.

С помощью CLI можно:

- Переименовывать, перемещать, удалять, преобразовывать файлы и каталоги;
- Открывать и закрывать программы;
- Управлять различными процессами на компьютере;
- Управлять приложениями облачных сервисов;
- Управлять правами доступа к ресурсам;
- Оптимизировать выполнение повторяющихся задач.

Недостатки CLI:

- Ограниченные возможности для представления информации;
- CLI кажется более сложным, чем GUI (графический пользовательский интерфейс).

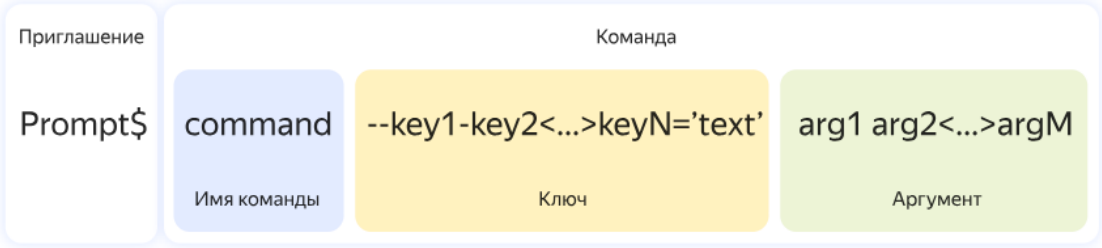
Преимущества CLI:

- **Направленность внимания.** Интерфейс CLI отображает информацию только по тому проекту, над которым в настоящий момент идет работа. Это позволяет меньше отвлекаться, глубоко погружаясь в процесс.
- **Информативность.** CLI всегда содержит более широкий набор команд, чем GUI.
- **Компактность.** CLI занимает гораздо меньше места на компьютере, чем любой графический интерфейс. Это позволяет выполнять аналогичные задачи с меньшими затратами ресурсов.

- **Автоматизация.** Выполнение повторяющихся задач в GUI занимает большое количество времени и ресурсов. С помощью CLI можно автоматизировать такие процессы, написав список команд, которые должен выполнять ваш компьютер.

Оболочка - это пользовательский интерфейс, отвечающий за обработку команд, введенных в CLI. Оболочка является посредником между пользователем и операционной системой. CLI по умолчанию реализован в различных оболочках в зависимости от того, какая операционная система установлена на компьютере.

Командная срока в CLI имеет определенный синтаксис, включающий в себя несколько элементов:



Приглашение - обязательный элемент командной строки, располагающийся в самом начале. Приглашение оканчивается специальным символом, который сигнализирует о том, что CLI готов к вводу команды. Приглашение появляется при открытии командной строки и после завершения очередной команды.

Команда - это выраженное в текстовом виде предписание для компьютера. Команда состоит из нескольких частей:

- **Имя команды** - обязательный элемент, располагающийся в самом начале команды и сразу после приглашения. Имя команды обозначает действие, которое вы хотите выполнить.
- **Ключ (флаг)** - необязательный элемент команды, который располагается после названия и начинается с дефиса «-» или двойного дефиса «–». Обозначает вспомогательные параметры, которые вы применяете к команде. К одной команде можно применить несколько ключей одновременно.
- **Аргумент** - необязательный элемент команды, который располагается в конце. Аргумент обозначает различные входные данные, необходимые для выпол-

нения команды. Например, аргументом может быть название файла, который вы хотите удалить. Некоторые команды могут принимать несколько аргументов одновременно.

3.4 Переменные окружения и оболочки

Оболочка может хранить временные переменные, называемые *переменными оболочки*, содержащие значения текстовых строк. Переменные оболочки очень полезны для отслеживания значений в скриптах, а некоторые из них управляют поведением оболочки. Чтобы присвоить значение переменной оболочки, используйте:

```
$ SOME_ENV=some_value
```

Здесь значение переменной с именем `SOME_ENV` устанавливается равным значению `some_value`. Чтобы обратиться к этой переменной, используйте `$SOME_ENV`

Переменная окружения похожа на переменную оболочки, но она не специфична для оболочки. Все процессы в системах Unix имеют хранилище переменных окружения.

Основное различие между переменными окружения и оболочки заключается в том, что операционная система передает все переменные окружения системным процессам и программам, выполняемым оболочкой, в то время как переменные оболочки доступны только внутри текущего экземпляра оболочки.

