



Bash & Linux CLI

Mastering Linux: The complete guide

Who am I?

- ▶ My name is Jannis Seemann
- ▶ I'm German , with a degree in computer science (TU München)
- ▶ **Important work experience:**
 - ▶ With 18, first internship at Google in London 
 - ▶ At age 20, another one in Mountain View, California 
 - ▶ Nowadays, I'm a freelance web developer (and server admin)
 - ▶ And one of the most popular instructors on the German speaking market on Udemy (250.000+ students)
- ▶ **In this course:**
 - ▶ You benefit from my work experience
 - ▶ ... and from my online teaching experience

Streams

File management

Networking

Processes on
Linux

Pipes

Bash environment

Cronjobs

SSH
(client & server)

Mounts

Unix: What is a file?

Colorful prompt
(PS1)

The boot process
(GRUB)

wget

SELinux

Shell conditionals

File permissions

Deploying a
LAMP server

Symlinks

DHCP

LVM (Logical
Volume Manager)

Users & Groups

Background jobs
in Bash

curl

DNS

Various Quizzes

Firewall with
`firewalld`

Shell expansions

Package management on CentOS
and Ubuntu (`apt` and `dnf`)

System services:
`systemd`

Full software
upgrade on
Ubuntu

The structure of this course

Part 1: Introduction

Setting up your system



Part 2: Bash

Using the Command line (CLI)



Part 3: Linux

Understand core Linux concepts



Part 4: Advanced Bash

Automate your workflow with Bash Scripts

About this course

- ▶ **This course is the most comprehensive course I've ever developed**
- ▶ **It also took the most amount work ever:**
 - ▶ 8+ months of development
 - ▶ On top of it, support from 2 amazing freelancers for additional quizzes, proof watching, exercises,...
- ▶ **Thank you for watching and buying this course:**
 - ▶ This allows us to create such a comprehensive guide
 - ▶ It really means a lot to me



macOS

- ▶ This course is mostly about Linux
- ▶ But whenever there's a low hanging fruit (no pun intended)...
we'll just pick it... but we won't take large detours for this
- ▶ **Thus, if you're using macOS:**
 - ▶ There'll be some additional lectures here and there
 - ▶ This allows me to highlight minor differences between
Linux and macOS

Reviews

- ▶ Udemy will ask you to leave a review soon
- ▶ This is outside of my control
- ▶ If you don't feel ready to leave a review, you can skip this, and
Udemy will ask you later again
- ▶ Otherwise, I appreciate any honest review - they keep me
motivated to develop more courses!



Bash & Linux CLI

What is your goal?

What is your goal?

- ▶ **Why do you want to learn Linux?**

- ▶ Do you want to work with servers?
- ▶ Do you want to develop and deploy applications?
- ▶ Are you preparing for a professional certification?
- ▶ Do you want to boost your career?
- ▶ Do you want to learn Linux for personal projects?
- ▶ Are you using Linux as your primary operating system?

- ▶ **For all those answers - and probably even more:**

- ▶ **This is the right course for you!**

How to approach this course?



► **Follow the course structure:**

- ▶ Start with the foundational concepts
- ▶ Gradually move towards more complex topics
- ▶ Utilize the provided quizzes and exercises and labs
- ▶ You can of course also skip chapters if they're not relevant for you

► **Practice makes the master:**

- ▶ Try to set a few hours per week aside just for studying Linux
- ▶ Adapt the commands, and try them on your own system
- ▶ Experiment and challenge yourself
- ▶ **Remember:** It's a marathon, not a sprint

► **Seek support when necessary:**

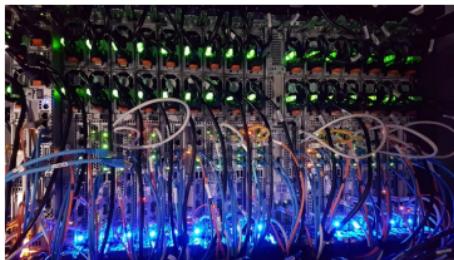
- ▶ Research questions online first
- ▶ But if you don't find any answers, reach out in the QA area to other fellow students (and us)



Bash & Linux CLI

What is Linux?

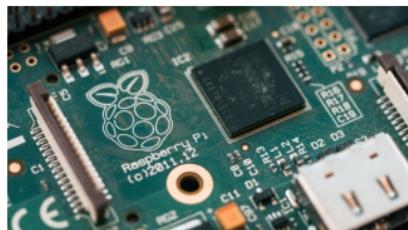
Use Cases



Servers
(Web, Email, File, Database, Game,...)



Data Centers
(Cloud Computing)



Micro-Computers



Cyber-Security

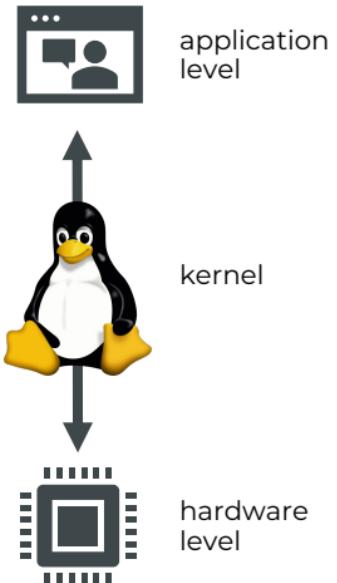


Internet of Things (IoT)



What is Linux?

- ▶ **Linux** is the kernel of a family of open-source Unix-like operating systems (OS)
- ▶ **OS:** A system software that manages computer hardware and provides various services for computer programs
- ▶ **Unix:** A powerful, multitasking, multi-user OS that serves as the basis for many modern OS
- ▶ **Kernel:** The **kernel** is the core part of an OS
 - ▶ It is responsible for managing the hardware, including the CPU, memory, and other devices



What is the GNU project?

- ▶ GNU stands for **GNU's Not Unix**:
 - ▶ **The goal:** To create a free and open-source operating system
 - ▶ Initiated in 1983
 - ▶ Most GNU software is released under the **GNU General Public License** (GPL), promoting the freedom to use, modify, and distribute the code
 - ▶ The Linux kernel, created in 1993 by Linus Torvalds, is released under the GPL
- ▶ GNU provides many of the additional tools & utilities that we want to use
- ▶ Linux provides the kernel
- ▶ Together: **GNU/Linux**



Bash & Linux CLI

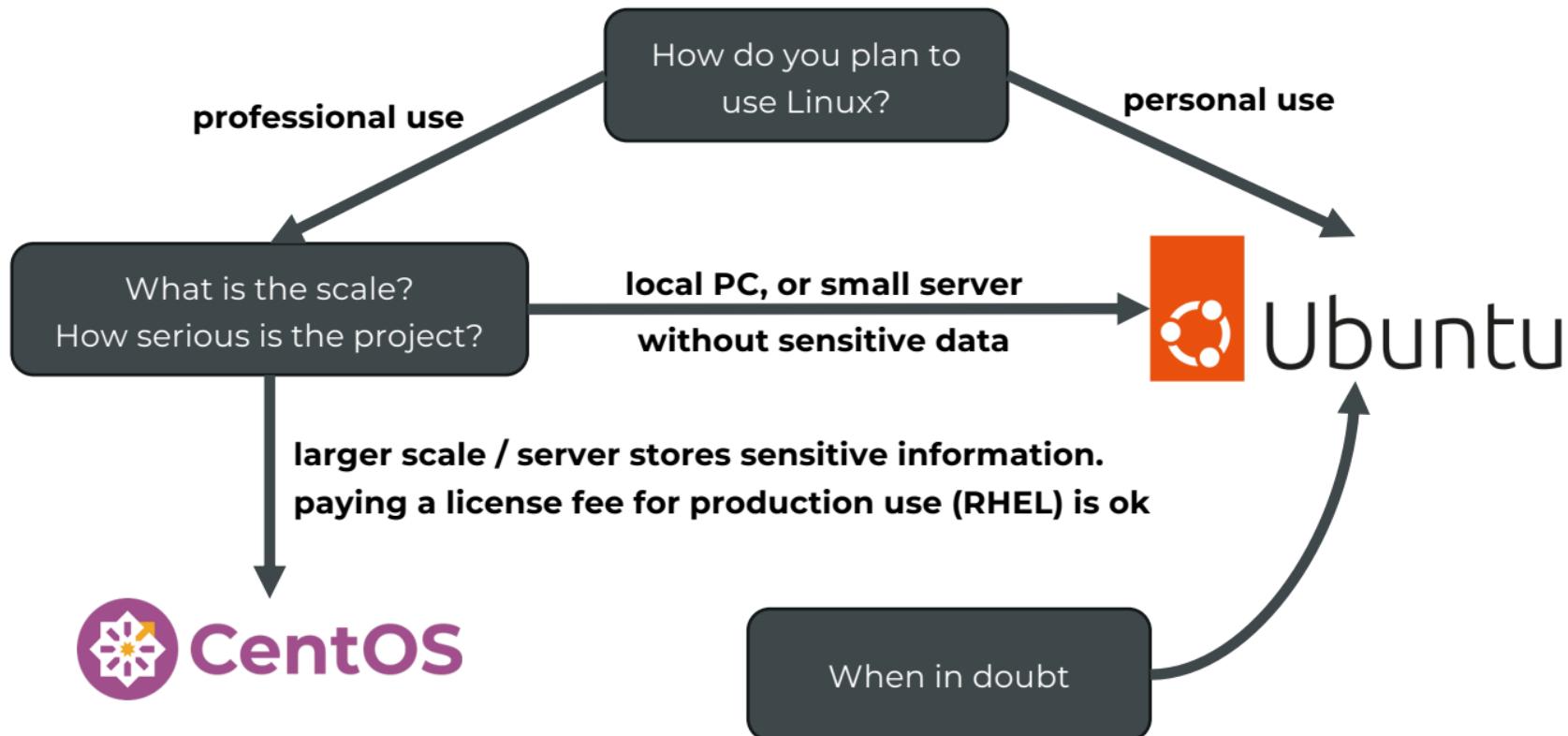
Linux distributions

Linux distributions?

- ▶ There are various OS built upon GNU/Linux, called Linux distributions (distros)
- ▶ Each distribution then adds additional, specific software:
 - ▶ This allows to cater to the needs of a specific user group
 - ▶ Most Linux distributions are free
- ▶ **In this course, we will explore 2 popular distributions:**
 - ▶ **Ubuntu:**
 - ▶ Part of the Debian family
 - ▶ Designed to be user-friendly
 - ▶ **CentOS Stream:**
 - ▶ Part of the Red Hat family
 - ▶ It's free to use, and allows us to familiarize ourselves with the Red Hat family
- ▶ **We will later have an overview about additional Linux distributions!**



Which Linux distribution should you choose?



Bash & Linux CLI

How to run Linux?

How to run Linux?

- ▶ **We could install Linux as a main operating system:**

- ▶ You could then reboot and choose to boot into Linux
(depending on how you install it: "dual-boot")

- ▶ **But for learning Linux:**

- ▶ It's better to run Linux virtualized
 - ▶ This means we create a virtual computer that runs on your computer
 - ▶ And on this virtual computer, we'll run Linux

- ▶ **The advantages:**

- ▶ If we destroy your system - nothing to worry about
 - ▶ We can just reinstall it
 - ▶ And we can easily switch between different Linux systems...
 - ▶ ... or even run them at the same time!



Ubuntu



Windows

If you're using a Mac

► **If you have an Intel processor:**

- ▶ You can use VirtualBox
- ▶ You can keep following this guide

► **If you have an Apple processor:**

- ▶ VirtualBox doesn't work properly on your system
- ▶ Thus, I've provided you with a specialized guide to follow (UTM)



PowerPoint

About This Mac

System Settings...

App Store...

Recent Items



MacBook Pro

16-inch, 2021

Chip Apple M1 Max

Memory

Startup disk

Serial number

macOS



More Info...

What is VirtualBox?

- ▶ VirtualBox is a **virtualization software** that allows you to create and run virtual machines (VMs) on your computer
 - ▶ Developed by Oracle Corporation
 - ▶ Free & open-source software
- ▶ **Installation:**
 - ▶ <https://www.virtualbox.org/wiki/Downloads>



Bash & Linux CLI

How to install Ubuntu

Installing Ubuntu in VirtualBox

- ▶ **Ubuntu:**

- ▶ You also need to download the **Ubuntu Desktop image**
- ▶ This image contains the whole operating system - including an installer
- ▶ We can then create a virtual machine, and let VirtualBox install Ubuntu for us
- ▶ <https://ubuntu.com/>

- ▶ **Important:**

- ▶ **We will be using the latest LTS version of Ubuntu in this course**
- ▶ **Let's have a look at this!**

Bash & Linux CLI

VirtualBox: Configuring Ubuntu

Configuring Ubuntu

- ▶ We need to install some drivers to be able to work smoothly with our virtual machine
- ▶ **For this:**
 - ▶ We need to update our system
 - ▶ We need to install tools required to compile the drivers
 - ▶ And then we need to install the drivers

Bash & Linux CLI

VirtualBox: How to install CentOS Stream

Installing CentOS Stream in VirtualBox

- ▶ **CentOS Stream:**

- ▶ We need to manually create an empty virtual machine
- ▶ And then install CentOS Stream into it

- ▶ **For this:**

- ▶ **We need to go to the website of CentOS:**

- ▶ <https://www.centos.org/>
- ▶ And download the latest version of CentOS Stream from there!
- ▶ **Let's have a look at this!**

Bash & Linux CLI

VirtualBox: How to configure CentOS Stream

VirtualBox: How to configure CentOS Stream

- ▶ We need to install some drivers to be able to work smoothly with our virtual machine
- ▶ **For this:**
 - ▶ We need to update our system
 - ▶ We need to install additional ways to fetch additional software
 - ▶ We need to update our system (one more time)
 - ▶ We need to install tools required to compile the drivers
 - ▶ And then we need to install the drivers

Bash & Linux CLI

VirtualBox: How to create snapshots

How to create snapshots

- ▶ **A snapshot:**

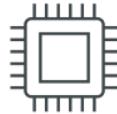
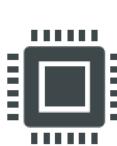
- ▶ Allows you to go back in time
- ▶ Then you could destroy your virtual machine
- ▶ And just go back to the previous snapshot!

Bash & Linux CLI

[Mac, UTM]: Installing UTM

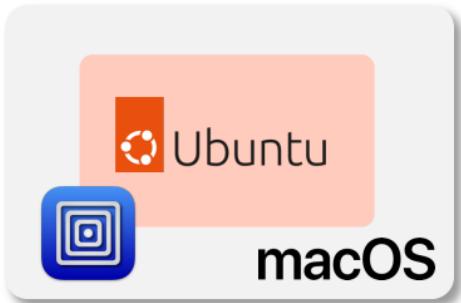
Why do we need UTM?

- ▶ The two most common CPU architectures are "x86" and "ARM"
 - ▶ **x86:**
 - ▶ Developed by Intel
 - ▶ Widely used by AMD and Intel
 - ▶ Extended to 64 bit (called `x86_64` or `amd64` then)
 - ▶ **ARM:**
 - ▶ Dominant in the mobile OS space (iOS, Android)
 - ▶ Nowadays, expanding into traditional computing
 - ▶ Especially certain servers and newer Macs (M1, M2, M3,...)
- ▶ VirtualBox mainly supports x86 architectures for host systems



What is UTM?

- ▶ **UTM:**
 - ▶ Virtualization software, optimized for macOS
- ▶ **UTM offers 2 ways to run a virtual machine:**
 - ▶ **Apple virtualization:**
 - ▶ This is the virtualization that we will use
 - ▶ Quite fast, but we can only run software that is compiled for ARM
 - ▶ **QEMU:**
 - ▶ Slower, but would allow us to run software that is compiled for x86
 - ▶ Not necessary for Ubuntu / CentOS
 - ▶ **I highly recommend against this!**
- ▶ **Installation:**
 - ▶ <https://mac.getutm.app/>



Bash & Linux CLI

How to install Ubuntu

Installing Ubuntu in UTM

- ▶ **Ubuntu:**

- ▶ You also need to download the **Ubuntu Desktop image for ARM**
- ▶ Currently, this is a little bit difficult to find

- ▶ **But we can download it from here (Ubuntu 22.04):**

- ▶ <https://cdimage.ubuntu.com/jammy/daily-live/current/>

- ▶ This image contains the whole operating system - for arm64 systems
- ▶ We can then create a virtual machine in UTM

- ▶ **Important:**

- ▶ **We will be using the latest LTS version of Ubuntu in this course**

- ▶ **Let's have a look at this!**

Bash & Linux CLI

How to install Ubuntu

Bash & Linux CLI

[Mac, UTM]: Configuring Ubuntu

Bash & Linux CLI

[Mac, UTM]: Installing CentOS Stream

Installing CentOS Stream in UTM

- ▶ **CentOS Stream:**

- ▶ We need to download CentOS Stream image for arm64
- ▶ We can then create a virtual machine in UTM, and boot from this image

- ▶ **Let's have a look at this!**

Bash & Linux CLI

[Mac, UTM]: Configuring CentOS Stream

Bash & Linux CLI

Outlook

Outlook

► **Congratulations:**

- You now have a running Linux system that you can use
- Feel free to play around a bit and discover how things work

► **Now, we're prepared for the next chapter:**

- There, we will have a look at the principles of Linux
 - And we will start to explore the Terminal
- **So: Stay tuned!**

Bash & Linux CLI

First steps in the Terminal

First steps in the Terminal

► In this chapter:

- ▶ We will have a look at what is a Shell
- ▶ We will go through a quick history of different shells (sh, Bash, Zsh,...)
- ▶ We will open and configure the Terminal
- ▶ We will execute our first few Bash commands

Bash & Linux CLI

What is a shell?

What is a shell? (generally)

- ▶ **A shell:**

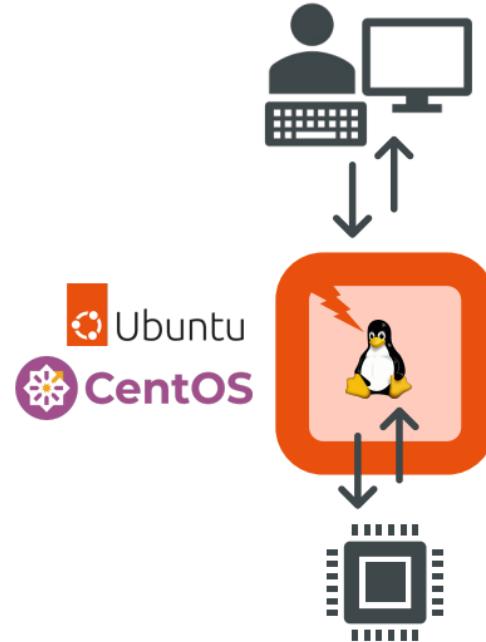
- ▶ **Outer layer** of the operating system
- ▶ It takes commands from the user and translates them into a form that the kernel can understand
- ▶ It can also display the result of those commands to the user

- ▶ **In general:**

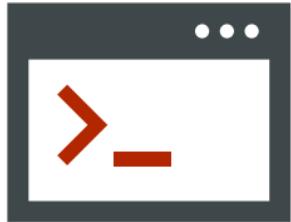
- ▶ Everything that allows the outer world to access the operating system

- ▶ **So:**

- ▶ The graphical user interface is always a shell!



What is a shell (command line)



- ▶ **However:**

- ▶ Quite often, the term "shell" only refers to the **command line interface (CLI)** of an operating system
- ▶ **CLI:** A text-based interface that allows users to interact with systems by typing commands

- ▶ **Thus, "Linux shell":**

- ▶ This usually only refers to the CLI / "terminal" / "console"

- ▶ **The Linux shell:**

- ▶ Allows us to work on devices that don't support a graphical user interface (GUI)
- ▶ Quite often, it's more efficient to use the CLI instead of the GUI

Bash & Linux CLI

A history of shells

What are the most important shells?

- ▶ There're many different shells:

- ▶ They depend on the operating system that is being used

- ▶ Let's go through the most important of them:

- ▶ **bourne shell**:

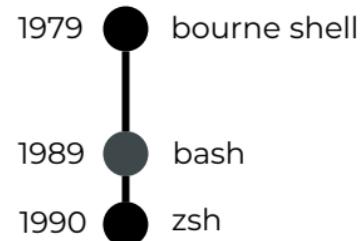
- ▶ The shell that was the default shell in Unix 7
 - ▶ (back then: as the program "sh")

- ▶ **bash**: Replacement for the *bourne shell* (**Bourne Again Shell**)

- ▶ It's the default shell for many Linux distributions
 - ▶ Open-source software, GPL license
 - ▶ If we nowadays start a "sh"-shell, we usually start bash

- ▶ **Z shell (zsh)**:

- ▶ It's the default shell for various Linux-distributions (Kali Linux,...) and new installations of macOS



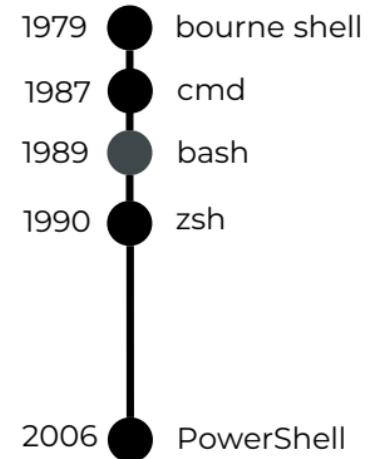
What are the most important shells?

► Microsoft Shells:

- ▶ **cmd** (command prompt)
- ▶ Developed by Microsoft
- ▶ It allows us to use MS-DOS commands on a windows computer
- ▶ Completely different commands than Bash

▶ PowerShell

- ▶ Developed by Microsoft
- ▶ Also available for Linux and macOS
- ▶ Uses different commands than Bash
- ▶ *(though some commands resemble their Unix counterparts)*





Bash & Linux CLI

Why should you learn Bash?

Why Bash?



► Bash:

- ▶ The most common command language used in CLI
- ▶ We can easily launch unix tools, and combine them together
- ▶ This allows us to be extremely efficient in what we write
- ▶ We can also write our commands down and create shell scripts

But, be careful

- ▶ The syntax of Bash is extremely short and compact
- ▶ Slightly different commands can mean something completely different
- ▶ **Example (no need to understand for now):**
 - ▶ `cat $PWD/*.txt`
 - ▶ `cat "$PWD/*.txt"`
 - ▶ `cat '$PWD/*.txt'`
- ▶ **Thus:**
 - ▶ We will focus on deep understanding in this course
 - ▶ And I'll guide you through everything - step by step!



Bash & Linux CLI

First steps in the Terminal

Let's confirm that you're using Bash

- ▶ Let's first confirm that you're running a Bash
- ▶ **We can do this with the following command:**
 - ▶ `echo "${BASH_VERSION}"`
- ▶ **If it doesn't output anything (or just an empty line):**
 - ▶ Be sure to launch Bash with the following command:
 - ▶ `bash`
 - ▶ For now, you will need to do this any time you open a new window / tab in your terminal



Bash & Linux CLI

A first command in the Terminal

The command echo

- ▶ echo allows us to output text in the terminal
- ▶ **We can test this:**
 - ▶ echo 'Bash is amazing!'
- ▶ **For now:**
 - ▶ The string that we want to print should be wrapped in **single quotes**
 - ▶ We will learn more about why in the chapter about *Expansions*
- ▶ **Let's have a look at this!**

```
echo 'Bash is amazing!'
```

command / program argument(s)

Bash command

The command echo

- ▶ **By default:**

- ▶ echo will by default output a line break at the end
- ▶ We can disable this with the option: -n

- ▶ **Example:**

- ▶ `echo -n 'Bash is amazing!'`

- ▶ **Let's have a look at this!**

- ▶ **We can also use different options:**

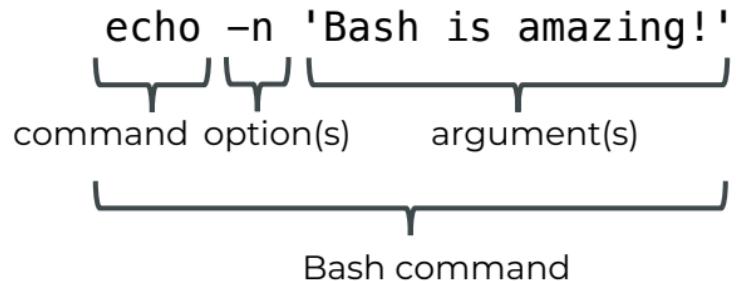
- ▶ The option -e enables backslash escapes

- ▶ **Example:**

- ▶ `echo -e 'Line 1\nLine 2'`

- ▶ Here, \n will now be converted into a line break

- ▶ **Let's have a look at this!**



Combining options

- ▶ Quite often, we want to combine multiple options
- ▶ **Usually, all of those are equivalent:**
 - ▶ `echo -e -n 'Hello\nworld'`
 - ▶ `echo -en 'Hello\nworld'`
 - ▶ `echo -ne 'Hello\nworld'`

Bash & Linux CLI

Navigating the filesystem

The pwd command

- ▶ **In our terminal:**

- ▶ We're always in a directory
- ▶ This directory is called the *working directory*
- ▶ It is the folder that the shell is currently operating at
- ▶ Commands can access this folder and act relative to this directory

- ▶ **To find out our current directory:**

- ▶ `pwd`
- ▶ (**p**rint **w**orking **d**irectory)

Changing the directory: cd

- ▶ If we want to change the directory, we can use the cd command
- ▶ cd stands for: "c~~h~~ange d~~i~~rectory"

- ▶ **Examples:**

- ▶ **cd [directory_name]:**

- ▶ Moves into the directory specified by [directory_name]

- ▶ **Example:**

- ▶ cd Desktop

- ▶ cd /

- ▶ **cd ..:**

- ▶ Move into the parent directory

- ▶ **cd ~ or cd:**

- ▶ Move to the user's home directory

- ▶ **cd ~/Desktop:**

- ▶ Move to the user's desktop

Bash & Linux CLI

Listing contents of a folder

Listing the contents: `ls`

- ▶ To list the contents of the current working directory:

- ▶ `ls`

- ▶ We can also add additional options:

- ▶ **Syntax:** `ls [option...] [path]`

- ▶ **-a:** List all entries, including hidden files starting with `.`

- ▶ We will learn more about hidden files later

- ▶ **-r:** Reverse order while sorting

- ▶ **-t:** Sort by modification time, newest first

- ▶ **--color:**

- ▶ Enables colorful output

- ▶ `--color={always,never,auto}`

- ▶ Example:

- ▶ `ls -ta --color ~/Desktop`

- ▶ `ls -ta --color=auto ~/Desktop`

Bash & Linux CLI

Absolute vs. relative paths

Absolute vs. relative paths

- ▶ **Absolute paths:**

- ▶ Start with a "/"
- ▶ They define the complete path to a file
- ▶ Thus, they work everywhere - no matter our current working directory

- ▶ **Example:**

- ▶ `/home/jannis/Desktop`
- ▶ `~/Desktop`

- ▶ **Relative paths:**

- ▶ Are being resolved according to our current working directory

- ▶ **Example:**

- ▶ `./Desktop`
- ▶ `Desktop`
- ▶ `../Desktop`

- ▶ **Let's have a look at this! ☺**

Bash & Linux CLI

Executing multiple commands

Executing multiple commands

- ▶ We can also execute multiple commands
- ▶ **For this, we can just add a semicolon between them:**
 - ▶ command1; command2;
 - ▶ **For example:**
 - ▶ echo -n 'Hello '; echo 'World'

Bash & Linux CLI

How to get help?



How to get help?

▶ **--help / -h:**

- ▶ For a lot of commands, we can just add a `-h` or a `--help`:
- ▶ `ls --help`
- ▶ We will then be shown a list of possible options and arguments

▶ **man:**

- ▶ If this doesn't work, we can check the built-in manuals
- ▶ `man ls`

▶ **Important:**

- ▶ man pages (documentation) must be installed on your system

▶ **Otherwise:**

- ▶ Many tools have extensive online documentation
- ▶ Communities such as Stack Overflow, or Reddit's Linux communities can also be great for help

Bash & Linux CLI

First steps with Linux

First steps with Linux

- ▶ **In this chapter:**

- ▶ User management
- ▶ Elevating privileges with sudo

- ▶ **Package management:**

- ▶ Keeping your system up to date
- ▶ Installing / removing software

- ▶ **Additional bonus for macOS users:**

- ▶ How to run a Bash on macOS
- ▶ (without needing to launch a VM)

Bash & Linux CLI

User Management

Basics about user management

- ▶ In Linux, users can be categorized into three general categories:

- ▶ **System accounts:**

- ▶ They are responsible for running background tasks on your system
(such as: webserver, database,...)
- ▶ They don't have a home directory

- ▶ **Regular users:**

- ▶ They have access to their own files and directories
- ▶ They cannot perform administrative tasks or access other user's files
without permission

- ▶ **Superuser (root):**

- ▶ The superuser (root) has unrestricted access to the entire system
(including files in the home directories of regular users)
- ▶ Can add / remove users, install software
- ▶ Can change the configuration of the system

Bash & Linux CLI

Elevating privileges: sudo

Elevating privileges: sudo

- ▶ If we want to temporarily elevate our privileges...

... we can put a `sudo` in front of our command

- ▶ **Example:**

- ▶ `sudo ls /root`
- ▶ Usually, we could not access this directory
- ▶ Only the root user can access it
- ▶ But `sudo` elevates our privileges
- ▶ This allows us to access it anyway!

- ▶ **But be careful:**

- ▶ Always make sure you understand what a command does
- ▶ Especially when `sudo` is involved

- ▶ **Example:**

- ▶ **Important: DO NOT EXECUTE!**
- ▶ `sudo rm -rf /etc`

Bash & Linux CLI

Optional:

What to do if sudo doesn't work

If sudo doesn't work

- ▶ In that case:
 - ▶ You must turn the user into a regular user
(but with "Administrative" rights)
 - ▶ sudo might not work properly
 - ▶ In this case, you might want to create a new user and give this user administrative privileges
 - ▶ Let's have a quick look at this!

Bash & Linux CLI

Package management (part 1):

How does Ubuntu / CentOS install software

Package management on Linux

► Package management:

- ▶ Most Linux distributions offer a centralized way to install software
- ▶ This process is called package management
- ▶ This is an enormous benefit of many Linux systems
- ▶ As it helps us to keep our system up to date

► Also:

- ▶ A lot of applications no longer need to include their own updater
(Example: Firefox, Chrome,...)

► How does it work?

