
HIGHWAY
TO
SHELL

Highway to Shell

Быстрый, практичный заезд в командную строку Linux: меньше слов, больше сигнала.

Alexander Tarasov

This book is available at https://leanpub.com/highway_to_shell

This version was published on 2025-09-10

Contents

Предисловие

Цель книги – дать короткое, практичное введение в командную строку Linux (на примере Ubuntu 24.04, оболочка `bash`). Здесь – факты, примеры и минимум «лирики».

Как пользоваться:

- подготовьте окружение;
- держите терминал открытым и повторяйте примеры;
- `$` – команда обычного пользователя, `#` коментарий;
- если команда не работает – проверьте права, оболочку и версию утилиты.

Материалы по книге: github.com/chrns/highway_to_shell_book

Благодарности

Спасибо за обратную связь следующим господам: `@an2shka`, `@MuratovAS`, `@bobko`, `@milordevops`, `@viiri`.

Эпиграф

She sells `csh` by the sea shore. The `shs` she sells are surely `cshs`. So if she sells `shs` on the sea shore, I'm sure she sells seashore `shs`.

Введение

Когда компьютеры стали чем‑то большим, чем «считывателями» перфокарт, понадобилась операционная система, обеспечивающая удобный доступ к ресурсам: памяти, диску и т. д. Действительно, проще написать программу, обращаясь к интерфейсу доступа к ресурсам, чем встраивать драйверы непосредственно в программу. Так, в 1970‑х в стенах Bell Labs появился UNIX и более или менее та самая командная строка. Последнюю ещё называют оболочкой, так как по сути она является внешним интерфейсом операционной системы.

Командная строка — не что иное, как интерпретатор команд, позволяющий запускать программы. Чаще всего интерпретатор — это не просто прослойка между вами и ядром, но и язык программирования с переменными, циклами, условными операторами и функциями. Используя специальный синтаксис, можно писать скрипты, а в сочетании с другими программами‑утилитами они становятся незаменимым инструментом в повседневных задачах. Например, в экосистеме Arch активно используются bash‑скрипты; можно даже сказать, что Arch это ядро и набор скриптов.

> **Личное мнение автора**
>
> Несмотря на то что оболочка‑интерпретатор — это язык программирования, **это плохой язык программирования**. Для сложных задач лучше использовать подходящий инструмент, а не писать плохо читаемые портянки на `bash`.

В UNIX‑подобных системах можно менять командный интерпретатор в зависимости от предпочтений. Самые популярные сегодня — `bash`, `zsh`, `fish`, `csh/tcsh`, `dash` и другие. Системный интерпритатор, `/bin/sh`, в Ubuntu указывает на `dash`, в то время как интерактивная оболочка для пользователя по умолчанию является `bash`. В macOS, также UNIX‑подобной, по умолчанию используется `zsh`. Функциональность, а также синтаксис оболочек часто различаются до несовместимости: `dash` ближе к POSIX, а `bash` и `zsh` — нет. Зато в последних двух есть

автодополнение по клавише Tab. В `zsh`, в отличие от `bash`, можно подключать плагины. У `csh` – Сипподобный синтаксис; у остальных – нет.

Графический интерфейс (GUI) удобен, но не так гибок и функционален, как командная строка. Просматривать вебпстраницы комфортнее в графическом браузере (хотя существуют и консольные), но для автоматизации и многих других задач командная строка – незаменимый инструмент.

Командная строка стала удобной во многом благодаря философии UNIX – набору культурных норм и подходов к разработке ПО. Кратко и упрощённо: (а) лучше писать маленькие программы, которые выполняют одну задачу и делают это максимально хорошо; (б) следует использовать текстовый ввод/вывод и делать так, чтобы входом одной программы мог быть вывод другой.

Команд (программ, скриптов) и способов их использования (флагов и параметров) очень много. Мы не рассмотрим их всех и не покроем и нескольких процентов возможностей. Дистрибутивы по разным причинам содержат разный набор утилит, их версий и реализаций: изпза лицензий или стремления к легковесности (см. Alpine Linux). Поэтому пытаться исчерпывающе описать каждую программу бессмысленно. Если чтопто не работает – RTFM.

> К слову, в книге мы будем использовать слова «команда», «программа» и «утилита». В нашем контексте они взаимозаменяемы.

Многие задачи уже решены до вас. Прежде чем изобретать велосипед, проверьте: скорее всего, вашу задачу можно решить стандартными средствами или установкой готовой утилиты. Если решения нет – пишите свою программу и решайте только свою задачу: большую часть рутины, скорее всего, можно переложить на уже имеющиеся инструменты.

> **Примечание**
>
> Последовательность изложения будет слегка нарушена: в первых двух главах встретятся команды до их формального обзора. Это сделано в целях повествования.

Подготовка окружения

Не запускайте на своей машине скрипты из непроверенных источников – это небезопасно. Если вас кто-то убеждает, что это не так, – не верьте. Теперь тот самый прыжок веры:

```
bash <(wget -q -O -
↪  "https://raw.githubusercontent.com/chrns/highway_to_shell_book/refs/heads/
main/materials/gen_data.sh")
```

Команда создаст в домашнем каталоге директорию `highway` и необходимые подкаталоги. Все примеры в книге предполагают, что вы находитесь в `~/highway`.

Про Linux

Наиболее удачная (оценочное суждение) реализация концепции командной строки получилась в UNIX/Linux – недаром мы обсуждаем `bash` в рамках одного из дистрибутивов, Ubuntu. Многие разработчики (и не только) используют именно её (чаще всего `bash`) в повседневной работе – даже под Windows, будь то установленный терминал с утилитой `git` или WSL (Windows Subsystem for Linux). Чтобы последующая информация имела смысл, коротко обсудим особенности самого Linux.

Всё есть файл

В UNIX-подобных системах «файл» – это не просто текстовые или бинарные данные; для упрощения дизайна системы принята парадигма *Everything is a file*. Видеокарта – файл. Звуковая карта – файл. Терминал – это тоже файл.