Переменная окружения назначается с помощью команды `export`:

```
$ SOME_ENV=some_value
$ export SOME_ENV
```

3.5 Основные команды CLI

- `pwd` - вывести название текущей/рабочей директории.
- `ls` - вывести содержимое директории.

  ```
  $ ls [OPTION]... [FILE]...
  ```

  Основные опции:

  - `-l` - вывести содержимое как список;
  - `-a, --all` - также вывести файлы, начинающиеся с «.»;

- `cd` - изменить рабочую директорию.

```
$ cd [DIR]
```

Команда без аргумента изменяет директорию на домашнюю (неявно подставляет в качесте аргумента `$HOME`). Команда `cd -` перемещает в предыдущую рабочую директорию (вместо `-` подставляет `$OLDPWD`)

- `mkdir` - создать директорию/директории, если они не существуют.

```
$ mkdir [OPTION]... DIRECTORY...
```

Основные опции:

  - `-p, --parents` - не вызывать ошибку, если директория существует, создать родительские каталоги при необходимости;

- `rmdir` - удалить пустые директории.

```
$ rmdir [OPTION]... DIRECTORY...
```

Основные опции:

  - `-p, --parents` - удалить `DIRECTORY` и ее предков. Например, `rmdir -p a/b` аналогично `rmdir a/b a`;

- `touch` - изменить временную метку файла.

```
$ touch [OPTION]... FILE...
```

В самом простом варианте создаёт пустой файл: `touch some_file`

- `cat` - конкатенировать файлы и вывести на стандартный вывод.

```
$ cat [OPTION]... [FILE]...
```

Основные опции:

  - `-n, --number` - пронумеровать строки;
  - `-E, --show-ends` - вывести `$` в конце каждой строки;
  - `-T, --show-tabs` - вывести символы табуляции как `^I`;

- `mv` - переместить (переименовать) файлы.

```
$ mv [OPTION]... [-T] SOURCE DEST
$ mv [OPTION]... SOURCE... DIRECTORY
$ mv [OPTION]... -t DIRECTORY SOURCE...
```

Основные опции:

- `-T, --no-target-directory` - воспринимать `DEST` как обычный файл;
- `-t, --target-directory=DIRECTORY` - переместить все `SOURCE`-аргументы в `DIRECTORY`;

- `cp` - копировать файлы и директории.

```
$ cp [OPTION]... [-T] SOURCE DEST
$ cp [OPTION]... SOURCE... DIRECTORY
$ cp [OPTION]... -t DIRECTORY SOURCE...
```

Основные опции:

- `-T, --no-target-directory` - воспринимать `DEST` как обычный файл;
- `-t, --target-directory=DIRECTORY` - переместить все `SOURCE`-аргументы в `DIRECTORY`;
- `-R, -r, --recursive` - копировать директории рекурсивно;

- `rm` - удалить файлы или директории.

```
$ rm [OPTION]... [FILE]...
```

Основные опции:

- `-r, -R, --recursive` - удалить директории и их содержимое рекурсивно;
- `-d, --dir` - удалить пустые директории;

- `man` - интерфейс к справочным руководствам по системе. Каждый аргумент `page`, передаваемый в man, обычно является именем программы, утилиты или функции.

  Для удобства страницы сгруппированны в следующие разделы в соответствии с нумерацией:

  1. Пользовательские команды;

2. Системные вызовы;
3. Высокоуровневая документация библиотек программирования Unix;
4. Интерфейсы устройств и информация о драйверах;
5. Описание файлов конфигурации системы;
6. Игры;
7. Форматы файлов, соглашения и кодировки (ASCII, суффиксы и др.);
8. Системные команды и серверы.

Например команда `man printf` выведет страницу руководства пользовательской команды `printf`, а `man 3 printf`, где 3 это номер раздела, - страницу руководства к функции `printf` стандартной библиотеки Си.

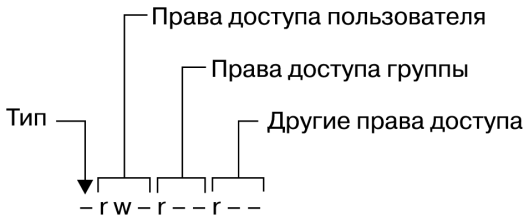
С помощью `man` вы можете узнать более подробную информацию об использовании вышеперечисленных и многих других команд

3.6 Режимы файлов и права доступа

Каждый файл Unix имеет набор доступов, которые определяют, можете ли вы читать, записывать или запускать файл. Команда `ls -l` отображает права доступа:

```
-rw-r--r-- 1 someuser somegroup  530 Nov 27 23:56 somefile.txt
```

Режим файла `-rw-r-r-` представляет права доступа к файлу и некоторую дополнительную информацию.



Составные части режима файла

Первый символ режима - это тип файла. `-` в этой позиции обозначает обычный файл, в котором хранятся двоичные или текстовые данные. Каталоги обозначаются буквой `d`.

Оставшаяся часть режима файла содержит права доступа, которые разбиваются на три равных набора: пользователь (user), группа (groups) и остальные (other). Каждый набор прав доступа может содержать четыре основных варианта:

- `r` - файл доступен для чтения;
- `w` - файл доступен для записи;
- `x` - файл является исполняемым;
- `-` - означает «ничего», то есть никаких прав на этот слот не было предоставлено.

Права доступа пользователя (первый набор) принадлежат пользователю - владельцу файла. В примере это пользователь `someuser`. Второй набор - права доступа группы - предназначен для группы файла (в примере - для группы `somegroup`). Любой пользователь, относящийся к данной группе, может воспользоваться этими правами. Все остальные в системе имеют доступ в соответствии с третьим набором - правами доступа всех остальных, которые иногда называются правами доступа world (то, что доступно всему миру).

