

# Справочная информация по системе Linux

Алексей Полухин

2023

# *Содержание*

1	Справочная информация. Полезные сведения . . . . .	2
2	Операционные системы . . . . .	5
2.1	Ядро ОС . . . . .	5
2.2	<i>UNIX</i> . . . . .	7
2.3	Структура каталогов в Linux . . . . .	9
2.4	Установка ПО в Linux . . . . .	11
2.5	Создание Linux . . . . .	13
2.6	Файловые системы . . . . .	13
2.7	Hardlink и Softlink . . . . .	14
2.8	Работа с файлами . . . . .	15
2.9	Потоки . . . . .	16
2.10	Условия . . . . .	17
2.11	Диски и монтирование . . . . .	17
3	Полезные bash-команды . . . . .	18
3.1	Пример сборки кода из исходников . . . . .	20
3.2	Зависимости . . . . .	21
4	Ответы на важные вопросы . . . . .	21

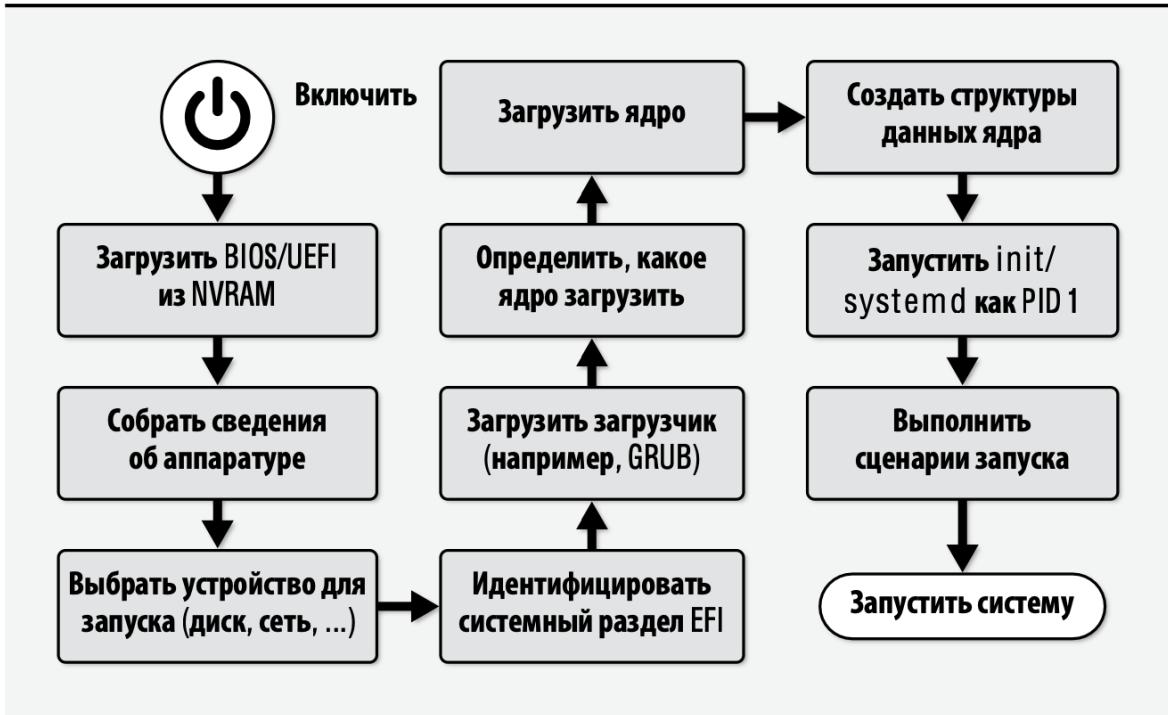


Рис. 1: Процессы загрузки Linux и Unix

# 1 Справочная информация. Полезные сведения

**Демон в Linux (служебный пользователь)** — фоновый процесс. Программа на уровне пользователя

Информация о пользователях содержится в файле `/etc/passwd`. Данные хранятся в следующем порядке: (имя, x, UID, группа, коммент, дом\_каталог, команда\_оболочка\_пользователя)

Информация о группах содержится в файле `/etc/group`. (имя группы, индентификатор группы, кто входит)

Информация о паролях содержится в файле `/etc/shadow` (там хранится хэш)

*Хеш-функция* — такая функция, что при известном  $y = f(x)$  невозможно восстановить  $x$ . Пример  $y = x^2$ . (Ещё примеры:  $y = \text{sign}(x)$ ,  $y = \cos(x)$ , функция Дирихле). Однозначно восстановить  $x$  невозможно. То есть это функция с неоднозначным соответствием. Так же хеш (результат работы функции) всегда одной

и той же длины.