> Несмотря на то, что «всё есть файл», в Linux есть исключения: потоки, сетевые пакеты, внутренние структуры ядра (планировщик, буферы), сигналы, адреса памяти, прерывания и некоторые другие вещи. Их иногда сложно, а иногда нерационально (из соображений производительности) представлять в виде файла. Тем не менее, с большинством из перечисленного выше вам, вероятно, работать не придётся, поэтому можно принять допущение, что «всё есть файл».

Принимая такую философию, заметно упрощается дизайн системы. Не нужно выдумывать сложные конструкции и концепции для описания разнообразных объектов. Любое устройство, как и файл, можно «открыть» (активировать), «записать» (отправить данные), «прочитать» (получить данные), а по завершении – «закрыть» (деактивировать). Если речь о потоках данных, почему нельзя видеокарту представить как файл? Команды `write`, `read`, `open`, `close` называются системными вызовами.

Файлы можно разделить на категории:

- обычные файлы (текстовые или бинарные);
- директории (имеют метаданные: дата создания, модификации, имя);
- файлы устройств (блочные устройства – жёсткий диск `/dev/sda`; или символьные, *character devices*, `/dev/ttyS0`);
- конвейеры и FIFO;
- сетевые сокеты;
- символические ссылки;
- специальные файлы (`/dev/null`, `/dev/random`, `/proc/*`).

Каждый открытый файл, устройство или ресурс описывается дескриптором (число, назначаемое в рамках сессии). Первые три зарезервированы под стандартные потоки – поговорим о них позже.

> Количество файлов, которое может открыть ОС одновременно, ограничено и контролируется параметром, описанным в `/proc/sys/fs/file-max`. Обычно это большое число, например `9223372036854775807`. При этом каждый процесс может открыть, как правило, не более `1024` файлов. Узнать текущее ограничение можно командой:

```
$ ulimit -n
1024
```

> Временно увеличить лимит можно, добавив в конце команды через пробел требуемое число. Также есть понятия «мягкого» и «жёсткого» лимитов, но останавливаться на них не будем.

Если вы программировали на C, то уже слышали про дескрипторы:

```
int fd = open("file.txt", O_RDONLY);
```

Функция `open` (обёртка над системным вызовом) возвращает число, назначенное операционной системой. Используя этот дескриптор, вы можете записывать в файл и читать из него. При закрытии дескриптор будет «освобождён» операционной системой.

> При аварийном завершении программы ОС самостоятельно закроет все ассоциированные с умершим процессом файлы и освободит дескрипторы. Тем не менее лучше не полагаться на такое поведение и обрабатывать сигналы от ОС (см. graceful termination/exit).

Файловая система

Файлы — это данные, хранящиеся на диске. Каким образом их хранить отвечает файловая система. Linux поддерживает широкий спектр ФС, со своими плюсами и минусами. Не вдаваясь в подробности, перечислим самые популярные: ext2, ext3, ext4, Btrfs, ZFS, XFS.

Файловая система накладывает ограничения на операции с файлами. Во-первых, от ФС зависит максимальная длина имени файла. Как правило, это **255 байт** (символ в UTF-8 может занимать от 1 до 4 байт). Во-вторых, ограничивается максимальная длина абсолютного пути — обычно **4096 байт**.

Метаданные файлов и директорий хранятся (в большинстве ФС, таких как ext4, ZFS и Btrfs) в отдельных структурах файловой системы, называемых inode (*index node*). В метаданных содержатся информация о размере и типе файла, разрешения доступа, данные о владельце, временные метки создания, модификации и последнего доступа и т. д.

Эти служебные структуры не доступны напрямую из файлового менеджера; с ними работает сама файловая система. Но вы можете получать и изменять некоторые атрибуты через утилиты `lsattr` и `chattr`.

> В macOS есть удобная функция, интегрированная в файловый менеджер, — *tags*. Они как раз реализованы на уровне файловой системы. Можно помечать файлы и директории метками и мгновенно находить их без обхода дерева каталогов.

Кроме стандартных атрибутов есть расширенные — `xattr`. Их поддерживают не все файловые системы, но, например, они доступны в ext4. С утилитами `setfattr` и `getfattr` можно создавать, модифицировать и удалять пользовательские метаданные. Пример:

```
$ setfattr -n user.tag -v "important" README.md
$ getfattr -n user.tag README.md
important
```

Графический файловый менеджер не умеет работать с пользовательскими метаданными, поэтому получить такой же удобный пользовательский опыт, как в macOS, не получится. Однако собственные

метки могут быть полезны для других задач. Например, `rsync` использует `xattr` для хранения статуса бэкапа и некоторых служебных данных.

Структура директорий

В Linux есть понятие корневой файловой системы — `/`. Все ваши файлы лежат относительно этого пути. Даже если у вас несколько дисков, путь всё равно идёт через корень (расположенный на одном из дисков). Наиболее важные директории корневой системы:

- `/bin/` — основные пользовательские программы, например `bash`;
- `/etc/` — конфигурационные файлы системы;

  > В оригинале `etc` происходит от латинского *etcetera*, но в простонародье его расшифровывают как **E**ditable **T**ext **C**onfiguration.

- `/sbin/` — основные системные программы (работа с дисками, перезапуск и выключение компьютера);
- `/usr/` — пользовательские приложения, а также документация для программ;

  > Несмотря на то, что `usr` напоминает слово *user*, на самом деле это сокращение от **U**nix **S**ystem **R**esources. Исторически внутри есть поддиректории `/usr/bin` и `/usr/sbin`.

- `/var/` — временные/часто изменяемые файлы;
- `/dev/` — файлы устройств;
- `/home/` — домашние директории пользователей;
- `/lib/` — библиотеки и модули ядра;
- `/mnt/` — точки монтирования файловых систем;
- `/proc/` — процессы и файлы информации ядра.