У некоторых исполняемых файлов в наборе прав доступа пользователя вместо `x` указана `s`. Это говорит о том, что исполняемый файл имеет значение setuid, что означает: при выполнении программы она запускается так, как будто файлом владеет пользователь, а не вы.

Изменение прав доступа. Чтобы изменить права доступа для файла или каталога, используйте команду `chmod` (изменение режима, change mode). Сначала выберите набор прав доступа, которые вы хотите изменить, а затем - бит для изменения. Например, чтобы добавить в файл file права доступа для группы (g, от group) и остальных (o, от other) на чтение (r, от read), можете выполнить команды:

```
$ chmod g+r file
$ chmod o+r file
```

или

```
$ chmod go+r file
```

Для удаления прав доступа замените `+` на `-`. Также есть возможность изменять права доступа с помощью цифр:

```
$ chmod 644 file
```

Это называется *абсолютным* изменением, потому что оно устанавливает все биты прав доступа одновременно. Чтобы понять, как это работает, вам нужно уметь представлять биты прав доступа в восьмеричной системе счисления (каждая цифра представляет число с основанием 8, от 0 до 7, и соответствует набору прав доступа).

r	w	x	Цифра	Текстовый аналог	Значение
0	0	0	0	`-rwx`	Удалить все права
0	0	1	1	`+x-rw`	Нельзя читать и изменять, испольняемый
0	1	0	2	`+w-rx`	Можно только записывать в файл
0	1	1	3	`+wx-r`	Можно записывать в файл, испольняемый
1	0	0	4	`+r-wx`	Можно только читать из файла
1	0	1	5	`+rx-w`	Можно читать из файла, исполняемый
1	1	0	6	`+rw-x`	Можно читать и записывать
1	1	1	7	`+rwx`	Все права

Директории также имеют права доступа. Вы можете перечислить содержимое каталога, если оно доступно для чтения, но получить доступ к файлу в каталоге можете, только если каталог исполняемый. В большинстве случаев вам нужно и то и другое. Распространенная ошибка, которую пользователи совершают при настройке разрешений каталогов, — это случайное удаление разрешения на выполнение при использовании абсолютных режимов.

3.7 Управление пользователями и группами

Под *пользователем* понимается любой, кто работает с компьютером. Как правило, для каждого пользователя в системе создаётся отдельный аккаунт, которому присваивается специальное имя. С помощью имени пользователь получает доступ к своему аккаунту и, в конечном счёте, к системе. Некоторые системные службы запускаются либо привилегированными аккаунтами, либо строго определёнными аккаунтами пользователей.

Механизм пользователей был разработан по соображениям безопасности, для ограничения доступа к различным частям системы. Суперпользователь (root) имеет полный доступ к операционной системе и её настройкам. Этот аккаунт используется только для целей системного администрирования; обычные пользователи могут временно повысить уровень своих прав командами `su` и `sudo`.

Пользователи часто объединятся в "группы". Добавление пользователя в определённую группу обычно даёт ему дополнительные права, связанные с этой группой.

Информация о пользователях храниться в файле `/etc/passwd`.

Список пользователей, которые в настоящий момент выполнили вход в систему, можно вывести командой `who`.

Добавление нового пользователя выполняется командой `useradd`:

```
$ useradd -m -G <additional_groups> -s <login_shell> <user_name>
```

- `-m, --create-home` - создать домашний каталог пользователя по адресу `/home/user_name`. В каталоге создаётся набор стартовых файлов, владельцем которых назначается новый пользователь.
- `-G, --groups` - список дополнительных групп, в которые должен войти новый пользователь. Группы в списке разделяются запятыми. По умолчанию новый пользователь добавляется только в начальную группу.
- `-s, --shell` - путь к файлу оболочки входа (login shell) пользователя.

Группы в параметре `-G` задаются именами или номерами существующих групп. Если этот парамет не указан, то поведение useradd зависит от переменной `USERGROUPS_ENAB` в файле `/etc/login.defs`. Стандартное поведение - создать группу, название которой совпадает с именем нового пользователя.

Когда оболочка входа не нужна, например, если аккаунт пользователя создаётся для работы неинтерактивной службы, вместо обычной оболочки можно указать `/usr/bin/nologin`. В этом случае процедура входа будет корректно отклоняться.

После создания пользователя необходимо задать для него пароль:

```
$ passwd <user_name>
```

В файле `/etc/group` перечислены существующие группы. Вывести список групп, в которых состоит пользователь можно с помощью команды `groups <user_name>`. Если `<user_name>` не указан, то будет выведен список групп текущего пользователя.

Для создания новой группы используется команда `groupadd <group>`. Чтобы добавить пользователя в группу, примените команду:

```
$ usermod -aG <additional_groups> <user_name>
```

3.8 Управление пакетами

Системы управления пакетами («менеджер пакетов» или «пакетный менеджер») - это набор программного обеспечения, позволяющего управлять процессом установки, удаления, настройки и обновления различных компонентов ПО.