*автентификация* — проверка наличия такого пользователя (совпадение логина и пароля)

*авторизация* — проверка прав доступа к серверу (уже после корректного совпадения логина и пароля)

Система Linux не хранит пароли. Только хеши паролей. То есть после ввода пароля рассчитывается хеш и, совпадении хранимого и рассчитанного хешей выполняется вход.

Права доступа к файлам с системе Linux рассмотрены на Рис. 2.

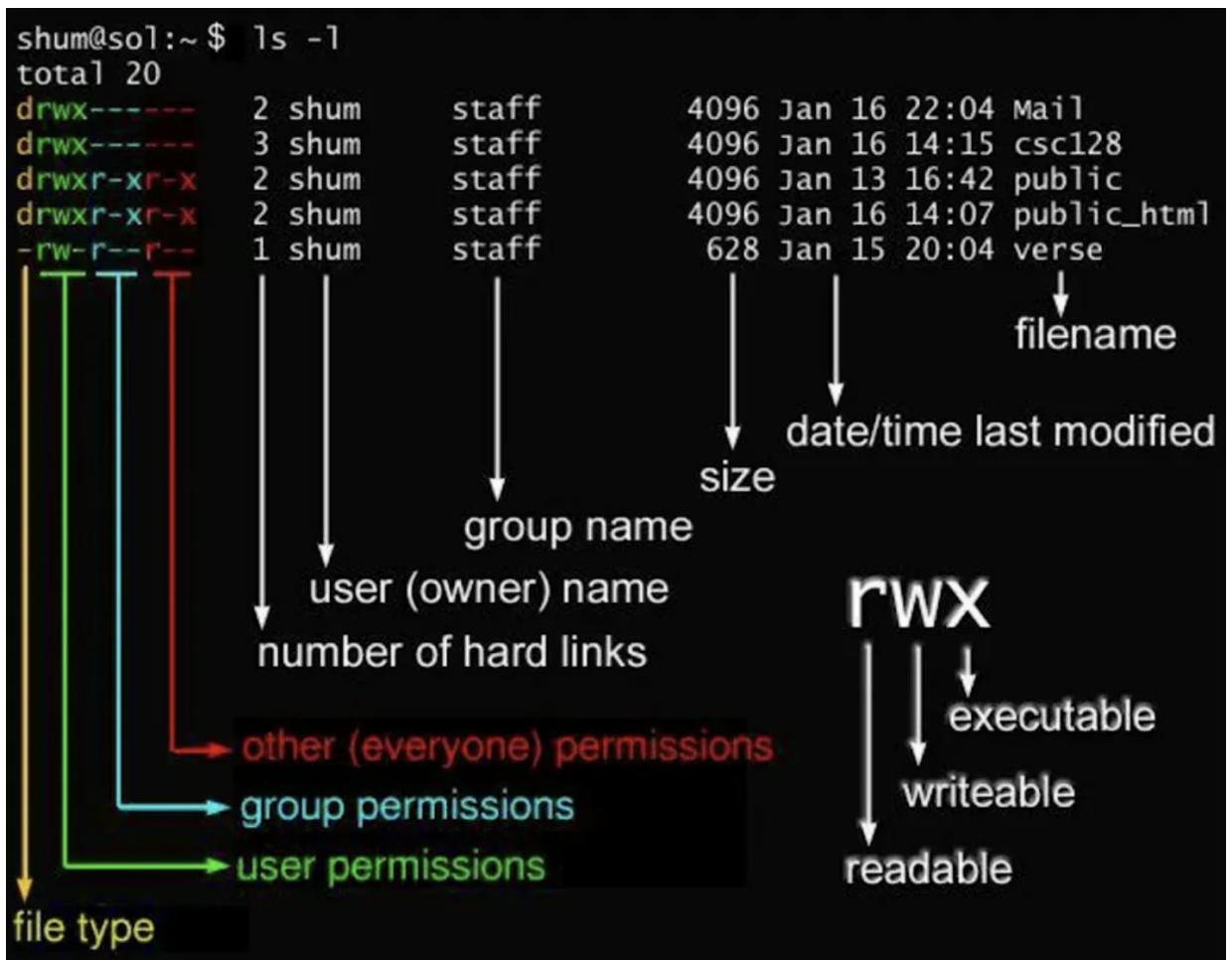


Рис. 2: Права доступа к файлам

Так же права доступа можно задавать, используя восьмиричную систему (Таблица 1). Сумма чисел из таблицы даёт права. То есть 761 означает, что владелец файла обладает всеми правами, группа владельца обладает правами на чтение

и запись, остальные обладают только правом хранить молчание на исполнение.

Права	Восьмиричное представление
Read (Чтение)	4
Write (Запись)	2
Execute (Выполнение)	1

Таблица 1: Права доступа к системе

Интересно, что право *x* для директории, означает, что мы можем зайти в директорию с помощью команды `cd`. А право *r* означает, что мы можем прочитать содержимое директории (с помощью команды `ls -l`).