  > Депфакто это отдельная файловая система — `procfs` — для представления структур ядра.

Структура директорий описывается спецификацией FHS (*Filesystem Hierarchy Standard*). Её можно прочитать, набрав `man hier`.

Пользователи и группы

Linux — многопользовательская операционная система. Пользователи могут работать за одной машиной одновременно, даже если создан только один аккаунт.

> Узнать имя текущего пользователя можно, выполнив команду `whoami`.

У всех пользователей есть идентификатор — UID (*User ID*). Самый главный пользователь — `root`; его UID равен `0`. Другие системные пользователи, которые используются для разных задач (например, `www-data` для вебпсервера), имеют UID меньше `1000`. Остальные, обычные пользователи, как правило, имеют UID начиная с `1000`.

Пользователь `root` обладает максимальными привилегиями и может получать доступ к любому файлу. Обычно от его имени никто не работает, так как это небезопасно, поэтому используется утилита, позволяющая повысить привилегии до уровня `root`. В Ubuntu это `sudo` (*superuser do*). Полностью переключиться на `root` можно командой `sudo su`.

> `sudo` — наиболее распространённая утилита, но есть и альтернативы, например `doas`.

Упоминавшийся файл `/etc/passwd` содержит строки формата:

```
1  username:x:UID:GID:comment:home_directory:shell
```

`shell` — это оболочка, `username` — имя пользователя, `home_directory` — каталог, где у пользователя по умолчанию максимальные права. Для обычных пользователей (UID ≥ 1000) эта директория обычно находится в `/home/username`. Про `GID` мы ещё поговорим.

> К слову, все пароли хранятся в виде хэша, а не в открытом виде. Они находятся в файле `/etc/shadow`:

```
username:encrypted_passwd:last_passwd_change:min:max:warn:expire:disable
```

> У обычного пользователя нет права читать этот файл; работать с ним может только `root`.

Прописывать права на каждый файл для каждого пользователя — не самое элегантное решение. Проще ввести сущность «группа» и выдавать разрешения на файлы и директории через неё (в том числе).

Каждый пользователь имеет так называемую «основную группу». Для вашего пользователя она будет называться так же, как `username`; её GID указан в `/etc/passwd`. Получить название основной группы можно командой `id` с флагами `-gn`:

```
$ id -gn <username>
<username>
```

Кроме основной, у пользователя может быть несколько дополнительных групп. Получить полный список можно той же командой без флагов:

```
$ id <username>
```

Каждый пользователь может состоять в нескольких группах, но это число конечно (см. код ядра, макрос `NGROUPS_MAX`). Группы и их члены описываются в файле `/etc/group`:

```
group_name:x:GID:user1,user2
```

> В принципе, ничто не мешает добавить своего пользователя в группу `root`, но это небезопасно.

Управление пользователями

Для управления пользователями нужны три (четыре) команды: `useradd`, `usermod`, `userdel` — создание, модификация и удаление соответственно, и `passwd` — для управления паролями.

При добавлении нового пользователя создаётся его домашняя директория по шаблону из `/etc/skel`.

Примеры:

```
# Create a new user
$ sudo useradd -m <username> [-s <shell>] [-d </custom/home/dir>]
# Set a password for the user
$ sudo passwd <username>
# Set a specific UID for a user
$ sudo useradd -u <UID> <username>
# Delete user (without home directory; add -r to delete the directory)
$ sudo userdel <username>
# Change primary group
$ sudo usermod -g <groupname> <username>
# Add to auxiliary group; remove in the same way, but without 'a'
$ sudo usermod -aG <groupname> <username>
# Block and unblock a user
$ sudo usermod -L <username>
$ sudo usermod -U <username>
```

Кто может использовать `sudo`?

Пользоваться `sudo` могут не все. В системе есть список пользователей, которым разрешено повышать привилегии до уровня `root`. Вряд ли вы хотели бы, чтобы гость имел такие же права, как вы. Во-первых, уберите пользователя из группы `sudo`; во-вторых, с помощью `visudo` (редактирует `/etc/sudoers`) исключите его из списка. Можно не забирать полный доступ, а ограничить исполнение отдельных утилит и программ.

Управление группами

Для управления группами также существуют три команды: `groupadd`, `groupmod`, `groupdel`. Наиболее частые случаи применения:

```
# Create a new group
$ sudo groupadd [-g <GID>] <groupname>
# Delete a group
$ sudo groupdel <groupname>
# Rename a group
$ sudo groupmod -n <newname> <oldname>
```

Через `groupmod` можно сменить GID группы, но имейте в виду: все созданные ранее файлы будут иметь старый GID. Поменять можно так:

```
$ sudo find / -group OLD_GID -exec chgrp NEW_GID {} \;
```

Список всех групп хранится в `/etc/group`.

```
# Alternative way to view user groups
$ groups <username>
```

Одна из системных групп – `dialout`: в неё входят последовательные порты `/dev/ttyS*`, `/dev/ttyUSB*`, `/dev/ttyACM*`. Поскольку ваш пользователь в неё не входит, при работе с этими файлами требуются привилегии `sudo`. Вы можете добавить своего пользователя в эту группу:

```
$ sudo usermod -aG dialout $USER
```

Права на файлы и директории

Разрешения определяют права действий с файлами и директориями. Базовых прав три: чтение (`r`), запись (`w`) и исполнение (`x`).

- **Чтение** позволяет просматривать содержимое.
- **Запись** позволяет изменять содержимое. Для директории – добавлять и удалять файлы в ней.
- **Исполнение** позволяет запускать файл как программу/скрипт. Для директории – входить в неё.