Различные дистрибутивы ОС Linux имеют свои форматы пакетов:

- `.deb` - Debian и производные (Ubuntu, Mint и т.д.)
- `.rpm` - Red Hat и производные (CentOS, Fedora и т.д.), OpenSUSE
- `.apk` - Android
- `.ebuild` - Gentoo

Все системы управления пакетами имеют схожие команды. Разберём основные на примере `apt`. **APT** (Advanced Package Tool) является инструментом управления пакетами, который широко используется в дистрибутивах Debian и Ubuntu.

Основные команды `apt`:

- `apt update` - обновить список доступных пакетов из репозиториев;
- `apt upgrade` - обновить все установленные пакеты до последних версий;
- `apt install <package_name>` - установить или обновить указанный пакет из репозиториев;
- `apt remove <package_name>` - удалить указанный пакет, оставляя файлы конфигурации;
- `apt purge <package_name>` - удалить указанный пакет, удаляя все файлы конфигурации;
- `apt search <keyword>` - искать пакеты по ключевому слову в репозиториях;
- `apt show <package_name>` - показать подробную информацию о пакете.

3.9 Знакомство с bash-скриптами

Bash-скрипты - это сценарии командной оболочки, написанные для оболочки bash.

Сценарии командной оболочки - наборы команд командной строки, которые объединены в один файл с целью выполнения определённых составных задач (К примеру, автоматическое обновление репозиториев и установка обновлений). При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии — это мощный способ автоматизации часто выполняемых действий.

Вы можете в приглашении командной строки написать две команды, указав точку с запятой между ними для разделения:

```
pwd; whoami
```

Это простой bash-скрипт, состоящий из двух команд. Сначала команда `pwd` выводит на экран сведения о текущей рабочей директории, потом команда `whoami` показывает данные о пользователе, под которым вы вошли в систему.

Каждый bash-скрипт начинается со специальной строки

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

В ней мы сообщаем, что хотим использовать оболочку Bash, указав путь до неё.

Команды отделяются знаком перевода строки или через точку с запятой, комментарии выделяют знаком решётки.

```
#!/bin/bash
pwd
whoami
# comment
ls; cd /var
```

Переменные. Переменные позволяют хранить информацию, например — результаты работы команд для использования их другими командами.

Существует два типа переменных в bash-скриптах:

- переменные среды;
- пользовательские переменные.

Переменные среды. Иногда в командах оболочки нужно работать с некими системными данными.

```
#!/bin/bash
echo "Home for the current user is: $HOME"
```

Сценарий выше выводит сообщение с указанием домашней директории, используя системную переменную `$HOME`

Пользовательские переменные. В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.

Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

```
#!/bin/bash
# testing variables
grade=5
person="Adam"
echo "$person is a good boy, he is in grade $grade"
```

Подстановка команд. Это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами:

- знаком апострофа «‘»;

  ```
  mydir=‘pwd‘
  ```

- конструкцией `$()`.

  ```
  mydir=$(pwd)
  ```

Математические операции. Для выполнения математических операций используется конструкция вида `$((a+b))`.

```
#!/bin/bash
var1=$(( 5 + 3 ))
```

```
var2=$(( $var1 * 2 ))
echo $var2
```

Условная конструкция if-then-else. Для обработки условий применяется конструкция `if-then-else`.

```
if <command1>    # начало блока и условие вхождения
then
<commands>       # тело
elif <command2> # иначе если
<commands>
else
<commands>
fi               # конец блока
```

Рассмотрим пример:

```
#!/bin/bash
if pwd
then
    echo "It works"
fi
```

Если выполнение команды `pwd` завершится успешно (со статус кодом 0), в консоль будет выведен текст «It works».

Сравнение чисел. В скриптах можно сравнивать числовые значения.

- `n1 -eq n2` - возвращает истинное значение, если `n1` равно `n2`;
- `n1 -ge n2` - возвращает истинное значение, если `n1` больше или равно `n2`;
- `n1 -gt n2` - возвращает истинное значение, если `n1` больше `n2`;
- `n1 -le n2` - возвращает истинное значение, если `n1` меньше или равно `n2`;
- `n1 -lt n2` - возвращает истинное значение, если `n1` меньше `n2`;
- `n1 -ne n2` - возвращает истинное значение, если `n1` не равно `n2`.

```
#!/bin/bash
val1=6
if [ $val1 -gt 5 ]
then
    echo "The test value $val1 is greater than 5"
else
    echo "The test value $val1 is not greater than 5"
fi
```

Обратите внимание на то, что выражение заключено в квадратные скобки.

Сравнение строк. В сценариях можно сравнивать и строковые значения.