**Переменная окружения** — это специальные переменные, которые использует система для облегчения своих настроек со стороны пользователя (начинаются с символа `$`)

`$HISTSIZE` — размер истории команд *bash*

```
$ echo $HISTSIZE
```

```
1000
```

Ещё полезная переменная `$PATH` — каталоги, в которых командная оболочка будет искать исполняемый файл. (если в них положить свой *script* на bash, то потом можно просто вводить команду *script* в терминал и скрипт будет запускаться)

*Киби-, меби-, гиби-* байты

*репозиторий* — место, где хранятся и **поддерживаются** какие-то данные  
Примеры репозиториев: *App Store*, *Google Play*

Эмулятор терминала — это программное обеспечение, которое имитирует работу физического терминала или консоли на компьютере. Он предоставляет пользователю интерфейс командной строки, позволяя взаимодействовать с операционной системой или другими удаленными системами через текстовый интерфейс. Физический терминал был устройством, аналогичным монитору с клавиатурой,

используемым для ввода и вывода текстовых данных. Физический терминал изображён на Рис. 3 и Рис. 4.



Рис. 3: Физический терминал

Логин в Linux не имеет значения для самой ОС, система ориентируется по UID (User Identifier), который принимает значение от 2 до  $2^{32} - 1$ . Именно UID определяет права пользователя. UID root = 0

## 2 Операционные системы

### 2.1 Ядро ОС

*Ядро ОС* – центральная часть ОС, обеспечивающая приложениям координированный доступ к ресурсам компьютера Рис. 5. В современных ОС приложение не может напрямую обратиться к ресурсам компьютера, поэтому приложения обращаются к **ядру**.



Рис. 4: Физический терминал

*Архитектура ядра операционной системы* — это структура и дизайн основной части операционной системы, которая обеспечивает основные функции управления ресурсами компьютера, планирование выполнения задач, обработку прерываний и обеспечивает взаимодействие между аппаратными устройствами и пользовательскими процессами.

В ОС, основанных на ядре, приложения имеют собственные независимые окружения. То есть свои участки памяти, своё процессорное время, свой доступ к устройствам ввода и вывода.

Современные ОС имеют пространство ядра (где идёт работа с оборудованием) и пространство пользователя.

### *Архитектуры ядер*

- Монолитное ядро (самое быстрое) — Linux
- Микроядро (самое отказоустойчивое)



Рис. 5: Ядро ОС

- Гибридное ядро — Windows

Ближе к оборудованию — быстрее, ближе к пространству пользователя — стабильнее. (Интересная аналогия: Python, C++, assembler)

Создание ОС с ядром возможно, когда на аппаратном уровне появляются кольца защиты. (методы разграничения ресурсов компьютера)

*Кольца защиты* — аппаратная реализация механизма разграничения ресурсов компьютера

Интерпретатор команд — это обычное приложение. В Linux они бывают разные: *shell, bash, ksh, csh, psh* . . . Он не является частью ядра ОС.

Для Linux не существует расширений файлов, они сделаны исключительно для удобства пользователя и для программ, которые взаимодействуют с этими файлами

Всё в Unix/Linux — это файл, просто поток байт

## 2.2 *UNIX*

### *Философия UNX*

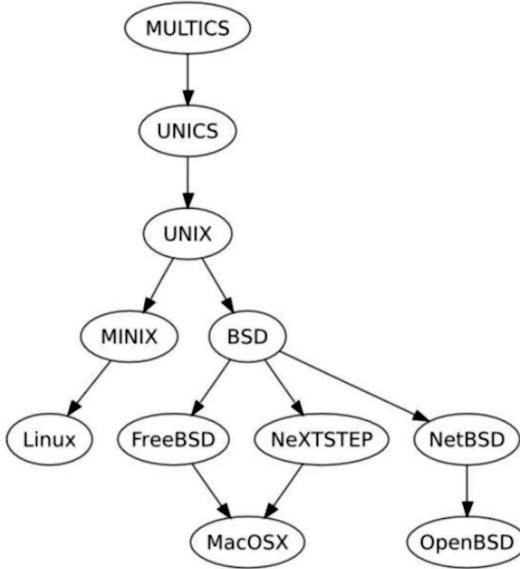


Рис. 6: Развитие операционных систем

1. **Пишите программы, которые делают что-то одно, но хорошо**
2. Пишите программы, которые работают вместе
3. Пишите программы, которые поддерживали бы текстовые потоки, так как это универсальный интерфейс

*POSIX* — набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой (в *UNIX*). (Средство общения с ядром ОС). Обеспечивает совместимость *UNIX*-подобных ОС.