Типичный вывод `ls`:

```
drwx------  3 oliver oliver   4096 Nov 30 21:17 .local
-rw-r--r--  1 oliver oliver    807 Mar 31  2024 .profile
```

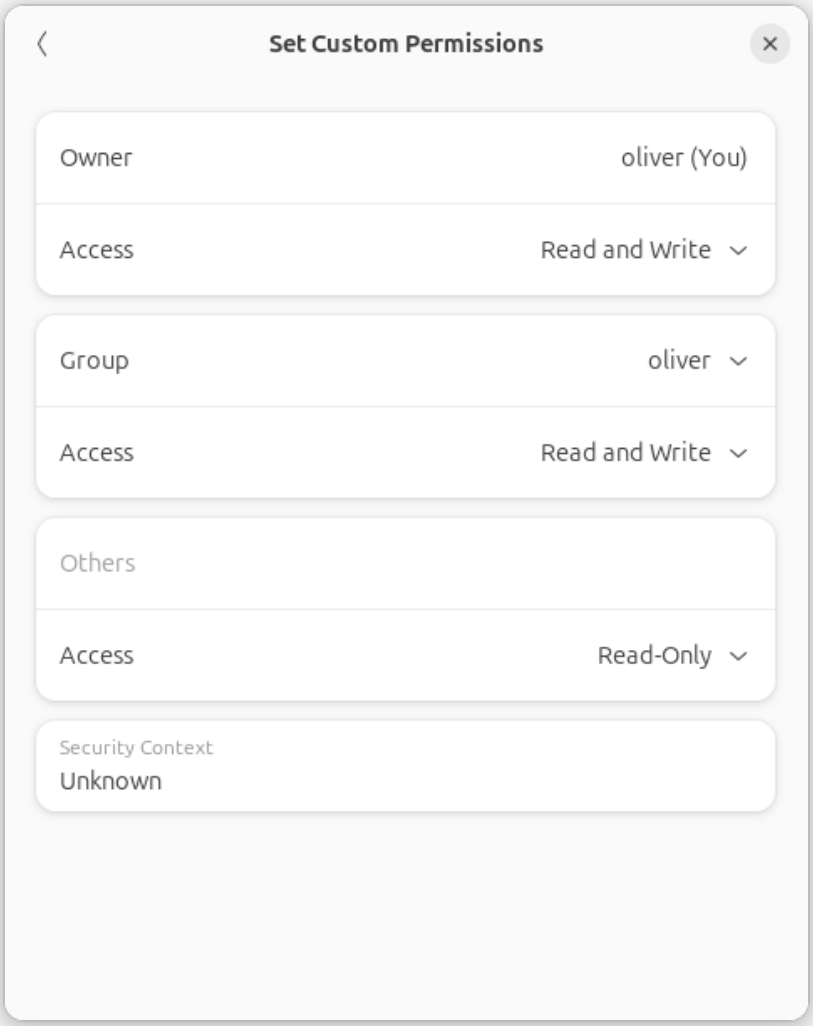
Первый столбец – строка из 10 символов. Первый символ обозначает тип: для директорий – `d`, для обычных файлов – `-`; блочные устройства – `b`, символьные – `c`, именованные каналы – `p`, символические ссылки – `l`. Далее идут три группы по три символа (`rwx`), соответствующие правам пользователя, группы и остальных. Если установлена буква – право выдано; если `-` – права нет.

> Существует и более продвинутая система контроля доступа – SELinux.

В примере у пользователя есть полные права на директорию `.local`, тогда как все остальные (кроме `root`) не могут даже просмотреть её содержимое; члены основной группы пользователя – тоже. `.profile` – обычный текстовый файл; его нет смысла выполнять, поэтому права на исполнение отсутствуют (их можно добавить). Все остальные могут лишь читать содержимое.

> Файлы, начинающиеся с точки, считаются скрытыми.

У каждого объекта есть пользователь-владелец и группа-владелец. Это `oliver oliver` в выводе `ls` в примере выше.



Управлять правами можно через три команды: `chown` (изменение пользователя), `chgrp` (изменение группы; можно и через `chown`) и `chmod` (изменение прав).

```
$ sudo chown <user>:<groupname> <filename/dirname>
$ sudo chgrp <groupname> <filename/dirname>
# same, but using chown
$ sudo chown :<groupname> <filename/dirname>
```

Есть два способа задания прав через `chmod` — символьный и числовой. Синтаксис:

```
chmod [reference][operator][mode] <filename/dirname>
```

`reference` — чьи права менять: `u` — пользователь; `g` — группа; `o` — остальные; `a` — все. `operator`: `+` разрешить, `-` запретить, `=` скопировать. `mode` — комбинация `r`, `w`, `x`.

Примеры:

```
# Prevent all other users from doing anything with a file
$ chmod o-rwx README.md
# The group can write, everyone else can only execute
$ chmod g+w,o-rw,a+x README.md
# Copy user rights to group rights
$ chmod g=u README.md
# Add the ability to execute the file for everyone
$ chmod +x script.sh
```

Права описываются тремя битами (`rwx`), поэтому их удобно задавать числом. Например, `r--` — это `0b100`, то есть `4` в восьмеричной записи. Чтобы не писать много букв:

```
# 400 = r-- --- ---
$ chmod 400 README.md
```

> Ключи в директории `~/.ssh` имеют как раз значение `400`. Приватные ключи должны быть приватными.

Если речь о директории, чаще всего нужно менять права и внутри неё. Используйте ключ `-R`.

Наиболее частые комбинации:

- `777` (`rwxrwxrwx`) — всем можно всё;
- `755` (`rwxr-xr-x`) — пользователю можно всё, остальные могут читать и выполнять;
- `666` (`rw-rw-rw-`) — все могут читать и писать;
- `640` (`rw-r-----`) — пользователь может читать и писать, его группа — только читать.

Попробуйте самостоятельно составить такие числа.

> Права на файлы хранятся в служебных структурах файловой системы. Если ФС не поддерживает POSIXправа (например, NTFS), то реализовать их невозможно. При подключении диска NTFS все файлы и директории будут отображаться как `777`.

При изменении прав в своей домашней директории `chmod` можно вызывать без `sudo`. Будьте внимательны: можно случайно делегировать права другому пользователю/группе и лишиться доступа от имени своего пользователя.