- ▶ Our system connects to centralized repositories
- ▶ They provide a list of available packages
(including available versions, and their dependencies)
- ▶ This list can then be used to install updates, or install additional tools



Package management

- ▶ **Package management is slightly different for each distribution**
- ▶ **However:**
 - ▶ The main idea remains the same
 - ▶ But different Linux distributions implement it slightly differently
 - ▶ Thus, I need to split this into 2 separate lectures for CentOS vs. Ubuntu

Bash & Linux CLI

[Ubuntu]: How to install software (apt)

| Hello from Ubuntu! |
=====

\ \ ^ ^
 (oo)_____
 (_)\) \ / \ /
 ||----w ||

Ubuntu: How to update software with apt

- ▶ On Debian-based distributions (such as Ubuntu), we can use the tool `apt` to keep our system up to date
- ▶ **It provides several commands for us:**
 - ▶ **`apt update`:**
 - ▶ Refreshes the list of available packages
 - ▶ We should run this before doing anything else with `apt`
 - ▶ **`apt upgrade`:**
 - ▶ Runs a small upgrade of our system
 - ▶ Small means: Upgrades existing packages (and when using `apt`, also allows the install of additional dependencies)
 - ▶ **`apt full-upgrade` or `apt dist-upgrade`:**
 - ▶ Runs a large upgrade of our system
 - ▶ Large means: Upgrades existing packages, and removes / installs additional packages (dependencies)

Ubuntu: How to install / remove software

- ▶ **To install / remove software:**

- ▶ **apt install [package]:**

- ▶ This will install an additional package on our system

- ▶ Example:** apt install cowsay

- ▶ **apt remove [package]:**

- ▶ Removes a package from our system

- ▶ **apt autoremove:**

- ▶ Removes packages that are no longer needed

- ▶ You can run this if there're any errors during an **upgrade** or
full-upgrade

Important

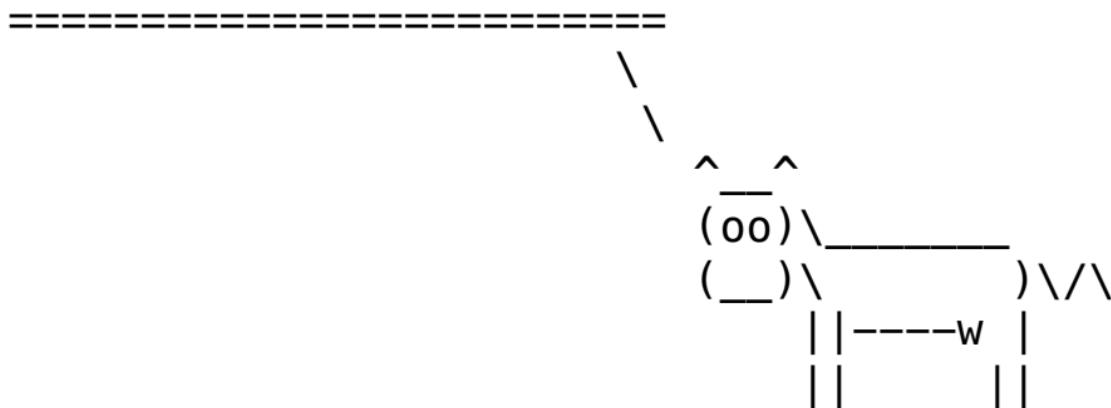
- ▶ There're additional commands for `apt`
- ▶ We will have a more in-depth look at package management on Ubuntu later in this course
- ▶ **Also, a quick heads-up:**
 - ▶ `apt-get` is another implementation of the `apt` tool
 - ▶ Sometimes I might use this as well: `apt-get update`

Bash & Linux CLI

[CentOS]:

How to install software (dnf)

| Hello from CentOS Stream! |



CentOS: How to update software with dnf

- ▶ On RHEL-related distributions (such as CentOS Stream), we can use the tool `dnf` to keep our system up to date
- ▶ **With just one command, we can keep our system up to date:**
 - ▶ **`dnf upgrade` or `dnf update`:**
 - ▶ Fetches the latest version of the package list
 - ▶ And upgrades our system
- ▶ **Important:**
 - ▶ `dnf` always keeps the local package list up to date
 - ▶ We don't have to manually refresh it (in contrast to Ubuntu)

CentOS: How to install / remove software

- ▶ **To install / remove software:**

- ▶ **dnf install [package]:**

- ▶ This will install an additional package on our system

- ▶ **Example:**

- ▶ `dnf install epel-release`
 - ▶ `dnf install cowsay`

- ▶ **dnf remove [package]:**

- ▶ Removes a package from our system

- ▶ **Important:**

- ▶ For now, please install `epel-release` on your system
 - ▶ This adds additional package lists to be able to install additional tools on your system
 - ▶ Command: `dnf update; dnf install epel-release; dnf update;`
 - ▶ We should also run: `crb enable`
 - ▶ We will have a more detailed look at `epel-release` later

Heads-up: dnf

- ▶ **Important:**

- ▶ There're additional commands for dnf
- ▶ We will have a more in-depth look at package management on CentOS later in this course

- ▶ **Also, a quick heads-up:**

- ▶ yum used to be a package manager on CentOS previously
- ▶ Nowadays, it is replaced by dnf

- ▶ **But yum commands still works perfectly:**

- ▶ `yum install cowsay`



Bash & Linux CLI

[Bonus, macOS only]: Using Bash on macOS

Bonus: Using the Terminal on macOS

- ▶ Bash comes pre-installed on macOS
- ▶ Thus, we can just use it in the built-in Terminal app
- ▶ **However:**
 - ▶ Bash is an open-source project
 - ▶ It is released under the GPLv license
- ▶ **The issue:**
 - ▶ Version 3.x, Bash was being released under the GPLv2 license
 - ▶ Starting at Version 4.x, Bash switched to GPLv3
 - ▶ Apple does not include GPLv3 software in their OS
(probably due to licensing issues)
- ▶ **How did Apple solve this issue?**

Installation Guide: The solution

► **The solution for Apple**

- Bash 3.x is pre-installed on macOS, even though it is quite outdated
- Z shell (Zsh) is the default shell for new installations

► **In the next lecture:**

- We will install Bash 5.x on your Mac
- This allows you to use all the latest features
- And you can run Bash commands directly on your Mac!

► **But be careful:**

- You don't have a separate environment then anymore

Bash & Linux CLI

**[Bonus, macOS only]:
Package management & Installing Bash**

Installing Software



- ▶ On Mac, we can use Homebrew to install additional packages
- ▶ **We can find it here:**
 - ▶ <https://brew.sh/>
- ▶ This gives us a package manager, that we can then use to install additional packages
- ▶ **Once we've installed Homebrew:**
 - ▶ **To update the package definitions:**
 - ▶ `brew update`
 - ▶ **To install a package:**
 - ▶ `brew install bash`
 - ▶ **And to install available updates:**
 - ▶ `brew upgrade`

Launching Bash on macOS

- ▶ **Once we've installed Bash on a Mac:**
 - ▶ We can launch Bash in Version 5.x+ by launching a terminal, and executing the following command:
 - ▶ `bash`
 - ▶ **We can confirm that we're running a recent version of Bash:**
 - ▶ `echo "${BASH_VERSION}"`
 - ▶ **To launch the original (3.x) version of Bash:**
 - ▶ `/bin/bash`
- ▶ **Let's have a look at this! ☺**

Bash & Linux CLI

Outlook

You now know quite a bit about Linux

- ▶ Now, you know quite a bit of the Linux operating system
- ▶ This is now a great foundation we can keep building on top
- ▶ **You can:**
 - ▶ Launch a terminal
 - ▶ Keep your system up to date
 - ▶ Install additional software
 - ▶ And you have even explored the first few commands in the Terminal

Outlook

- ▶ **Outlook:**

- ▶ **In the next chapter:**

- ▶ We will have a look at how you can work with normal files
- ▶ How to copy files
- ▶ How to move files
- ▶ How to create empty files
- ▶ How to work with folders,...

- ▶ **And then, in the chapter after the next:**

- ▶ We will explore how to work with text files
- ▶ How to print them int he terminal
- ▶ How to only print a few lines
- ▶ How to edit text files,...

Later in this course

► Later in this course:

- We will deep dive into package management (`apt` / `dnf`)
- We will explore all the intricacies of user management, `sudo`....
- We will have a more in-depth look at how commands are structured and executed

Bash & Linux CLI

Part 2: Bash CLI

Part 2: Bash CLI

► In the next few chapters:

- ▶ We will have a look how to manage files through the command line
- ▶ We will see how we can redirect the output of programs into files / use files as input for programs
- ▶ We will see how we can use pipes to combine tools
- ▶ We will see how the Bash environment works and how this influences the way programs are executed
- ▶ And we will configure your prompt to achieve a more beautiful terminal



Bash & Linux CLI

File Management in Bash

File Management in Bash

► **In this chapter:**

- ▶ You will learn how to create folders and files
- ▶ You will learn how to copy and move files
- ▶ You will learn how to delete files and how dangerous Bash can be
(and how you can mitigate that)

► **In addition, you will solve some real-world problems:**

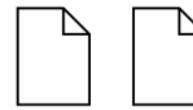
- ▶ Extracting photos from a folder structure
(such as an SD-card from a camera)
- ▶ And you will extract all Excel- and PDF-files from a nested folder
structure

Bash & Linux CLI

File Management in Bash: touch and mkdir

File Management in Bash

- ▶ `touch`
 - ▶ typical use case: create an empty file (or multiple files)
 - ▶ **more precisely:**
 - ▶ to modify the timestamp of a file
 - ▶ if the file already exists, only its timestamp will be modified
 - ▶ otherwise, a new (and empty) file is created
- ▶ `mkdir (make directory)`
 - ▶ create a new directory

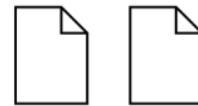


Bash & Linux CLI

File Management in Bash: mv and cp

File Management in Bash

- ▶ **mv** (**move**)
 - ▶ move an existing file to another location
 - ▶ can also be used to rename an existing file
- ▶ **cp** (**copy**)
 - ▶ to copy an existing file
 - ▶ **cp -R**: copies a whole folder

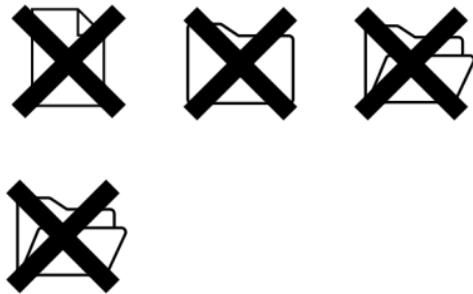


Bash & Linux CLI

How to delete files and folders

How to delete files and folders

- ▶ **rm** (**remove**)
 - ▶ to remove a file (or multiple files at once)
 - ▶ for deleting a directory, you need to use the option **-r**
 - ▶ works for empty and non-empty directories
- ▶ **rmdir** (**remove directory**)
 - ▶ to delete an empty directory



Bash & Linux CLI

Exercise: File management

Exercise: File management

- ▶ In the next lecture, I will present you an exercise
- ▶ It's pretty much just a play-along to do some file management
- ▶ **The goal is:**
 - ▶ You can practice the file management commands for Bash
 - ▶ You can use a file browser to see what each command does and how those commands work
 - ▶ So please don't just execute the commands, also have a look at what they do and how they change the files!
- ▶ **Outlook:**
 - ▶ Later exercises in this course will be more practical, but we need to practice the basics first

Bash & Linux CLI

Solution: File management

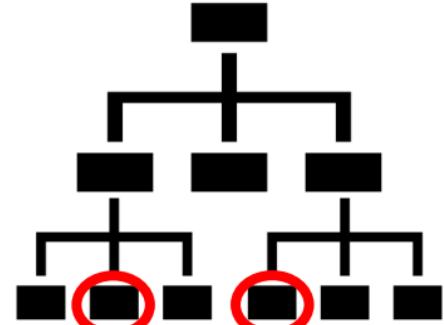
Bash & Linux CLI

Filename expansion (globbing)

Filename expansion (globbing)

► **Filename expansion:**

- ▶ Bash can rewrite our command before it is being executed
- ▶ Globbing recognizes and expands pre-defined wildcard characters
- ▶ It will then search for files that match this pattern and expand (rewrite) our command
- ▶ This allows us to easily access multiple files
- ▶ **Wildcard character: ***
 - ▶ Matches 0 to any number of characters



Filename expansion

► **Important:**

- The wildcard characters must not be quoted (so neither in 'single quotes' nor "double quotes")
- Globbing does not use regular expressions

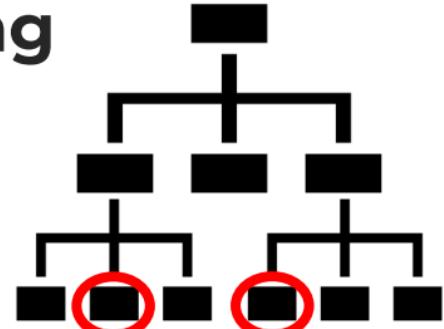
Bash & Linux CLI

Additional wildcards for globbing

Additional wildcards for globbing

► Additional ways to use globbing:

- ▶ ? -> matches any single character
- ▶ [0-9] -> The square brackets allow us to specify a character range (here: all numbers)
- ▶ ** -> matches zero up to arbitrarily many characters (including /)
 - ▶ Only supported in Bash 4.0 or higher!
 - ▶ It might be necessary to enable this:
 - ▶ `shopt -s globstar`



Bash & Linux CLI

Be careful with globbing!

Be careful with globbing!

- ▶ Globbing can be dangerous sometimes!
- ▶ Bash doesn't differentiate between a folder and a parameter
- ▶ And a name of a file may be interpreted as a parameter
- ▶ This is especially dangerous if a file has a name such as:
 - ▶ `-rf`
- ▶ **If we then execute `rm *` in that folder:**
 - ▶ 1.: The `*` will be expanded, so `-rf` will appear in the command
 - ▶ 2.: `rm` will think that `-rf` is a parameter
 - ▶ `-r` means: recursive
 - ▶ `-f` means: don't ask
- ▶ **What's the solution?**
 - ▶ Especially if the files are in the current folder...
 - ▶ We should prefer globbing with `./*` over `*`

Bash & Linux CLI

Installing software on Unix

Installing Software



- ▶ A lot of Linux / Mac systems have a package manager
- ▶ A package manager is a software that helps us install other software
- ▶ On Linux, this package manager is usually build-in
- ▶ On Mac, we can use Homebrew (<https://brew.sh/>)
- ▶ **For installing Software, there're usually 2 steps to be made:**
 - ▶ **Update the package definitions:**
 - ▶ Updates the list of available packages, their versions,...
 - ▶ Mac: brew update
 - ▶ Ubuntu: sudo apt-get update
 - ▶ **Install a package:**
 - ▶ Mac: brew install tree
 - ▶ Ubuntu: sudo apt-get install tree
 - ▶ **(Update existing packages):**
 - ▶ **Mac:** brew upgrade
 - ▶ **Ubuntu:** sudo apt-get dist-upgrade

Bash & Linux CLI

Exercise: Extracting Files

Exercise: Extracting Files

- ▶ Imagine we're running a company
- ▶ And we need to urgently provide documents for a court hearing
- ▶ **For our exercise:**
 - ▶ Right now, it's march
 - ▶ We need to provide all Excel- and PDF-Files for January and February
 - ▶ How do we do this?
 - ▶ Can you use globbing for this?

✓	📁 Purchasing	--	Folder
✓	📁 01 - January	--	Folder
 Balance Sheet.xlsx	Zero bytes	Micros...k (.xlsx)	
 Invoice.pdf	Zero bytes	PDF Document	
✓	📁 02 - February	--	Folder
 additional-table.xlsx	Zero bytes	Micros...k (.xlsx)	
 important-business-figures.xlsx	Zero bytes	Micros...k (.xlsx)	
 important-invoice.pdf	Zero bytes	PDF Document	
 not-important.mp4	Zero bytes	MPEG-4 movie	
✓	📁 03 - March	--	Folder
 numbers-from-march.xlsx	Zero bytes	Micros...k (.xlsx)	
✓	📁 Sales	--	Folder
✓	📁 01 - January	--	Folder
 Sales-January.xlsx	Zero bytes	Micros...k (.xlsx)	
✓	📁 02 - February	--	Folder
 Balance-Sheet-February.pdf	Zero bytes	PDF Document	
✓	📁 03 - March	--	Folder
 Additional-Invoice-March.pdf	Zero bytes	PDF Document	
 Sales-March.xlsx	Zero bytes	Micros...k (.xlsx)	

Tips

- ▶ You can use custom ranges; this may help you with selecting the proper months. For example, [0-4] would match one character of the following: 0, 1, 2, 3, 4
- ▶ You can combine custom ranges with wildcards, for example: A[0-4]*
 - ▶ The filename needs to start with an "A", followed by a 0, 1, 2, 3, 4, and then followed by zero to unlimited characters
- ▶ **You can combine multiple patterns into a single cp command:**
 - ▶ cp [pattern1] [pattern2] [dest]
 - ▶ cp */**/*.jpg */**/*.dng folder

The exercise

- ▶ The folder structure needed for this project is attached to this lecture
- ▶ Just download the .zip file and extract it somewhere
- ▶ Try to solve this on your own
- ▶ You will see how easy this is with bash 😊

Bash & Linux CLI

Solution: Extracting Files

Bash & Linux CLI

Bonus the find program

Bonus: The find program

- ▶ Instead of searching by name we can use the find program for a more sophisticated search
- ▶ **find**
 - ▶ We should provide a path as a first parameter
(e.g. . for current working directory)
- ▶ **Examples:**
 - ▶ **If we want to limit the file type:**
 - ▶ Only files: `find . -type f`
 - ▶ Only directories: `find . -type d`
 - ▶ **Find all files that have been modified within the last 7 days:**
 - ▶ `find . -type f -mtime -7`
 - ▶ **mtime:** modification timestamp
 - ▶ **Find all files that are larger than 10MB:**
 - ▶ `find . -type f -size +10M`
 - ▶ **Delete all empty files in a directory:**
 - ▶ `find . -empty -delete`

Bash & Linux CLI

File management:

Part 2 (text files)

File management: Part 2 (text files)

- ▶ **In this chapter:**

- ▶ We will learn how to work with text files

- ▶ **We will:**

- ▶ Explore different commands to print text files to the terminal
 - ▶ See how we can calculate the size of a file
 - ▶ We will see how we can edit text files in the terminal
 - ▶ And then, there'll be a quick exercise about how to work with files

Bash & Linux CLI

How to read and write files

Bash & Linux CLI

How to read files

How to read files



`data.txt`

- ▶ `cat`:
 - ▶ The command `cat` allows to print the contents of a file:
 - ▶ `cat data.txt`
- ▶ `head`:
 - ▶ Shows us the start of a (text) file
 - ▶ We can specify the Parameter `-n` for the number of lines
 - ▶ `head -n 5 data.txt`
- ▶ `tail`:
 - ▶ Read a file from the end
 - ▶ Same options as `head`
 - ▶ `tail -n 5 data.txt`

Bash & Linux CLI

How to read large files: less

How to read large files: less

► **less**

- ▶ Allows us to read large files
- ▶ We can use the arrow keys to navigate through the file
- ▶ Or we can use f (**f**orward) and (**b**ackward) to navigate a whole page ahead or back
- ▶ We can also navigate to a percentage of our content:
 - ▶ We just enter **50p** to navigate to 50% of our content

How to read large files: less

- ▶ **less**

- ▶ We can use the = to show info about our current position
- ▶ We can also use the Option -N to display row numbers

- ▶ **Find in file:**

- ▶ Forward search: /Search Term
 - ▶ Backwards search: ?Search Term
- ▶ **To quit:** q

Bash & Linux CLI

How to get the size of a file

How to get the size of a file

- ▶ If we just `cat` a large file... the whole file will be printed in the terminal
- ▶ This may take quite long... can't we check the size of a file ahead of time?
- ▶ `wc` (**w**ord **c**ount):
 - ▶ `wc file.txt`
 - ▶ `wc -lwc file.txt`
 - ▶ Prints out the number of lines, the number of words, and the number of bytes in the file
- ▶ `wc -l file.txt`
 - ▶ Counts the number of lines
- ▶ `wc -w file.txt:`
 - ▶ Counts the number of words
- ▶ `wc -c file.txt`
 - ▶ Counts the number of bytes (=8 bit ASCII-characters)

How to get the size of a file: du

- ▶ **du** (**d**isk **u**sage):
 - ▶ It will calculate the size of all items in this folder
 - ▶ We can also specify the parameter "-s", to just get a summary

Bash & Linux CLI

How to edit files

Bash & Linux CLI

Exercise: Analyze a logfile

How to edit files

- ▶ There's no build-in text editor for bash
- ▶ **We have to install additional software for that**
- ▶ **4 quite popular options are:**
 - ▶ pico / nano: A simple editor for text files in bash
 - ▶ vi / vim: A more advanced text editor
- ▶ The install process depends on the system you're using:
 - ▶ **Mac:**

```
brew install nano
```
 - ▶ **Ubuntu:**

```
apt-get update
apt-get install -y nano
```

```
:::  
iLE88Dj. :jD88888Dj:  
.LGitE888D.f8Gjjjl8888E;  
iE :8888Et. .G8888.  
;i E888, ,8888,  
D888, :8888:  
D888, :8888:  
D888, :8888:  
D888, :8888:  
888W, :8888:  
W88W, :8888:  
W88W: :8888:  
DGGD: :8888:  
:8888:  
:8888:  
:8888:  
E888i  
tW88D
```



Exercise: Analyze a logfile

- ▶ In the next lecture, you can download the logfile for this exercise
- ▶ Your exercise is to answer the following questions:
 - ▶ The logfile is based on some real-world data. What could it be?
 - ▶ How large is this file?
 - ▶ How many lines does it contain?
- ▶ Hints / Tips:
 - ▶ Important: Try to all the questions through the shell!
 - ▶ The size of the file is still "manageable" for our shell (usually)
 - ▶ Still, try to not open the file with the cat command
 - ▶ Try to use different commands to browse the file!

Bash & Linux CLI

Solution: Analyze a logfile

Bash & Linux CLI

Overview: Streams

Overview: Streams

► In this chapter:

- ▶ You will learn how you can redirect the output of a program into a file
- ▶ You will learn how you can use a file as the input for another program
- ▶ You will learn how you can handle errors and redirect them

► This is important:

- ▶ It helps us to understand general Unix concepts
(though later in this course we'll go into even more details)
- ▶ It enables us to build more complex commands in Bash
- ▶ It allows you to confidently interact with the terminal
- ▶ And of course, it allows you to save the output of a command to a file ☺

Bash & Linux CLI

How to write output to a file

Write output to a file

- ▶ Let's say we have a command that outputs something
- ▶ How do we get this output into a file?
- ▶ How can we redirect this output into a file?
- ▶ We can do this with the operator ">"
- ▶ **For example:**
 - ▶ echo 'Bash is amazing' > file.txt
 - ▶ If the file doesn't exist, it will be created
 - ▶ Otherwise, the file will be overwritten!
- ▶ **To append to a file**, we can use ">>" instead of ">":
 - ▶ echo 'Bash is amazing' >> file.txt

'Bash is amazing'



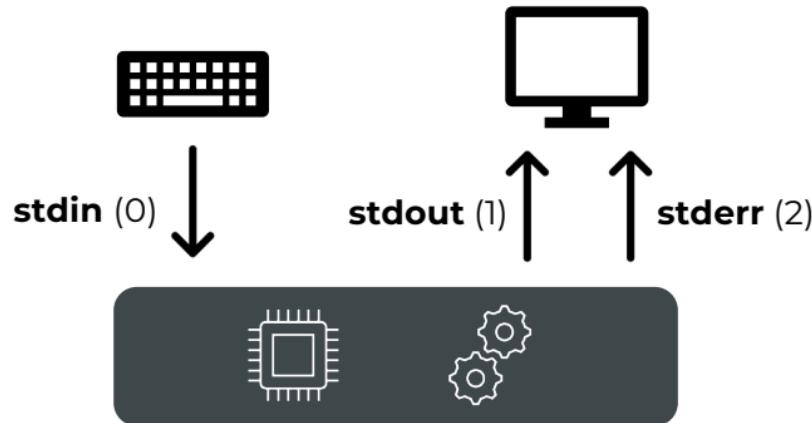
file.txt

Bash & Linux CLI

Standard streams: stdin, stdout, stderr

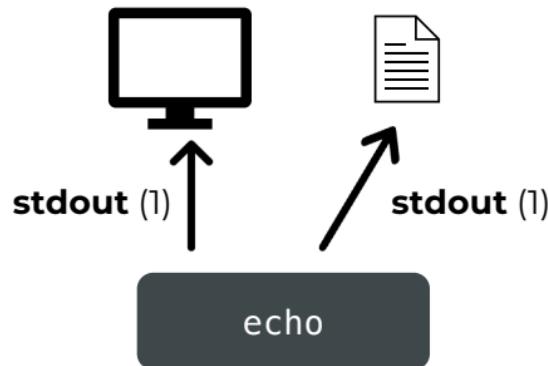
Standard streams: `stdin`, `stdout`, `stderr`

- ▶ By default, there are 3 communication channels for data:
 - ▶ **0**: standard input (from the keyboard); `stdin`
 - ▶ **1**: standard output (on the screen); `stdout`
 - ▶ **2**: standard error output (on the screen); `stderr`



So what does ">" or ">>" do?

- ▶ By using ">" or ">>", we can redirect the stdout to a file
- ▶ Thus, the stdout is no longer being sent to the terminal,
but to a file



Bash & Linux CLI

How to handle errors: Redirect stderr

Why do we want to redirect stderr?

- ▶ **Sometimes, we want to redirect the stderr**
- ▶ **Why?**
 - ▶ A program may print errors, and we want to ignore them
 - ▶ We are only interested in the errors and want to redirect them into a file (to have a look at them at a later date)
- ▶ **How do we do that?**

Redirect stdout and stderr

- ▶ **So far, we have only redirected stdout (1):**

- ▶ `du -h IMG_9328.jpg > output.txt`

- ▶ **However, there's also a more verbose way to do this:**

- ▶ `du -h IMG_9328.jpg 1> output.txt`

- ▶ **We can use this way to also redirect stderr (2):**

- ▶ `du -h IMG_9328.jpg 2> error.txt`

- ▶ **We can also combine those:**

- ▶ `du -h IMG_9328.jpg > output.txt 2> error.txt`

- ▶ **Let's have a look at this in the terminal ☺**

Bash & Linux CLI

Redirect stderr to stdout

Can we redirect stderr to stdout?

- ▶ **First:** Why do we want to redirect stderr to stdout

- ▶ It allows us to easily store both outputs in the same file
- ▶ Right now we would have to provide the filename multiple times:

```
du -h IMG_9328.jpg IMG_1234.jpg 1>> out.txt 2>> out.txt  
du -h IMG_9328.jpg IMG_1234.jpg 1> out.txt 2> out.txt
```

- ▶ And later, when we learn how to chain commands together (with pipes):
 - ▶ We can only pipe stdout into the next command

- ▶ **How can we redirect stderr to stdout?**

- ▶ &1 stands for: *current stdout*

```
[command] 2>&1  
[command] > out.txt 2>&1
```

- ▶ [command] could be anything, for example it could be:

```
du -h IMG_9328.jpg IMG_1234.jpg
```

Bash & Linux CLI

Redirecting stderr... why is the order so important?

Can we redirect stderr to stdout?

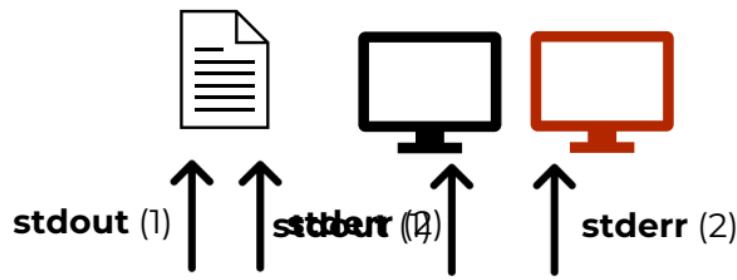
- ▶ Let's compare those 2 commands:
 - ▶ [command] > out.txt 2>&1
 - ▶ [command] 2>&1 > out.txt
- ▶ Let's have a look in the terminal first!

Can we redirect stderr to stdout?

► What are the differences?

► [command] > out.txt 2>&1

- As a first step, stdout is redirected to the file out.txt
- Afterwards, stderr is redirected to the same destination as the current stdout...
- So both outputs redirect their contents into the file out.txt

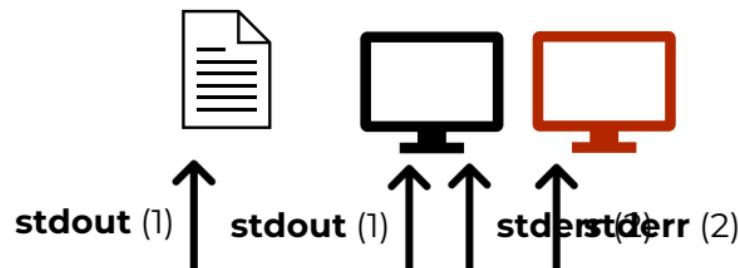


Can we redirect stderr to stdout?

- ▶ What are the differences?

- ▶ [command] `2>&1 > out.txt`

- ▶ At first, stderr is redirected to the same destination as current stdout
- ▶ Meaning: stderr is being redirected to the terminal, all errors will show up there
- ▶ After that, stdout is being redirected into the file out.txt



Can we redirect stderr to stdout? (print version)

► What are the differences?

- ▶ [command] > out.txt 2>&1
 - ▶ As a first step, stdout is redirected to the file out.txt
 - ▶ Afterwards, stderr is redirected to the same destination as the current stdout...
 - ▶ So both outputs redirect their contents into the file out.txt
- ▶ [command] 2>&1 > out.txt
 - ▶ At first, stderr is redirected to the same destination as current stdout
 - ▶ Meaning: stderr is being redirected to the terminal, all errors will show up there
 - ▶ After that, stdout is being redirected into the file out.txt

Bash & Linux CLI

What about stdin?

What about stdin?

- ▶ Can we also redirect a file into stdin?
- ▶ Some programs also accept user input
- ▶ **For example:**
 - ▶ `wc -l`
 - ▶ Let's have a look at this in our terminal!
- ▶ How to we send stdin to a program?
- ▶ **We can also use redirects!**
 - ▶ `wc -l < out.txt`
 - ▶ However, a lot of Unix tools can work with files directly...
- ▶ **This command should be preferred:**
 - ▶ `wc -l out.txt`

Bash & Linux CLI

Motivation: Pipes

Motivation: Pipes

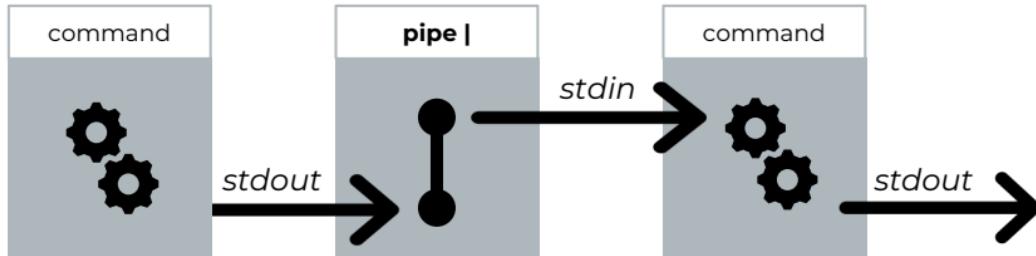
- ▶ Let's start with a simple example...
 - ▶ How to we count the number of files in a directory?
- ▶ **Right now, this would be quite inefficient**
 - ▶ (and the output.txt should even be in a different folder):
 - ▶ `ls > output.txt`
 - ▶ `wc -l output.txt`
 - ▶ `rm output.txt`
 - ▶ Isn't there a more efficient way to achieve the same result?
 - ▶ Yes, there is! In this chapter, you will learn about Pipes in Bash.
 - ▶ Also, this will allow us to use some programs to perform basic string manipulation

Bash & Linux CLI

What is a pipe?

What is a pipe | ?

- ▶ The **pipe** is a mechanism that passes the output of one command as input to another command
- ▶ Thus, you can chain multiple commands together and build more complex functionalities
- ▶ General syntax: command | command | ... | command

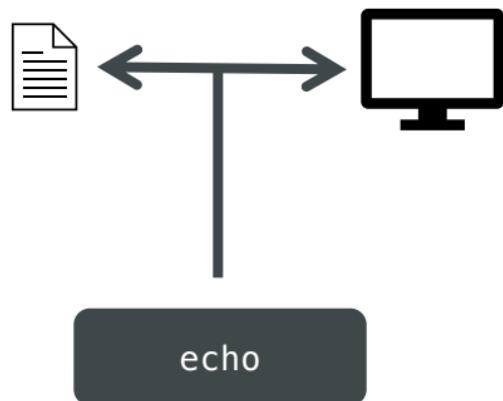


Bash & Linux CLI

Commonly used tool: tee

Commonly used tool: tee

- ▶ With the combination of a pipe and the `tee` command you can create a standard output and write it to a file at the same time
 - ▶ `echo 'Hello World!' | tee hello.txt`
 - ▶ **To append:** `echo 'Hello World!' | tee -a hello.txt`



Bash & Linux CLI

Commonly used tools: sort, uniq

Commonly used tools: sort

- ▶ sort
 - ▶ **sort the contents in a file or stdin:**
 - ▶ By default: Alphabetical order
 - ▶ `sort -r`: sort in reverse alphabetical order
 - ▶ `sort -n`: sort numbers according to numerical order
 - ▶ `sort -c`: check whether the contents in a file are sorted and find unsorted elements
 - ▶ `sort -k column_number (starting at 1)`: sort data by a specific column

Find unique lines

- ▶ Remove duplicate lines:

- ▶ `sort users.txt | uniq`
- ▶ `uniq` only checks for consecutive(!) lines with the same content

- ▶ However, there's also a shorter way:

- ▶ `sort -u users.txt`

- ▶ Find the duplicate lines in a file:

- ▶ `sort users.txt | uniq -d`

Bash & Linux CLI

How to filter the pipe

Filter lines: grep

- ▶ Grep is a tool that can find a pattern in an output or a file
- ▶ **Basic usage:**
 - ▶ `grep -F 'pattern' file.txt`
- ▶ **However, it can also work on stdin:**
 - ▶ `[command] | grep -F 'pattern'`
- ▶ **By default, grep uses "basic regular expressions"**
 - ▶ We haven't looked into regular expressions (regex) yet...
 - ▶ ... so for now, we disable this feature of grep by passing the parameter: `-F`

Grep and binary data

- ▶ Grep works with every file - it will just find the pattern in the file
- ▶ Thus, we could also execute grep on files that contain binary data
- ▶ **But we shouldn't:**
 - ▶ We may find files we are not intending to find
 - ▶ Grep is meant to be used on text files, not on binary files!
 - ▶ Especially when using regular expressions, grep can be extremely slow on binary files
 - ▶ Binary files may contain non-printable characters, or characters that change the behavior of our terminal once printed

Bash & Linux CLI

How to work with strings

How to work with strings?