Когда разработчики создают программы, используя стандарт *POSIX*, эти программы могут работать на разных операционных системах без необходимости изменять их код. Это облегчает перенос программного обеспечения с одной системы на другую, так как программы, сделанные в соответствии с *POSIX*, будут использовать одни и те же команды и функции, доступные во всех системах, поддерживающих этот стандарт.

*POSIX* описывает работу в пространстве пользователя. (видимо, именно из-за этого команды в терминалах MacOS, Unix и Linux совпадают)

Команды вроде ls (список файлов и директорий) и pwd (текущая рабочая директория) являются стандартными командами, определенными в стандарте POSIX.

POSIX определяет интерфейс и поведение для командной оболочки и других системных команд в операционных системах, таких как UNIX, Linux, macOS и другие, чтобы обеспечить переносимость программ между различными системами.

Сейчас UNIX используется в серверах и мейнфреймах

*Мейнфрейм (Mainframe)* — это тип большого и мощного компьютера, который обычно используется в корпоративных средах для обработки больших объемов данных и критически важных бизнес-приложений. (Например, используется в центрах управления (космическими) полётами, в банках, на биржах — в отраслях, где важна каждая *наносекунда*). (Используется в критически важных задачах, где важна скорость и бесперебойность)

На всех суперкомпьютерах установлен Linux, потому что они решают сложные математические задачи, если они вдруг остановятся, ничего критического не произойдёт.

## 2.3 Структура каталогов в Linux

См. Рис. 7

Основные каталоги

- / — root
- /bin — Необходимые утилиты, необходимые при работе всем пользователям (и в однопользовательском режиме)
- /boot — загрузочные файлы (файлы загрузчика, ядро, initrd, System.map)
- /dev — основные файлы устройств

- **/etc** — общесистемные конфигурационные файлы (настройки) (настройки ОС и служб ОС)
- **/home** — домашние каталоги пользователей (их персональные настройки и данные)
- **/lib** — Основные библиотеки, необходимые для работы программ из **/bin** и **/sbin**
- **/media** — Каталог, где производится монтирование сменных носителей (USB, CD-ROM)
- **/mnt** — каталог содержит временно монтируемые файловые системы
- **/opt** — Дополнительные программное обеспечение
- **/proc** — каталог, которые содержит информацию о всех процессах в нашей ОС
- **/root** — домашний каталог пользователя *root*
- **/run** — информация о системе с момента её загрузки (что запущено и чем это работает)
- **/sbin** — основные системные исполняемые файлы (основные программы для настройки и администрирования системы, *init*, *ifconfig*, *iptables*)
- **/srv** — данные для сервисов, представляемых системой (*www* или *ftp*)
- **/sys** — содержит информацию об устройствах, драйверах и некоторых свойствах ядра
- **/tmp** — временные файлы
- **/usr** — Большинство пользовательских приложений и утилит (используемых в многопользовательском режиме)

- `/var` — изменяемые файлы. (файлы регистрации, временные почтовые файлы, файлы спулеров)
- `/var/log` — логи
- `/home/username` — домашний каталог пользователя

## 2.4 Установка ПО в Linux

*Установка ПО в Linux/Unix — это просто переписывание бинарных файлов пакет — скомпилированный бинарный файл и перечень зависимостей*

### 1. Из исходных кодов (файлы на языке С) — **плохой способ**

- Можно модифицировать и устанавливать без прав администратора
- Поиск зависимостей очень долгий (как и сам процесс компиляции)
- Нет контроля ПО. (что, какой версии, куда, когда и кто установил — нет возможности узнать)
- Правильнее будет скопиллировать, собрать пакет и пакет установить с помощью пакетоного менеджера (будет вестись запись установленного ПО и версий)

### 2. Из пакетов

- Сразу видны зависимости
- Не тратится время на компиляцию
- Просто распаковка архива и копирование файлов в нужное место ОС (если есть все зависимости)
- Есть контроль версий ПО
- Нужны права администратора

- Пакеты создаются под определённый дистрибутив Linux

### 3. Из репозитория — лучший способ

- Сразу виден перечень зависимостей
- Есть контроль версий ПО
- Как правило, все зависимости устанавливаются автоматически из репозитория
- При установке пакетный менеджер сообщает, что нужно и сколько места это займет
- Репозиторий может быть локально или где-то на серверах