> При создании файла или директории присваиваются максимальные права за вычетом значения `umask` (обычно `022`). Для файлов максимальные права — `666`, для директорий — `777`. При `umask=022` файлы будут иметь `644` (`rw-r--r--`), директории — `755` (`rwxr-xr-x`). Значение `umask` часто задают в `.profile`.

Специальные права

Кроме обычных прав (`rwx`) есть специальные — *setuid*, *setgid* и *sticky bit* — полезные в ряде случаев.

setuid и *setgid* полезны, когда нужно предоставить доступ к выполнению команды с правами владельца. Например, команда `passwd` позволяет поменять пароль текущего пользователя без `sudo`.

> Команда `passwd` может менять пароль любого пользователя, но при запуске не от `root` она проверяет, от чьего имени запущен процесс (UID). При попытке сменить пароль пользователя, отличного от текущего, команда выдаст ошибку.

```
$ passwd root
passwd: You may not view or modify password information for root.
```

Такое поведение обеспечивает именно *setuid*. Обратите внимание на вывод:

```
$ ls -la /usr/bin | grep -w "passwd"
-rwsr-xr-x  1 root root        64152 May 30  2024 passwd
```

Вместо привычной записи `rwx` для пользователя указано `rws`. Аналогично работает *setgid*. К слову, применив *setgid* к каталогу, все новые файлы будут наследовать группу этого каталога.

```
# Grant setuid to user
$ chmod u+s script.sh
# In octal notation you need to add the number 4 at the beginning
$ chmod 4755 script.sh
# The same for GID: instead of 4 - 2, i.e. 2755
$ chmod g+s script.sh
```

> Будьте осторожны. Если программа, помеченная `setuid`, уязвима (переполнение буфера, инъекции команд, гонка состояний), злоумышленник может получить контроль над системой. Для повышения безопасности сценария можно запускать `bash` с ключом `--` — обработчик не примет дополнительные параметры.
>
> Найти все файлы с таким атрибутом:

```
$ find / -perm -4000 -type f 2>/dev/null
```

Sticky bit полезен, когда права выданы на директорию, но нужно ограничить возможность другим пользователям изменять или удалять чужие файлы. Пример:

```
$ chmod +t ./somedir
$ ls -ld ./somedir
drwxrwxr-t  2 oliver oliver 4096 Aug 15 22:35 somedir
```

Ссылки

В Linux есть специальный тип файлов, действующий как указатель на другой файл или директорию, – ссылка (*link*). Такой файл хранит путь до цели, а не её содержимое. Когда пользователь обращается к объекту через ссылку, ОС перенаправляет его в нужное место.

Существует два типа ссылок: *soft link* (symbolic) и *hard link*. Первая (символическая) может указывать на директорию и на объект в другой ФС. Если оригинальный файл удалён, символическая ссылка становится «битой» (*dangling link*). Жёсткая ссылка указывает на тот же `inode`, что и исходный файл; её нельзя создать на директорию и на объект в другой ФС.

Символьные ссылки удобны, когда в системе есть несколько версий программ, и нужно переключаться между ними. Допустим, у вас установлен `gcc` 13ой версии, но для работы нужна 11ая. В `/usr/bin/` уже лежит символьная ссылка (префикс `l` в выводе `ls`):

```
$ ls -la /usr/bin/gcc | grep -w "gcc"
lrwxrwxrwx 1 root root 6 Jan 31  2024 /usr/bin/gcc -> gcc-13
```

Останется поменять ссылку на `gcc-11`. Другой пример: вы скачали старую версию компилятора для ARM, которого нет в репозитории вашего дистрибутива. Распакуйте его в `/opt/`, затем создайте символьную ссылку и положите её в `/usr/bin/`:

```
# ln -s [target] [symlink name]
$ ln -s /opt/old_uc_compiler /usr/bin/uc_compiler
```

Чтобы выяснить, куда указывает ссылка, используйте `readlink`:

```
$ readlink /usr/bin/gcc
gcc-13
$ which gcc-13
/usr/bin/gcc-13
```

Ничто не мешает создать ссылку на саму ссылку:

```
$ ln -s symlink_name symlink_name
```

Это *circular symlink*. Он может увести некоторые утилиты в бесконечный цикл. Не делайте так.

Жёсткая ссылка указывает на тот же `inode`, что и оригинал. В отличие от мягкой, она не может указывать на директорию и не может ссылаться на другой диск (например, на что-то из `/mnt`). Если оригинальный файл удалить, жёсткая ссылка всё равно будет работать, так как указывает на тот же `inode`.

```
$ ln target_filename hardlink_filename
```

Проверить, используют ли файлы одну и ту же `inode`, можно через `ls`:

```
$ touch orig_file.txt another_file.txt
$ ln orig_file.txt hardlink_file.txt
$ ls -i orig_file.txt hardlink_file.txt
11537876 hardlink_file.txt  11537876 orig_file.txt
$ ls -i hardlink_file.txt another_file.txt
11537877 another_file.txt  11537876 hardlink_file.txt
```

Также можно посмотреть количество жёстких ссылок (вторая колонка):

```
$ ls -l orig_file.txt
-rw-rw-r-- 2 user user 0 Jan 18 16:15 orig_file.txt
$ ls -l another_file.txt
-rw-rw-r-- 1 user user 0 Jan 18 16:15 another_file.txt
```

Видно, что у `another_file.txt` одна жёсткая ссылка. Другими словами, любой файл в Linux и так является жёсткой ссылкой (как минимум одной).

Терминал

В Linux часто встречается аббревиатура TTY – **T**ele**TY**pewriter – отсылка к механическому устройству для коммуникации. Он нужен для общения пользователя и системы и может быть как физическим, так и виртуальным.