- ▶ **To replace (on a character level):**

- ▶ using `tr` (**translate**)
 - ▶ `echo 'bash' | tr 'b' 'd'`

- ▶ **Convert upper and lower case:**

- ▶ `echo 'awesome' | tr 'a-z' 'A-Z'`

- ▶ **To delete characters:**

- ▶ `echo 'Bash is amazing' | tr -d ' '`

- ▶ **To reverse a string:**

- ▶ using `rev` (**reverse**)
 - ▶ `echo 'Was it a cat I saw?' | rev`

Bash & Linux CLI

The program: cut

The program: cut

- ▶ **The program cut:**

- ▶ It allows us to process and extract data from a file or standard input

- ▶ **We can use it for different modes:**

- ▶ **Cutting by bytes:**

- ▶ `cut -b`

- ▶ **Example:** `uptime | cut -b 1-10`

- ▶ Here, we select the bytes 1-10 from each line of the input

- ▶ **Cutting by characters:**

- ▶ `cut -c`

- ▶ It otherwise works exactly as we would cut by bytes

- ▶ **Cutting by fields:**

- ▶ `cut -d ' ' -f 1`

- ▶ `uptime | cut -d ' ' -f 1`

- ▶ And we select the first item

- ▶ In case the output starts with a whitespace character... then you would need to use `-f 2`

Bash & Linux CLI

The tool: sed

The tool: sed

- ▶ The tool sed allows us to easily execute commands on a file or on stdin

- ▶ **For example:**

- ▶ `sed 'command1; command2; ...'`

- ▶ **Those commands can modify the string, such as:**

- ▶ delete lines
 - ▶ insert lines

- ▶ **or... replace lines (the most common use case for sed):**

- ▶ `s/[pattern]/[replacement]/[flags]:`

- ▶ Replaces a String with another
 - ▶ The most common flag is "g":
 - ▶ This means we want to replace all occurrences, not just the first

The tool: sed

- ▶ There're several different implementations of sed
- ▶ **For example:**
 - ▶ OS X uses an implementation from FreeBSD from 2005...
 - ▶ Linux usually uses GNU sed
- ▶ **This means for us:**
 - ▶ Most of the code works exactly the same on each platform
 - ▶ However there might be some minor differences
 - (especially when running complex sed commands)

Bash & Linux CLI

Exercise: Analyze a logfile (part 2)

Exercise: Analyze a logfile

- ▶ Let's now analyze the logfile of one of the previous chapters again
- ▶ Btw, you can just download it again, in the next lecture I will provide you the link again
- ▶ **The questions:**
 - ▶ How many requests have downloaded .zip-files?
 - ▶ How many different .zip-files have been downloaded?
- ▶ **Tip:**
 - ▶ You can detect Firefox users by their user agent, it contains "Firefox/"
 - ▶ You can't fully parse the logfile as a logfile - you might have to treat it as lines of text, and for example match the user agent on the whole line of text

Bash & Linux CLI

Solution: Analyze a logfile (part 2)

Bash & Linux CLI

The shell environment: Introduction

The shell environment: Introduction

- ▶ The shell environment is a collection of settings, variables, aliases, and configurations
- ▶ It defines the context in which our programs are run
- ▶ It thus influences how our programs are being run
- ▶ **In this chapter:**
 - ▶ You will learn how the shell environment works
 - ▶ You will understand environment variables, and how you can set, use and modify them
 - ▶ You will learn how you can make persistent changes to your shell environment



Bash & Linux CLI

Environment variables

Overview: Using environment variables

- ▶ **What are environment variables:**
 - ▶ Used to store configuration information and settings
 - ▶ They influence the shell and program behavior
- ▶ **By convention:**
 - ▶ Environment variables are written in uppercase letters
- ▶ **We can list all of them with the following command:**
 - ▶ `env`
- ▶ **Example (to be preferred):**
 - ▶ `echo "${PWD}"`
 - ▶ `echo "${USER}"`
 - ▶ `echo "${PATH}"`
- ▶ **We can also use:**
 - ▶ `echo "$PATH"`
- ▶ **We should avoid:**
 - ▶ `echo $PATH`

Bash & Linux CLI

Important environment variables: **HOME, PWD, USER**

Important environment variables

- ▶ HOME
 - ▶ Stores the current user's home directory path
 - ▶ Example: /home/username or /Users/username
- ▶ PWD
 - ▶ Stores the current working directory path
 - ▶ For the last working directory before: OLDPWD
 - ▶ Not to be confused with the command `pwd`!
- ▶ USER
 - ▶ Stores the current user's username
 - ▶ Example: username

Bash & Linux CLI

How to set / unset environment variables

How to set / unset environment variables

► How do we set an environment variable?

- ▶ We can use the export command to create an environment variable:
 - ▶ `export VAR=value`
- ▶ We can also overwrite an existing variable:
 - ▶ `VAR=new-value`

► How do we delete an environment variable?

- ▶ `unset VAR`
- ▶ This is useful for troubleshooting and cleaning up of the environment

Bash & Linux CLI

The environment variable: PATH

The environment variable: PATH

- ▶ **The environment variable PATH is one of the most important variables in our shell:**
 - ▶ Stores a list of directories
 - ▶ Directories searched for executable programs
 - ▶ **Order matters:** directories searched from left to right
 - ▶ Multiple directories separated by colons (":")
 - ▶ **Let's have a look at my variable PATH**

Bash & Linux CLI

Bonus: Why do we have different PATHs?

Why do we need different paths?

- ▶ **Important definitions:**

- ▶ Filesystem Hierarchy Standard: Unified standard of where to place files
- ▶ Single-user mode: A special way to launch Linux for repairing a broken system - we need access to essential tools

- ▶ **Different paths:**

- ▶ `/bin`: Essential binaries, that need to be always available
- ▶ `/sbin`: Essential binaries, that are usually executed as root, and need to be always available
- ▶ `/usr/bin`: Non-essential binaries, for all users. Could be shared with other computers
- ▶ `/usr/sbin`: Non-essential binaries, usually executed as root. Could be shared with other computers
- ▶ `/usr/local/bin`: Non-essential binaries, for all users. Specific to this host
- ▶ `/usr/local/sbin`: Non-essential binaries, usually executed as root. Specific to this host

Bash & Linux CLI

Modifying the PATH variable

Modifying the PATH variable

- ▶ **Sometimes, we want to modify the PATH variable**
- ▶ **Why?**
 - ▶ For example, we have a separate directory in which we want to install executable files on our system
 - ▶ For example: /opt/homebrew/bin
- ▶ **Usually, we want to append a directory to our PATH:**
 - ▶ `PATH="${PATH}:/new/path"`
 - ▶ **Let's have a look at how PATH works ☺**

Troubleshooting PATH issues

- ▶ **Common issues: "command not found"**

- ▶ In that case, we need to verify the contents of the variable PATH:
- ▶ `echo "${PATH}"`
- ▶ The desired directory should be included
- ▶ The order of the entries in PATH must be correct

- ▶ **If we need to find a program:**

- ▶ `which program`

PATH: Best practices

► **Best practices:**

- ▶ Keep system directories at the beginning
- ▶ Place user-specific directories after system directories
- ▶ Avoid unnecessary duplication of directories
- ▶ Minimize the number of directories to improve search efficiency
- ▶ Regularly review and clean up the PATH
- ▶ Be cautious when modifying the PATH for system-wide changes

Bash & Linux CLI

Bonus: Our own program!

Bash & Linux CLI

Environment variables: Available to programs

Environment variables

► **Important:**

- Environment variables are not a Bash feature
- They are a feature provided by our operating system

► **This is important:**

- This will help us understand why they work seamlessly across programs and why they can "cross" those boundaries

Example: Available to Python

- ▶ Environment variables are automatically available for all programs that we launch
- ▶ We can for example access them, also in other applications
- ▶ **Why is this important?**
 - ▶ Environment variables are often used to pass configuration data to programs
 - ▶ Such as key, URLs, and even passwords
 - ▶ If you run your own application, you might also consider passing in the configuration as environment variables!
- ▶ **Example Python:**
 - ▶

```
import os
print(os.environ)
```

Bash & Linux CLI

The environment variable: SHELL

The environment variable: SHELL

- ▶ **The variable: SHELL:**

- ▶ Stores the path to the user's default shell

- ▶ **Important:**

- ▶ It is inherited just as a normal environment variable
 - ▶ So even if we start a bash...
 - ▶ ... SHELL might still be /bin/zsh

- ▶ **If we wanted to change it:**

- ▶ `chsh -s "/bin/bash"`
 - ▶ It must be a shell that is listed in the file `/etc/shells`
 - ▶ It should then take effect after the next login

Bash & Linux CLI

How can we store our configuration?

How can we store our configuration?

- ▶ Let's say we've made some changes to our PATH, or to other environment variables
- ▶ How can we store it, so that Bash automatically loads those changes?
- ▶ **Unfortunately, there're many different startup files for bash:**
 - ▶ `~/.bash_profile`
 - ▶ `~/.bash_login`
 - ▶ `~/.profile`
 - ▶ `~/.bashrc`
- ▶ **The problem:**
 - ▶ In which one should we put our configuration?
- ▶ **Let's have a look!**

Bash startup mode

- ▶ Bash can start in different startup modes:

- ▶ **Interactive login shell:**

- ▶ We directly log into a computer
- ▶ For example: tty-terminal on our machine / ssh

- ▶ **Interactive non-login shell:**

- ▶ We open a terminal on our desktop environment, or we use "bash" within an existing terminal

- ▶ **Non-interactive non-login shell:**

- ▶ We run a .sh-Script, for example "bash script.sh", or "./script.sh".

- ▶ **Non-interactive login shell:**

- ▶ Very rare. One way would be, if we send a command with ssh to a remote server without starting a shell
- ▶ `echo 'command' | ssh server`

Bash startup files

- ▶ **Interactive login shell:**

- ▶ /etc/profile
- ▶ **First one of:**
 - ▶ ~/.bash_profile
 - ▶ ~/.bash_login
 - ▶ ~/.profile

- ▶ **Interactive non-login shell:**

- ▶ ~/.bashrc

- ▶ **Non-interactive shells (login & non-login):**

- ▶ Bash looks for an environment variable BASH_ENV
- ▶ If found, will try to execute this file (without looking in PATH)

Bash & Linux CLI

How to edit startup files

Bash startup files

- ▶ **Interactive login shell:**

- ▶ /etc/profile
- ▶ **First one of:**
 - ▶ ~/.bash_profile
 - ▶ ~/.bash_login
 - ▶ ~/.profile

- ▶ **Interactive non-login shell:**

- ▶ ~/.bashrc

- ▶ **Non-interactive shells (login & non-login):**

- ▶ Bash looks for an environment variable BASH_ENV
- ▶ If found, will try to execute this file (without looking in PATH)

Bash & Linux CLI

Aliases

Alias: Overview

- ▶ We can use an alias to shorten commands
- ▶ **We can setup a new alias with the following command:**
 - ▶ `alias gohome='cd ~'`
- ▶ **We can list existing aliases with the following command:**
 - ▶ `alias`
- ▶ **We can remove an alias as well:**
 - ▶ `unalias gohome`
- ▶ **Important:**
 - ▶ An alias is only valid within the current session
 - ▶ We can add it to a startup file, so it is loaded every time we start our shell

Bash & Linux CLI

Configuring the shell: set

Configuring the shell: set

- ▶ One way to change the behavior of the shell is the set command
- ▶ It provides pre-defined configuration options that we can use

- ▶ **To enable a feature:**

- ▶ `set -[feature]`

- ▶ **To disable a feature:**

- ▶ `set +[feature]`

- ▶ **What features can we enable / disable?**

- ▶ **set -x:**

- ▶ xtrace: Enables that each command that the shell executes will be printed
 - ▶ Allows us to debug the commands

- ▶ **But there're way more:**

- ▶ https://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html

Bash & Linux CLI

Configuring the shell: shopt

Configuring shell options: shopt

- ▶ We can configure how the shell is behaving in certain scenarios through the `shopt` built-in
- ▶ **To enable a configuration option:**
 - ▶ `shopt -s [optname]`
- ▶ **To disable a configuration option:**
 - ▶ `shopt -u [optname]`
- ▶ **Btw, why do we need set and shopt?**
 - ▶ **set:** Configuration options that are inherited from `sh`
 - ▶ **shopt:** Configuration options that are specific to Bash

Configuration options for shopt

- ▶ shopt allows us to change many configuration options for Bash

- ▶ **Example:**

- ▶ **autocd:**

- ▶ Allows us to change into another directory without the cd command

- ▶ **cdspell:**

- ▶ If a minor spelling error occurs for the cd command - still navigate into the folder

- ▶ **We can see more options in the Bash manual:**

- ▶ https://www.gnu.org/s/bash/manual/html_node/The-Shopt-Builtin.html

Bash & Linux CLI

Project: Colorful bash prompt

The project in this chapter

- ▶ **In this chapter:**

- ▶ We will create a colorful prompt

- ▶ **You will learn:**

- ▶ How to customize the prompt (PS1)
 - ▶ How colors work
 - ▶ Advanced cursor navigation in bash (tput)
 - ▶ And we will then combine this into a colorful prompt

→ **Desktop** |

Bash & Linux CLI

The environment variable: PS1

The PS1 environment variable

- ▶ **PS1 environment variable:**
 - ▶ Stands for "Prompt String 1"
 - ▶ Defines the appearance of the primary shell prompt
 - ▶ Highly customizable with escape sequences and special characters
 - ▶ Enhances user experience and workflow efficiency
- ▶ **Let's have a look at what happens when we change PS1**

PS1 customization

- ▶ **PS1 supports various placeholders:**
 - ▶ Username: \u
 - ▶ Hostname:
 - ▶ Up to the first ".": \h
 - ▶ The complete hostname: \H
 - ▶ Current working directory:
 - \w (full path)
 - \W (last directory of the full path)
 - ▶ Time in 24-hour format: \t
 - ▶ Time in 12-hour format (with am / pm): \@
- ▶ **Example:**
 - ▶ `PS1='\u@\h:\w$ '`

Bash & Linux CLI

Terminal capabilities: color

History: VT100

- ▶ Back in the days, displays used to be different than what we're used to today
- ▶ **As an example, let's look at the VT100:**
 - ▶ This is an ASCII-Terminal, produced between 1978 and 1983
 - ▶ It is not connected through HDMI or anything like that
 - ▶ A computer would send text (and "escape sequences") to this terminal, and the terminal would just display it
 - ▶ It was pretty much just the terminal app that you might be using right now to use bash



Terminal capabilities: TERM

- ▶ Terminals have different abilities
- ▶ We can tell bash which abilities should be used by the environment variable:
 - ▶ `TERM`
- ▶ For example, my terminal contains: `xterm-256color`
- ▶ **There're different terminals available:**
 - ▶ Text-only terminals (no colors)
 - ▶ Terminals, that support 16 different colors
 - ▶ Terminals, that support 256 different colors (`xterm-256color`)
- ▶ **We can list the available modes:**
 - ▶ `toe`
 - ▶ (table of terminfo entries)

Terminal escape sequences

- ▶ In order to control this terminal, we can send "escape sequences":
 - ▶ echo -e "\e[30;40m":
 - ▶ 30: Black foreground
 - ▶ 40: Black Background
 - ▶ echo -e "\e[36;41m":
 - ▶ **36: Cyan foreground**
 - ▶ **41: Red Background**
- ▶ The \e[is the escape character to tell the terminal, that an ANSI escape sequence now starts
- ▶ And the m tells the terminal, that the colors, etc. should be applied to text

FG	BG	Name	VGA
30	40	Black	0,0,0
31	41	Red	170,0,0
32	42	Green	0,170,0
33	43	Yellow	170,85,0
34	44	Blue	0,0,170
35	45	Magenta	170,0,170
36	46	Cyan	0,170,170
37	47	White	170,170,170
90	100	Bright Black (Gray)	85,85,85
91	101	Bright Red	255,85,85
92	102	Bright Green	85,255,85
93	103	Bright Yellow	255,255,85
94	104	Bright Blue	85,85,255
95	105	Bright Magenta	255,85,255
96	106	Bright Cyan	85,255,255
97	107	Bright White	255,255,255

Bash & Linux CLI

Additional terminal capabilities

Terminal escape sequences

- ▶ We can list our terminals capabilities with:
 - ▶ infocmp
- ▶ We then get a list of supported escape sequences
- ▶ Let's have a look and create bold text!

Bash & Linux CLI

Bash: Command substitution

Bash: Command substitution

- ▶ We can collect the output of a program and use it:
 - ▶ `echo "This is the output: $(ls)"`
- ▶ **What does this line do?**
 - ▶ `$(ls)`:
 - ▶ Executes the command "ls", and collects the output

Bash & Linux CLI

The program: tput

tput & escape sequences

- ▶ We can use it **tput** generate escape sequences:

- ▶ **tput clear**: Clears the terminal
- ▶ **tput cup 5 20**:
 - ▶ Moves the cursor to position 5 (vertical) and 20 horizontal
- ▶ **tput bold**: Start bold font
- ▶ **tput sgr0**:
 - ▶ Rest our font / terminal configuration
- ▶ **tput smul/tput rmul**:
 - ▶ Start / end underlined text
- ▶ **tput setaf [color]**: Set foreground color
- ▶ **tput setab [color]**: Set background color

The program: tput

- ▶ We can also use it to query the terminal:

- ▶ `tput colors`
- ▶ `tput cols`
- ▶ `tput lines`

Bash & Linux CLI

The problem: Escape sequences + PS1

The problem: Escape sequence & PS1

- ▶ **The problem:**

- ▶ The Bash needs to know how long our PS1 prompt is
- ▶ This is being calculated automatically

- ▶ **But:**

- ▶ Escape sequences also count as characters
- ▶ This can lead to issues, especially with multi-line commands
- ▶ Let's have a look at those problems!

- ▶ **The solution:**

- ▶ We need to wrap all escape sequences into the following code:
 - ▶ `\[... \]`
- ▶ Then they won't be counted as normal characters

Bash & Linux CLI

Let's combine: tput + PS1

Exercise: tput + PS1

► **Your task:**

- ▶ Create a nice custom PS1 prompt using custom placeholders and tput

▶ **You can find the placeholders here:**

- ▶ https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html

→ Desktop |

Tips

► Tip 1:

- ▶ If you want to use the \${} or \${} commands when assigning PS1,
be sure to use the double quotes:
 - ▶ **Use:** PS1="...before... \$(command) ...after..."
 - ▶ **Don't use:** PS1='...before... \$(command) ...after...'

► Tip 2, be aware:

- ▶ PS1 is only assigned / evaluated once during assignment
- ▶ If you would use something like this:
 - ▶ PS1="\$(ls)"
 - ▶ It would only collect the output of the "ls" program once
(during assignment of PS1) and reuse it
- ▶ Thus, it's best just to use tools like tput in this declaration (and
no commands that could change their output)
- ▶ **And for everything else, you can use the placeholders**

Tips

► Hint 3, also:

- You can disable characters with a backslash (*escaping*):
 - `PS1="\$HELLO"`
 - Will set the environment variable "PS1" to "\$HELLO" and prevent the dollar from referring to a variable

► Suggestion 4:

- You can also add all unicode characters (if your terminal app supports them):



Tips

► Tip 5:

- ▶ At the end of your PS1, be sure to reset the terminal configuration
- ▶ This will allow the command that you can enter to be printed in normal letters
- ▶ `PS1=". $(tput sgr0)"`

Bash & Linux CLI

Solution: tput + PS1

Bash & Linux CLI

Bash: Expansions

Shell Expansions?

- ▶ **In this chapter:**

- ▶ You will learn about Shell Expansions
- ▶ They allow you to easily express complex operations in bash
- ▶ You've used many of them before
- ▶ **You will also learn about the quotes in bash,**

- and the difference between:**

- ▶ `cat $PWD/*.txt`
- ▶ `cat "$PWD/*.txt"`
- ▶ `cat '$PWD/*.txt'`

Bash & Linux CLI

Filename expansion

Shell expansion?

- ▶ Before executing, the shell "rewrites" and parses the command
- ▶ This happens before the command is being executed
- ▶ **Let's have a look at our first expansion:**
 - ▶ Filename expansion

Filename expansion

- ▶ **Filename expansion:**

- ▶ `ls *`
- ▶ In this case, the `*` is being "expanded"
- ▶ For the asterisk, this means Bash gets the contents of the current directory
- ▶ And passes those items as parameters to the `ls` command

- ▶ **This command prints the contents of our current folder:**

- ▶ `echo *`
- ▶ **We can also filter the files that Bash should find for us:**
- ▶ **Example:**
 - ▶ `echo *.txt`
 - ▶ `echo ?.txt`

Bash & Linux CLI

Tilde expansion

Tilde expansion

- ▶ **Tilde expansion:**

- ▶ `ls ~`
- ▶ This lists our home folder
- ▶ By the `~` character, we're using the tilde expansion
- ▶ This character is being expanded to the value of the environment variable `HOME`

- ▶ **We can test this:**

- ▶ `echo ~`

- ▶ **We could also use it with a plus:**

- ▶ `~+`: Expands to `$PWD`
- ▶ `echo ~+`

Bash & Linux CLI

Variable & parameter expansion

Variable expansion

- ▶ The variable expansion allows us to access a variable:
 - ▶ Possible ways to write it:
 - ▶ `echo $HOME`
 - ▶ `echo ${HOME}`
 - ▶ `echo "$HOME"`
 - ▶ `echo "${HOME}"`
 - ▶ Bash rewrites the command for us, and fills in the variable
 - ▶ We should prefer to use double quotes around it
 - (more on that soon)
 - ▶ We should try to use curly braces to make it clear where the variable ends:
 - ▶ `echo "${HOME}path"`
 - ▶ `echo "$HOMEPATH"` (means something different)

Shell parameter expansion

- ▶ Shell parameter expansion allows us to work with strings
- ▶ However, we need to have a variable first!

- ▶ Query the length of a string:
 - ▶ `${#HOME}`
- ▶ Cut out a substring (substring extraction):
 - ▶ `${HOME:start:length}`
- ▶ Replace substring (one time):
 - ▶ `${HOME/pattern/replacement}`
- ▶ Replace substring (all occurrences):
 - ▶ `${HOME//pattern/replacement}`

Bash & Linux CLI

Word splitting

Word splitting

- ▶ **Word splitting:**

- ▶ Bash performs word splitting on our input
- ▶ This happens, after our command has been (potentially) rewritten by expansions

- ▶ **Example:**

- ▶ touch a.txt b.txt
- ▶ **This command will be splitted into 3 different words:**
 - ▶ touch
 - ▶ a.txt
 - ▶ b.txt
- ▶ The first entry will then be the program (**touch**)
- ▶ And **a.txt** the first parameter / argument
- ▶ And **b.txt** will be the second parameter / argument

Word splitting

► Where will it happen?

- ▶ Word splitting will occur at any character that is listed in the variable IFS
- ▶ By default, this is at every space, every tab, and every newline
- ▶ Sequences of IFS characters are treated as a single delimiter
- ▶ Thus, this would be equivalent:
 - ▶ `touch a.txt b.txt`
 - ▶ `touch a.txt b.txt`

Word splitting

- ▶ We can disable word splitting by wrapping parts of the command into quotes:
 - ▶ touch 'a file.txt'
 - ▶ touch "a file.txt"
- ▶ Without word splitting:
 - ▶ 2 files would've been created ('a' and 'file.txt')
 - ▶ Word splitting is disabled for characters between those quotes
- ▶ Thus, the filename will be:
 - ▶ a file.txt
- ▶ But are the quotes really that simple? ;)

Bash & Linux CLI

What more do the quotes achieve?

A quick heads-up

- ▶ My coworker came up with the perfect way to describe how quoting works in Bash:
 - ▶ "*The more you know about programming, the more confusing it is*"
- ▶ So... try to forget everything you've learned from other programming languages... and let's start from scratch ☺

What do the quotes do?

- ▶ **What's the difference between the following commands?**

- ▶ `cat $PWD/*.txt`
- ▶ `cat '$PWD/*.txt'`
- ▶ `cat "$PWD/*.txt"`

- ▶ **No quotes:**

- ▶ All available shell expansions are being applied

- ▶ **Single quotes:**

- ▶ All expansions are disabled, word splitting is disabled

- ▶ **Double quotes:**

- ▶ Disables most expansions, such as tilde expansion (~), filename expansion (*), word splitting,...
- ▶ However certain expansions are still enabled: Variable and parameter expansion are still working

What do the quotes do?

- ▶ The quotes only define how Bash will expand / split the command
- ▶ **They do nothing else!**
- ▶ **Thus, this would be a completely valid bash command:**
 - ▶ `echo 'hello'"world"`
- ▶ **Or this:**
 - ▶ `'echo' 'hello world'`

Bash & Linux CLI

Expansions: Be careful!

Expansions: Be careful!

- ▶ **Be careful:**

- ▶ **The process is always the same:**

1. First, the command is being expanded
2. Then, word splitting is applied
3. Quotes are being removed
4. Command is being executed

- ▶ This also applies to the program name that we want to execute!

- ▶ **Example:**

- ▶ Let's say we got a folder and the filename of the first file was echo

- ▶ **Then this would be a valid command:**

- ▶ *

Expansions: Be careful!

- ▶ You should use double quotes around variables
- ▶ Otherwise, word splitting will occur
- ▶ **Try this in a path that contains a space:**
 - ▶ `touch $PWD/file.txt`
 - ▶ `touch "${PWD}/file.txt"`

Best practice

- ▶ Try to refer to filenames in the same directory as `./file.txt`
- ▶ And, for the quotes:
 - ▶ Always use the quoting style that is as restrictive as possible
 - ▶ Thus:
 - ▶ Prefer single quotes: `'hello world'`
 - ▶ If this is not possible, use double quotes:
 - ▶ `echo "hello ${USER}"`
 - ▶ `echo 'hello '\"${USER}"'`
 - ▶ Use no quoting only if there's no ambiguity, or you want all expansions to be applied:
 - ▶ Example for no ambiguity:
 - ▶ `ls -al` (would be annoying: `'ls' '-al'`)
 - ▶ Here we want all expansions:
 - ▶ `echo ./*.txt`

Bash & Linux CLI

Escaping

Escaping

- ▶ Sometimes we want to disable Bash from performing the default actions

- ▶ **Example:**

- ▶ We autocomplete a filename that contains a whitespace character

- ▶ **Bash might autocomplete it to:**

- ▶ `cat a\ file.txt`

- ▶ **What does the backslash (\) do?**

- ▶ It disables the normal behavior of the next character

- ▶ Here, that would be word splitting

- ▶ Thus, '`a file.txt`' is the filename, that will be passed as an argument to `cat`
(instead of 2 arguments: `a` and `file.txt`)

- ▶ **Escaping would also allow us to print a single double quote in the following way:**

- ▶ `echo \"\""`

- ▶ Though I would prefer to write it in this way: `echo ''''`

Escaping & quotes

- ▶ **But:**

- ▶ Escaping is also a feature that "rewrites" / expands our command as well
- ▶ The single quotes disable all "rewrites" / expansions
- ▶ Thus, also the backslash is disabled

- ▶ **Thus, this one does not print out a single single quote:**

- ▶ `echo '\''`
- ▶ Instead, this command is not yet finished, as the last single quote has not yet been terminated

Bash & Linux CLI

Brace expansion

Brace expansion

- ▶ As of **Bash 4**, it is possible to automatically expand strings of characters (brace expansion):
 - ▶ {item1,item2}
 - ▶ echo data.{csv,txt} -> data.csv data.txt
 - ▶ {start..end}
 - ▶ echo {A..Z}
 - ▶ echo {a..z}
- ▶ Brace expansion **does not** work within quotes
(neither double nor single quotes)!
- ▶ **But of course, this would work:**
 - ▶ echo 'test'{.txt,.csv}

Bash & Linux CLI

Command substitution

Command substitution

- ▶ We can also execute a command and use the output as a replacement
- ▶ For this, we can use command substitution: \$(...)

- ▶ **Example:**

- ▶ `echo "$(cat file.txt)"`
- ▶ It's best to still have double quotes around it, to disable word splitting
- ▶ This will execute the command in a subshell environment and collect the output
(subshell = new bash process)

- ▶ **Sometimes this can be useful:**

- ▶ `echo 'The size of my home directory is: '"$(du -sh ~)"'`
- ▶ `echo 'There'"'"'re '"$(ls | wc -l)"' files in the current directory'`

- ▶ **Another way to write it are the backticks:**

- ▶ `echo `cat file.txt`"`
- ▶ **Though this is considered less readable, and I would not recommend using it**

Bash & Linux CLI

Process substitution

Process substitution

- ▶ Process substitution allows us to use the input or the output of a process as a temporary file
- ▶ **To use the output of a process as a temporary file:**
 - ▶ <(command)
 - ▶ **Example (will print out the filename):**
 - ▶ echo <(ls)
 - ▶ **We can also use it in the following way (and combine it with a redirect):**
 - ▶ wc -l < <(ls)
- ▶ **To use a temporary file as the input to a command:**
 - ▶ >(command)
 - ▶ **Example (combined with a redirect):**
 - ▶ echo "test" > >(cat)

Bash & Linux CLI

Expansions: Outlook

Expansions: Outlook

► You now know:

- ▶ Why and when you should use quotes in Bash
- ▶ And you have seen the most important expansions that you need for day to day use
- ▶ This will allow you to confidently use Bash

► But:

- ▶ We have just explored the most important ways to expand our commands
- ▶ Quite a few expansions support additional parameters

► You can have a look at those here:

- ▶ https://www.gnu.org/software/bash/manual/html_node/Shell-Expansions.html
- ▶ Also, we will be using those expansions a lot, once we start learning about bash scripting / .sh files

Bash & Linux CLI

Part 3: Linux

Part 3: Linux

► In the next few chapters:

- ▶ We will explore in more detail how files work on Linux, what the different directories do, what a device is,...
- ▶ We will also look at user management and explore how file permissions decide if a user can access a certain file
- ▶ We will explore (in detail) how package management works on Ubuntu vs. CentOS
- ▶ We will have a look at the boot procedure and how `systemd` manages the system start
- ▶ We will investigate mounts and how they allow you to access additional drives
- ▶ We will learn how networking works and what tools allow us to manage this on Linux
- ▶ We will setup our VM so that we can connect to it through SSH

Part 3: Linux

► Additional topics:

- ▶ LVM: Logical Volume Manager
- ▶ Cronjobs: Run tasks at specific intervals
- ▶ SELinux: Further enhance the security of your system

▶ Linux distributions:

- ▶ Why are there so many?
- ▶ What are the main differences?

▶ Bonus:

- ▶ Deployment of a LAMP server (for web developers)
- ▶ LAMP = Linux, Apache, MySQL, PHP
- ▶ Setting up a firewall with firewalld

Bash & Linux CLI

Introduction: Files on Unix

Files on Unix



- ▶ **In this chapter:**
 - ▶ We will have a look at the folder structure on Linux systems
 - ▶ After that, we will have a look at what a file is:
 - ▶ This sounds simple, but there're special kinds of files
 - ▶ **As an example:**
 - ▶ Symlinks (symbolic links)
 - ▶ Files that represent devices
- ▶ **This chapter is important:**
 - ▶ To give us an understanding of how the files are organized
 - ▶ To understand symbolic links and their implications
- ▶ **We will also investigate potential problems:**
 - ▶ Our drive might be "full" even though enough disk space is available
(inode limit)

Bash & Linux CLI

Files on Unix systems

What is a file?



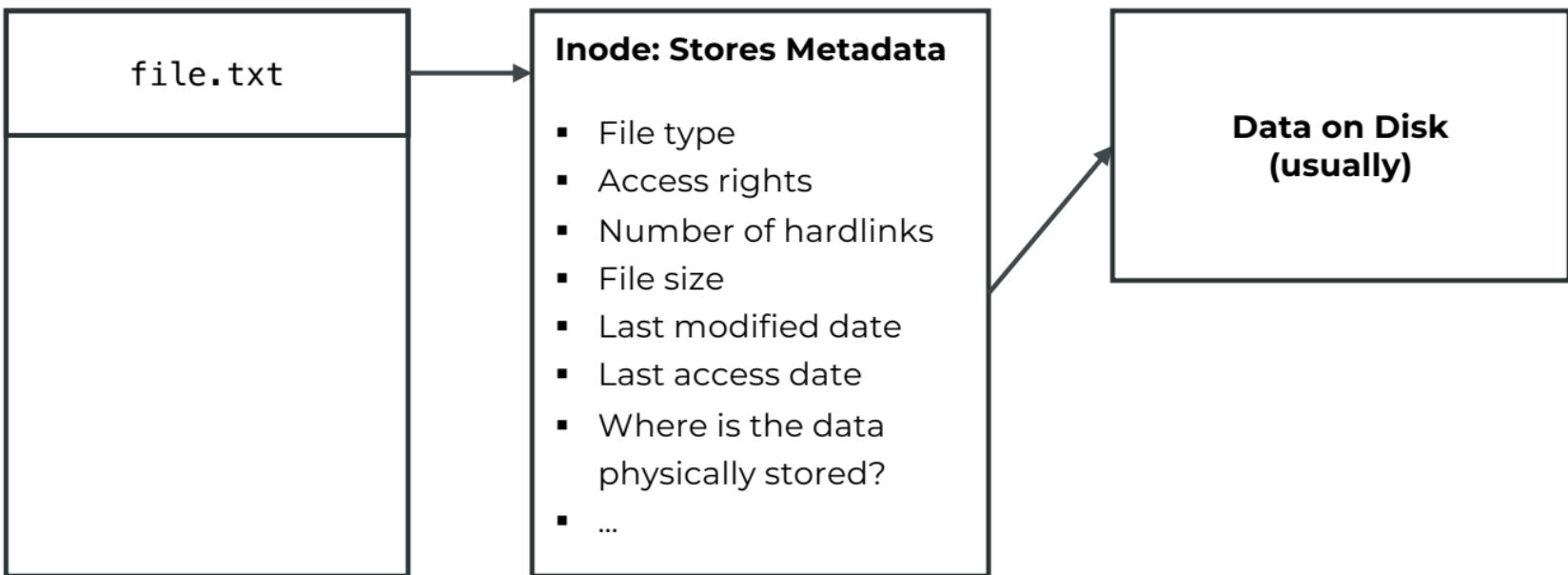
- ▶ **What is a file?**

- ▶ A container for storing, accessing and / or managing data
- ▶ Typically associated with a unique identifier or filename
- ▶ This name, combined with its path, provides a unique location for each file in a filesystem.