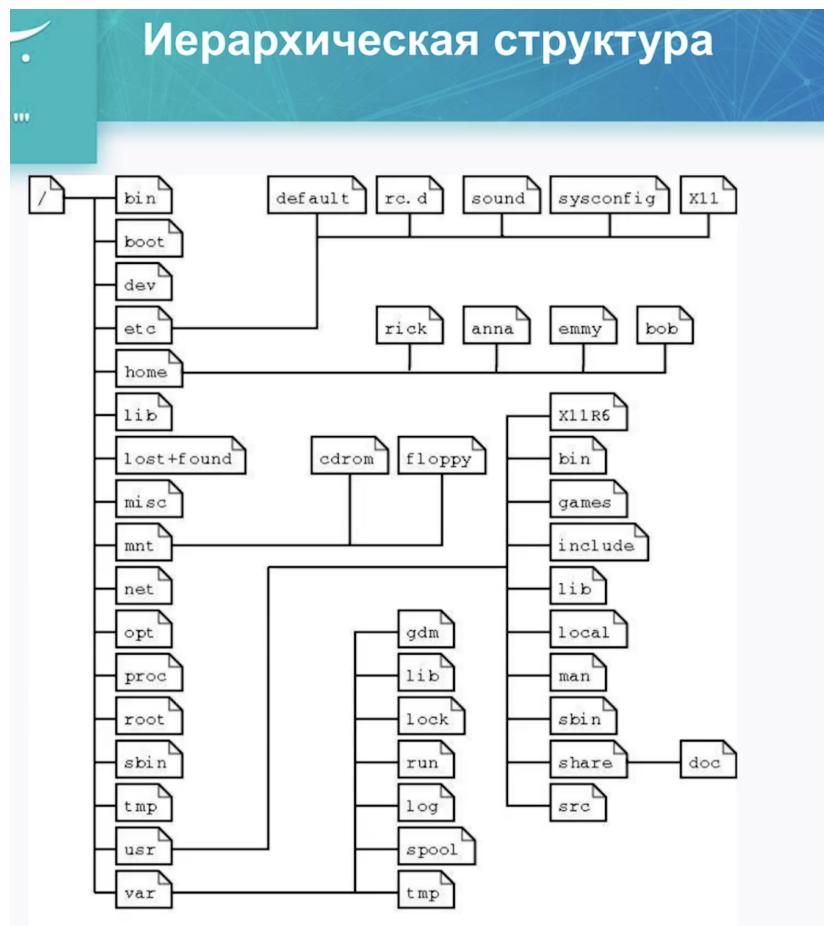


Рис. 7: Структура каталогов в Linux

## 2.5 Создание Linux

Linus Torvalds в 1991 создал **ядро** операционной системы Linux. Ядро — часть ОС, которая отвечает за взаимодействие с оборудованием и предоставляет определённый интерфейс (в данном случае *POSIX*) Пользователи и администраторы не работают с самим ядром, они работают в пространстве пользователя

У Richard M. Stallman было готово окружение GNU, но не было ядра.

Проекты объединились и образовалась ОС GNU/Linx. ОС GNU/Linux — это ядро и набор программ.

Официальная версия ядра vanilla kernel [www.kernel.org/](http://www.kernel.org/)

### ***Различия дистрибутивов***

- Разные версии ядра
- Разная структура каталогов
- Разные менеджеры пакетов

## 2.6 Файловые системы

В Linux достаточно часто используются файловые системы типа ext (ext2, ext3, ext4). Минимальная единица хранения информации на диске — блок.

В Linux и других Unix-подобных операционных системах, inode (или инод, inode) представляет собой структуру данных, используемую для хранения метаданных о файлах в файловой системе. Inode (инод) хранит информацию о файле, такую как разрешения доступа, владелец, временные метки (время создания, доступа и модификации), размер файла, количество жестких ссылок и местоположение данных файла на диске.

Когда вы создаете новый файл, операционная система резервирует для него соответствующий inode, который хранит информацию о файле. Каждый inode

имеет уникальный номер, который идентифицирует файл внутри файловой системы.

Когда вы выполняете различные операции с файлами (читаете, записываете, изменяете права доступа и т.д.), операционная система использует информацию из `inode` для обработки этих операций.

Посмотреть иноды можно командой `$ ls -li` или `$ stat file`

Каталог — представляет собой соответствие имён файлов и их инод. Имя каталога хранится в самом каталоге.

При удалении файла просто удаляется его имя из каталога. Удаляется вся информация о месторасположении файла, удаляется его инода. Вручную восстановить данные можно (найти эти данные на диски и убедиться, что это именно они). Но утилиту для восстановления написать невозможно, потому что после удаления инода не будем ничем отличаться от никогда не используемой иноды. (по крайней мере для файловой системы `ext4`).

Каталог и директория являются *синонимами*

## 2.7 Hardlink и Softlink

### Hardlink

*Hardlink* — это ещё одна запись в каталоге о файле. В выводе утилиты `ls -l` во втором столбце мы видим количество hardlink для этого файла.

Для создания hardlink есть утилита `$ ln что куда` То есть

`$ ln file1 file3` — file3 это ссылка на file1

⇒ файл существует до тех пор, пока ∃ хотя бы один hardlink на него.