Linux создаёт (как файлы) множество TTY – виртуальные и физические (часто заглушки). Аппаратные системные терминалы имеют суффикс `S` на конце, например `ttyS1` (`COM1` в Windows). В Raspberry Pi порт UART, выведенный на внешние ножки, называется `serial0` (настройки системы). Другие физические порты, например устройства, подключённые через USB, определяются как `/dev/ttyUSBx` или `/dev/ttyACMx`. Аппаратные терминалы могут быть связаны с системой (можно получить доступ к ОС через UART) – зависит от настроек. Впрочем, нам интереснее виртуальные.

> Знак доллара `$` используется для обозначения готовности принять команду от текущего пользователя. Знак решётки `#` – когда команда выполняется от имени суперпользователя (об этом позже). Здесь и далее строка без значка означает вывод команды.
>
> Когда показывается синтаксис вызова, в квадратных скобках указывается необязательный аргумент/флаг `[optional]`, в угловых – обязательный `<mandatory>`.

При загрузке Linux запускает оболочку (в нашем случае `bash`) и связывает её с одним из TTY. Чтобы понять, через какой TTY вы работаете, выполните команду `tty`:

```
$ tty
/dev/pts/0
```

В Linux можно переключаться между виртуальными **текстовыми** терминалами сочетаниями клавиш `Ctrl` + `Alt` + `Fn`. Первый (`F1`) часто зарезервирован под графический интерфейс. Терминал, который вы открываете в GUI, называется псевдотерминалом: на самом деле это эмулятор терминала (PTY). Эмуляторы бывают разные. По умолчанию в Ubuntu – `gnome-terminal` (поэтому в выводе выше `pts`, а не `tty` –

подробнее позже), в Kubuntu — `konsole` (с KDE). Никто не ограничивает вас в выборе: например, Termit позволяет из коробки создавать несколько псевдотерминалов в одном окне — удобно, если вы следите за какимпнибудь процессом.

> Выполнив команду `who`, можно увидеть всех вошедших в систему пользователей. Каждый открытый терминал обозначается как отдельная сущность, например:

```
$ who
user            console      Jan 11 09:19
user            ttys000      Jan 17 23:00
user            ttys001      Jan 18 08:14
```

UART-порт

Допустим, вы подключили USBптопUART преобразователь к машине. Вероятно, вы потянетесь за программой вроде `minicom` или `GTKTerm`. Но помните: всё — файл.

Если попытаться работать напрямую без настроек, ничего не выйдет. Сначала задайте параметры связи: скорость, размер, количество стоппбитов и наличие/отсутствие бита чётности. Сделать это можно через `stty`:

```
$ stty -F /dev/ttyUSB0 115200 cs8 -cstopb -parenb
# Check that the parameters are accepted
$ stty -F /dev/ttyUSB0 -a
```

Получить данные легко:

```
# Ctrl+C — to stop reading
$ cat /dev/ttyUSB0
```

Отправить ещё проще:

```
$ echo "Heinrich Hertz" > /dev/ttyUSB0
```

Типы команд

Есть разные типы команд. Есть встроенные, т.е. (реализованные самой оболочкой). Оболочка также использует зарезервированные слова для реализации действий (циклы, условные операторы). Другие команды – это внешние программы, которые запускаются так же, как и встроенные. Поверх всего есть псевдонимы `alias` (встроенная команда), позволяющие длинной команде дать короткое имя. Узнать, с чем вы имеете дело, поможет команда `type`:

```
$ type type
type is a shell builtin
$ type rm
rm is /usr/bin/rm
$ type select
select is a shell keyword
$ type ls
ls is an alias for ls --color=tty
```

Команда `alias` помогает сократить длинную команду. Чтобы перестать использовать псевдоним, воспользуйтесь `unalias`.

> Некоторые команды имеют и встроенные, и внешние реализации:
>
> ```
> $ type -a echo
> echo is a shell builtin
> echo is /usr/bin/echo
> echo is /bin/echo
> ```

Если команда внешняя, путь до неё можно найти утилитой `which` (поиск идёт по директориям из `PATH`):

```
$ which ls
/usr/bin/ls
```

Другая команда — `whereis` — покажет не только исполняемый файл, но и исходные коды, а также `man`страницы. Поиск происходит не только по `PATH`, но и по системным директориям (например, `/usr/share/man`).

> Некоторые команды реализованы через скрипты. Например, в Ubuntu наряду с бинарной программой `userdel` есть скриптовая `deluser`:

```
$ file /usr/sbin/userdel
/usr/sbin/userdel: ELF 64-bit LSB pie executable, x86-64, version 1
↪  (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
↪  BuildID[sha1]=b3e6378d1a0d34675134e28002df84a31c1e734d, for GNU/Linux
↪  3.2.0, stripped
$ file /usr/sbin/deluser
/usr/sbin/deluser: Perl script text executable
```

> Команда `file` полезна, когда нужно узнать тип файла: она не смотрит на расширение, а анализирует содержимое, проверяя «магические» последовательности байт (см. `/usr/share/file/magic`). Если это символьная ссылка, `file` покажет, куда она ведёт.

Процессы и код возврата

Все программы при завершении (штатном или аварийном) возвращают оболочке код возврата. Если вы писали программы на C, это тот самый `return` в функции `main`:

```
int main() {
    return 0;
}
```

Код ошибки — число. Если программа завершилась штатно, следует возвращать `0`, в противном случае — любое ненулевое значение. За некоторыми числами закреплён смысл согласно POSIX. Не уверены, что вернуть? Возвращайте `1` — ошибка общего характера.

Сессию терминала можно завершить командой `exit`. Команда принимает аргумент – код возврата. Если его не указать, будет использован код последней выполненной команды.

```
$ exit 0
```

Последний код возврата хранится в `$?`:

```
$ echo $?
0
```

Все процессы в Linux описываются числом – PID (*Process ID*). Узнать PID текущей оболочки:

```
$ echo $$
```

Некоторые команды имеет смысл запускать в фоновом режиме, чтобы они не блокировали ввод других команд. Узнать PID последней фоновой команды:

```
$ echo $!
```

Подробнее о работе с процессами и запуске программ в фоне – в следующей главе.