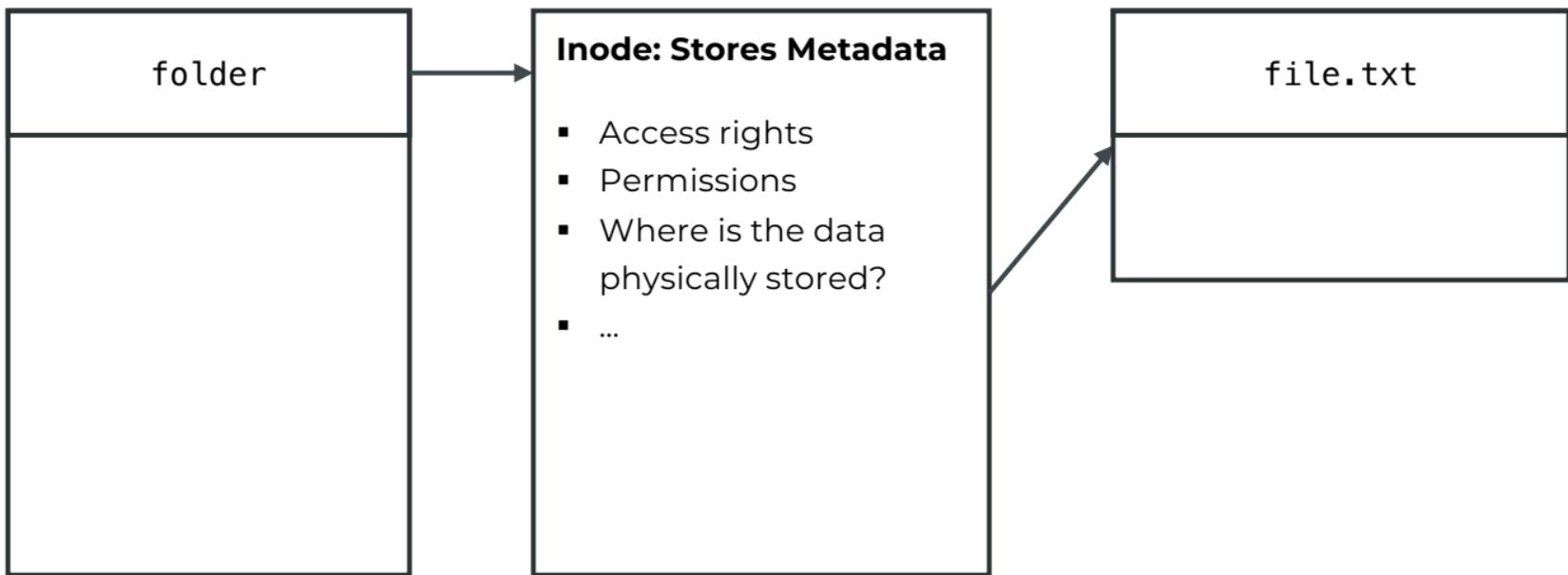
- ▶ **Files can have various attributes (stored in inode):**

- ▶ **Size:** The amount of data stored in the file
- ▶ **Permissions:** Who can read, write, or execute the file
- ▶ **Ownership:** Which user and group owns the file
- ▶ **Timestamps:** When the file was created, last accessed, or modified

How is the data stored?



How does a folder work?



Files on Unix



- ▶ **Files on Unix systems:**

- ▶ In Linux and Unix-like systems, (almost) everything is considered a file
- ▶ This is part of the Unix philosophy

- ▶ **For example:**

- ▶ Ordinary file (-)
- ▶ Directories (d)
- ▶ Symbolic Links (l)
- ▶ Character device (c)
- ▶ Block device (b)
- ▶ Named pipes (p)
- ▶ Sockets (s)

- ▶ **We can show the type of a file with the ls command:**

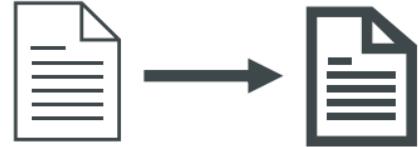
- ▶ `ls -l [folder / file]`
- ▶ The type then shows up as the first character of the first column:

```
-rw-rw-r-- 1 janniss janniss 9019 Apr 29 11:46 document.odt
```

Bash & Linux CLI

What is a symlink?

What is a symlink?



- ▶ A symlink (symbolic link) is a special kind of file on Unix systems

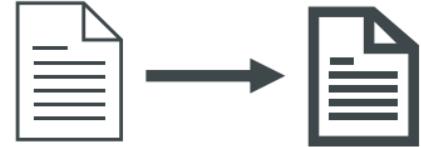
- ▶ **Purpose:**

- ▶ It serves as a reference to another file or directory
- ▶ It's a special way of "shortcut" to another destination

- ▶ **The idea:**

- ▶ We create a special file, that contains a reference to the destination path
- ▶ This reference is being resolved on access of the symlink (or a file within it)
- ▶ This affects read and write operations

What is a symlink?



- ▶ **How to create a symlink?:**

- ▶ We can use the `ln -s` command for this
- ▶ `ln` is a tool to make links between files
- ▶ We can use it to create symlinks

- ▶ **Example:**

- ▶ `ln -s target link`

- ▶ **How do we see if a file is a symlink?**

- ▶ We can use the `ls` command:
- ▶ `ls -l`
- ▶ If the first character is an "l", this file is a symlink:
- ▶ `lrwxrwxrwx`

Bash & Linux CLI

Bonus: Symlinks on Windows

Bonus: Symlinks on Windows



- ▶ This lecture is completely optional
- ▶ **You can use symlinks on Windows as well:**
 - ▶ On Windows, symlinks are quite often called "soft links"
 - ▶ If you want to create one, you can open the PowerShell
 - ▶ And we can use the following command to create a new soft link:
 - ▶ `New-Item -ItemType SymbolicLink -Path Symlink.rtf -Target .\Document.rtf`

Bash & Linux CLI

Filesystem Hierarchy Standard

Filesystem Hierarchy Standard

► **Definition:**

- The Filesystem Hierarchy Standard (FHS) defines the directory structure and directory contents in Unix-like operating systems.
- It provides a consistent and predictable location for specific types of files and directories

► **Purpose:**

- Ensures compatibility across different distributions.
- Makes it easier for users, administrators, and developers to locate files

What are the important folders? (Part 1)

- ▶ **/:**
 - ▶ The root directory of the hierarchy
- ▶ **/bin:**
 - ▶ Contains essential command binaries required for users
 - ▶ **Nowadays:** Being merged into /usr/bin (usrmerge)
- ▶ **/boot:**
 - ▶ Contains files for the bootloader
- ▶ **/dev:**
 - ▶ Contains device files that represent hardware and software devices.
 - ▶ **Examples:** /dev/null, /dev/sda (first SATA drive), /dev/tty (terminal device file)
- ▶ **/etc:**
 - ▶ Contains system-wide configuration files and directories
 - ▶ These files are generally text files that can be edited for system configuration

Bash & Linux CLI

Filesystem Hierarchy Standard (2)

What are the important folders? (Part 2)

- ▶ **/home:**
 - ▶ Contains personal directories for users
 - ▶ User-specific settings, documents, and other files are stored here
- ▶ **/lib:**
 - ▶ Contains library files that supports the binaries located under /bin and /sbin
 - ▶ Depending on the system, we might also have additional lib folders for additional architectures
 - ▶ **Example:** /lib32, /lib64
 - ▶ Nowadays: Being merged into /usr/lib (usrmerge)
- ▶ **/media:**
 - ▶ Contains mount points for removable storage media
- ▶ **/mnt:**
 - ▶ Mount points for additional filesystems
- ▶ **/opt:**
 - ▶ Optional application software packages can be stored here

Bash & Linux CLI

Filesystem Hierarchy Standard (3)

What are the important folders? (Part 3)

- ▶ **/proc:**
 - ▶ Virtual filesystem (usually procfs)
 - ▶ Provides information about processes and the kernel
- ▶ **/root:**
 - ▶ Contains the personal data for the root user (home folder of the root user)
- ▶ **/run:**
 - ▶ Run-time data
 - ▶ Files here will be removed / emptied during boot, or will be discarded on shutdown
- ▶ **/sbin:**
 - ▶ Contains essential system binaries that are generally used by the root user
 - ▶ **Nowadays:** Being merged into /usr/sbin (usrmerge)
- ▶ **/srv:**
 - ▶ Files for services (if we don't store them in /var)
 - ▶ Quite often, data offered by FTP servers
- ▶ **/sys:**
 - ▶ Information about devices, drivers and kernel features

Bash & Linux CLI

Filesystem Hierarchy Standard (4)

What are the important folders? (Part 4)

- ▶ **/tmp:**
 - ▶ Contains temporary files created by system and users
 - ▶ These files are typically deleted on reboot
- ▶ **/usr:**
 - ▶ Contains shareable, read-only data
 - ▶ This includes system binaries, libraries, documentation, and source-code for various system programs
 - ▶ Usually, files in this folder could be shared between multiple computers
- ▶ **But:**
 - ▶ **/usr/local:**
 - ▶ This folder is for files that should not be shared between multiple computers
- ▶ **/var:**
 - ▶ Contains variable data files such as logs, databases, websites, and email, among other things
 - ▶ This directory's contents changes as the system runs

Bash & Linux CLI

Background: The `usrmerge` project



What is the usrmerge project?

- ▶ **We can answer the following question:**

- ▶ Why is /bin identical to /usr/bin?
- ▶ Why is /sbin identical to /usr/sbin?
- ▶ Why is /lib identical to /usr/lib?

- ▶ **Usrmerge Project:**

- ▶ A project to simplify the filesystem hierarchy on linux systems
- ▶ It merges several directories that traditionally exist on the root level (/) into their counterparts in /usr

- ▶ **Back then:**

- ▶ It was important to differentiate between /lib and /usr/lib
- ▶ Reasoning: Back then, we might have wanted to store /usr on a different, physical drive
- ▶ /bin, /sbin, /lib would only contain applications that are absolutely necessary to fix a broken system

- ▶ **Nowadays, this is no longer necessary:**

- ▶ Thus, we can now simplify our directory structure

Bash & Linux CLI

What is a device?



What is a device?

- ▶ **Devices are files???**

- ▶ **Idea:**

- ▶ "*Everything is a file*"
 - ▶ "*Everything is a stream of bytes*" (Linus Torvalds)

- ▶ **How it works:**

- ▶ (Almost) all hardware devices are represented as a "file"
(or to be more precise: as a stream of bytes)
 - ▶ Through this "file", we can enable access to the underlying hardware,
without knowing its technical details

What is a device?

- ▶ A device refers to a physical or virtual entity that can be accessed through a file-like interface
- ▶ Devices in Unix serve as the interface between the operating system and various hardware or virtual components
- ▶ They allow applications and users to interact with these components by reading from and writing to their corresponding device files

What is a device?

- ▶ **What kind of devices does Unix support?**

- ▶ **Character devices (c):**

- ▶ We gain unbuffered, direct access to the hardware
 - ▶ Usually, we can access those devices by reading a byte (character)
 - ▶ Though there might be additional restrictions / requirements for certain character devices

- ▶ **Block devices (b):**

- ▶ We gain buffered access to the hardware
 - ▶ Multiple bytes are bundled into a block
 - ▶ And we can access this device through accessing those blocks

- ▶ **Pseudo devices:**

- ▶ Those are devices that don't necessarily refer to a physical device
 - ▶ Depending on the type, they may show up as a block device or a character device

Bash & Linux CLI

Important pseudo devices

Important pseudo devices

- ▶ **Important pseudo devices:**

- ▶ **/dev/null:**

- ▶ When reading: returns EOF (end-of-file)
 - ▶ When writing: Discards the information

- ▶ **/dev/random:**

- ▶ Produces a stream of random numbers
 - ▶ Only produces random data, as long as enough "environmental noise" is available

- ▶ **/dev/urandom:**

- ▶ Just as /dev/random, but:
 - ▶ Always produces data, and may reuse already used "environmental noise"

- ▶ **/dev/stdin, /dev/stdout, /dev/stderr:**

- ▶ Facilitates input / output / errors for normal Unix programs

Bash & Linux CLI

Important /proc files (procfs)

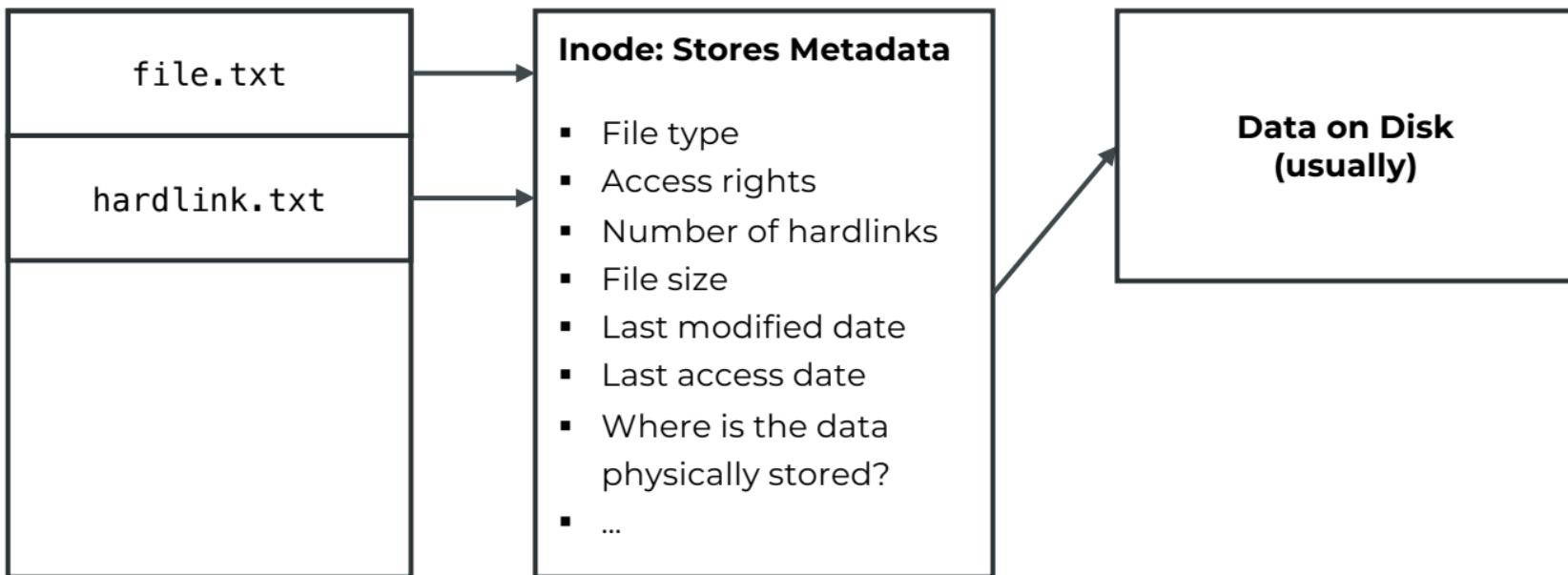
Important /proc files (procfs)

- ▶ If we want to inspect our system, we can look into files in the /proc folder:
 - ▶ **/proc/cpuinfo:**
 - ▶ Contains information about the system's CPU(s), like vendor, model, and speed
 - ▶ **/proc/meminfo:**
 - ▶ Displays the current memory usage of the system, including physical and swap memory
 - ▶ **/proc/version:**
 - ▶ Contains version information about the kernel, gcc, and the operating system
 - ▶ **/proc/uptime:**
 - ▶ Displays the time the system has been up (in seconds) and the amount of time it has been idle
 - ▶ **/proc/loadavg:**
 - ▶ Shows the system load averages for the past 1, 5, and 15 minutes, number of currently running processes / and number of threads, and the last PID

Bash & Linux CLI

What is a hard link?

What is a hardlink?



Hardlinks

► **What is a hardlink?**

- ▶ A hardlink is a directory entry or reference to an existing inode
- ▶ Technically, the first filename of a file is already a hardlink
- ▶ But one file can have multiple hardlinks

▶ **Thus:**

- ▶ Hardlinks behave as if they were the same file
- ▶ A hardlink can only link to files on the same filesystem
- ▶ The filesystem must support additional hardlinks

▶ **If we delete a hardlink:**

- ▶ The other filenames / hardlinks remain intact
- ▶ The data is only deleted if all hardlinks are removed

▶ **How can we create a hardlink?**

- ▶ We can use the `ln` command for this (without the `-s`):

- `▶ ln target hardlink`

▶ **Important:**

- ▶ Hardlinks cannot be created for directories (to prevent loops)

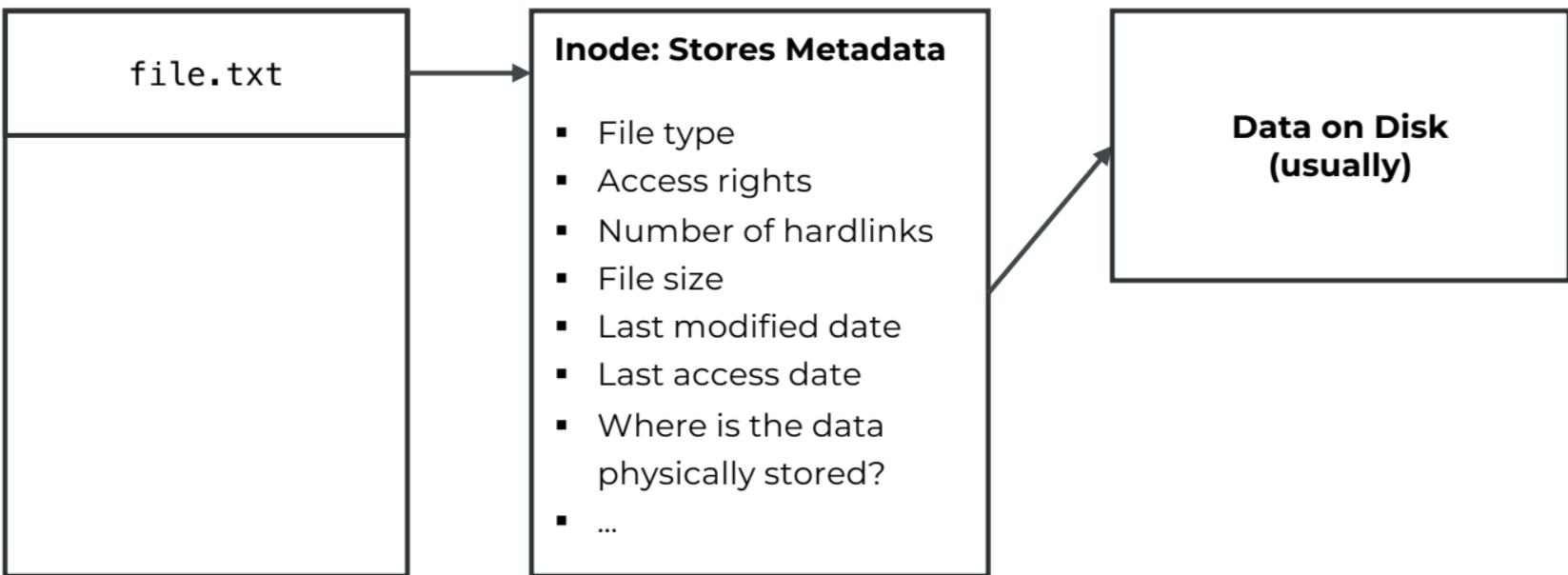
Copy files with a hard link

- ▶ **Copy whole folder structure with hard links:**

- ▶ `cp -al source dest`
- ▶ This will copy the whole source folder, and create hard links for all files
- ▶ We will not need any additional storage for this
- ▶ We can now organize the files in multiple different folder structures, without needing any additional disk space

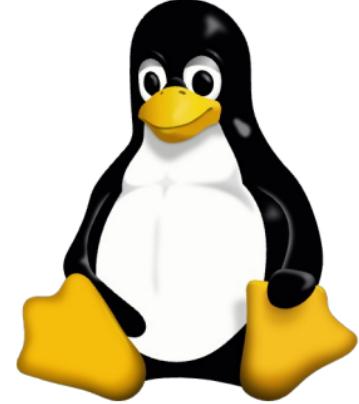
Bash & Linux CLI

Troubleshooting: The inode limit



Increasing the inode limit

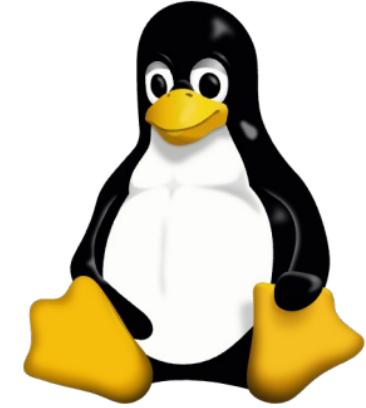
- ▶ During the creation of a filesystem, space is reserved for inodes
- ▶ This space can't be used for anything else
- ▶ **We can show how many inodes are being used:**
 - ▶ `df -ih`
- ▶ Usually, the limit should be sufficiently high, so we'll never reach it
- ▶ **But if your application uses many small files:**
 - ▶ This can be (theoretically) reached
 - ▶ Processes or applications crash or can't be started
 - ▶ Operating system crashes
- ▶ **To solve this:**
 - ▶ You can remove files that are no longer needed
 - ▶ You can compress multiple files into an archive (for example: `.tar`)
 - ▶ You can recreate your filesystem with a higher inode limit
 - ▶ You can store data on additional drives (and mount them)



Bash & Linux CLI

User management in Linux

User management in Linux



- ▶ You will learn:

- ▶ Managing users:

- ▶ Creating new users: `useradd`
 - ▶ Setting and resetting passwords: `passwd`
 - ▶ Deleting users: `userdel`
 - ▶ Modifying user details: `usermod`
 - ▶ Managing privileges: Configure `sudo` and `/etc/sudoers`

- ▶ Managing groups:

- ▶ Organizing users into groups: `groupadd`, `groupdel`, `groupmod`

- ▶ Managing files:

- ▶ Understand file and directory permissions

- ▶ Important:

- ▶ User management works differently on a Mac. We will only investigate user management on Linux systems

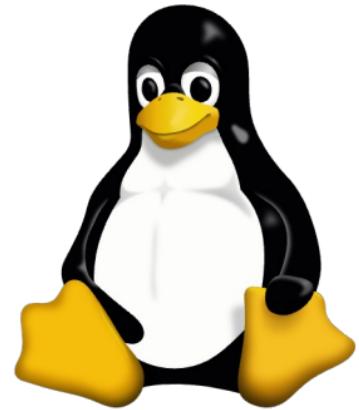
User management in Linux

- ▶ **This chapter is important for:**
 - ▶ **System administrators:**
 - ▶ Managing users, permissions, and security.
 - ▶ **IT professionals:**
 - ▶ Ensuring secure access and resource sharing.
 - ▶ **DevOps engineers:**
 - ▶ Configuring user environments and access controls.
 - ▶ **Technical support specialists:**
 - ▶ Troubleshooting user-related issues.



Bash & Linux CLI

Managing users



How do users in Linux work?

- ▶ **Linux has different kind of users:**

- ▶ **Root user:**

- ▶ Highest privileges
 - ▶ It has the user ID: 0
 - ▶ There can only be one root user on the system

- ▶ **Regular users:**

- ▶ Limited privileges
 - ▶ We can allow regular users to temporarily get root access through sudo

- ▶ **Service users:**

- ▶ For specific tasks
 - ▶ This allows us to safely run a webserver, database,...

- ▶ **Groups:**

- ▶ All users have a primary group
 - ▶ And can be assigned to zero to unlimited additional groups

Managing users



- ▶ **On Linux, user information is stored in various files:**

- ▶ **/etc/passwd:**

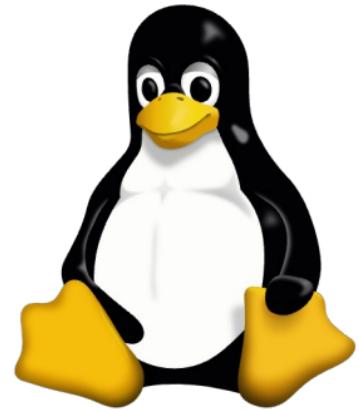
- ▶ Contains basic user account information
- ▶ Username, user ID (UID) , group ID (GID), user description (full name), home directory and default shell
- ▶ Readable by all users

- ▶ **/etc/shadow:**

- ▶ Stores encrypted user passwords and password aging information
- ▶ Also stores additional information, such as the date of the last password change, expiry dates,...
- ▶ Readable only by the root users (or users with root privileges)

- ▶ **/etc/group:**

- ▶ Contains information about the groups, and their members
- ▶ Readable by all users



Bash & Linux CLI

Adding users: useradd, passwd

Managing users: useradd

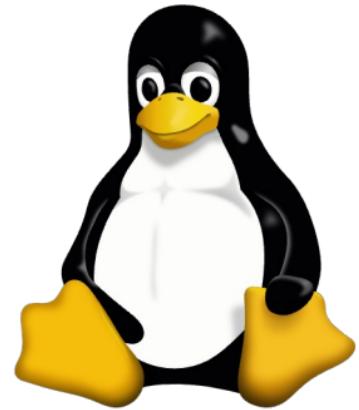


- ▶ With the useradd command, we can create new users
 - ▶ Syntax: `useradd [options] username`
 - ▶ **The most important options are:**
 - ▶ `-m`: Create home directory
 - ▶ `-d`: Set custom home directory
 - ▶ `-s`: Specify default shell
 - ▶ **Manage groups:**
 - ▶ `-g`: Specify primary group instead of using the default configuration
 - ▶ Often, default configuration is that a new group will be created with the same name as the username
 - ▶ `-G`: Add user to secondary groups
 - ▶ **Let's see this in action and create a new user 😊**

Managing users: passwd

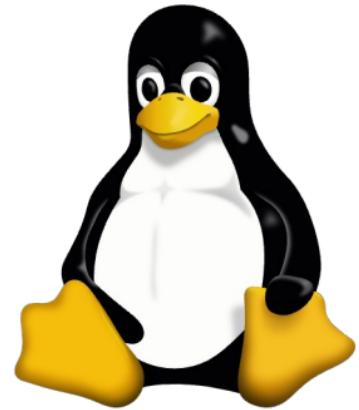
- ▶ With the passwd command, we can set the password for users
 - ▶ Syntax: `passwd [options] [username]`
 - ▶ **The most important options are:**
 - ▶ `-S`: Display password status
 - ▶ `-d`: Delete password
 - ▶ `-n`: Set minimum password age (days)
 - ▶ `-x`: Set maximum password age (days)
 - ▶ **We can also use it to lock / unlock the account:**
 - ▶ `-l`: Lock user account
 - ▶ `-u`: Unlock user account





Bash & Linux CLI

Bonus: Password expiration?



Bash & Linux CLI

Changing users: usermod

Managing users: usermod

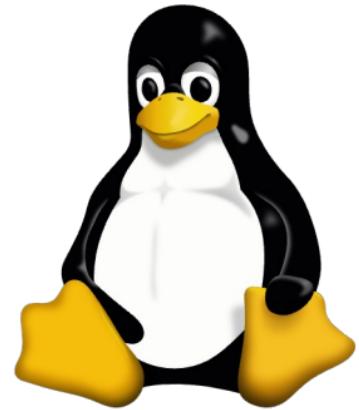


- ▶ With the usermod command, we can modify another user's details
 - ▶ Syntax: `usermod [options] username`
 - ▶ **The most important options are:**
 - ▶ `-c`: Change user description (full name)
 - ▶ `-s`: Change default shell
 - ▶ **We should really consider if we want to use those options:**
 - ▶ `-d`: Change home directory (`-m` to also move the existing home directory to the new location)
 - ▶ `-l`: Change username
 - ▶ **Manage groups:**
 - ▶ `-g`: Change primary group
 - ▶ `-G`: Change secondary groups
 - ▶ `-aG`: Add secondary group

Changing default shell

► Repetition:

- If the user wants to change their default shell:
 - It must be one of /etc/shells
 - `chsh -s /bin/bash`



Bash & Linux CLI

Deleting users: userdel

Deleting users: userdel

- ▶ **We can also delete an existing user:**

- ▶ `userdel [options] username`

- ▶ **Example:**

- ▶ `userdel max`

- ▶ **There're quite a few options:**

- ▶ **-r:**

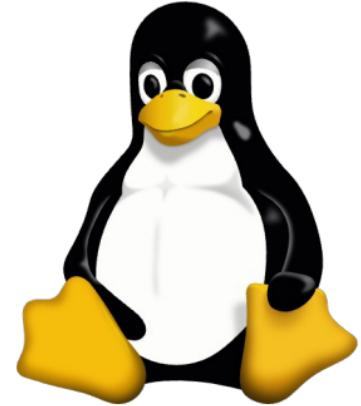
- ▶ Removes the home directory + mails

- ▶ **-f:**

- ▶ Also removes home directory + mails

- ▶ Forces the removal of the user, even if the user is still logged in

- ▶ Might also delete a group with the same name as this user
(depending on system configuration)



Bash & Linux CLI

How do groups work?

Groups on Linux systems

- ▶ **Motivation:** Why do we need groups on Linux systems?
- ▶ **They help us with:**
 - ▶ Organizing users with similar access rights
 - ▶ Simplifying permission management
 - ▶ Enhancing collaboration and resource sharing
 - ▶ Controlling access to files and directories
- ▶ **Thus, overall:**
 - ▶ Strengthening system security
- ▶ **We will later see:**
 - ▶ How we can give a group access to a specific command
 - ▶ How we can give a group access to specific files and directories



How do groups work?

- ▶ **How do groups work?**

- ▶ Each user has a primary, and zero to many secondary groups

- ▶ **Primary group:**

- ▶ Stored in /etc/passwd
 - ▶ Default ownership for new files

- ▶ **We can test this:**

- ▶ touch file.txt
 - ▶ ls -al file.txt

- ▶ **Secondary group(s):**

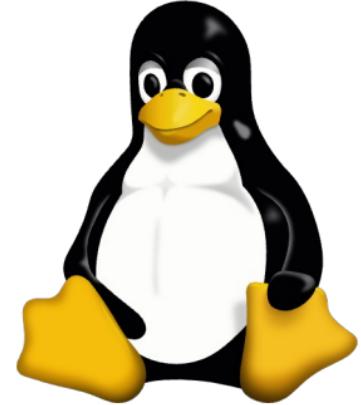
- ▶ Multiple memberships allowed
 - ▶ Stored in /etc/group
 - ▶ This allows us to give this user fine grained access rights to our system

- ▶ We can list a user's groups with the following command:

- ▶ `groups [username]`

Existing groups in Ubuntu

- ▶ There're quite a few existing groups on Ubuntu
- ▶ A selection of the most important groups:
 - ▶ **root**: The superuser group with administrative privileges, allowing complete control over the system.
 - ▶ **sudo / wheel**: Members can use sudo. May also be called "wheel".
 - ▶ **adm**: Allows members to read log files
 - ▶ **lpadmin / lp**: Members may manage printers and print queues (CUPS). May also be called "lp"
 - ▶ **www-data**: A group for web server processes (such as Apache or Nginx), gives access to web content
 - ▶ **plugdev**: Allow this user to manage pluggable devices (USB sticks, external HDDs,...)

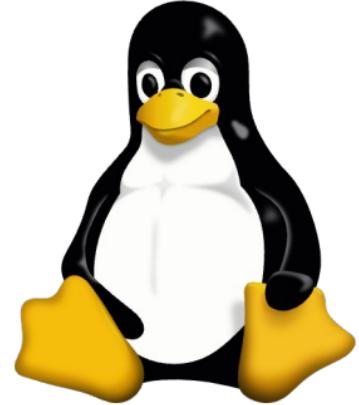


Bash & Linux CLI

Add a user to a group

How do we add a user to a group?