**То есть при удалении одного hardlink (одной ссылки на файл) с самим файлом ничего не произойдёт.** Все созданные hardlink равноправны.

Нет главного hardlink

**hardlink нельзя создавать на каталоги!**

**hardlink можно создавать только в пределах одного диска, одной файловой системе.** Потому что иноды уникальны только в рамках одного диска.

## Softlink

Это обычные ярлыки (как в Windows) То есть это файлы, которые содержат с себе ссылку на другой файл.

```
$ ln -s file1 file3
```

У Softlink права 777 (но, если перейти к файлу, на который ведёт ссылка, вступят в силу его права)

В выводе ls -l первая буква в правах у softlink — l

**Softlink можно создавать на каталоги и между разными файловыми системами (дисками)**

Чаще используются Softlink.

**Пример:** если вышло обновление java, можно не переписывать путь к новой java в всех программах, которые её используют. Можно просто создать один softlink java, который будет смотреть на нужную версию java.

## 2.8 Работа с файлами

### grep

```
$ grep шаблон путь
```

```
$ grep "Failed" ./log
```

```
$ grep -i — вне зависимости от регистра шаблона
```

### head

```
$ head ./ log — вывод первых 10 строк файла
```

\$ head -3 ./log — вывод первых 3 строк файла (или \$ head -n 3 — рекомендуется так)

### **tail**

\$ tail ./log — последние 10 строк файла

\$ tail -n 3 ./log

**важный ключ -f** — продолжит в режиме реального времени выводить строки, если они будут добавляться

\$ tail -f ./log

### **more**

\$ more file — просмотр файла (можно пролистывать файлы)

### **less**

\$ less file — просмотр файла (можно пролистывать стрелочками)

если нажать / — откроется поиск по файлу

## **2.9 Потоки**

\$ ls -i file1 2> file2 — перенаправим поток ошибок в файл file2

\$ ls -li file1 2> file2 1>file3 — ещё и поток вывода (1) перенаправим в file3 одной командой

При этом >> добавляет информацию в файл, а > перезаписывает файл целиком

**Команды в *bash* выполняются справа налево**

\$ tail -n 50 log > log — в log ничего не будет, так как сначала будет проверено, что файл существует, затем из-за > он будет обнулён. И только потом будет вывод файла

**Pipe (Конвейер) |**

`$ ls -li file1 | grep ^-` (применяем утилиту и перенаправляем её вывод в утилиту grep, которая выводит только те файлы, которые начинаются (из за галочки) с символа -)

Чтобы вывести информацию и на экран и в стандартный поток вывода

`$ ls -l | tee file`

`$ ls -l | tee -a file` — чтобы не перезаписывать, а добавит информацию в файл

Альтернатива этому — перенаправлять вывод команды в файл, а в другом окне терминала читать этот файл с командой `$ tail -f`

`$ ls -l file 1>file2 2 > &1` — поток 2 направляется туда же, куда и поток 1  
`&` — адрес чего-либо

Всё, что направлено в поток `/dev/null` — уходит в «Чёрную дыру»

## 2.10 Условия

*Интересный факт:* в bash *True* — это 0 (а не 1), а не 0 — *False*

`$ echo $?` — (`$` — обозначение переменной, а `?` — код возврата последней команды). То есть таким образом мы можем посмотреть код возврата последней команды

`&&` — оператор И

`||` — оператор ИЛИ

`;` — оператор НЕ ИМЕЕТ ЗНАЧЕНИЯ

`command1 ; command2` — `command2` будет выполнена вне зависимости от кода возврата `command1`

## 2.11 Диски и монтирование

Диски находятся в специальном каталоге `/dev`

`sda, sdb` и тд — это физические диски

`sda1, sdb2` — (с цифрами) это логические диски

**VFS Virtual File System** — программный интерфейс между ядром и драйвером конкретной файловой системы

Монтирование — связь VFS с реальной файловой системой

Какой диск куда смонтирован можно посмотреть через утилиту `df -h` (-h чтобы в GiB был размер)

Для монтирования используются команды `mount` и `umount`

### 3 Полезные bash-команды

Формат:

**команда** *ключи аргументы*

`$ man cmd` — более подробная документация, чем в `--help`

`$ man -k word` — ищёт в документации ключевое слово *word*

`$ uname` — выводит информацию о версии ядра

`$ date` — выводит текущую дату и время

`$ ls -l` — более подробный `ls`

(Если в первой колонке вывода `$ ls -l` стоит `-`, то это файл. А если `d` — то директория)

`$ ls -la` (или `-l -a`) — посмотреть скрытые файлы (имя начинаются с `.`)

`$ ls -la ..` — содержимое родительского каталога

`$ comand --help` — справочная информация

`$ touch existing_file` — изменить время создания файла

`$ mkdir -p dir1/dir2/dir3` — рекурсивное создание директорий

Двойное нажатие TAB выдаст список возможных дополнений. Кроме того, если дополнение единственное, будет дополнено автоматически

`$ cd` — переводит в домашнюю директорию пользователя (аналог `cd ~`).