- `str1 = str2` - возвращает истину, если строки идентичны;
- `str1 != str2` - возвращает истину, если строки не идентичны;
- `str1 < str2` - возвращает истину, если `str1` меньше, чем `str2`;
- `str1 > str2` - возвращает истину, если `str1` больше, чем `str2`;
- `-n str1` - возвращает истину, если длина `str1` больше нуля;
- `-z str1` - возвращает истину, если длина `str1` равна нулю.

```
#!/bin/bash
user ="someuser"
if [$user = $USER]
then
    echo "The user $user is the current logged in user"
fi
```

Существует особенность сравнения строк: операторы `>` и `<` необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится. Скрипт интерпретирует знак `>` как команду перенаправления вывода.

```
#!/bin/bash
val1=text
val2="another text"
if [ $val1 \> "$val2" ]
```

```
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
```

Ещё одна особенность операторов `>` и `<` заключается в том, как они работают с символами в верхнем и нижнем регистрах: в командах сравнения прописные буквы меньше строчных. Сравнение строк здесь выполняется путём сравнения ASCII-кодов символов, порядок сортировки, таким образом, зависит от кодов символов.

Проверки файлов. Следующие команды позволяют проверять различные условия, касающиеся файлов.

- `-e file` - проверяет, существует ли файл;
- `-f file` - проверяет, существует ли файл и является ли он файлом;
- `-d file` - проверяет, существует ли файл и является ли он директорией;
- `-s file` - проверяет, существует ли файл и не является ли он пустым;
- `-r file` - проверяет, существует ли файл и доступен ли он для чтения;
- `-w file` - проверяет, существует ли файл и доступен ли он для записи;
- `-x file` - проверяет, существует ли файл и и является ли он исполняемым;
- `file1 -nt file2` - проверяет, новее ли `file1`, чем `file2`;
- `file1 -ot file2` - проверяет, старше ли `file1`, чем `file2`;
- `-O file` - проверяет, существует ли файл и является ли его владельцем текущий пользователь;
- `-G file` - проверяет, существует ли файл и соответствует ли его идентификатор группы идентификатору группы текущего пользователя;

Циклы for. Оболочка bash поддерживает циклы for, которые позволяют организовывать перебор последовательностей значений. Вот какова базовая структура таких циклов:

```
for var in list
do
<commands>
done
```

```
#!/bin/bash
for var in first second third fourth fifth
do
    echo The  $var item
done
```

В данном примере списком является `first second third fourth fifth`, в котором элементы разделены по пробелу. На каждой итерации цикла в переменную `var` будет записываться следующее значение из списка `list`.

Обратите внимание на то, что переменная `var` сохраняет значение при выходе из цикла, её содержимое можно менять и работать с ней как с любой другой переменной.

Перебор сложных значений. В списке, использованном при инициализации цикла `for`, могут содержаться не только простые строки, состоящие из одного слова, но и целые фразы, в которые входят несколько слов и знаков препинания.

```
#!/bin/bash
for var in first "the second" "the third" "'Ill do it"
do
    echo "This is: $var"
done
```

Инициализация цикла списком, полученным из результатов работы команды. В цикл можно передать список, который является результатом работы какой-либо команды.

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

```
file="myfile"
for var in $(cat $file)
do
    echo " $var"
done
```

В этом примере с помощью команды `cat` считывается содержимое файла, состоящего из списка слов, разделённых символом перевода строки.

Надо учесть, что подобный подход, если ожидается построчная обработка данных, не сработает для файла, в строках которого может содержаться по несколько слов, разделённых пробелами. Цикл будет обрабатывать отдельные слова, а не строки.

Разделители полей. Причина вышеописанной особенности заключается в специальной переменной окружения, которая называется `IFS` (Internal Field Separator) и позволяет указывать разделители полей. По умолчанию оболочка bash считает разделителями полей следующие символы:

- пробел;
- знак табуляции;
- знак перевода строки.

Если bash встречает в данных любой из этих символов, он считает, что перед ним — следующее самостоятельное значение списка.

Для того, чтобы решить проблему, можно временно изменить переменную среды `IFS`.

```
#!/bin/bash
file="/etc/passwd"
IFS=$'\n'     # переопределение переменной среды
for var in $(cat $file)
do
    echo " $var"
done
```

Разделителями могут быть любые символы.

Обход файлов, содержащихся в директории. Один из самых распространённых вариантов использования циклов `for` в bash-скриптах заключается в обходе файлов, находящихся в директории, и в обработке этих файлов.

```
#!/bin/bash
for file in /home/someuser/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
```

Инициализация цикла происходит с помощью подстановочного знака «*» в конце адреса каталога. Этот символ воспринимается как шаблон «все файлы с любыми именами».

Циклы for в стиле C. В bash-скриптах можно использовать циклы `for`, описание которых выглядит очень похожим на циклы в стиле C:

```
for (( начальное значение переменной ; условие окончания цикла; изменение
   переменной ))
```

```
#!/bin/bash
for (( i=1; i <= 10; i++ ))
do
    echo "number is $i"
done
```

Цикл while. В таком цикле можно задать команду проверки некоего условия и выполнять тело цикла до тех пор, пока проверяемое условие возвращает ноль, или сигнал успешного завершения некоей операции. Когда условие цикла вернёт ненулевое значение, что означает ошибку, цикл остановится.

```
while <command>
do
<commands>
```

```
done
```

```
#!/bin/bash
var1=5
while [ $var1 -gt 0 ]
do
    echo $var1
    var1=$(( $var1 - 1 ))
done
```

Управление циклами. С помощью команд `break` и `continue` можно менять стандартное поведение циклов.