- ▶ We can use the usermod command for this!
- ▶ Syntax: `usermod [options] username`
- ▶ **Manage groups:**
 - ▶ `-g`: Change primary group
 - ▶ `-G`: Change secondary groups
 - ▶ `-aG`: Add secondary group
- ▶ **Depending on the system, we might also be able to use additional tools:**
 - ▶ `adduser [user] [group]`
 - ▶ `deluser [user] [group]`



Bash & Linux CLI

Creating our own groups

How can we create a group?

- ▶ **How can we create a group?**

- ▶ We can use the groupadd command
- ▶ Syntax: `groupadd [options] groupname`

- ▶ **Important option:**

- ▶ `-g`: option to set custom GID
- ▶ **Example:** `groupadd -g 1005 newgroup`
- ▶ The group info is then stored in the `/etc/group` file

How can we change a group?

- ▶ **How can we change a group?**

- ▶ We can use the groupmod command
- ▶ Syntax: `groupmod [options] groupname`

- ▶ **Important option:**

- ▶ `-n`: Change the name of the group
- ▶ `-g`: Change the group Id (GID)
- ▶ This will then update `/etc/group` and `/etc/passwd`

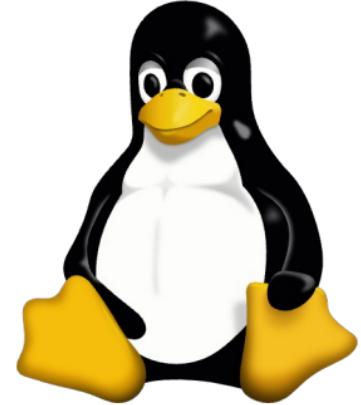
How can we delete a group?

- ▶ **How can we delete a group?**

- ▶ We can use the `groupdel` command
- ▶ Syntax: `groupdel groupname`

- ▶ **Important:**

- ▶ This command does not delete group-owned files
- ▶ This command will update `/etc/group` and `/etc/passwd`
- ▶ This will fail, if this group is the primary group of a user



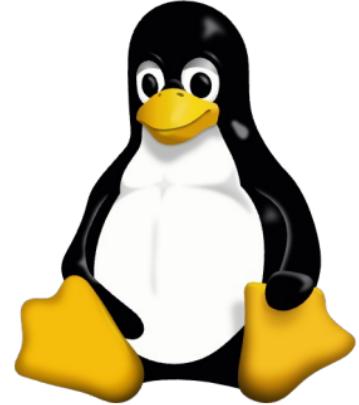
Bash & Linux CLI

Switch user: su

Switch user: The su command

- ▶ We can switch the user with the su command
- ▶ su stands for "switch user"
- ▶ **We can just enter:**
 - ▶ `su [other-user]`
- ▶ We will have to provide the password for the other user
- ▶ And then we will be logged in as the other user!
- ▶ **Let's have a look at how this works 😊**





Bash & Linux CLI

Temporary change privileges: sudo

The sudo command in Linux

► What does the sudo command do?

- ▶ sudo stands for "superuser do"
- ▶ It gives us temporarily privileges of another user - by default from the root user
- ▶ It thus executes commands with elevated / changed permissions
- ▶ Requires user authentication (with the user's password)
- ▶ The user must be allowed to use sudo

Configuring sudo access

- ▶ **How do we allow a user to use sudo?**

- ▶ Those are configured in the file /etc/sudoers
- ▶ We should only use the visudo command to safely edit this file
- ▶ We can allow access for specific users or groups

- ▶ **Example, gives the user "jannis" full sudo access:**

- ▶ jannis ALL=(ALL:ALL) ALL

- ▶ **Example, gives the group "jannis" full sudo access:**

- ▶ %jannis ALL=(ALL:ALL) ALL

- ▶ **We could also use a group to enable sudo access:**

- ▶ Group: sudo
- ▶ usermod -aG sudo jannis
- ▶ On other systems, this group might also be called wheel

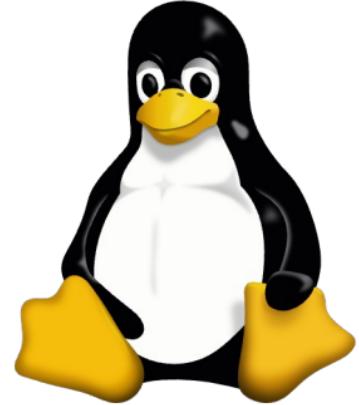
Using sudo in the Terminal

- ▶ **How can we use sudo in the Terminal?**

- ▶ We can just prefix the command with sudo

- ▶ **Example (Ubuntu / Debian):**

- ▶ `sudo apt-get update`
 - ▶ We then need to enter our user password
 - ▶ `sudo` will be stored in a session (by default: 15 minutes)
 - ▶ We can expire the session with the command `sudo -k`
 - ▶ Or we can also start a shell with additional privileges: `sudo -s`

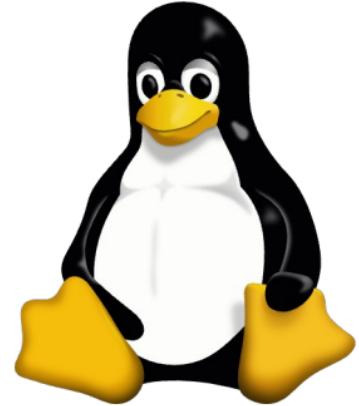


Bash & Linux CLI

Changing privileges: sudo into different user

sudo into a different user

- ▶ **We can also use sudo to start a program as a different user:**
 - ▶ `sudo -u [user] -g [group]`
 - ▶ **For example:**
 - ▶ `sudo -u lauren bash`
 - ▶ This would start the program "bash" as the user lauren
- ▶ **Let's have a look at how this works!**



Bash & Linux CLI

The sudo config file: /etc/sudoers

Advanced sudo: /etc/sudoers

► What does this line mean?

- ▶ `jannis ALL=(ALL:ALL) ALL`
 - ▶ **jannis:** The username, this rule should be applied to this user
 - ▶ **ALL:** The hostname that this rule should apply to
 - ▶ **(ALL:ALL):** The run configuration
 - ▶ The first **ALL:** The user that the user jannis can `sudo` into
 - ▶ The second **ALL:** The group that the user jannis can `sudo` into
 - ▶ If we don't specify this, we can only `sudo` into the root user
 - ▶ **ALL:** The command specification, **ALL** means any command

► sudo without password?

- ▶ We can also specify `NOPASSWD:`, to allow `sudo` without a password (potential security risk):
- ▶ `jannis ALL=(ALL:ALL) NOPASSWD: ALL`

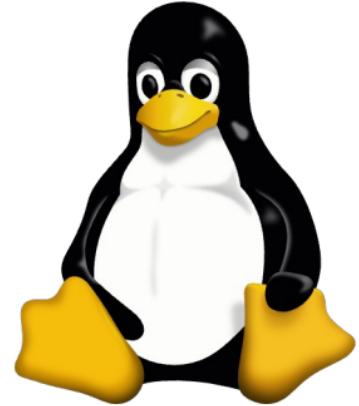
Advanced sudo: /etc/sudoers

- ▶ **We can also just allow specific programs:**

- ▶ jannis ALL= NOPASSWD: /usr/bin/apt-get
- ▶ This would give jannis access to only the program apt-get

- ▶ **This might also be a security issue:**

- ▶ I could also remove any program I want
- ▶ I could install any program I want
- ▶ Some of those programs might start a service that runs in the background... and executes files
- ▶ Those then might be executed with additional privileges



Bash & Linux CLI

File permissions in Linux

File permissions in Linux

- ▶ **Why do we need file permissions?**

- ▶ Control access to files and directories
- ▶ Determine who can read, write and execute those files

- ▶ **There're 3 important levels of permissions:**

- ▶ Owner (u)
- ▶ Group (g)
- ▶ Others (o)

Type of permissions

- ▶ **Type of permissions:**

- ▶ **Read (r / 4):**

- ▶ Allows viewing file contents or listing directory contents

- ▶ **Write (w / 2):**

- ▶ Allows modifying file contents or creating / deleting files in a directory

- ▶ **Execute (x / 1):**

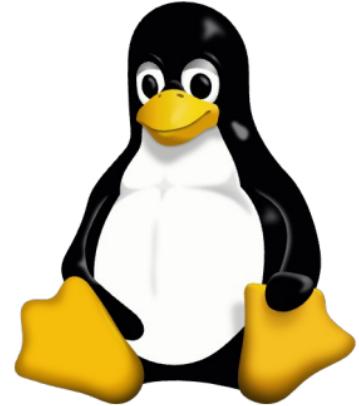
- ▶ Allows running a file as a program or accessing a directory's contents

Assigning permissions to a file

- ▶ **Permission levels:** Owner (u), group (g), others (o)
- ▶ **Type of permissions:** Read (r / 4), write (w / 2), execute (x / 1)
- ▶ **How do we assign permissions to a file?**
 - ▶ We can use the chmod command for this
- ▶ **Examples:**
 - ▶ `chmod u+x file.txt`:
 - ▶ Would give the owner (u) executable rights
 - ▶ `chmod g-w file.txt`:
 - ▶ Would remove the write permission for the group(g)
 - ▶ `chmod o+r file.txt`:
 - ▶ Would give other users (o) read access to this file or directory

Changing owner / group of a file

- ▶ We can also change the owner / group of a file:
 - ▶ `chown user:group file.txt`
 - ▶ This sets the owner to "user", and the group to "group"
- ▶ Let's see how this works 😊

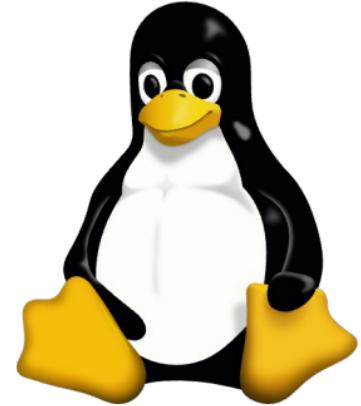


Bash & Linux CLI

File permissions: chmod 777

chmod with numeric values

- ▶ **Permission levels:** Owner (u), group (g), others (o)
- ▶ **Type of permissions:** Read (r / 4), write (w / 2), execute (x / 1)
- ▶ **How do we assign permissions to a file?**
 - ▶ We can use the chmod command for this
- ▶ **Examples:**
 - ▶ `chmod 754 file.txt`
 - ▶ **First digit is for the owner:**
 - ▶ $7 = 4 + 2 + 1 \Rightarrow$ read, write, execute
 - ▶ **Second digit is for the group:**
 - ▶ $5 = 4 + 1 \Rightarrow$ read, execute
 - ▶ **Third digit is for all others:**
 - ▶ $4 \Rightarrow$ Read
 - ▶ **Let's have a look at how this works ☺**



Bash & Linux CLI

File permissions (directories)

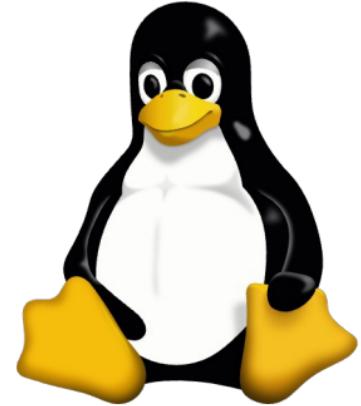
How do file permissions work for directories?

- ▶ **File permissions for directories:**

- ▶ Read (r): Access directory contents
- ▶ Write (w): Add or remove files
 - ▶ (we also need execute permissions for this though)
- ▶ Execute (x): Enter and traverse directory

Changing permissions

- ▶ If we want to change permissions / ownership for a whole directory structure...
- ▶ ... we can use the parameter **-R**:
 - ▶ `chown jannis:jannis -R ./directory`
 - ▶ `chmod 777 -R ./directory`
- ▶ Let's have a look at how this works 😊



Bash & Linux CLI

Advanced file permissions: umask

Why do we need a umask?

- ▶ The umask allows us to specify who should be able to access new files
- ▶ Thus, this is an important security feature
- ▶ If you're the sysadmin, you might want to evaluate this, and see which value is most appropriate for the umask in your organization

Advanced file permissions: umask

- ▶ **The idea:**

- ▶ It thus determines the default permissions for new files / directories

- ▶ **How does it work?**

- ▶ We have some base permissions
 - ▶ And from those, we subtract the umask value (technically: we apply a bitmask according to the binary representation of our umask value)

- ▶ **Base permissions are usually:**

- ▶ 777 for directories, 666 for files

- ▶ **If we set the umask to 022:**

- ▶ **Directories will have 755**

- ▶ Owner: read + write + execute
 - ▶ Group and others: read + execute (but not write)

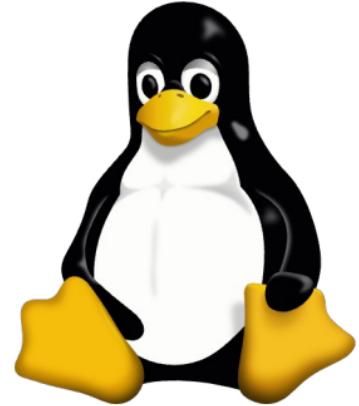
- ▶ **And files will be 644:**

- ▶ Owner: Read + Write
 - ▶ Group and others: read (but not execute + write)

- ▶ **We can show the current umask value:** `umask`

How to change umask?

- ▶ **How can we change the umask value?**
- ▶ **Temporarily change it (shell session):**
 - ▶ `umask [new umask]`
- ▶ **Permanently change it (for shell):**
 - ▶ We can add the umask command to one of the startup files our bash
(Example: `~/.bashrc`)
- ▶ **Permanently change it (for all programs):**
 - ▶ Usually, we can edit this in the following file:
 - ▶ `/etc/login.defs`
 - ▶ This should then also affect new GUI sessions!
 - ▶ **Be aware:**
 - ▶ Changes umask command in a shell overwrite this for the current shell session
 - ▶ Those changes should usually be applied automatically during startup - if a new shell doesn't pick them up, be sure to make sure the umask is not overwritten during startup of the shell



Bash & Linux CLI

Advanced file permissions: Sticky bit

Advanced file permissions: Sticky bit

- ▶ The sticky bit is an extra bit that we can set for all files or directories
- ▶ **It has different meaning:**
 - ▶ **For files:**
 - ▶ Obsolete, no longer used
 - ▶ It used to indicate that an executable file can remain in memory, to be loaded more quickly on next launch
 - ▶ **For directories:**
 - ▶ Without the sticky bit, any user with write + execute permissions for a directory can rename / delete files in it
 - ▶ If the sticky bit is set, only the owner (and root) of a file or the directory owner can rename or delete a file
- ▶ The sticky bit is especially used for the `/tmp` folder

Advanced file permissions: Sticky bit

- ▶ **How can we set the sticky bit?**

- ▶ `chmod +t [folder]`

- ▶ **Or in octal notation:**

- ▶ Set sticky bit: `chmod 1777 [folder]`

- ▶ Unset sticky bit: `chmod 0777 [folder]`

- ▶ **How can we inspect the sticky bit?**

- ▶ **We can use the `ls -l` command for this:**

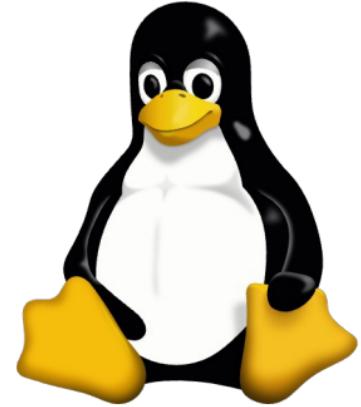
- ▶ `ls -l`

- ▶ **If the sticky bit is set:**

- ▶ If the file / directory also has executable permissions for others:

- ▶ Last character will be a "t"

- ▶ Otherwise, the last character will be a "T"



Bash & Linux CLI

Advanced file permissions: SUID / SGID

SUID / SGID

► **Have you ever wondered:**

- ▶ How do certain programs have additional privileges?
- ▶ How does that work?

► **Example:**

- ▶ sudo, su, mount, ...

Advanced file permissions: SUID

- ▶ **What is the SUID?**

- ▶ SUID = Set User ID

- ▶ **The Idea:**

- ▶ We can set a special bit for executable files

- ▶ **If this bit is set:**

- ▶ The executable will gain the rights of the owner
 - ▶ This allows unprivileged users to access privileged resources

- ▶ **Be careful:**

- ▶ This can be a major security issue if used improperly
 - ▶ On most systems, the SUID bit is limited to executable binary files
 - ▶ It is usually not supported for executable scripts (.sh, .py,...)

- ▶ **If we set it on our own programs, we can easily create major security vulnerabilities. Be extremely careful!**

Advanced file permissions: SUID

- ▶ **How can we inspect the SUID bit?**

- ▶ `ls -l file`

- ▶ **Example:**

- ▶ `-rwsrwxrwx`

- ▶ Lowercase s: SUID bit + execute bit

- ▶ Uppercase S: SUID bit, but without execute bit

- ▶ **We can also set the SUID bit:**

- ▶ `chmod u+s file`

- ▶ **Important:**

- ▶ We should also limit write access to this file as much as possible!

Advanced file permissions: SGID

- ▶ **We can give additional privileges based on the group**

- ▶ This is the SGID: Set Group ID

- ▶ **Example:**

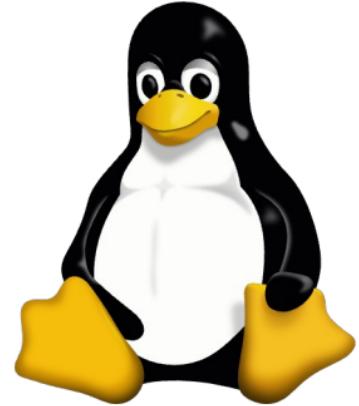
- ▶ `ls -l file`
 - ▶ `-rwxrwsrwx`

- ▶ **It works the same as for the SUID:**

- ▶ Lowercase s: SGID bit + execute bit
 - ▶ Uppercase S: SGID bit, but without execute bit

- ▶ **We can also set the SGID bit:**

- ▶ `chmod g+s file`



Bash & Linux CLI

User management: Best practices

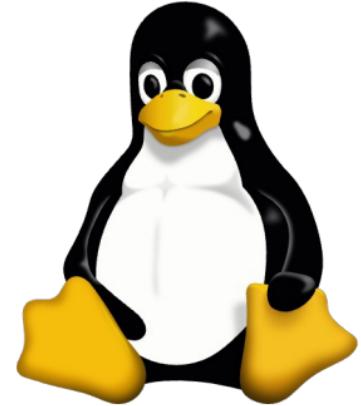
Groups: Best-practices

- ▶ **Best-practices:**

- ▶ Prefer groups to manage privileges
- ▶ Use meaningful group names

- ▶ **Follow the principle of least privilege:**

- ▶ Assigning users only the necessary permissions and group memberships required
- ▶ Avoiding the use of overly permissive access rights
- ▶ Don't give write access to everybody
- ▶ Minimizing the number of users with elevated privileges
- ▶ Keep group memberships up-to-date
- ▶ Regularly review group permissions



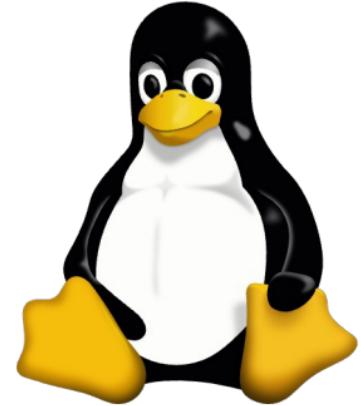
Bash & Linux CLI

Linux: Processes and signals

Linux: Processes and signals

► What will you learn in this chapter?

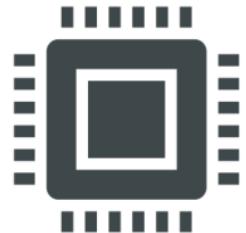
- ▶ Understand Linux processes and their attributes
- ▶ How you can send signals to processes:
 - ▶ This allows you to pause, resume or kill processes
- ▶ Use commands for managing processes and priorities
- ▶ Control processes with signals (e.g., `kill`, `killall`)
- ▶ How you can list the active processes (`top`, `htop`,...)



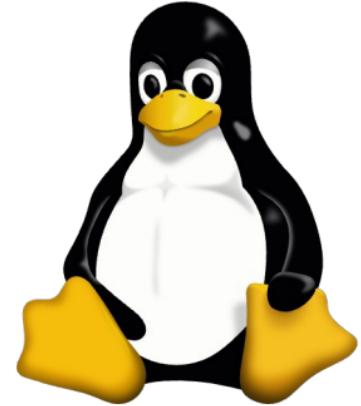
Bash & Linux CLI

How do processes work?

Processes



- ▶ **A process:**
 - ▶ An instance of a program
 - ▶ Independent execution unit with its own resources:
 - ▶ CPU & Memory resources
 - ▶ Opened files, network connections,...
 - ▶ It is managed by the kernel (lowest level of the operating system)
- ▶ **Each process has:**
 - ▶ A process ID (pid)
 - ▶ A user under which this process runs under
 - ▶ A state (running, waiting, stopped, zombie)
 - ▶ And various other properties
- ▶ All processes are organized in a hierarchy!
- ▶ **Let's have a look at an example!**



Bash & Linux CLI

The ps command

The program: ps

► By default:

- ▶ ps stands for "Process Status"
- ▶ Displays information about running processes
- ▶ Useful for monitoring system resources
- ▶ It works (almost) the same on macOS and Linux

► How do we use it?

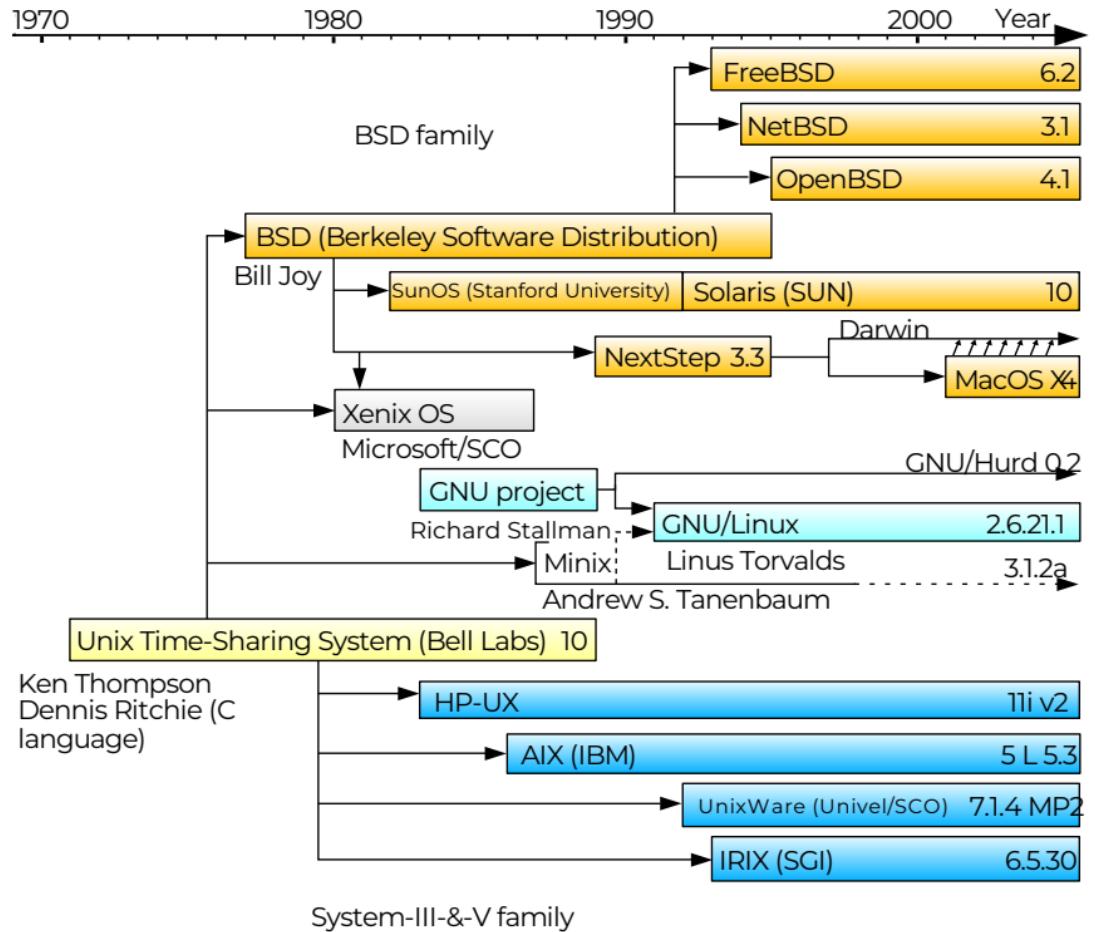
- ▶ On Linux, default usage: ps
 - ▶ Shows all processes of the current terminal (TTY)
- ▶ **Let's have a look at this in the terminal ☺**

Parameters for ps

- ▶ How do we list processes with ps?
- ▶ **We can select (and combine) from several parameters:**
 - ▶ `ps -A`, `ps -e`:
 - ▶ Show all processes, from all users and all sessions
 - ▶ `ps -f`:
 - ▶ Full-format listing: Show extended information, such as user, terminal, and parent process (PPID)
 - ▶ `ps -p 1234,1235`:
 - ▶ Show processes with process ID 1234 and 1235
 - ▶ We are also allowed to omit (leave out) the `-p`, and even the comma:
 - ▶ `ps 1234 1235`
 - ▶ `ps --forest`:
 - ▶ Only Linux: Show the processes as an ASCII tree
 - ▶ `ps -l`:
 - ▶ Show entries in long format (a few more columns)

Bash & Linux CLI

ps on macOS



Original source (edited):
[https://commons.wikimedia.org/
 wiki/File:Unix_timeline.de.svg](https://commons.wikimedia.org/wiki/File:Unix_timeline.de.svg)

For mac users: ps on macOS

- ▶ **Linux:**

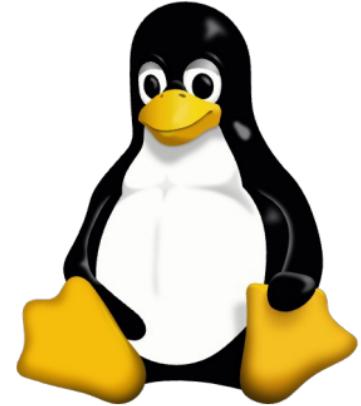
- ▶ Follows the POSIX standard for ps
- ▶ ps by default only displays the active processes of the current terminal

- ▶ **Mac:**

- ▶ Follows the BSD standard for ps
- ▶ ps by default displays all active processes of all terminals run by our current user ...
- ▶ ... even if those are in another tab or in another terminal application

- ▶ **To replicate the same behavior as on Linux:**

- ▶ `ps -T`



Bash & Linux CLI

What is BSD? And POSIX?

BSD vs. Linux?

► Unix:

- ▶ Released as UNICS in September 1969
- ▶ Then later renamed to UNIX
- ▶ Originally developed by Bell Laboratories / AT&T
- ▶ Started as free software - but in the 1980, AT&T started to monetize the software project

► GNU / Linux:

- ▶ Project started by Richard Stallman in 1983
- ▶ Idea: Open alternative to UNIX
- ▶ In the early 90s: Viable alternative to UNIX

► BSD (Berkeley Software Distribution):

- ▶ Started as a variant of UNIX
- ▶ Developed at the University of California, Berkeley
- ▶ Completely rewritten in the 90s, to prevent copyright issues with the authors of UNIX
- ▶ macOS builds on top of Darwin, which is built on top of BSD

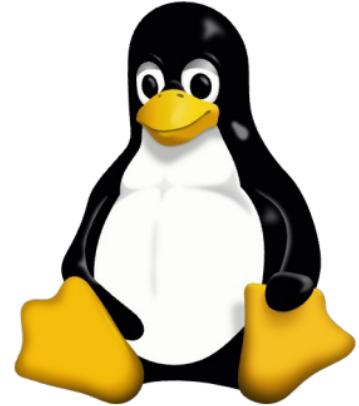
What is POSIX?

- ▶ **POSIX:**

- ▶ Portable Operating System Interface (IEEE 1003)
- ▶ Started in 1988 by IEEE Computer Society

- ▶ **Idea:**

- ▶ Family of standards for maintain compatibility between operating systems
- ▶ Defines common APIs
- ▶ Defines common shell utilities and tools



Bash & Linux CLI

Bonus: BSD style parameters: ps aux

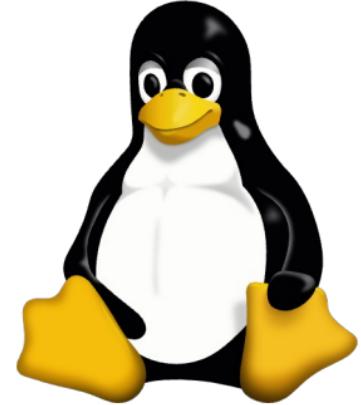
BSD style parameters for ps

- ▶ **ps also supports BSD style parameters:**

- ▶ BSD style parameters are parameters without a dash
- ▶ If we use a BSD style parameter, certain defaults are different
- ▶ And (at least some of) those parameters do behave differently!

- ▶ **Because this example is so common... let's have a look:**

- ▶ `ps aux` / `ps a u x`
- ▶ `ps a:`
 - ▶ Show all processes of all users
- ▶ `ps u:`
 - ▶ Displays the information in a more user-oriented format
(additional columns)
- ▶ `ps x:`
 - ▶ Show processes without a tty (=processes outside of a terminal)

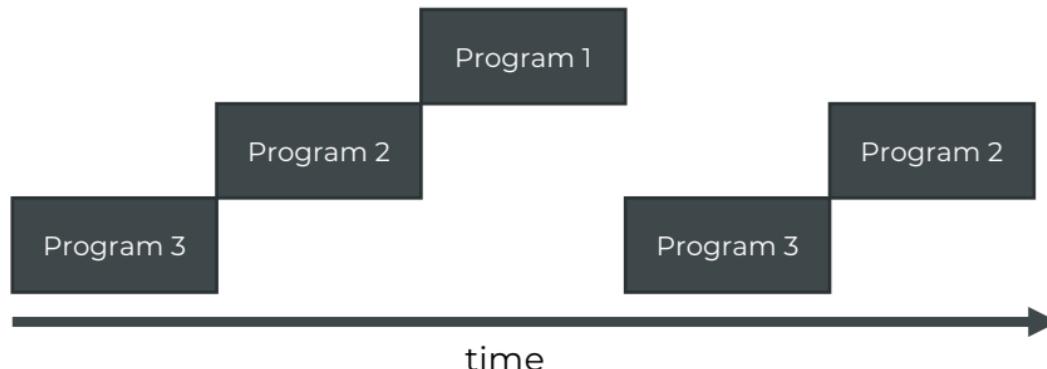


Bash & Linux CLI

Multitasking: The scheduler

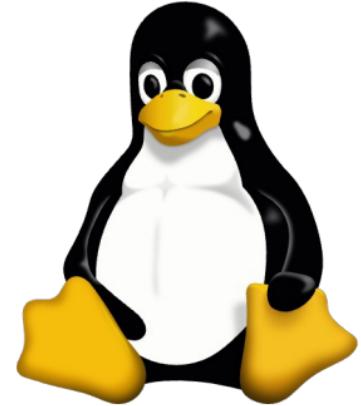
How does multitasking work?

- ▶ If we only have one CPU in our system
- ▶ How can we execute multiple programs at the same time?
- ▶ **The idea:**
 - ▶ We let our CPU switch between those
 - ▶ If we switch fast enough, it will seem like we're running all programs at the same time
 - ▶ This switching is called scheduling



Bonus: Can we inspect this?

- ▶ If our CPU switches from one program to another, it's called "context switch"
- ▶ **You don't need to remember this, but we can visualize those context switches:**
 - ▶ `cat /proc/[process ID]/status | grep ctxt`
 - ▶ `cat /proc/12345/status | grep ctxt`
- ▶ **We can use the watch command:**
 - ▶ This allows us to automatically re-execute a command
 - ▶ **Here:** Refresh the output every 0.5 seconds
 - ▶ `watch -n 0.5 grep ctxt /proc/12345/status`



Bash & Linux CLI

Setting the priority of a process: Niceness

Setting the priority of a process

- ▶ Now that we know how our operating system switches between processes... can we influence this?
- ▶ We can! We can use the niceness for this
- ▶ **Niceness:**
 - ▶ Niceness ranges from -20 (highest priority) to +19 (lowest priority)
 - ▶ Default niceness for new processes is typically 0
 - ▶ Processes with lower niceness (higher priority) receive more time from the scheduler
- ▶ **Typically:**
 - ▶ We need administrative privileges to lower the niceness / increase the priority of a process
 - ▶ We don't need any privileges to increase the niceness / lower the priority of a process

How do we set the niceness?