Конструкция `$ cd alex` эквивалентна `$ cd ./alex`

## Маски для файлов («Регулярные выражения»)

\* — любой набор любых символов

? — один любой символ

\$ rm \*2 — (всё, что оканчивается на 2)

\$ rm file\* — (всё, что начинается с file)

\$ rm \*.pdf — (все pdf файлы)

\$ rm garbadge.\*

\$ rmdir *dir1* — удаление **пустой** директории

\$ cp -r *dir1* *dir2* — рекурсивное копирование директории и всего её содержимого  
(из директории *dir1* в *dir2*)

При этом

\$ mv *dir1* *dir2* — перемещение директории (работает без ключей)

\$ type *command* — выводит сведения о команде (внутренняя — принадлежит ОС, или внешняя — просто исполняемый файл (в Linux большинство команд именно такие))

### Пример

\$ type cd

cd is a shell builtin — команда встроена в оболочку

\$ type cp

cp is /bin/cp — просто исполняемый файл (как и mv, rm . . . )

\$ type ls

ls is aliased to ‘ls –color=auto’

\$ which *cmd* — показывает путь к бинарному файлу *cmd*

### Пример

\$ which ls

```
/bin/ls
```

\$ who — кто сейчас работает на этом сервере, к какому терминалу он подключен, когда он подключился и с какого адреса

### Пример

```
$ who
```

```
ubuntu pts/0 2023-07-26 09:53 (192.168.64.1)
```

```
ubuntu pts/1 2023-07-26 09:55 (192.168.64.1)
```

```
$ (MacOS) who
```

```
alexey console 24 июл 08:14
```

```
alexey ttys002 26 июл 09:51
```

\$ id *user* — информация о пользователе

\$ chmod u-w — убирает права на запись для пользователя

(u — user, g — group, o — others, a — all, — — убрать, + — добавить)

\$ chmod u+rwx, g-x+rw, o-rwx file

% chmod a+rwx file — дать всем полные права

\$ chmod +x file — сделать исполняемым для всех (эквивалентно chmod a+x file)

Внимание: если изменить права для директории, они не изменятся для содержимого директории для этого нужно использовать -R (R — заглавная!)

Информация о дисках

```
$ df или $ df -i
```

## 3.1 Пример сборки кода из исходников

```
$ git clone https://github.com/the-tcpdump-group/tcpdump.git
```

```
$ cd tcpdump
```

```
$ ./configure
```

Wants	Модули, которые должны быть активированы одновременно, если это возможно, но не обязательно
Requires	Строгие зависимости; отказ от каких-либо предварительных условий прекращает работу этой службы
Requisite	Аналогично <i>Requires</i> , но модуль должен быть активным
BindsTo	Аналогично <i>Requires</i> , но модуль должен быть связан еще более тесно
PartOf	Аналогично <i>Requires</i> , но влияет только на запуск и остановку
Conflicts	Отрицательные зависимости; не может взаимодействовать с этими единицами

Таблица 2: Явные зависимости

\$ make

\$ sudo make install

## 3.2 Зависимости

# 4 Ответы на важные вопросы

### Что такое процесс в Linux?

Программа — это исполняемый файл, который может быть запущен в операционной системе, и когда программа запускается, операционная система создает **процесс**, который представляет эту работающую программу.

Каждый процесс имеет свой собственный уникальный идентификатор (PID), который используется для управления и отслеживания процессов операционной системой.

Процессы могут быть запущены как фоновые задачи, которые работают в фоновом режиме без взаимодействия с пользователем, или как интерактивные задачи, которые требуют ввода и вывода с помощью терминала или графического интерфейса пользователя.

Каждый процесс имеет свою собственную память и ресурсы, которые используются во время его выполнения.

## **Переменные окружения и для чего они нужны**

В контексте Linux окружение — это набор переменных окружения и их значений, которые определяют поведение и конфигурацию среды выполнения для запущенных процессов.

Переменные окружения в операционной системе (в том числе в Linux) представляют собой именованные значения, которые определяют окружение, в котором работают запущенные процессы. Эти переменные содержат информацию о системе, пользовательских настройках, путях к исполняемым файлам и другую важную информацию.