Команда break. Позволяет прервать выполнение цикла.

```
#!/bin/bash
for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Number: $var1"
done
```

Команда continue. При появлении этой команды текущая итерация заканчивается досрочно и начинается следующая.

Параметры и ключи командной строки. Иногда скриптам необходимо передавать некоторые данные. Самый распространённый вариант - использование параметров командной строки.

```
$ ./somescript param1 param2
```

Чтение параметров командной строки. Оболочка bash назначает специальным переменным (позиционным параметрам) введённые при вызове скрипта параметры командной строки:

- `$0` - имя скрипта;

- `$1` - первый параметр;
- `$2` - второй параметр и так далее до `$9`;

Параметры командной строки разделяются пробелами.

Если скрипту надо больше девяти параметров, при обращении к ним номер в имени переменной надо заключать в фигурные скобки (`{$10}`).

Подсчёт параметров. Существует специальная переменная `$#`, которая содержит количество переданных параметров.

Захват всех параметров командной строки. В некоторых случаях нужно захватить все параметры, переданные скрипту. Для этого можно воспользоваться переменными `$*` и `$@`. Обе они содержат все параметры командной строки, что делает возможным доступ к тому, что передано сценарию, без использования позиционных параметров.

Переменная `$*` содержит все параметры, введённые в командной строке, в виде единого «слова».

В переменной `$@` параметры разбиты на отдельные «слова». Эти параметры можно перебирать в циклах.

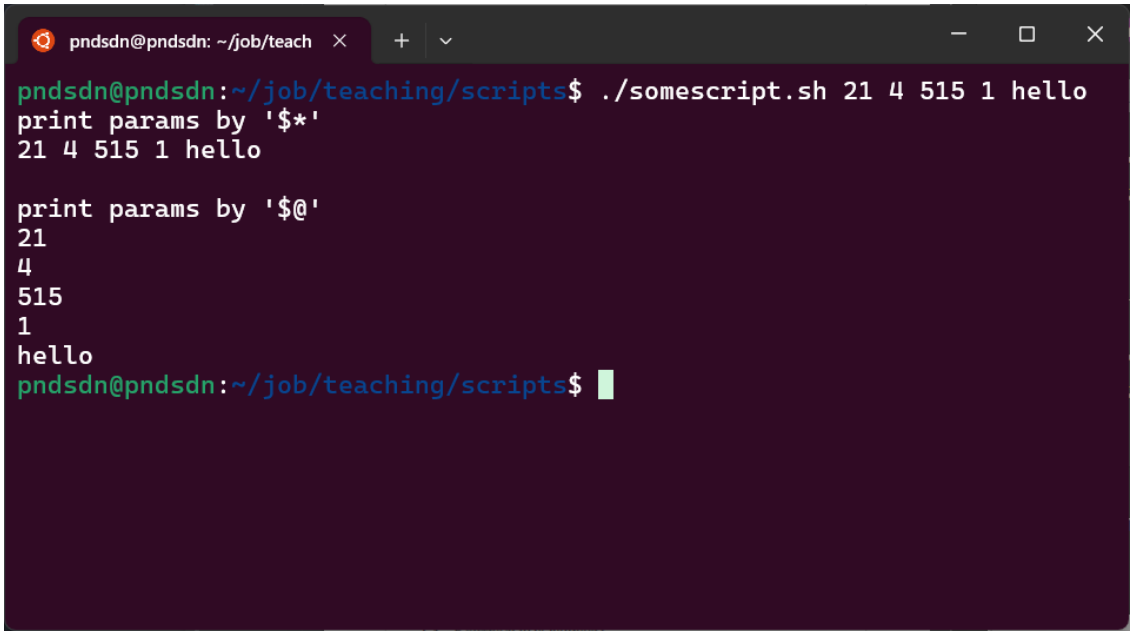
Чтобы увидеть разницу, выполним следующий сценарий:

```
#!/bin/bash
echo "print params by '\$*'"
for param in "$*"
do
    echo "$param"
done

echo

echo "print params by '\$@'"
for param in "$@"
do
    echo "$param"
done
```

В результате получится следующий вывод:



Команда shift. Сдвигает значения позиционных параметров влево. Например, значение переменной `$3` становится значением переменной `$2`, значение `$2` переходит в `$1`, а то, что было до этого в `$1`, теряется. Однако, значение переменной `$0`, содержащей имя скрипта, не меняется.

Ключи командной строки. Ключи служат для управления сценариями. К примеру, у команды `ls` есть ключ `-a`, который отвечает за вывод всех файлов, находящихся в директории.

```
#!/bin/bash
echo
while [ -n "$1" ]
do
    case "$1" in
        -a) echo "Found the -a option" ;;
        -b) echo "Found the -b option" ;;
        -c) echo "Found the -c option" ;;
        *) echo "$1 is not an option" ;;
    esac
    shift
done
```

В примере выше использовалась конструкция `case`, которая сверяет переданный ключ со списком обрабатываемых скриптом ключей. Если при вызове скрипта будет

использован любой ключ, обработка которого не предусмотрена, будет исполнена ветвь «*».