- ▶ Set the niceness for a program:
 - ▶ `nice -n [niceness] [program]`:
 - ▶ Sets the niceness to [niceness] for a program (lower priority than default)
 - ▶ **Example:**
 - ▶ `nice -n 19 gedit`
- ▶ We can also change the priority of an existing process:
 - ▶ `renice -n 19 [process ID]`
- ▶ **Let's have a look at this!**



Bash & Linux CLI

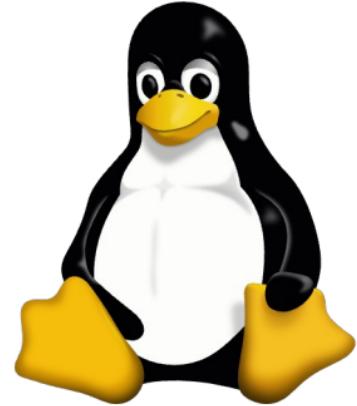
Getting the process ID for a program

Getting the process ID for a program

- ▶ Let's say we have a process of the program "firefox"
- ▶ How can we get it's process ID?
- ▶ We could use the ps program, and grep in the output
 - ▶ `ps -ef | grep -F 'firefox'`
- ▶ **But we will also get additional output :/**
- ▶ **How can we only get the process ID?**
 - ▶ We can use the pgrep program!
 - ▶ This allows us to search in our programs
 - ▶ And it only returns the process ID(s)!
 - ▶ `pgrep firefox`
- ▶ **Why is it so useful that we need a special tool?**
 - ▶ **Reason:** We can combine it with expansions!
 - ▶ `renice -n 19 $(pgrep firefox)`

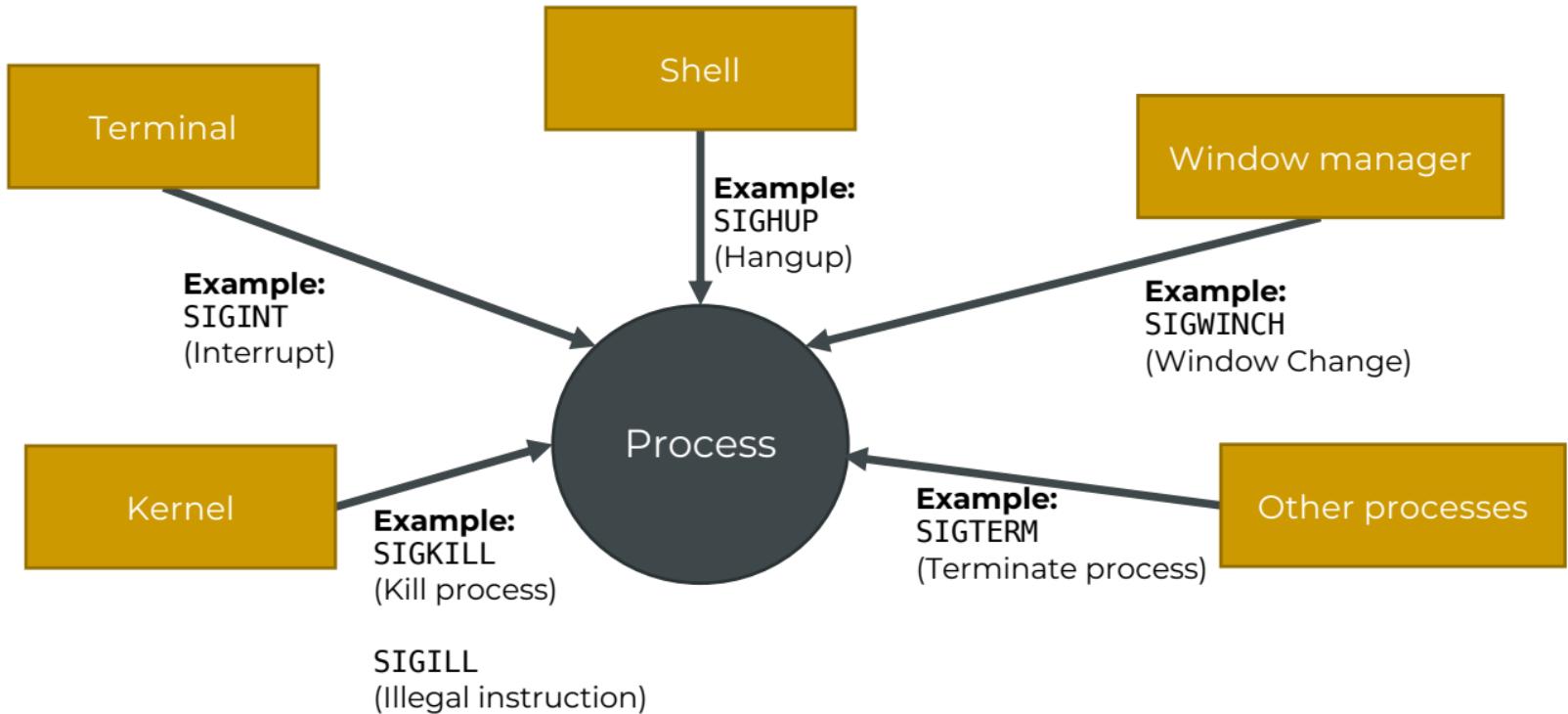
Bash & Linux CLI

Signals



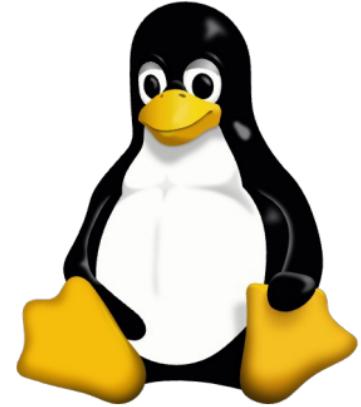
Signals

- ▶ Signals can be sent to processes, and they will interrupt the process flow at a *convenient* time
- ▶ It's a mechanism to asynchronously notify a process of an event
- ▶ **The operating system:**
 - ▶ Is responsible for delivering the signal to the process
 - ▶ Maintains a signal queue, so we can send a signal to every process (even to one that is not active right now)
- ▶ **What kind of messages do signals deliver?**



Let's send our first signal

- ▶ **We've used signals before!**
- ▶ When we end a command with CTRL + C... we're sending the SIGINT signal to the program
- ▶ **SIGINT:**
 - ▶ Usually sent from the terminal to a program
 - ▶ Meant to indicate we would like it to stop and regain control over our terminal
 - ▶ Programs can listen to that signals and perform custom actions - including ignoring this signal!
- ▶ **Let's have a look at this in our terminal**

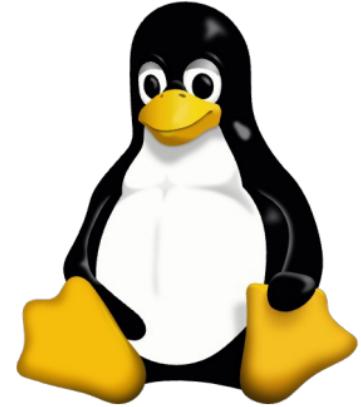


Bash & Linux CLI

The kill command

The kill command

- ▶ We can also send signals from other places to a process
- ▶ Usually, we use the `kill` command for this
- ▶ **Important:**
 - ▶ The program is just called `kill`...
 - ▶ But all it does is to send signals to programs
- ▶ **How do we send a SIGINT with kill?**
 - ▶ `kill -s [SIGNAL] [process-ID]`
 - ▶ We first need to get the process ID of the process we would like to send the SIGINT to
- ▶ **And then we can send the signal to the process:**
 - ▶ `kill -s SIGINT 12345`
 - ▶ `kill -SIGINT 12345`



Bash & Linux CLI

The killall command

The killall command

- ▶ We can also use the `killall` command for sending signals
- ▶ The difference to `kill` is, that we select the process by name

- ▶ **Example:**

- ▶ `killall firefox`

- ▶ **How do we send a SIGINT with killall?**

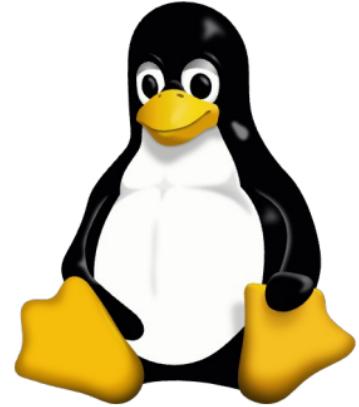
- ▶ `killall -s [SIGNAL] [process-name]`

- ▶ **macOS:**

- ▶ If we're on macOS, `-s` has a different meaning - it will make the program more verbose and prevent any signal from being sent

- ▶ We can only use the following syntax there:

- ▶ `killall -SIGINT firefox`



Bash & Linux CLI

Additional signals, SIGTERM, SIGKILL

More signals we can send to a program

- ▶ The program kill can also send various signals to programs
- ▶ We can show all available signals with the command: `kill -l`
- ▶ **Example, on my Linux machine:**

1) SIGHUP	2) SIGINT	3) SIGQUIT	4) SIGILL	5) SIGTRAP
6) SIGABRT	7) SIGBUS	8) SIGFPE	9) SIGKILL	10) SIGUSR1
11) SIGSEGV	12) SIGUSR2	13) SIGPIPE	14) SIGALRM	15) SIGTERM
16) SIGSTKFLT	17) SIGCHLD	18) SIGCONT	19) SIGSTOP	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) SIGPROF	28) SIGWINCH	29) SIGIO	30) SIGPWR
31) SIGSYS	34) SIGRTMIN	35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3
38) SIGRTMIN+4	39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12	47) SIGRTMIN+13
48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14	51) SIGRTMAX-13	52) SIGRTMAX-12
53) SIGRTMAX-11	54) SIGRTMAX-10	55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7
58) SIGRTMAX-6	59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX			

The signal: SIGTERM

- ▶ **The signal: SIGTERM**

- ▶ Is used to tell the process that it should terminate

- ▶ **We can imagine the meaning like this:**

- ▶ Hey process, would you please come to an end? If you can't that's totally alright as well, just please try to come to an end

- ▶ **We can send this signal the following ways:**

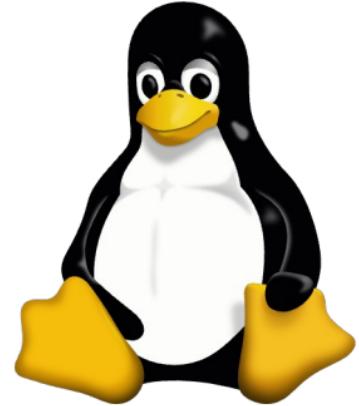
- ▶ `kill [process ID]`
 - ▶ `kill -s SIGTERM [process ID]`
 - ▶ `kill -s 15 [process ID]`
 - ▶ `kill -SIGTERM [process ID]`
 - ▶ `kill -15 [process ID]`

The signals: SIGTERM vs. SIGINT

- ▶ **What's the difference between SIGTERM and SIGINT (CTRL + C)?**
 - ▶ SIGTERM tells the process to terminate
 - ▶ SIGINT tells the process to terminate, because we're in a shell and we would like to regain control immediately

The signal: SIGKILL

- ▶ SIGTERM and SIGINT could be ignored from our program - it's up to the program to handle those
- ▶ **How to we kill a process?**
 - ▶ We send the SIGKILL signal!
 - ▶ This forcefully terminates the program
- ▶ **Be careful:**
 - ▶ This may cause data loss
 - ▶ Files might not have been written completely, or might be in an inconsistent state
- ▶ **This signal is unique:**
 - ▶ It is not handled by the process, but by the kernel
 - ▶ The program isn't even informed that there's a SIGKILL event
- ▶ **We can send this signal the following ways:**
 - ▶ `kill -s SIGKILL [process ID]` / `kill -SIGKILL [process ID]`
 - ▶ `kill -s 9 [process ID]` / `kill -9 [process ID]`



Bash & Linux CLI

Additional signals: SIGHUP, SIGSTOP, SIGCONT

The signal: SIGHUP

- ▶ **For usual programs:**

- ▶ The SIGHUP signal is used to communicate that the terminal has been closed
- ▶ The program then usually exists

- ▶ **For daemons (background processes):**

- ▶ It may also mean that the program should reload its configuration

- ▶ **We can also send a SIGHUP ourselves:**

- ▶ `kill -s SIGHUP [process ID]`

The signal: SIGSTOP

- ▶ **The signal: SIGSTOP**

- ▶ The process is being put into a stopped state (=paused)
- ▶ Non-catchable, the process can't ignore it
- ▶ We can later resume the process again (with SIGCONT)

- ▶ **We can also send a SIGSTOP ourselves:**

- ▶ `kill -s SIGSTOP [process ID]`

- ▶ **Let's test this with 2 programs:**

- ▶ `cmatrix`
- ▶ `Wget`

- ▶ **You might have to install them:**

- ▶ `sudo apt-get install cmatrix wget`
- ▶ `brew install cmatrix wget`

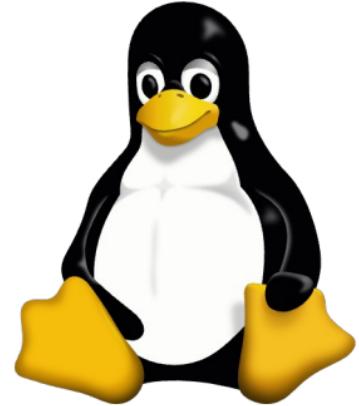
- ▶ **On CentOS:**

- ▶ `sudo yum install wget`

The signal: SIGCONT

► The signal: SIGCONT

- ▶ We can send this signal to a process to resume it again
- ▶ Example:
 - ▶ We had stopped a process before (for example, with SIGSTOP)
 - ▶ And now we want to resume it
 - ▶ `kill -s SIGCONT [process ID]`



Bash & Linux CLI

kill vs. /usr/bin/kill vs. /bin/kill

kill vs. /usr/bin/kill vs. /bin/kill

- ▶ **A lot of commands exists twice:**

- ▶ one time as a shell build-in,
- ▶ and one more time as an executable file

- ▶ **We can check this:**

- ▶ type kill

- ▶ **Output:**

- ▶ kill is a shell builtin

- ▶ **What does this mean?**

- ▶ The shell is providing this functionality
- ▶ When we call the kill command, we are not calling the executable file from the operating system

- ▶ **Why does it matter?**

kill vs. /usr/bin/kill vs. /bin/kill

► Why does it matter?

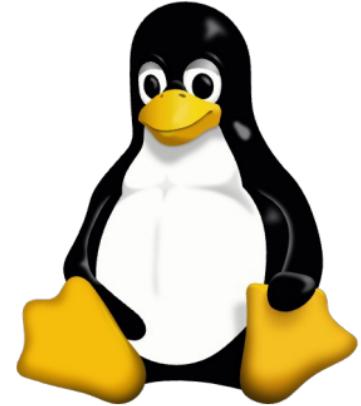
- ▶ kill provides additional functionality and behaves differently than the executable from the OS
- ▶ Let's just compare those two

► How do we find the executable from the OS?

- ▶ We could manually investigate our PATH
- ▶ Or let a program do this for us:
 - ▶ Bash: which kill
 - ▶ Zsh: where kill
- ▶ It should usually be /usr/bin/kill (Linux) or /bin/kill (macOS)

► Let's now compare:

- ▶ kill -l
- ▶ /usr/bin/kill -l



Bash & Linux CLI

The killall command

The command: killall

- ▶ **The command killall:**

- ▶ Can search for a process with a given name
- ▶ And then send a signal to it

- ▶ **killall firefox:**

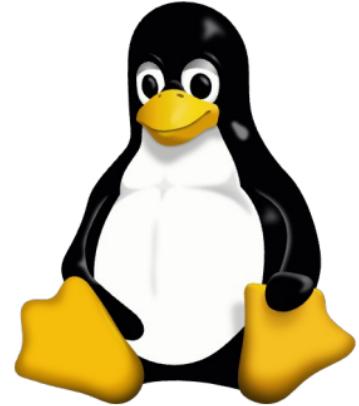
- ▶ Searches for all processes that contain "firefox" in their name
- ▶ Sends SIGTERM to those processes
- ▶ Alternative: `killall -s SIGTERM firefox`

- ▶ **By default:**

- ▶ It only matches the first 15 characters of a program name
- ▶ For programs with very long names, we should use the `-e` parameter (stands for: exact)

- ▶ **We can also enable interactive mode:**

- ▶ `killall -i firefox`



Bash & Linux CLI

What happens when a process exits?
What are zombie processes?

What happens when a process exits?

- ▶ First, most of the resources are made available again
- ▶ Parent Process:
 - ▶ Child Termination:
 - ▶ When a child process terminates, the kernel sends a SIGCHLD signal to the parent
 - ▶ The parent can then retrieve the child's exit status
 - ▶ Process Reaping:
 - ▶ The parent process uses a system call (wait() or waitpid()) to collect the child's exit status
 - ▶ This action is known as "reaping" the child process
 - ▶ In a Bash, we can access the child's exit code: \$? (more on those later)
 - ▶ Orphan Process:
 - ▶ If the parent process ends before the child, the child becomes an orphan and is adopted by the init process

Zombie process



► **Zombie Process:**

- ▶ Definition: A zombie process is a process that has finished executing but still has an entry in the process table
- ▶ This usually occurs when the parent process hasn't read the child's exit status (yet)

► **Characteristics:**

- ▶ Zombies can lead to process table overflow
- ▶ They are marked as "Z" in the output of `ps -l`

► **Removal:**

▶ **When the parent process ends:**

- ▶ They're usually removed automatically

▶ **If not, they can be manually reaped:**

- ▶ We can send a `SIGCHLD` signal to their parent

▶ **Or we can kill the parent process:**

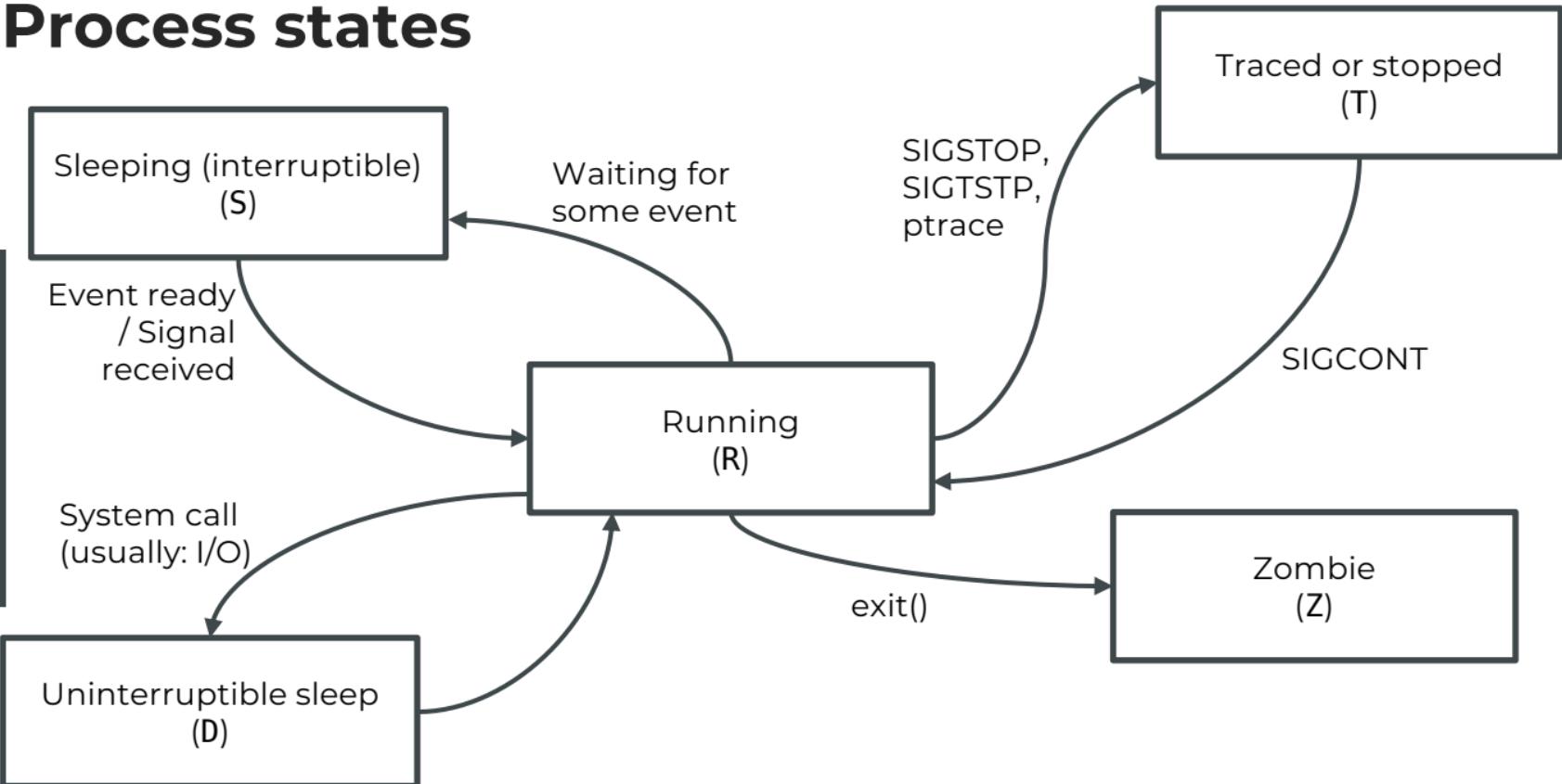
- ▶ Then, the init process will adopt this process, and reap it

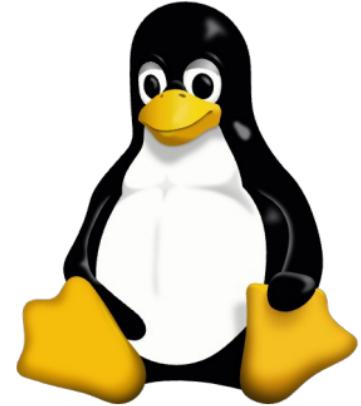
Bash & Linux CLI

Process states



Process states





Bash & Linux CLI

The program: top

The program: top

- ▶ The program top allows us to display all processes from our system
- ▶ **We can start this program with the following command:**
 - ▶ `top`
 - ▶ `sudo top`
- ▶ **Let's have a look at it!**

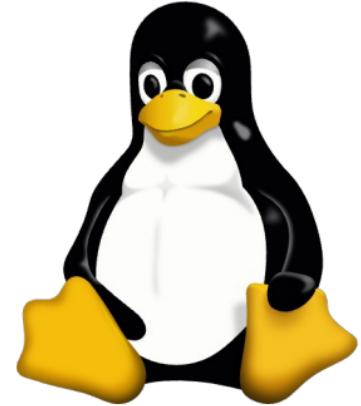
top vs. htop

► **I now had a problem in this course:**

- ▶ There're more ways to use the top program, and additional ways to configure it
- ▶ But on most (unix) systems, there's a better program available: htop
- ▶ But not on all!

► **The solution:**

- ▶ In the next lecture, we will have a deep dive into the top command
- ▶ But you can just skip it, if you're able to use htop on your system
- ▶ For example, install on Ubuntu: `sudo apt-get install htop`
- ▶ Or on CentOS (with EPEL): `sudo dnf install htop`
- ▶ And after that, we will investigate the htop program



Bash & Linux CLI

Deep dive: The program top

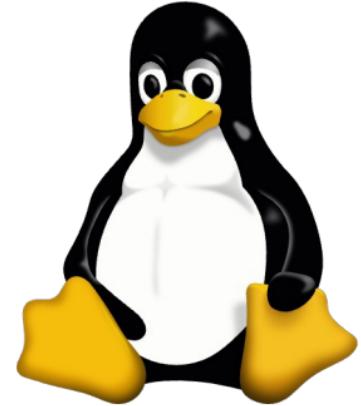
Important parameters for top

- ▶ **The most important parameters are:**
 - ▶ **-u [username]**: Show only processes owned by the specified user
 - ▶ **-d [seconds]**: Set the delay between display updates, in seconds. The default value is 3 seconds
 - ▶ **-i**: Start top with the "idle" processes hidden. This shows only the processes that are currently using CPU resources
 - ▶ **-c**: Display the full command line used to start each process, instead of just the command name

How to change the output of top?

- ▶ **We can also interactively change the output:**

- ▶ **f** key: We can press the f key to further customize the output
- ▶ **k** key: We can send signals to processes with this key (k = kill)
- ▶ **r** key: We can change the niceness of processes
- ▶ **z** key: We can switch to color mode, and with the uppercase Z key, we can configure it
- ▶ **W** key: Write the current configuration, so it is loaded the next time we start top



Bash & Linux CLI

The program: htop

The program: htop

- ▶ **The program htop:**
 - ▶ An easier to use alternative to top
 - ▶ It's available on most Linux systems, but might have to be installed as an additional software
- ▶ **For example, on Ubuntu:**
 - ▶ `sudo apt-get install htop`
- ▶ **The big advantage:**
 - ▶ **It's quite self-explanatory. Let's just have a look!**



Bash & Linux CLI

Bash: Job control



Bash: Job control

- ▶ In the last chapter, you have learned how you can manage processes on Linux
- ▶ In this chapter, we will look at something that builds on top of that: We will look at how jobs are being managed by Bash
- ▶ **You will learn:**
 - ▶ How you can start a command in the background
 - ▶ How exactly pipes are being executed
 - ▶ How you can easily move a command into the background
 - ▶ How to easily pause and resume a command
 - ▶ How to disconnect a program from the terminal, so that it stays open even if you close the terminal

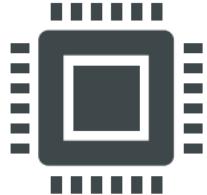
Bash & Linux CLI

Background jobs in Bash

What is a job?

- ▶ A job is a command that is being executed
- ▶ **A job can consist out of multiple programs:**
 - ▶ `cat file.txt | wc`
 - ▶ In this case, we have 2 programs that are being executed at the same time: `cat` and `wc`
 - ▶ Both of those programs are bundled into one job that is being executed
- ▶ **So far:**
 - ▶ **Foreground jobs:**
 - ▶ We have looked at foreground jobs
 - ▶ Those occupy our shell: Bash will wait for its completion before accepting a new command
 - ▶ **Background jobs:**
 - ▶ We will have a look at those in this lecture!

Background jobs in Bash



- ▶ **We can also start background jobs in Bash:**
 - ▶ [command] &
 - ▶ **Example:**
 - ▶ `ping -c 10 google.com &`
 - ▶ The -c option limits the number of ping packets to 10
 - ▶ Because of the & symbol, the command will be started as a background job
 - ▶ The output will still be displayed in the shell
(unless we redirect it)
 - ▶ It will display the job ID and the process ID
 - ▶ **Let's have a look at how this works!**

Bash & Linux CLI

Job management in Bash

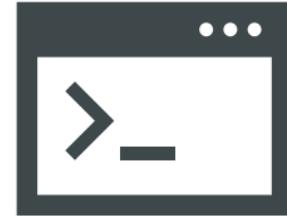
List running jobs: jobs

- ▶ ping is now running in the background
- ▶ How can we see all the background jobs?
- ▶ **We can use the jobs command for this!**
 - ▶ **Command:**
 - ▶ jobs
 - ▶ **Output:**
 - ▶ [1]+ Running ping -c 10 google.com &
- ▶ **Let's have a look at it 😊**



Job to foreground: fg

- ▶ If we want to get a job back into foreground, we can use the command: `fg`
- ▶ `fg [%job-ID]`
- ▶ **For example:**
 - ▶ `fg %1`
 - ▶ If there's just one job, it will automatically switch to that, and we don't need to specify the job ID:
 - ▶ `fg`
 - ▶ If there're multiple jobs, it will switch to the current job (marked with a + in the jobs table)
- ▶ **Let's have a look at it 😊**



Important

- ▶ Only foreground jobs can receive keyboard input
- ▶ **This also includes signals, such as SIGINT:**
 - ▶ Ctrl + C
 - ▶ This will only ask foreground jobs / processes to quit themselves
 - ▶ But it will not work for background jobs!

Bash & Linux CLI

Job management: Suspend program, resume it

Send a job to the background

- ▶ If we are in a foreground job, and would like to send it into the background...
- ▶ **We can suspend the job:**
 - ▶ We enter the suspend character (usually: Ctrl + Z)
 - ▶ This internally sends a SIGTSTP signal to the programs involved
- ▶ **SIGTSTP?**
 - ▶ It's a nicer version of the SIGSTOP signal: It asks the program to pause execution and return control to the terminal
 - ▶ A program could ignore it though
 - ▶ The job can be resumed later
- ▶ **Let's have a look at how this works 😊**

How can we resume the job?



- ▶ **How can we resume a job?**
 - ▶ We can resume a job in the foreground:
 - ▶ `fg %[job-ID]`
 - ▶ We can also resume a job as a background job:
 - ▶ `fg %[job-ID] &`
 - ▶ `bg %[job-ID]`
- ▶ **When continuing:**
 - ▶ A SIGCONT signal is sent to the program
- ▶ **Let's have a look at how this works ☺**

Bash & Linux CLI

Job management in Bash: kill a job

How can we kill a job?



- ▶ **How can we kill a job?**
 - ▶ We can kill a job with the kill command:
 - ▶ `kill %[job-ID]`
 - ▶ **This will send SIGTERM to the job and will ask the job to terminate itself**
 - ▶ If this doesn't work, we could of course also send SIGKILL to a job:
 - ▶ `kill -s SIGKILL %[job-ID]`
 - ▶ **Let's have a look at how this works ☺**

Bash & Linux CLI

Stop jobs with output: stty

Stop jobs with output

- ▶ Sometimes, we want to stop jobs that create output
- ▶ We want to see the whole output of a program
- ▶ And we don't want the program to just directly write to our terminal
- ▶ How do we solve this?

Stop jobs with output

- ▶ **We can set a bash option:**

- ▶ `stty tostop`
- ▶ `stty` is a tool to change / print the terminal line settings
- ▶ By setting the `tostop` option, we can tell it to suspend the job if it creates any output
- ▶ When we now start a background job it will only run until it creates any output
- ▶ Once it writes any output, the job will be suspended immediately
- ▶ **To disable this feature again:**
- ▶ `stty -tostop`

Bash & Linux CLI

Waiting for background jobs: wait

Waiting for jobs: wait



- ▶ Let's say we have 3 downloads, and they're all running in the background
- ▶ How can we wait for them and execute another command once all of them are done?

Waiting for jobs: wait

- ▶ **The wait command:**

- ▶ With `wait`, we can wait for background jobs
- ▶ **`wait`**
 - ▶ Waits until all currently running jobs have changed their state
- ▶ **`wait 123`**
 - ▶ Waits for process with the ID 123
- ▶ **`wait %1`**
 - ▶ Waits only for background job #1
- ▶ **`wait -n`**
 - ▶ Waits for any job to be completed

Bash & Linux CLI

Keep a program running

Keep a program running: nohup

- ▶ We can use the `nohup` command, to launch a program that will remain open even if we close our terminal:
 - ▶ `nohup ping -c 100 google.com &`
- ▶ **This will:**
 - ▶ Start the program `ping`, which will be run in the background
 - ▶ It will keep running, even if we close our terminal (or log out)
- ▶ **The standard output will be redirected:**
 - ▶ Usually to a file called "nohup.out" in the current folder
 - ▶ It may also be redirected into a "nohup.out" in the home directory, if the current directory is not writable

nohup vs. &

- ▶ **What's the difference between:**
 - ▶ nohup ping -c 10 google.com
 - ▶ ping -c 10 google.com &
 - ▶ nohup ping -c 10 google.com &
- ▶ **Let's now look at those three cases on the next slide**

nohup vs. &

- ▶ **nohup ping -c 10 google.com:**

- ▶ nohup disconnects the ping program from the SIGHUP signal. Ping is still a foreground process
- ▶ ping will remain running, even after the terminal has been closed

- ▶ **ping -c 10 google.com &:**

- ▶ ping is being launched as a background process for our current terminal.
- ▶ It will terminate when we close the terminal, because it still receives the SIGHUP signal

- ▶ **nohup ping -c 10 google.com &:**

- ▶ nohup disconnects the program from the SIGHUP signal, thus it will keep running if we close our terminal
- ▶ It's a background process, so it will run in the background of our current terminal



Ubuntu

Bash & Linux CLI

Introduction to package management on Ubuntu

What is package management?