## **Некоторые из распространенных переменных окружения в Linux**

- PATH: Одна из самых важных переменных окружения. Она содержит список каталогов, в которых операционная система ищет исполняемые файлы, когда вы вызываете команду в терминале. Когда вы вводите команду в терминале, Linux просматривает все каталоги, указанные в переменной PATH, чтобы найти соответствующий исполняемый файл.
- HOME: Указывает домашний каталог текущего пользователя. Когда пользователь входит в систему, его домашний каталог становится текущим рабочим каталогом.
- USER и USERNAME: Имя текущего пользователя
- TMP или TMPDIR: Указывает каталог для временных файлов, используемых различными программами.
- SHELL: Путь к интерпретатору командной строки (shell), используемому по

умолчанию для текущего пользователя.

- PS1 и PS2: Определяют строку приглашения командной строки (prompt), которая отображается перед вводом команды (PS1) и при продолжении многострочной команды (PS2).

## Как программа ищет библиотеку в момент запуска?

На порядок поиска могут влиять переменные окружения и настройки системы.

Но в целом он такой:

1. Исходные каталоги программы: Если библиотеки, требуемые программой, находятся в том же каталоге, что и сам исполняемый файл программы, они будут использоваться в первую очередь.
2. При запуске программы операционная система заранее знает несколько стандартных каталогов, в которых могут находиться общие библиотеки. Эти пути указаны в переменной окружения LD\_LIBRARY\_PATH. Обычно стандартные пути включают /lib и /usr/lib
3. Каталоги кэширования: Современные версии Linux используют кэширование динамических библиотек для повышения производительности. Кэш содержит информацию о расположении библиотек в системе. Кэшированные данные хранятся в /etc/ld.so.cache
4. Каталоги из файла конфигурации /etc/ld.so.conf: Файл /etc/ld.so.conf содержит список дополнительных каталогов, в которых могут находиться общие библиотеки. Если есть изменения в этом файле, обычно требуется запустить команду ldconfig, чтобы обновить кэш библиотек.
5. Системные пути: Если все остальные пути не дали результатов, операционная система будет искать библиотеки в системных каталогах, таких как /lib и /usr/lib.

Как только требуемая библиотека найдена, она будет загружена в память, и программа сможет использовать функции из этой библиотеки в процессе своего выполнения.

## **Порядок запуска программ**

1. ОС создает новый процесс для этой программы
2. ОС загружает исполняемый файл программы в память нового процесса.
3. ОС начинает разрешать зависимости библиотек, ища соответствующие библиотеки, необходимые для выполнения программы.
4. Когда требуемая библиотека найдена, операционная система загружает ее в память процесса, который выполняется.
5. После разрешения всех зависимостей и загрузки библиотек, процесс становится полностью загруженным и готовым к выполнению. Операционная система передает управление программе, и она начинает свое выполнение.

## **Чем отличается файл программы от файла библиотеки?**

Файл программы (исполняемый файл): Это файл, который содержит всю информацию и код, необходимые для запуска и выполнения отдельной программы. Когда вы запускаете программу, используется этот файл, чтобы программа могла выполнять свои функции. Файл программы — это конечный результат всего процесса разработки, и он может быть запущен напрямую.

Файл библиотеки (динамическая или статическая библиотека): Это файл, который содержит функции и код, которые могут быть использованы другими программами. Библиотеки создаются для облегчения повторного использования кода и экономии памяти. Этот файл не является полноценной программой, но предоставляет некоторый функционал, который может быть подключен к другим про-

граммам. Библиотеки могут быть использованы множеством программ, чтобы они могли обмениваться кодом между собой и не дублировать его в каждой программе.

## **Какие ресурсы нужны программе при запуске и при работе?**

1. Центральный процессор (CPU): Программе требуется центральный процессор для выполнения своего кода и обработки данных. Чем более сложные операции выполняет программа, тем больше ресурсов CPU она использует.
2. Память (RAM): Когда программа запускается, ей нужно занять определенный объем оперативной памяти (RAM) для хранения своего кода, данных и временных результатов. При работе программа может активно использовать память для хранения переменных, стека вызовов и других данных.
3. Ввод/вывод (I/O) устройства: Программы могут взаимодействовать с внешними устройствами через различные каналы ввода/вывода. Например, программы могут читать и записывать данные на жесткий диск, сеть, клавиатуру, монитор и т. д.
4. Файловая система: Программам может потребоваться доступ к файлам, чтобы считывать конфигурационные данные, записывать журналы и т. д. Для этого требуются права на чтение и запись в соответствующие файлы и директории.
5. Графический интерфейс: Если программа имеет графический пользовательский интерфейс (GUI), ей понадобятся ресурсы для отображения окон, кнопок, изображений и других элементов интерфейса.
6. Системные вызовы и библиотеки: Для выполнения различных операций, таких как чтение файлов, создание сетевых соединений и другие, программа может вызывать функции из системных библиотек операционной системы.