Ввод и вывод. В Linux всё есть файл, в том числе - ввод и вывод. Операционная система идентифицирует файлы с использованием дескрипторов.

Каждому процессу позволено иметь до девяти открытых дескрипторов файлов. Оболочка bash резервирует первые три дескриптора с идентификаторами 0, 1 и 2.

- `0`, `STDIN` - стандартный поток ввода;
- `1`, `STDOUT` - стандартный поток вывода;
- `2`, `STDERR` - стандартный поток ошибок.

STDIN - стандартный поток ввода оболочки. Для терминала стандартный ввод — это клавиатура. Когда в сценариях используют символ перенаправления ввода — `<`, Linux заменяет дескриптор файла стандартного ввода на тот, который указан в команде. Система читает файл и обрабатывает данные так, будто они введены с клавиатуры.

Многие команды bash принимают ввод из `STDIN`, если в командной строке не указан файл, из которого надо брать данные.

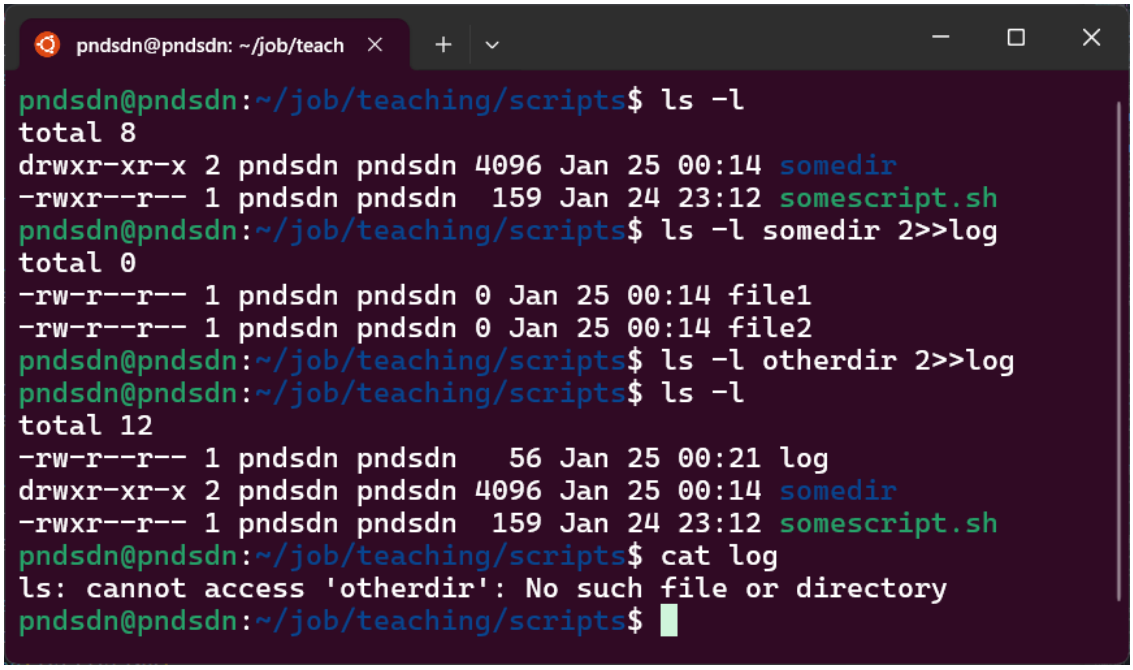
STDOUT - стандартный поток вывода оболочки. По умолчанию это — экран. Большинство bash-команд выводят данные в `STDOUT`, что приводит к их появлению в консоли. Данные можно перенаправить в файл, присоединяя их к его содержимому, для этого служит команда `»`.

```
$ pwd >> somefile
```

В примере выше результат команды `pwd` добавится в конец файла `somefile`.