- ▶ **Definition:**

- ▶ Process of installing, updating, configuring, and removing software

- ▶ **Purpose:** Streamline software handling on a Linux system

- ▶ **Why do we need it?**

- ▶ Facilitates easy software distribution
 - ▶ Simplifies software installation
 - ▶ Ensures software compatibility
 - ▶ Handles dependencies
 - ▶ Maintains system stability and security

- ▶ **But we need to be careful:**

- ▶ Package management is quite different between different Linux distributions
 - ▶ We will look at package management on Ubuntu / Debian

Package management on Ubuntu



► Package management on Ubuntu:

- ▶ Based on Debian package management
- ▶ Debian packages can ***usually*** be installed on Ubuntu as well
- ▶ Uses apt / apt-get and dpkg

► Package management with snap:

- ▶ We can also install snap packages
(completely different format for packages)

This chapter

- ▶ Is for people who really want to understand package management on ubuntu
- ▶ **The goal:**
 - ▶ You're not just able to install and manage software on a system
 - ▶ You will learn everything you need to know to handle urgent situations

Bash & Linux CLI

Let's start with: dpkg

dpkg: Debian Package Manager

► **dpkg:**

- ▶ On the lowest level, dpkg is responsible for installing software packages
- ▶ Those packages are distributed as **.deb** files, and we can install them through dpkg:
 - ▶ `dpkg -i package.deb`
- ▶ A **.deb** file is a compressed archive (ar file format) with all the files needed for the program, and its installation on the system



▶ **How do we get those software packages?**

- ▶ Later we will be able to get them automatically
- ▶ But let's now get them manually

▶ **For Ubuntu, we can find the repository here:**

- ▶ <https://packages.ubuntu.com/>

Bash & Linux CLI

Dependency management: apt / apt-get

Dependency management with apt

- ▶ dpkg was able to install .deb packages on our system
- ▶ But we were not able to install dependencies through it
- ▶ **Thus, we need another tool, that builds on top of dpkg. We can choose one of the following:**
 - ▶ apt-get:
 - ▶ Stable API, we should use this when we're writing scripts
 - ▶ apt:
 - ▶ API and parameters might change when deemed necessary
- ▶ **But how does it work?**

A package source / repository in apt

- ▶ apt-get / apt needs to know which packages are available, and where it can download them from
- ▶ We thus need to connect to central repositories, which provide our packages
- ▶ **Those central repositories are stored in the following files:**
 - ▶ Repositories from the system:
 - ▶ `/etc/apt/sources.list`
 - ▶ Additional (third party) repositories:
 - ▶ `/etc/apt/sources.list.d/*`
- ▶ **Important:**
 - ▶ Those repositories provide a list with packages, their versions, ...
 - ▶ We must fully trust those repositories
 - ▶ They could even replace existing software on our system

Updating package definitions

- ▶ Once we have our repositories, we need to update the "package definitions"
- ▶ This means, that we fetch the latest list of available packages from the repositories
- ▶ **We can do this with the following command:**
 - ▶ `sudo apt update / sudo apt-get update`
- ▶ **After this, we're able to install software on our system**
 - ▶ If we want to install neofetch, we could now do it:
 - ▶ `sudo apt install neofetch`
 - ▶ This package is then installed - and all the dependencies (additional packages) needed to execute `neofetch`
 - ▶ `apt` will remember, which package we installed manually, and which packages were just installed as dependencies (important for later)

Bash & Linux CLI

Keeping our system up to date

Managing upgrades

- ▶ We want to keep our system up to date
- ▶ We thus want to install available updates on our system

- ▶ **How do we do this?**

- ▶ `sudo apt upgrade`
- ▶ `sudo apt-get upgrade --with-new-pkgs`

- ▶ **What will this do?**

- ▶ This will install all available and possible updates - and even install additional dependencies (if they become necessary)
- ▶ It will never remove any packages from our system, even if they're no longer needed

Managing updates (full-upgrade / dist-upgrade)

- ▶ If we want to have a bigger upgrade, we need to allow apt to uninstall packages as well:
 - ▶ `sudo apt full-upgrade`
 - ▶ `sudo apt-get dist-upgrade`
- ▶ **This will:**
 - ▶ Try to install all available and possible upgrades
 - ▶ Uninstall dependencies if this is required in order to install the upgrade
- ▶ **Example for this:**
 - ▶ Package A depends on Package B (v1.0)
 - ▶ Package A is updated, now depending on Package C (v1.0), which conflicts with Package B (v1.0)
 - ▶ Package B is uninstalled during dist-upgrade.

Warning: dist-upgrade

- ▶ Be careful with the dist-upgrade / full-upgrade
- ▶ It can remove programs you were relying on, in order to solve dependency problems
- ▶ **Thus, be sure to:**
 - ▶ Really read the output of what it wants to do
 - ▶ Have a backup
 - ▶ Have enough time to fix potential problems

Bash & Linux CLI

Auto-removing packages: autoremove

Auto-removing packages

- ▶ `full-upgrade` / `dist-upgrade` only uninstalls dependencies, if they're exclusive and need to be uninstalled
- ▶ All other dependencies will remain installed, even if they're no longer needed
- ▶ **To uninstall those, we need to execute the following command:**
 - ▶ `sudo apt autoremove`
 - ▶ `sudo apt-get autoremove`

How does the sources.list work?

- ▶ **The system reads the following file(s) for its repositories:**
 - ▶ /etc/apt/sources.list
 - ▶ /etc/apt/sources.list.d/*
- ▶ **Let's just have a look at those files first**

How does the sources.list work?

- ▶ What's the syntax of the /etc/apt/sources.list?

- ▶ deb http://ports.ubuntu.com/ubuntu-ports/ jammy main restricted
- ▶ <type> <uri> <distribution> <domain1> <domain2> ...
- ▶ <type>:
 - ▶ deb: This repository contains binary packages
 - ▶ deb-src: This repository contains source packages
- ▶ <uri>: The address of the repository
- ▶ <distribution>: The ubuntu version, we want to download the packages for

<domain>:	Free software	Non-free software
Officially supported by Canonical	main	restricted
Community supported/Third party	universe	multiverse

Bash & Linux CLI

Custom repositories

Custom repositories

- ▶ We can add additional repositories to our apt sources
- ▶ **If we were to do this manual:**
 - ▶ We create a new file in `/etc/apt/sources.list.d`
 - ▶ **Usually:** We need to add the GPG key to our system for this repository
 - ▶ This is needed so that our system trusts the third-party repository
 - ▶ Usually wherever you find an installation guide for this, they will explain all the steps required
- ▶ **Important:**
 - ▶ The third-party repository could install any software on our system
 - ▶ It could for example say: I got bash in version 100000 - and our system would trust this and update it
 - ▶ Even if bash was installed from the official repository before
- ▶ **Let's have a look at an example:**
 - ▶ <https://wiki.winehq.org/Ubuntu>

Bash & Linux CLI

PPA on Ubuntu

Third party packages: ppa

► Personal Package Archive:

- ▶ This will usually work only on Ubuntu, not on Debian systems
- ▶ It's a website where users can easily provide repositories for others:
<https://launchpad.net/ubuntu/+ppas>
- ▶ For example, for the latest php version, we could just add a ppa:
▶ `sudo add-apt-repository ppa:ondrej/php`
 `sudo apt update`
- ▶ We would then have access to the latest php packages

▶ To remove a ppa:

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

▶ But:

- ▶ We're installing a third party repository here
- ▶ They could install any software on our system when we start a system
 update / upgrade
- ▶ We need to trust the authors, not just now but also in the future!

Bash & Linux CLI

Dependency management

Dependency management

- ▶ **Dependency management:**

- ▶ Let's say we got 2 tools, with at least one dependency:
 - ▶ bash: Wants libc6 >= 2.35
 - ▶ zsh: Wants libc6 >= 2.36

- ▶ **In this case, the solution is easy:**

- ▶ We can install libc6 in version 2.36
- ▶ And if we're currently on version 2.35, when we install zsh, we just upgrade libc6 to version 2.36
- ▶ So usually, `apt` should resolve dependency issues automatically

Show dependencies

- ▶ We can show the dependencies of packages with the following commands:
 - ▶ `sudo apt show bash`
 - ▶ `sudo apt-cache show bash`

Bash & Linux CLI

Conflict resolution

Conflict resolution

- ▶ **Dependency conflicts:**
 - ▶ Let's say we got 2 tools, with different dependencies:
 - ▶ bash: Wants libc6 <= 2.35
 - ▶ zsh: Wants libc6 >= 2.36
 - ▶ **We now would want to install both tools on our system**
 - ▶ **Which version of libc6 should we install?**
 - ▶ **We now have a problem:**
 - ▶ We can't satisfy either dependency requirement
 - ▶ Apt is not able to resolve this dependency conflict
 - ▶ **We must manually resolve this! But how?**

Our conflict (example):

Conflict resolution

```
bash: Wants libc6 <= 2.35  
zsh: Wants libc6 >= 2.36
```

- ▶ **How do we solve a dependency conflict?**

- ▶ **We ask apt to automatically fix the dependency issue for us:**
 - ▶ `sudo apt install -f / sudo apt-get install -f`
 - ▶ `-f` stands for "`--fix-broken`"
 - ▶ `apt` will analyze the current package state, identify inconsistencies, and solve them by installing, upgrading or removing packages
 - ▶ In 75%+ of the cases, this should resolve the problem (for packages from the default repositories)

Conflict resolution

- ▶ **How do we solve a dependency conflict? (Part 2)**
 - ▶ **We run an update / upgrade of our system:**
 - ▶ `sudo apt full-upgrade` / `sudo apt-get dist-upgrade`
 - ▶ This works especially well before we try to install additional software
 - ▶ **If this doesn't work:**
 - ▶ `sudo apt autoremove` / `sudo apt-get autoremove`
 - ▶ Let's automatically uninstall all packages that are no longer needed
 - ▶ This also sometimes solves the problem

Best practices

- ▶ Avoid third-party repositories (if possible)
- ▶ Avoid installing software with .deb files (if possible)
- ▶ Regularly install all normal software updates (always)
- ▶ **Before upgrading to a new Ubuntu version:**
 - ▶ Plan some extra time for dependency issues
 - ▶ Wait a month or two after release of new Ubuntu version
(especially when using third-party repositories)
 - ▶ Create a complete backup for your system
- ▶ Use **LTS** (=Long Term Support) versions of Ubuntu for servers -
less upgrades, less problems
- ▶ Consider using tools like docker to containerize your application

Bash & Linux CLI

Bonus: Trigger reconfigure with dpkg

Reconfiguring packages

- ▶ Sometimes packages asks us questions when we install them
- ▶ Or they run configuration when we install them on our system
- ▶ **Example:**
 - ▶ A Bootloader (for example: grub) might want to install itself as the main bootloader
 - ▶ A webserver might set itself up in a way that a user and group is created and that it automatically starts with our system
 - ▶ A display manager (manages the login screen) might start a dialog to ask us which one should be the default display manager
- ▶ Sometimes we want to re-run those scripts
- ▶ **We can do if with the following command:**
 - ▶ `sudo dpkg-reconfigure <package>`

Reconfiguring packages

- ▶ **I want to show you another example:**

- ▶ We install the package "locales" the first time
- ▶ This package asks us which locales we want to generate

- ▶ **What is a locale?**

- ▶ It defines how numbers and dates should be displayed
- ▶ A lot of programming languages rely on the operating system to provide this functionality
- ▶ For example, if we want to have a website (for example programmed in PHP), and we want to use the build-in functions to format dates
- ▶ The corresponding locale must be generated

- ▶ **To reconfigure a package:**

- ▶ `sudo dpkg-reconfigure locales`

Bash & Linux CLI

Verifying installation: debsums

Verifying installation: debsums

- ▶ A lot of packages ship with a list of md5 checksums of the files they're installing on our system
- ▶ We can use those checksums to verify if the files that have been installed are still identical
- ▶ **Important:**
 - ▶ md5 is considered insecure
 - ▶ A malicious actor might be able to generate an executable, that does something different, but still yields the same checksum
 - ▶ Still, it's enough for a 99%+ exact overview

Verifying installation: debsums

- ▶ We can verify all packages that contain an md5 sum with the command `debsums`:
 - ▶ `debsums [package / .deb file]`
 - ▶ **We might have to install it first:**
 - ▶ `sudo apt install debsums`
- ▶ **Important parameters:**
 - ▶ `-a`: List all files (including configuration files)
 - ▶ `-l`: List packages, that don't have a list of files with their md5 sum
 - ▶ `-s`: Silent, only list errors

Bash & Linux CLI

Snap packages

Package management with snap

- ▶ We have now seen how delicate `apt` works
- ▶ One dependency can be extremely problematic and cause `apt` to fail or to be unable to resolve it
- ▶ All the dependencies must be installed globally
- ▶ **Snap solves this:**
 - ▶ We bundle an application with all its dependencies
 - ▶ The download might be larger, because the dependencies are included
 - ▶ But this allows each application can have different dependencies
 - ▶ And this package can be independent from the Linux distribution
 - ▶ Packages should be updated automatically in the background (`snapd`), otherwise we can trigger it: `snap refresh`

How do snap packages work?

- ▶ **Snap is a centralized repository for complete applications:**
 - ▶ People can publish their applications to that repository
 - ▶ If we install a snap application, we should make sure that we trust the authors!
 - ▶ We can have a look at the available packages here:
 - ▶ <https://snapcraft.io/>
- ▶ **We can install an application:**
 - ▶ `snap install gimp`

Bash & Linux CLI

Package management: Red Hat / CentOS

A quick heads-up

- ▶ In this chapter, I will assume that you have skipped the chapter "*Package management on Ubuntu*"
- ▶ If you have watched the previous chapter, you might encounter quite a few similarities
- ▶ **But we will also focus on a different package approach:**
 - ▶ We will have a look at DNF modules, and how they will allow you to choose which version of a specific software you want to install
 - ▶ Thus, there's less need on Fedora / CentOS / Red Hat for that many third-party repositories

What is package management?

- ▶ **Definition:**

- ▶ Process of installing, updating, configuring, and removing software

- ▶ **Purpose:** Streamline software handling on a Linux system

- ▶ **Why do we need it?**

- ▶ Facilitates easy software distribution
 - ▶ Simplifies software installation
 - ▶ Ensures software compatibility
 - ▶ Handles dependencies
 - ▶ Maintains system stability and security

- ▶ **But we need to be careful:**

- ▶ Package management is quite different between different Linux distributions
 - ▶ In this chapter, we will look into package management on CentOS

Package management: Red Hat / CentOS

- ▶ **Simplified package flow:**

- ▶ Fedora -> CentOS Stream -> RHEL

- ▶ **Red Hat Enterprise Linux:**

- ▶ CentOS Stream is a rolling preview of RHEL
 - ▶ Thus, package management should behave extremely similar
 - ▶ This chapter also prepares you for RHEL
 - ▶ There will just be less potential issues in RHEL
(better tested system)

- ▶ **It will even prepare you for Fedora:**

- ▶ Fedora is a rolling preview of CentOS Stream

This chapter

► The goal:

- You're not just able to install and manage software on a system
- You will learn everything you need to know to avoid critical situations
- But you will also learn how to handle critical situations

Bash & Linux CLI

The .rpm package format

The .rpm package format

- ▶ **.rpm without package manager:**

- ▶ .rpm are archives that contain all the files & configuration required to install a software

- ▶ **We could download .rpm files manually, for example from:**

- ▶ <https://mirror.stream.centos.org/>
- ▶ We should usually avoid this, as it can easily lead to conflicts
- ▶ Let's still do it, to have a look at what CentOS will do for us automatically

- ▶ **We can then inspect the .rpm file:**

- ▶ `rpm -qpl [file].rpm`

- ▶ **Or install the .rpm file manually:**

- ▶ `rpm -i [file].rpm`

Bash & Linux CLI

Let's use DNF!

Package management on CentOS

- ▶ **Package management on CentOS:**

- ▶ It uses .rpm packages
 - ▶ For this, it uses the DNF package manager ("Dandified yum")
 - ▶ DNF replaced the yum package manager
- ▶ **Thus, we can access DNF through the following command:**
- ▶ dnf (and usually also available through yum)

Managing software

- ▶ **We can use dnf to search for software:**

- ▶ `dnf search [term]`

- ▶ **And we can install software through it:**

- ▶ `dnf install links`

- ▶ With this command, dnf will first install all the required dependencies, and
install the tool

- ▶ **And uninstall software:**

- ▶ `dnf remove links`

Bash & Linux CLI

Repositories

Repositories: Where do our packages come from?

- ▶ **Let's say we want to install a tool:**

- ▶ `dnf install gimp`

- ▶ How does DNF know where to download the software from?

- ▶ This is what repositories are for!

- ▶ **How do they work?**

- ▶ We can define those repositories in the following files:

- ▶ `/etc/dnf/dnf.conf`

- ▶ `/etc/yum.repos.d/*.repo`

- ▶ **A full system of RHEL / CentOS needs at least:**

- ▶ BaseOS

- ▶ AppStream

- ▶ We don't have to refresh this index manually

Bash & Linux CLI

What are dependencies?

What are dependencies?

► **Software Dependencies:**

- ▶ Required resources by a software for its proper functioning
- ▶ Includes other software, libraries, or operating system features
- ▶ DNF will handle the resolution of dependencies for us
- ▶ This will happen recursively - also dependencies of dependencies will be installed!

► **Why do we need dependencies?**

- ▶ Dependencies only need to be installed 1x on our system
- ▶ This saves storage space, and ensures applications all use the same dependencies

Provides, requires,...

- ▶ **Each package:**

- ▶ Can provide certain features:
 - ▶ `dnf repoquery --provides [package_name]`
- ▶ ... and also require certain features:
 - ▶ `dnf repoquery --requires [package_name]`

- ▶ **We can also use this to find packages:**

- ▶ We can also list which package provides a certain feature:
 - ▶ `dnf repoquery --whatprovides`
- ▶ We can also list which package requires a certain feature:
 - ▶ `dnf repoquery --whatrequires`

- ▶ **We can list the dependencies with the following command:**

- ▶ `dnf deplist [program]`

Bash & Linux CLI

What are weak dependencies?

What are weak dependencies?

- ▶ We have different "levels" of dependencies:

- ▶ **Required dependencies:**

- ▶ Essential for our software
 - ▶ Without those, we can't execute the program

- ▶ **Weak dependencies:**

- ▶ Optional software dependencies
 - ▶ They enhance the functionality, but aren't required

Types of weak dependencies

- ▶ **Types of weak dependencies:**

- ▶ **Recommends:**

- ▶ The package recommends another package
 - ▶ Most users would also want to install this other package

- ▶ **Suggests:**

- ▶ The package can use another package - but usually we wouldn't need this other package

- ▶ **By default:**

- ▶ DNF will install recommended weak dependencies as well
(if it doesn't trigger a conflict)

What are weak dependencies?

- ▶ **Querying:**
 - ▶ **List recommended weak dependencies:**
 - ▶ `dnf repoquery --recommends [package_name]`
 - ▶ **Lists suggested weak dependencies:**
 - ▶ (mostly for informational purposes)
 - ▶ `dnf repoquery --suggests [package_name]`
- ▶ **If we want to disable weak dependencies:**
 - ▶ We can set this option in `/etc/dnf/dnf.conf`:
 - ▶ `install_weak_deps=False`
 - ▶ Or set it as a parameter when installing:
 - ▶ `dnf install [package_name] --setopt=install_weak_deps=False`

Bash & Linux CLI

What are backward weak dependencies?

What are backward weak dependencies?

- ▶ **So far, our dependencies were mostly forward dependencies:**

- ▶ We want to install package B that depends on package A
 - ▶ Thus, we also install package A

- ▶ **What are backwards dependencies?**

- ▶ We have a package B that we want to install
 - ▶ We can imagine a backward dependency like this:
 - ▶ Another package C wants to say: *Hey, I'd enhance the functionality of B, when installing B, install me as well*
 - ▶ This is especially useful for third-party repositories, so they can connect to the install of existing packages

- ▶ **But we can also see it with language packs:**

- ▶ Let's have a look at: `langpacks-de`

Types of backward weak dependencies

► Supplements:

- ▶ This package will be installed (as a weak dependency) if another package is installed
- ▶ `dnf repoquery --supplements [package_name]`
- ▶ Of course, we can also check which package supplements another one:
 - ▶ `dnf repoquery --whatsupplements [package_name]`

► Enhances:

- ▶ This package could enhance the functionality of another package. But do not install by default
- ▶ `dnf repoquery --enhances [package_name]`

Bash & Linux CLI

Dependencies & removing packages

Automatic dependency uninstall

- ▶ **dnf automatically removes no longer needed dependencies**
- ▶ **Let's have a look at an example:**
 - ▶ `python3-matplotlib`
 - ▶ This tool has a dependency to `python3-numpy` (among many others)
 - ▶ Thus, when we install `python3-matplotlib`, `python3-numpy` is installed as well
- ▶ **We can check that:**
 - ▶ `python3 -c 'import numpy as np'`
 - ▶ **This will work!**
- ▶ **If we now uninstall `python3-matplotlib`:**
 - ▶ The package `python3-numpy` is uninstalled as well
 - ▶ If our application would depend on `numpy`... it would now fail:
 - ▶ `python3 -c 'import numpy as np'`

Automatic dependency uninstall

- ▶ **How to avoid this?**
 - ▶ **We could set an option to False in /etc/dnf/dnf.conf:**
 - ▶ `clean_requirements_on_remove=False`
 - ▶ And then we would have to remove old dependencies manually:
 - ▶ `dnf autoremove`
 - ▶ **We could remove a package without its dependencies:**
 - ▶ `dnf remove [package] --noautoremove`
 - ▶ **We can mark packages as installed manually:**
 - ▶ `dnf mark install [package]`

Bash & Linux CLI

DNF: Update & downgrade package(s)

DNF: Update packages: dnf upgrade

- ▶ Of course, we want to keep our system up to date
- ▶ **For this, we can use the following command:**
 - ▶ `dnf upgrade`
 - ▶ This will ensure the repository indexes are up to date
(and refresh them if required)
 - ▶ And install all available updates
 - ▶ **Let's have a look at this!**

DNF: How to downgrade a package

- ▶ **Usually, all updates should be sufficiently safe**
 - ▶ Especially on RHEL, but also on CentOS
 - ▶ They are important security updates, and minor updates
 - ▶ Large updates only happen when we update to a completely new version of CentOS / RHEL (more on that later)
- ▶ But in rare circumstances, we might still need to downgrade a package

DNF: How to downgrade a package

- ▶ Let's say an update is causing issues in our application
- ▶ Can we just downgrade a package?
- ▶ **For this, we can use the following command:**
 - ▶ `dnf downgrade [package]`
 - ▶ `dnf downgrade [package-with-version]`
- ▶ **Important:**
 - ▶ We should never downgrade the following packages: `dbus`, `glibc`, `selinux-policy`
 - ▶ We should never downgrade a system to a previous minor version of CentOS / RHEL
- ▶ **Example:** We should never downgrade from CentOS 9.2 to 9.1
- ▶ **We can list available versions:**
 - ▶ `dnf list [package] --showduplicates`

Bash & Linux CLI

DNF: How to prevent updates

How to prevent upgrade

- ▶ **Sometimes we want to prevent a package from being updated:**

- ▶ **Examples:**

- ▶ A more recent version is not compatible with our application
 - ▶ We have only tested our application against the previous version, and consider this to be more important than installing a bugfix / security update
 - ▶ We are relying on proprietary plugins that only run on a specific version
 - ▶ We need to manually compile a kernel module for each version - and thus, we might want to prevent our kernel from being updated automatically
 - ▶ We have downgraded a package and want to prevent it from being updated

How to prevent packages from being updated

- ▶ **If we want to prevent a package from being updated:**

- ▶ We can temporarily exclude it:
 - ▶ `dnf upgrade --exclude=[package]`
- ▶ Or permanently exclude it in our `/etc/dnf/dnf.conf`:
 - ▶ `excludepkgs=[packages]`

- ▶ **Aside from this, there's also a plugin for dnf:**

- ▶ `dnf-versionlock`
- ▶ **We will not have a look at this in this course**

Be careful

- ▶ Preventing a package from being updated can cause problems with the dependencies
- ▶ **Best practice:**
 - ▶ We should review those packages from time to time and consider if it's necessary to block updates
 - ▶ Before large system upgrades (more on those later), we should try to remove all excluded packages
 - ▶ This will allow DNF to update all software, and prevent many potential dependency issues

Bash & Linux CLI

DNF: Automatically updating the system

Important

► **General rule:**

- ▶ If a machine is a critical server, we should not use automatic updates
- ▶ If a machine is not critical (and unplanned downtime is acceptable) we can consider opting into automatic updates

DNF: Automatically keep the system up to date

- ▶ **First, we need to install dnf-automatic:**
 - ▶ `sudo dnf install dnf-automatic`
- ▶ **After this, we should have a look at the configuration file:**
 - ▶ `sudo nano /etc/dnf/automatic.conf`
- ▶ **And we need to enable the timer:**
 - ▶ `sudo systemctl enable --now dnf-automatic.timer`

Bash & Linux CLI

DNF: Modules

DNF: Modules



- ▶ In this lecture, we'll have a look at DNF modules
- ▶ This is one of the key features of the package management in CentOS / RHEL
- ▶ **The idea:**
 - ▶ For certain applications, we want to install specific versions of software
- ▶ **Example:**
 - ▶ Node.js
 - ▶ PHP
 - ▶ Database systems
- ▶ **Let's examine the problem with an example of Node.js:**
 - ▶ Node.js is a runtime that allows us to execute JavaScript on the server
 - ▶ We can write a highly concurrent web application with it
 - ▶ So what's the problem?

Why do we need modules?

- ▶ **The main requirement for an Enterprise Linux:**

- ▶ It needs to work reliably
- ▶ We do want software updates (security updates) for several years
- ▶ But other updates should be avoided if not necessary

- ▶ **This means:**

- ▶ We'd rather stick to an old version of Node.js for 10+ years
(with occasional security updates), instead of keeping it up to date
- ▶ And if we upgrade, we want this upgrade to be on our terms

- ▶ **The consequence:**

- ▶ Node.js will and should be outdated in the repositories
- ▶ If we want to use the latest JavaScript features, we can subscribe to
those updates manually

DNF: Modules

- ▶ **Luckily, dnf supports modules:**

- ▶ A dnf module is a package group - it usually represents an application, language runtime or a set of tools
- ▶ This allows us to choose the version we want to install

- ▶ **Listing available modules:**

- ▶ `dnf module list`

- ▶ **Enabling a module:**

- ▶ `dnf module enable [name]`
- ▶ `dnf module enable [name]:[stream]`

Bash & Linux CLI

DNF: Modules (part 2)

DNF: Modules

- ▶ **We can also install the module directly:**

- ▶ `dnf module install [name]`
- ▶ `dnf module install [name]:[stream]/[profile]`
- ▶ `dnf install @[name]:[stream]/[profile]`

Disabling and uninstalling a module

- ▶ **We can also remove installed module profiles:**

- ▶ This will uninstall the software that was installed through the profile
- ▶ `dnf module remove [--all] [name]`
- ▶ `dnf module remove [name]/[profile]`
- ▶ The `--all` flag will uninstall all software that was provided by this module

- ▶ **We can also disable a module:**

- ▶ The software will remain installed
- ▶ The module streams will become unavailable
- ▶ `dnf module disable [name]`

- ▶ **We can also reset a module:**

- ▶ This will reset the stream & profile to the default
- ▶ Software will not be uninstalled
- ▶ `sudo dnf module reset [name]`

Bash & Linux CLI

The repository: epel-release

The repository: epel-release

- ▶ **The idea:**

- ▶ EPEL: Extra Packages for Enterprise Linux
- ▶ It's a community-driven project, to offer high-quality Fedora packages for RHEL
- ▶ The goal: All packages should seamlessly install on RHEL
- ▶ But still, it's "just" a community project
- ▶ Thus, we get no enterprise support for those packages

- ▶ **Proceed with caution:**

- ▶ Especially on RHEL...
- ▶ ... and if you rely on long term support / enterprise support

- ▶ **But what about CentOS Stream?**

- ▶ Some packages might slightly differ between RHEL and CentOS Stream
- ▶ Thus, on CentOS Stream, we will also have to enable `epel-next-release`

How to enable epel-release

- ▶ **How to enable epel-release on CentOS Stream:**
 - ▶ On CentOS stream, we should be able to just install the additional repositories:
 - ▶ `dnf config-manager --set-enabled crb`
 - ▶ `sudo dnf install epel-release epel-next-release`
- ▶ **On Red Hat, the command might be different:**
 - ▶ Be sure to check out the documentation:
 - ▶ <https://docs.fedoraproject.org/en-US/epel/>

Bash & Linux CLI

DNF: Resolving dependency issues

Bash & Linux CLI

Optional: Snap packages

Snap on RHEL?

- ▶ **If you're running RHEL:**

- ▶ You need to consider if you really want to use snap
- ▶ Snap is part of the EPEL project (epel-release, no support)

- ▶ **Snap might break commercial support**

- ▶ **But it allows us to access certain software:**

- ▶ RHEL / CentOS Stream by default ships with an ER version of Firefox (ER = Extended Release)
- ▶ But let's say we're a web developer... and depend on the latest version of Firefox for our work (and we're using RHEL)
- ▶ In that case snap might be an option for us

- ▶ **Be aware:**

- ▶ This is something you may need to discuss with your system administrator

Package management with snap

- ▶ We have now seen how `dnf` works
- ▶ The dependency resolution should usually work without any issues
- ▶ **But still:**
 - ▶ All the dependencies must be installed globally
- ▶ **Snap solves this:**
 - ▶ We bundle an application with all its dependencies
 - ▶ The download might be larger, because the dependencies are included
 - ▶ But this allows each application can have different dependencies
 - ▶ And this package can be independent from the Linux distribution
 - ▶ Packages should be updated automatically in the background (`snapd`), otherwise we can trigger it: `snap refresh`

How do snap packages work?

- ▶ **Snap is a centralized repository for complete applications:**
 - ▶ People can publish their applications to that repository
 - ▶ If we install a snap application, we should make sure that we trust the authors!
 - ▶ We can have a look at the available packages here:
 - ▶ <https://snapcraft.io/>
- ▶ **First, we need to install snap:**
 - ▶ **EPEL must be activated for this!**
 - ▶ `dnf install snapd`
 - ▶ `systemctl enable --now snapd.service`
- ▶ **We can install an application:**
 - ▶ `snap install firefox`

Bash & Linux CLI

The system boot process & systemd

In this chapter

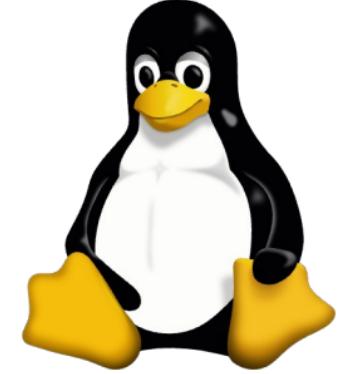
- ▶ **In this chapter, you will learn how the boot process works on Linux:**
 - ▶ How does the boot process work?
 - ▶ How does your system end up with a nice user interface?
 - ▶ How does a server start all the services?
 - ▶ How can we start a program that runs automatically in the background?
- ▶ **In order to understand this, you will learn about:**
 - ▶ The bootloader (GRUB 2)
 - ▶ Kernel
 - ▶ systemd
- ▶ We will not talk about the older init.d / sysvinit system

Bash & Linux CLI

Step 1: The bootloader

The bootloader: GRUB 2

- ▶ After the BIOS / UEFI, the bootloader is the first software to load run on startup
- ▶ Its goal is to load the operating system
- ▶ **How does it do it?**
 - ▶ It loads the kernel into memory and then hands control to it
 - ▶ The bootloader is only responsible for this and thus active for only a very short time
- ▶ **We can change the configuration of GRUB in the following file:**
 - ▶ `/etc/default/grub`
 - ▶ To update the GRUB configuration, after each change we must run:
 - ▶ **On Ubuntu:**
 - ▶ `sudo update-grub`
 - ▶ **On CentOS:**
 - ▶ `sudo grub2-mkconfig -o /boot/grub2/grub.cfg`
 - ▶ This will update the `/boot/grub/grub.cfg` file (we should never edit this file directly, as it would be overwritten during updates)



Bash & Linux CLI

Step 2: The kernel

Key functions of the kernel

▶ **Process management**

- ▶ Scheduling and resource allocation
- ▶ Inter-process communication (IPC)

▶ **Memory management**

- ▶ Physical and virtual memory
- ▶ Handles allocation and deallocation

▶ **File system management**

- ▶ supports various file systems (e.g., ext4, XFS, Btrfs)
- ▶ Handles the read and write to those

▶ **Networking stack**

- ▶ Implements various network protocols (e.g., TCP/IP, Ethernet)
- ▶ Routing, packet filtering, and traffic control

▶ **Hardware abstraction layer (HAL):**

- ▶ Enables applications to communicate with various devices

What are kernel modules?