Флаги и аргументы

Иллюстрации ради мы уже выполнили несколько команд в строке, и часть из них состоит не из одного «слова». Обобщённый вид:

```
$ command [flag/option] [argument(s)]
```

Всё, что идёт после имени команды, – параметры, задающие поведение программы. В UNIX/Linux есть соглашения о формате параметров. Не все утилиты им следуют – их пишут разные люди с разными соображениями.

- `[flag/option]` — задаёт поведение программы; обычно имеет префикс `-` или `--`;
- `[arguments]` — входные значения (имя файла, директории или другие данные).

Флаги за `-` во-первых, состоят из одного символа, во-вторых, их можно комбинировать (например, `-xfz`).

```
$ ls -l -a -h
$ ls -lah
```

> Оболочка запоминает аргументы последней программы и сохраняет их в `_`:
>
> ```
> $ echo Morse
> Morse
> $ echo $_
> Morse
> ```

Длинная форма (начинается с `--`) не комбинируется. Две команды ниже эквивалентны:

```
$ gcc main.c -o main
$ gcc main.c --output main
```

Часто встречаются оба синтаксиса:

```
$ command --flag argument
$ command --flag=argument
```

Если команда незнакома, прочитайте справку (`man`), если она есть, или воспользуйтесь `-h/--help`, чтобы понять, как с утилитой работать.

Страницы `man`

Системные утилиты (и не только они), а также некоторые системные файлы имеют документацию — *manual pages* (man pages). Доступ к справке предоставляет `man`:

```
man [section] <name>
```

Помимо `man` есть `info`, позволяющая работать с гиперссылками по тексту.

Если нужна справка по встроенным командам или зарезервированным словам, используйте встроенную `help`:

```
$ help select
select: select NAME [in WORDS ... ;] do COMMANDS; done
    Select words from a list and execute commands.
# ...
```

Страницы делятся на секции, обозначаемые цифрой от 1 до 9. Чтобы посмотреть перечень, вызовите:

```
$ man man
```

Выход из `man` (и многих других программ) – клавиша `q`.

Краткое описание страницы выдаёт `whatis`:

```
$ whatis man
man (1)              - an interface to the system reference manuals
```

Почему это важно? В системе могут быть программы и файлы с одинаковым именем. Например, секция `1` – исполняемые файлы, секция `5` – файлы конфигурации.

```
# Information about the utility
$ man 1 passwd
# Information about the /etc/passwd file
$ man 5 passwd
```

Не знаете, как пользоваться программой? Как говорится – RTFM!

Если вы не знаете, какая утилита нужна, воспользуйтесь поиском по `man`-страницам – `apropos` (предварительно прочитав `man` про неё):

```
$ apropos [optional] keyword
```

То же самое — через `man -k`. Поиск идёт по базе данных. Она обновляется автоматически в большинстве дистрибутивов, но не сразу. Если вы только что установили программу, обновите вручную:

```
$ sudo mandb --create
```

Обычно обновление осуществляется через `cron`. Проверить наличие задачи можно по файлу `/etc/cron.daily/man-db`. Либо обновление реализовано в виде сервиса `systemd`:

```
$ systemctl list-timers | grep man-db
```

Если известно, в какой секции искать (допустим, интересует программа), ограничьте поиск:

```
$ apropos -s 1 create
```

Если запрос сложнее одного слова, используйте регулярные выражения (Regex). Например, интересует создание или модификация:

```
$ apropos -r "create|modify"
```

Пакетный менеджер

В Linux есть специальная программа (в разных дистрибутивах своя) для управления пакетами и их зависимостями — пакетный менеджер. В Ubuntu это `apt`. Большинство программ и утилит можно установить через него из командной строки.

Перед установкой полезно обновить базу пакетов:

```
$ sudo apt update
```

Допустим, нужна консольная программа для более удобного просмотра директорий и файлов, с возможностью манипуляций. Утилита `mc` не входит в стандартный набор, поэтому её нужно установить:

```
$ sudo apt install mc
```

Если пакет больше не нужен, его можно удалить:

```
$ sudo apt remove mc
```

Логи ядра

При проблемах с оборудованием полезно посмотреть, что говорит ядро. События ядро записывает в кольцевой буфер (при переполнении старые записи затираются). Получить доступ можно через `dmesg`. Эта утилита понадобится нам дальше. Попробуйте ввести команду в терминале и посмотрите, что произойдёт:

```
$ sudo dmesg
```

Там, как правило, много текста. Дальше, узная другие утилиты, посик по логам упроститься, продолжайте читать.

Управление питанием

Выключать и перезагружать систему можно из консоли. Для перезапуска служит `reboot`:

```
# Restart now
$ sudo reboot now
# Restart at 7:40
$ sudo reboot 7:40
# Restart in 42 minutes
$ sudo reboot +42
```

Для выключения используйте `shutdown`:

```
$ sudo shutdown now
```

Кроме `shutdown` в системе присутствуют `poweroff` и `halt`. Они работают немного попразному. `poweroff` ведёт себя как `shutdown`, но без отсрочки. `halt` останавливает систему, но не отключает питание.

В современных дистрибутивах все эти команды — символьные ссылки на `systemctl`:

```
$ ls -la /sbin/ | grep "reboot"
lrwxrwxrwx  1 root root 16 Aug  8 14:51 reboot -> ../bin/systemctl
```

Про оболочку

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Переменные

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Инициализация оболочки

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Локализация

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Вложенность

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Выполнение команд

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Шаблоны

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Горячие клавиши

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

История

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Внешний вид

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Цвета

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

`tput` и `terminfo`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Псевдографика

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Арифметика

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Мультиплексор терминала

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Базовые команды

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Вывод текста

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Работа с файловой системой

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Работа с текстом

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Steam EDitor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

AWK

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Работа с процессами

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Прочие полезные команды

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Работа с сетью

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Конвейер и потоки

Что такое stream?