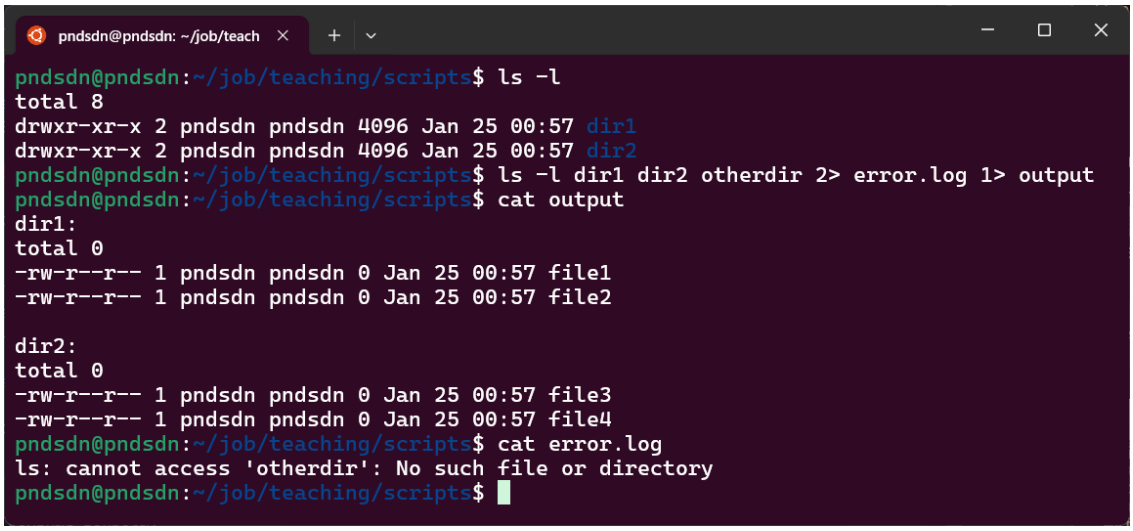
STDERR - стандартный поток ошибок оболочки. По умолчанию этот дескриптор указывает на то же самое, на что указывает `STDOUT`, именно поэтому при возникновении ошибки мы видим сообщение на экране.

Перенаправление потока ошибок. Можно перенаправить поток ошибок в файл. Так как дескриптор файла `STDERR` - 2, то это можно сделать с помощью команды перенаправления `2>` (`2»`).



В примере выше поток ошибок перенаправлялся в файл `log`. При указании существующей директории команда завершилась успешно и файл `log` не был создан. При указании директории `otherdir` в стандартный поток вывода ничего не было отправлено, но появился файл `log` с текстом ошибки.

Перенаправление потоков ошибок и вывода. При написании сценариев командной строки может возникнуть ситуация, когда нужно организовать и перенаправление сообщений об ошибках, и перенаправление стандартного вывода. Для этого необходимо использовать команды перенаправления для соответствующих дескрипторов с указанием файлов.



Из примера видно, что в файл `output` попал стандартный вывод команды `ls`, а в файл `error.log` - сообщение об отсутствии файла с именем `otherdir`.

`STDOUT` и `STDERR` можно перенаправить в один файл с помощью команды `&>`.

Перенаправление вывода в скриптах . Существует два метода перенаправления вывода в сценариях командной строки:

- временное перенаправление или перенаправление вывода одной строки;
- постоянное перенаправление или перенаправление всего вывода в скрипте либо в какой-то его части.

Временное перенаправление. Для этого достаточно использовать команду перенаправления, указав дескриптор `STDERR`, указав перед номером дескриптора символ амперсанда `&`:

```
#!/bin/bash
echo "This is an error" >&2
echo "This is normal output"
```

Постоянное перенаправление. Если в скрипте нужно перенаправлять много выводимых на экран данных, можно задать перенаправление вывода в определённый дескриптор на время выполнения скрипта, воспользовавшись командой `exec`:

```
#!/bin/bash
exec 1>outfile
echo "This is a test of redirecting all output"
echo "from a shell script to another file."
echo "without having to redirect every line"
```

В данном примере весь стандартный поток вывода перенаправляется в файл `outfile`.

Перенаправление ввода в скриптах. Для перенаправления ввода можно воспользоваться той же методикой, которую мы применяли для перенаправления вывода. Например, команда exec позволяет сделать источником данных для `STDIN` какой-нибудь файл:

```
exec 0< myfile
```

Эта команда указывает оболочке на то, что источником вводимых данных должен стать файл `myfile`, а не обычный `STDIN`.

4 Ход работы

Для выполнения этой и последущих лабораторных работ необходимо наличие любой UNIX-подобной ОС (Linux, macOS). Перед началом выполнения установите Linux как:

1. Основную ОС или паралелльно (dual-boot);
2. WSL2 для Windows;
3. Виртуальная машина.

Список отсортирован в порядке убывания приоритета используемого решения.

4.1 Написание скрипта

Создайте файл `deleteFiles.sh`, в котором напишите bash-скрипт для удаления файлов. Скрипт должен в качестве параметра принимать имя файла для удаления. Также скрипт должен предоставлять возможность указать флаг `-p`, указывающий на то, что значение параметра должно расцениваться как шаблон, и что удалению подлежат все файлы, соответсвующие шаблону.

Под шаблоном понимается любое вхождение подстроки параметра в названии файла. К примеру, если шаблон имеет значение `file`, то под критерий попадают: `file`, `file1`, `some.file`, `filefile`, `file.txt` и т.п. Не должны быть удалены файлы: `FILE`, `fiLe`, `fil0e` и т.п.

ПРЕДУПРЕЖДЕНИЕ: в скрипте в момент удаления файла явно указывайте относительный путь, иначе можете удалить больше, чем требовалось.

Если пользователь неправильно введёт команду, выведите подсказку с порядком указания ключей и параметров.

Сделайте файл исполняемым и проверьте правильность его работы.

5 Контрольные вопросы

1. Что такое Linux?

2. В чём отличие переменных окружения от переменных оболочки?

3. С помощью какой команды изменяются права доступа к файлу?