Что такое конвейер?

Скрипты

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Аргументы

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Регулярные выражения

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Функции

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Перехват сигналов

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Ветвление

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Циклы

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Обмен данными

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Утилита `set` и отладка скрипта

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/highway_to_shell.

Пишем shell-совместимую утилиту

Иногда bash-скрипта хватает. Иногда — нет. Нужна крошечная утилита, которую легко встроить в конвейер, передать ей поток, получить ровно один результат и пойти дальше. В этой главе мы напишем такую утилиту на Си и сразу встроим её в привычный Unix-поток работы.

Какой должна быть CLI-утилита

Чтобы инструмент «вписался» в экосистему, держим в голове несколько простых правил:

- коды возврата: `0` — успех, `>0` — ошибка;
- потоки: читаем из `stdin`, пишем в `stdout`, ошибки — в `stderr`;
- аргументы в стиле Unix: короткие `-o` и длинные `--output`;
- по умолчанию выводим минимум; детали — по флагам;
- никаких побочных эффектов: в файлы пишем только по явной просьбе;
- конвейеры — первоклассные граждане: всё должно работать с `|`.

Задача: контрольные суммы для прошивок

На микроконтроллерах STM32 есть периферия CRC32 — удобно для проверки целостности, например прошивки. Но есть нюанс: расчёт отличается от стандарта IEEE. Готовой «стандартной» утилиты под этот вариант нет — сделаем свою! Так же в целях демонстрации добавим расчёт контрольной суммы MODBUS.

Назовём программу `chsm` (checksum). Как ей хочется пользоваться:

- вход — файл или `stdin`;
- вывод — только контрольная сумма, без лишнего шума;

- по умолчанию шестнадцатеричный формат; десятичный включаем флагом `-i`;
- алгоритм выбираем флагом: `--algo stm32` (по умолчанию) или `--algo rtu` (Modbus RTU).

Примеры запуска:

```
$ ./chsm firmware.bin
58E99ECC

$ ./chsm -i firmware.bin
1491705548

$ ./chsm --algo rtu firmware.bin
1A80

$ echo -en "\xAA\xBB\xCC\xDD" | ./chsm
58E99ECC
```

Внутренний интерфейс

Сам алгоритм CRC32 разбирать не будем — важен единый интерфейс для разных реализаций. Под капотом каждая реализация должна уметь инициализироваться, «скармливать» порции данных и отдавать результат:

```
typedef enum {
    eALGO_STM32,
    eALGO_RTU,
} AlgoId;

typedef struct {
    AlgoId id;
    char *name;
    IfaceChecksum *iface;
} Algo;

typedef struct {
    void (*init)();
    void (*accumulate)(const uint8_t *data, uint32_t length);
    uint32_t (*get_crc)();
} IfaceChecksum;
// ...
IfaceChecksum stm32 = {
```

```
    .init = stm32_crc_init,
    .accumulate = stm32_crc_acc,
    .get_crc = stm32_crc_get,
};
// ...
Algo algorithms[] = {
    { eALGO_STM32,  "stm32",    .iface = &stm32 },
    { eALGO_RTU,    "rtu",      .iface = &rtu },
};
```

Разбор аргументов

Минимальная логика: `-i` переключает формат, `-a/--algo` выбирает реализацию, `-h` печатает справку.

```
int opt;
bool print_hex = true;
Algo *algorithm = &algorithms[0];

while ((opt = getopt_long(argc, argv, "ia:h",
        long_options, NULL)) != -1) {
    bool found = false;
    switch (opt) {
        case 'i':
            print_hex = false;
            break;
        case 'a':
            for (uint32_t id = 0;
                 id < sizeof(algorithms) / sizeof(Algo); id++) {
                if (strcmp(optarg, algorithms[id].name) == 0) {
                    found = true;
                    algorithm = &algorithms[id];
                    break;
                }
            }
            found ?: print_usage();
            break;
        case 'h':
        default:
            print_usage();
    }
}
```

Откуда брать данные

Сценария два: пользователь указал файл — читаем его; нет — пробуем `stdin`; если и там пусто (терминал), сообщаем об ошибке.

```
FILE *file = NULL;
if (optind < argc) {
    file = fopen(argv[optind], "rb");
    if (file == NULL) {
        fprintf(stderr, "%s: %s\n", PROGRAM_NAME, strerror(errno));
        return EXIT_FAILURE;
    }
} else if (!isatty(STDIN_FILENO)) {
    file = stdin;
} else {
    fprintf(stderr, "%s: No input provided.\n", PROGRAM_NAME);
    return EXIT_FAILURE;
}
```

Считаем и печатаем

Читаем маленькими порциями, накапливаем CRC и печатаем в нужном формате.

```
uint8_t buffer[4];
uint32_t bytes_read = 0;
algorithm->iface->init();
while (bytes_read = fread(buffer, 1, sizeof(buffer), file),
       bytes_read != 0) {
    algorithm->iface->accumulate(buffer, bytes_read);
}

printf(print_hex ? "%X\n" : "%d\n", algorithm->iface->get_crc());
```

Быстрая самопроверка

Сравним вывод «из файла» и «из stdin» — должны совпасть:

```
$ echo -en "\xAA\xBB\xCC\xDD" > firmware.bin
$ test $(./chsm firmware.bin) = $(echo -en "\xAA\xBB\xCC\xDD"|./chsm)
$ echo $?
0
```

`test` вернул `0` — значит, утилита ведёт себя одинаково в обоих режимах.

Что получилось

Небольшая, «вежливая», утилита: не засоряет вывод, дружит с конвейерами, понимает файл и поток, возвращает осмысленный код выхода и умеет работать с несколькими алгоритмами. Ровно то, что хочется иметь под рукой в сценариях сборки и проверки прошивок.