- ▶ **Kernel module:**

- ▶ Piece of code that can be loaded into the kernel on demand
- ▶ Allows us to extend the functionality of the kernel

- ▶ **A few examples:**

- ▶ **Device drivers:**

- ▶ Nvidia drivers
- ▶ Broadcom Wireless Drivers:
 - ▶ For some chipsets, proprietary drivers can enhance performance

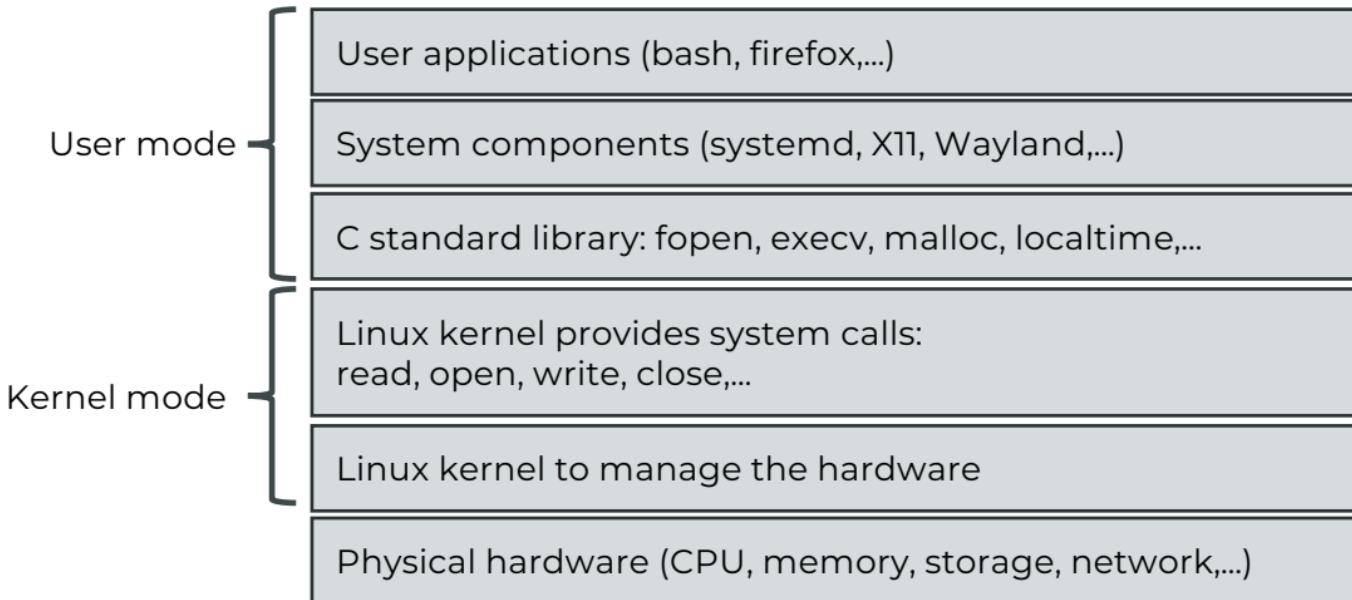
- ▶ **VFIO:**

- ▶ Virtual Function I/O
- ▶ Allows a Guest Operating System to access the host GPU

- ▶ **Others:**

- ▶ VirtualBox Guest Addons
- ▶ ZFS filesystem

How do we communicate with the hardware?



Bash & Linux CLI

Step 3: systemd

Step 3: systemd

- ▶ After the bootloader has given control to our kernel, the kernel will initialize and start the main process (PID = 1).
- ▶ On most Linux systems nowadays, it's the systemd process
- ▶ Let's first have a look at how this looks in the output of the ps program:
 - ▶ `ps 1`
- ▶ **This process will then initiate all the rest:**
 - ▶ Starting the system
 - ▶ Mounting drives
 - ▶ Starting services
 - ▶ Configuring network connections
 - ▶ Starting additional applications

The tool: systemd

- ▶ systemd is not just a tool, it's an extensive set of tools
- ▶ We will not be able to explore all of them
- ▶ **I want us to be guided by a few simple problems:**
 - ▶ We want a webserver to be launched in the background
 - ▶ We want systemd to execute a command on every boot
 - ▶ We want to have a command to be run every few minutes - no matter if we're logged in or not
- ▶ This will be enough to give you at least an overview about systemd



Bash & Linux CLI

The controversy about systemd

The controversy about systemd

- ▶ systemd seems to be rather polarizing in the Linux community
- ▶ **First:**
 - ▶ I want to distance myself from this discussion - it's used in quite a few Linux distributions, and we should know how it works
- ▶ **Critics said:**
 - ▶ It's overly complex, it's suffering from feature creep
 - ▶ Its architecture violates the Unix philosophy
 - ▶ (each program should do one thing well, programs should work together, and use standard input and output to allow combining)
 - ▶ It can only run on Linux systems, not on all Unix systems
 - ▶ Doing many things in parallel might be decremental of performance in certain scenarios (booting from HDD or DVD)

The advantages of systemd

► Systemd has enormous advantages

- A system can be highly dynamic, configuration might change
 - **Example:** We might have a server that only launches if we access a certain port on our machine
- Parallelizing the boot process makes sense on somewhat modern drives (even on slow SSDs)

► It offers many unique features, just for example:

- Just as an example, we can create a setup so that we can upgrade a service, and no connections are being lost
- We can have a backup script, that requires that a certain mount is present - and we can freely specify those dependencies

► No matter your side of this argument:

- systemd is currently used by many Linux distributions - we thus should learn it and become familiar with it



Bash & Linux CLI

systemd: General structure

systemd: General structure

- ▶ **system mode:**

- ▶ systemd manages the boot process and starts all services required for this (in parallel)
- ▶ It reads in configuration files (unit files), builds the dependency graph, and then executes all commands necessary to get to the result

- ▶ **user mode:**

- ▶ The same as system mode, but for user services (after the user logins)
- ▶ We will focus on system mode in this course

systemd

- ▶ **How does it do it?**

- ▶ **Let's have a look at the first 2 basic building blocks:**

- ▶ **Unit:**

- ▶ Basic building block
 - ▶ Various types available: Service, Target, Timer,...
 - ▶ Units can depend on another

- ▶ **Service:**

- ▶ Represents a service that should run on our system
 - ▶ Can be enabled, disabled, started, stopped, restarted, reloaded,...
 - ▶ Managed by .service unit files

Systemd manages "Units"

- ▶ systemd searches units in various paths on our system
- ▶ **Those paths are usually:**
 - ▶ `/lib/systemd/system` (sometimes also `/usr/lib/systemd/system`):
 - ▶ System configuration
 - ▶ This is the "default" place for configuration from the maintainer
(=authors of our Linux distribution / packages)
 - ▶ `/run/systemd/system`:
 - ▶ Non-persistent, runtime configuration
 - ▶ `/etc/systemd/system`:
 - ▶ Configuration for the administrator
 - ▶ Overwrites files from `/lib/systemd/system`
 - ▶ This is where we should store our custom configuration
- ▶ **We could find those paths with the following CLI command:**
 - ▶ `systemctl-analyze --system unit-paths`
- ▶ **Let's have a look at those folders!**



Bash & Linux CLI

systemd: How to manage a unit

How do we manage a unit?

- ▶ We can manage a unit with the following commands:
- ▶ **Get the status of a unit:**
 - ▶ `systemctl status [unit]`
- ▶ **Change the status of a unit:**
 - ▶ `systemctl {start, stop, restart, reload} [unit]`
 - ▶ start: starts a unit, stop: stops a unit, restart: restarts a unit
 - ▶ reload: asks the unit to reload its configuration (important: this is not the systemd configuration of this unit)
- ▶ **Example (apache2 needs to be installed for this):**
 - ▶ `systemctl start apache2.service`
 - ▶ `systemctl stop apache2.service`
 - ▶ `systemctl status apache2.service`



Bash & Linux CLI

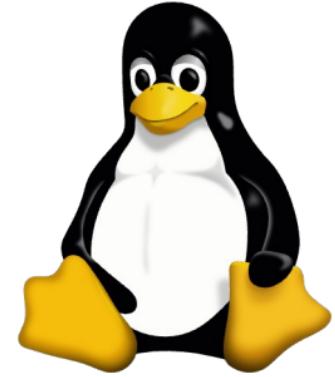
What is a cgroup?

What is a cgroup?

- ▶ Control groups (cgroups) are a feature of the Linux kernel
- ▶ It organizes processes hierarchically
- ▶ And allows us to more evenly distribute system resources
- ▶ Systemd heavily relies on cgroups to manage processes
- ▶ **Advantages**
 - ▶ A cgroup can span multiple processes - and even contain other cgroups
 - ▶ If we start a sub-process, it will automatically be in the same cgroup as the parent
- ▶ **Uses of cgroups include:**
 - ▶ Resource limiting:
 - ▶ We can configure a cgroup to not exceed a specific memory limit
 - ▶ We can define how much % of CPU resources this cgroup may occupy
 - ▶ We can also measure how much resources certain systems consume
 - ▶ ... or freeze a group of processes
- ▶ This can be useful when multiple services (in an application sense) / containers share the same server

How to inspect cgroup(s)?

- ▶ **Show processes on our system:**
 - ▶ `systemctl status`
- ▶ **If we want to inspect cgroups:**
 - ▶ We can use the following command for this:
 - ▶ `systemd-cgtop`
 - ▶ By default, it displays up to 3 levels of cgroups
 - ▶ We can change this level:
 - ▶ `--depth=5`



Bash & Linux CLI

Example for cgroups: Firefox @ 100MB



Cgroup example: Limit Firefox to 100MB RAM

- ▶ As an example, can we limit Firefox to 100MB of memory?

- ▶ **First step:**

- ▶ We need to create a systemd slice in our user account.

Here, I call the slice "browser":

- ▶ `~/.config/systemd/user/browser.slice`:

```
[Slice]  
MemoryHigh=100M
```

- ▶ And after this, we can launch Firefox. If the Firefox executable is placed at `/usr/bin/firefox`, the command is as follows:

- ▶ `systemd-run --user --slice=firefox.slice /usr/bin/firefox`

- ▶ **Important:**

- ▶ If the Firefox executable is a script... the script might change the cgroup configuration
 - ▶ In this case, we need to find out the real path to the real executable of Firefox

Bash & Linux CLI

Systemd: Targets

systemd: Targets

- ▶ **Target:**

- ▶ Groups units logically to a goal
- ▶ Managed by .target unit files

- ▶ **To get the (current) default target:**

- ▶ `sudo systemctl get-default`

- ▶ **To switch target, without rebooting:**

- ▶ `sudo systemctl isolate multi-user.target`

- ▶ **We can list available targets with the following command:**

- ▶ `sudo systemctl list-units --type target --all`

- ▶ **We could also change the current default target:**

- ▶ `sudo systemctl set-default multi-user.target`

Bash & Linux CLI

systemd: How to enable / disable a unit

systemd: How to enable / disable a unit

- ▶ Let's say we have a webserver...
- ▶ How is it started automatically?
- ▶ **We can enable a unit:**
 - ▶ `sudo systemctl enable --now apache2.service`
 - ▶ This will read the corresponding configuration file
(`apache2.service`)
 - ▶ If specified in the config file, it will be loaded on boot
 - ▶ Because of the `--now` parameter, this unit will also be loaded immediately
- ▶ **We can also disable a unit:**
 - ▶ `sudo systemctl stop apache2.service`
 - ▶ `sudo systemctl disable apache2.service`
 - ▶ This unit will then no longer be loaded on boot

Bash & Linux CLI

systemd: The format of a unit file

The unit config file

- ▶ A config file for a unit of a service can consist out of several sections:
 - ▶ [Unit]:
 - ▶ A general configuration for the unit
 - ▶ [Service]:
 - ▶ If this unit is a service, we should write all the configuration for the service into this section
 - ▶ [Install]:
 - ▶ Here, we specify, how the unit should be installed
- ▶ Let's have a look at this!

[Unit]

- ▶ **Description:**

- ▶ Brief explanation, helps users understand the purpose of the service

- ▶ **Documentation:**

- ▶ Links to relevant documentation

- ▶ **Requires:**

- ▶ Ensures other units are activated before the current unit
 - ▶ If the required unit fails to start, the current unit will not start

- ▶ **Wants:**

- ▶ Similar to Requires, but the current unit will start even if the wanted unit fails
 - ▶ Useful for optional dependencies

- ▶ **After:**

- ▶ Ensures the current unit starts after specified units
 - ▶ Helps to define the order of unit activation

- ▶ **Before:**

- ▶ Ensures the current unit starts before specified units
 - ▶ Also helps to define the order of unit activation

[Service]

- ▶ **Type:**

- ▶ Defines the process type and startup behavior
- ▶ Examples: simple, exec, forking, oneshot,...

- ▶ **ExecStart:**

- ▶ Command to start the service
- ▶ Can include arguments and options
- ▶ But it's not a full bash command!

- ▶ **ExecStop:**

- ▶ Command to stop the service (optional)

- ▶ **Restart:**

- ▶ Defines when the service should be automatically restarted
- ▶ Examples: no, on-success, on-failure, on-abnormal, on-abort, always

- ▶ **User:**

- ▶ User under which the service should be run

- ▶ **Environment:**

- ▶ Defines environment variables for the service

[Install]

► **WantedBy:**

- ▶ Specifies the target(s) that should include this unit as a dependency
- ▶ Common targets: multi-user.target, graphical.target
- ▶ Enables the unit to be started automatically at boot when "systemctl enable" is used

Bash & Linux CLI

systemd: How to edit a unit?

How can we edit a unit?

- ▶ How can we edit a unit file?
- ▶ **We can do it manually:**
 - ▶ We can copy the unit file from `/lib/systemd/system` to `/etc/systemd/system`
 - ▶ This will then take precedence over the original file in `/lib/systemd/system`
- ▶ **After this, we need to inform systemd about those changes:**
 - ▶ `sudo systemctl daemon-reload`

Bash & Linux CLI

systemd: How to edit a unit? (Part 2)

How can we edit a unit?

- ▶ **The easier way is to use build-in commands for this:**

- ▶ `sudo systemctl cat apache2.service`
 - ▶ This prints out the current configuration of apache2.service
- ▶ `sudo systemctl edit apache2.service`
 - ▶ This allows us to easily edit the configuration
 - ▶ We can only override this with the changes that we did -
we don't need to copy'n'paste the whole configuration

- ▶ **Internally, a new folder will be created:**

- ▶ `/etc/systemd/system/apache2.service.d`
 - ▶ From this folder, override files will be loaded, that can
change certain parts of the initial configuration

Bash & Linux CLI

systemd: Can we launch our own program?

Project: Let's launch our own program on boot!

- ▶ In this lecture, we want to create a new unit file for a new service
- ▶ It should launch on every boot and log the current time and ping a server
- ▶ **The goal in this lecture:**
 - ▶ You will get an overview about how a service on systemd works
 - ▶ You will be able to use the code from this lecture as a base for your own services

Our own program

- ▶ **This is quite simple:**

- ▶ We can create a file, for example `my-network-log.service` with the following contents:

- ▶ **[Unit]**

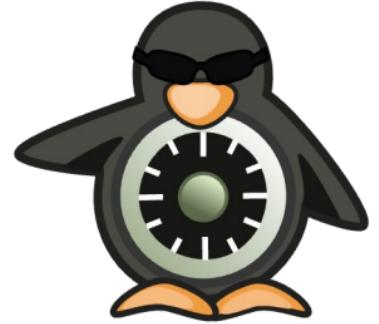
```
Description=Ping a server and log it
Requires=network.target
```

- [Service]**

```
Type=oneshot
StandardOutput=append:/network-log/log.txt
ExecStart=date '+%%T'
ExecStart=ping -c 4 google.com
```

- [Install]**

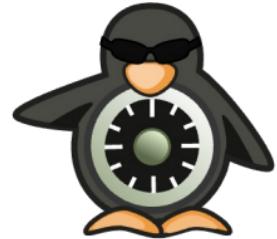
```
WantedBy=multi-user.target
```



Bash & Linux CLI

SELinux: Security-Enhanced Linux

SELinux: Security Enhanced Linux



- ▶ The idea:

Bash & Linux CLI

Bonus: Timers with `systemd`

Bonus: Timers with systemd

- ▶ With systemd, we can schedule tasks for a later date
- ▶ **Our task in this lecture:**
 - ▶ In the last lecture, we already created a tool to log the output of a few commands
 - ▶ In this lecture, we want to delay this by 5 minutes (for example for performance reasons)
- ▶ **How do we do this?**

Bonus: Timers with systemd

- ▶ First, we need a service, that is disabled:
 - ▶ sudo systemctl disable my-network-log.service
- ▶ Then, we can create a timer by creating the file
/etc/systemd/system/my-network-log.timer:
 - ▶ [Unit]
Description=Run the network logging service on boot
 - ▶ [Timer]
OnActiveSec=5min
 - ▶ Unit=my-network-log.service
 - ▶ [Install]
WantedBy=timers.target



Bash & Linux CLI

Bonus: Repeated timers with systemd

Bonus: Timers in systemd

- ▶ **Calendar events:**

- ▶ Refer to one or more points in time
- ▶ We can specify times (with wildcards) on which a service should be run

- ▶ **We can see the date format:**

- ▶ `systemd-analyze timestamp now`

- ▶ **Example:**

- ▶ [Timer]
`OnCalendar=*-*-* *:0,15,30,45`

- ▶ **We can use `systemd-analyze` to help us analyze the placeholders in the calendar format:**

- ▶ `systemd-analyze calendar '*-*-* *:0,15,30,45'`

Bash & Linux CLI

systemd and logging: journald



systemd-journald: Introduction

► **systemd-journald:**

- ▶ Part of the systemd suite
- ▶ Manages system logs
- ▶ Replaces traditional syslog

▶ **Key features:**

- ▶ Binary format for efficient storage
- ▶ Centralized logging solution
- ▶ Automatic log rotation and retention
- ▶ Indexing and querying capabilities
- ▶ Also logs messages that happened during boot



journalctl: basic usage

- ▶ **Display all logs:** journalctl
- ▶ **Important parameters:**
 - ▶ **Only show current boot:** journalctl -b
 - ▶ **Show all available boots:** journalctl --list-boots
 - ▶ **Filter by unit:** journalctl -u <unit>
 - ▶ **Filter by time range:**
 - ▶ journalctl --since "<time>" --until "<time>"
 - ▶ **Reverse output:** journalctl -r
 - ▶ **Follow logs in real-time:** journalctl -f
- ▶ **We can also send a message into the journalctl log:**
 - ▶ echo 'message' | systemd-cat

Bash & Linux CLI

Mounts and volumes

In this chapter

- ▶ **You will learn:**

- ▶ How does a filesystem work?
- ▶ How mounts work, and how you can use them to access external media
- ▶ How you can use the `/etc/fstab` to permanently declare mounts
- ▶ How you can mount external FTP servers into your filesystem to use those with all existing tools

- ▶ **And an important practical example:**

- ▶ How you can deal with a broken filesystem to still access your data



Bash & Linux CLI

Structure of a storage device (HDD, SSD, ...)



Structure of a storage device

- ▶ **Please note:**

- ▶ In this lecture, I'm mostly referring to HDDs, SSDs, and USB drives
- ▶ I'm excluding optical discs, or other storage devices (tape storage, cloud storage,...) from this lecture

- ▶ **What's the structure of such a storage device?**

- ▶ **Partition table:**

- ▶ **MBR (Master Boot Record):**

- ▶ Older partitioning scheme
- ▶ Limited to 4 primary partitions
- ▶ Limited to 2TiB disk size

- ▶ **GPT (GUID Partition Table):**

- ▶ Modern partitioning scheme
- ▶ Supports up to 128 partitions
- ▶ Supports larger disks

- ▶ We can visualize this with tools like `gparted`

Bash & Linux CLI

MiB vs. MB, GiB vs. GB,...

KiB vs. KB, MiB vs. MB, GiB vs. GB

- ▶ In the next lecture, I will create a partition
- ▶ For this, I will set the size of it to 50.000MiB...
- ▶ And then it will show up as just ~48GB
- ▶ **Why is that?**
 - ▶ Those 2 measurements are in different units!
 - ▶ 8 Bit = 1 Byte
 - ▶ 1 Kibibyte (KiB) = 1024 Byte (2^{10} Byte)
 - ▶ 1 Kilobyte (KB) = 1000 Byte (10^3 Byte)
 - ▶ Let's have a look at this in a table!

KiB vs. KB, MiB vs. MB, GiB vs. GB

Binary		Decimal		Difference
1024 bytes	KiB (kibibyte)	1000 bytes	kB (kilobyte)	2.4%
1024^2 bytes	MiB (mebibyte)	1000^2 bytes	MB (megabyte)	4.9%
1024^3 bytes	GiB (gibibyte)	1000^3 bytes	GB (gigabyte)	7.4%
1024^4 bytes	TiB (tebibyte)	1000^4 bytes	TB (terabyte)	10.0%
1024^5 bytes	PiB (pebibyte)	1000^5 bytes	PB (petabyte)	12.6%

JEDEC standard

- ▶ But now it will get confusing
- ▶ **The JEDEC standard from the 90s defines the following:**
 - ▶ 1 megabyte = 1024^2 bytes
 - ▶ 1 gigabyte = 1024^3 bytes
- ▶ **This is contrary to what we've seen before!**
- ▶ **This notion is used in several places:**
 - ▶ Early versions of macOS (until 10.6 Snow Leopard)
 - ▶ Windows
 - ▶ CPU cache
 - ▶ Memory (RAM)

Bash & Linux CLI

Preparing the Virtual Machine

Bash & Linux CLI

Filesystems?



Filesystems

- ▶ Each partition usually uses a filesystem to store the data
- ▶ **A filesystem is responsible for:**
 - ▶ Data organization: Storing files in folders and files
 - ▶ Space allocation: Managing free and used space, releasing space when files are deleted
 - ▶ Metadata management: Storing permissions, ownership, timestamps (file created at, last updated at,...)
- ▶ **Data integrity:**
 - ▶ Implementing mechanisms for error detection
 - ▶ Making sure the filesystem is always in a consistent state (preferable even after an unexpected power loss)
 - ▶ Journaling: Storing new changes in a temporary file, before committing them permanently

Most common filesystems (Linux)

- ▶ **Linux:**

- ▶ **ext3:** Third extended FS developed
 - ▶ Older but still common and in use
- ▶ **ext4:** Fourth version of the extended filesystem
 - ▶ Improved performance
 - ▶ Support for larger disks
 - ▶ Journaling improvements: Improved recovery after shutdown

- ▶ **xfs:**

- ▶ Especially proficient at managing large files and filesystems
- ▶ Optimized for parallel I/O
- ▶ Snapshot support, files can share the same data blocks

- ▶ **btrfs:**

- ▶ Support to easily create snapshots
- ▶ Enhanced RAID support

Most common filesystems (Windows / macOS)

▶ Windows:

- ▶ FAT32: Older file system, usually can only store files up to 4GB
- ▶ NTFS: Proprietary, sometimes only readable from Linux
- ▶ ReFS: Successor of NTFS
- ▶ exFAT: Proprietary until 2019, specification published now. Ideal for external drives

▶ macOS: APFS

- ▶ Filesystem for iOS and macOS devices
- ▶ Supports full disk encryption

In this course

- ▶ In this course we will focus on ext4
- ▶ ... but we will also have a look at xfs
- ▶ Those are the most common filesystems for Linux systems
- ▶ **Let's now create a partition with ext4!**

Bash & Linux CLI

What is a volume?

Bash & Linux CLI

Managing partitions on the CLI: parted

Managing partitions on the CLI: parted

- ▶ We can also manage partitions through the CLI
- ▶ We can do this with the command: `sudo parted`
- ▶ **Be careful:**
 - ▶ We can easily cause data loss
 - ▶ I personally have not used it that often:
 - ▶ Remote servers come fully partitioned already
 - ▶ And for external drives, I prefer gparted
- ▶ **Still, this lecture is important:**
 - ▶ When a server crashes, and you need to debug its filesystems
 - ▶ Then most likely you'll have to do this through a CLI
- ▶ **Let's still have a look at how we can use it!**

Creating the filesystem

- ▶ After we've created the partition...
- ▶ We still need to create the filesystem
- ▶ **We can do this with one of the following command:**
 - ▶ `mkfs.ext4 /dev/sdb1`

Bash & Linux CLI

How can we mount a volume?

What is a volume?

- ▶ **A partition:**

- ▶ A part of a physical drive
- ▶ Separated from other parts of the drive

- ▶ **A volume:**

- ▶ A logical storage unit on our computer
- ▶ It usually appears as an accessible drive or partition

- ▶ **It can be more than just a partition:**

- ▶ Multiple partitions can be combined into one logical volume (LVM)
- ▶ Or a volume can be stored on another computer and accessed through the network

What is a mount?

- ▶ **The idea:**

- ▶ We connect a filesystem to our directory tree
- ▶ This allows us to make it accessible to our programs

- ▶ **We can mount:**

- ▶ External, removable media (usually into a subfolder of /media)
- ▶ Internal, permanent volumes (usually into a subfolder of /mnt)
- ▶ External storage servers (for example: FTP)
- ▶ Folders into other folders (bind mount)
- ▶ After we mounted a drive, it will become part of our directory tree
- ▶ And we can just cd into that folder
- ▶ Let's now have a look at how Ubuntu Desktop handles mounting of external drives!

Bash & Linux CLI

How can we manually mount a drive

Get the name of a drive

- ▶ When we're running a server, drives won't be mounted automatically (usually)
- ▶ We need to manually mount the drives
- ▶ Before we can mount a drive, we need to get its name
- ▶ **We can do this with the following program:**
 - ▶ `lsblk -f`
 - ▶ The parameter `-f` tells the program that it should also show information about the filesystem
- ▶ **Let's have a look at this!**

How to manually mount a drive

- ▶ First, we need to create a folder to mount into:

- ▶ `mkdir /mnt/backups`

- ▶ Now we can mount the drive:

- ▶ `mount [device] [mount_point]`

- ▶ Example:

- ▶ `mount /dev/sdb2 /mnt/backups`

- ▶ We can also specify the filesystem manually: -t ext4:

- ▶ `mount -t ext4 /dev/sdb2 /mnt/backups`

- ▶ Or add additional options (mounts as read-only):

- ▶ `mount -o ro ext4 /dev/sdb2 /mnt/backups`

- ▶ We can now also verify the mount:

- ▶ `mount or df -h`

- ▶ We can unmount the drive with one of those commands:

- ▶ `umount /dev/sdb2`

- ▶ `umount /mnt/backups`

- ▶ Let's see this in action!

Bash & Linux CLI

Mount options?

Mount options

- ▶ Mount supports various mount options
- ▶ **Example options (ext4):**
 - ▶ `ro`: Read only
 - ▶ `rw`: (default): Read + write
 - ▶ `noexec`:
 - ▶ Disables execution of executable files on the mounted filesystem
 - ▶ `nosuid`:
 - ▶ Disables the set-user-identifier and the set-group-identifier on the mounted filesystem
 - ▶ Prevents a potential security risk!
 - ▶ `noatime`:
 - ▶ do not update access time when file is read
- ▶ **There're way more options that we could set:**
 - ▶ `man mount`

Bash & Linux CLI

Additional mount options

Mount options depend on filesystem!

- ▶ Depending on the filesystem, we can use different options
- ▶ exFAT for example doesn't support users and groups
- ▶ Thus, we can provide them as mount options
- ▶ **Example for exFAT:**
 - ▶ **gid=1001:**
 - ▶ All files should be owned by group with the id 1001
 - ▶ **uid=1001:**
 - ▶ All files should be owned by the user with the id 1001
 - ▶ **umask=0027:**
 - ▶ It starts with a zero to indicate that it's an octal number
 - ▶ This will set the permissions to 0777 - [our umask], meaning in this case to chmod 0750

Mount options depend on filesystem!

► Let's see this in action!

- ▶ For this, we will need an exFAT partition
- ▶ To be able to create and mount the exFAT partition, we will need the following packages:
 - ▶ **Ubuntu:**
 - ▶ `sudo apt install exfat-fuse exfatprogs`
 - ▶ **CentOS:**
 - ▶ `sudo dnf install exfatprogs`

Bash & Linux CLI

Mounting on boot: /etc/fstab

What is /etc/fstab

- ▶ **What is the /etc/fstab?**

- ▶ A system configuration file in Linux
- ▶ Defines how storage devices and partitions should be mounted
- ▶ It is being read during boot and allows us to automatically mount volumes

- ▶ **What is its format?**

- ▶ Each line represents a filesystem to be mounted
- ▶ Fields (columns) are separated by spaces or tabs
 - ▶ 1. Device identifier (UUID or device path)
 - ▶ 2. Mount point
 - ▶ 3. Filesystem type
 - ▶ 4. Mount options
 - ▶ 5. Dump option (dump is a backup utility, 0 = no backup)
 - ▶ 6. Filesystem check order
 - ▶ (fsck priority, 1 = root, 2 = non-root)

- ▶ **Let's have a look at this file!**

Mounting on boot: /etc/fstab

► Example entry of the /etc/fstab file:

► `UUID=aec067b7-c3cc-4b0d-97da-c4be187204f9 /mnt/backups ext4 defaults`

`0 2`

► Device identified by UUID: `aec067b7-c3cc-4b0d-97da-c4be187204f9`

► Mount point: `/mnt/backups`

► Filesystem type: `ext4`

► Mount options: defaults (rw, uid, dev, exec, auto, nouser, and async)

► auto: should be mounted automatically

► nouser: Can only be mounted with root privileges

► async: reads and writes should be async. Improves performance, but can be bad for data integrity on a power loss

► Dump option: 0 (disabled)

► Filesystem check order: 2 (non-root filesystem)

► To refresh our mounts (after editing):

► `sudo mount -a`

Bash & Linux CLI

Bonus: Mounting an FTP server

Only works on Ubuntu

- ▶ The next few lectures only work on Ubuntu
- ▶ You will also need FTP credentials to a remote server you want to access
- ▶ The next few chapters are mostly meant as an example to illustrate how FUSE allows access through additional filesystem types
- ▶ Thus, no need to follow-along!

What is FTP?

- ▶ **FTP: File Transfer Protocol**
- ▶ It allows us to transfer files from and to other computers
- ▶ **Important:**
 - ▶ FTP by default is unencrypted, the login data is transferred unencrypted
 - ▶ But there's also FTPS (FTP over SSL), which is encrypted
 - ▶ Or SFTP (but this is a completely different protocol)

Mounting a FTP server

- ▶ We can even mount FTP servers
- ▶ **First step:** We need to install all required software:
 - ▶ apt install fuse curlftpfs
- ▶ **We can then use it:**
 - ▶ sudo mkdir /mnt/ftp
 - ▶ sudo curlftpfs 'ftp://user:password@server/path-on-server/' /mnt/ftp
- ▶ **For unmounting:**
 - ▶ sudo fusermount -u /mnt/ftp

Bash & Linux CLI

Bonus: Mounting FTP, allow access

Allowing access

- ▶ **To allow access to all users:**

- ▶ `-o allow_other`:

- ▶ Allows everybody, to access this drive. Otherwise, it would only be accessible to the one who mounted it

Storing the credentials

- ▶ If we want to automatically mount this drive through /etc/fstab, we should avoid storing the credentials in that file
- ▶ **Luckily, we can create a `~/.netrc` file for the user root:**
 - ▶ `machine [server]`
 - `login [username]`
 - `password [password]`
- ▶ **We can then use curlftpfs without a username / password:**
 - ▶ `sudo curlftpfs 'ftp://server/path-on-server/' /mnt/ftp`

Bash & Linux CLI

Bonus: Mounting an FTP server through /etc/fstab

Automatically mount ftp drive

- ▶ **Can we automatically mount the drive through /etc/fstab?**
 - ▶ curlftpfs#username:password@server/ /mnt/ftp fuse noauto,allow_other,x-systemd.automount 0 0
- ▶ **Or, without username and password:**
 - ▶ curlftpfs#server/ /mnt/ftp fuse noauto,allow_other,x-systemd.automount 0 0

Bash & Linux CLI

Checking for errors: SMART

Checking for errors: SMART

- ▶ Physical drives deteriorate over time
- ▶ Most drives self-report their health through SMART
- ▶ **We can use SMART utility to check for errors:**
 - ▶ `smartctl --all /dev/sda`
 - ▶ **If we see something like this, everything is alright:**

```
==== START OF READ SMART DATA SECTION ====
SMART overall-health self-assessment test result: PASSED
```
- ▶ **Important:**
 - ▶ This only evaluates the physical drive on a self-reported hardware level
 - ▶ There might be problems with the partition table, filesystems,... - those will not be caught with SMART!
- ▶ **Btw, we might have to install smartctl:**
 - ▶ `apt install smartmontools`

Bash & Linux CLI

Checking for errors: fsck

Checking a filesystem for errors

- ▶ **Sometimes the filesystem might get corrupted:**
 - ▶ Bug in driver of filesystem
 - ▶ Unexpected power loss, system didn't shut down properly
 - ▶ Or the hard drive / SSD might be just old and loose data
(hopefully we can see it through SMART ahead of time)
- ▶ **Important:**
 - ▶ We need to check our drives regularly
 - ▶ This can prevent data loss

Checking a filesystem for errors

- ▶ **First:**
 - ▶ The drive needs to be unmounted and should not be encrypted
- ▶ **Then, we can trigger a check:**
 - ▶ `fsck /dev/sdb2`
 - ▶ **Note:** on some systems, we need to also specify the filesystem:
 - ▶ `fsck.ext4 /dev/sdb2`
- ▶ **So how do we check the / filesystem?**
 - ▶ We can set a kernel parameter through the GRUB menu:
`fsck.mode=force`
 - ▶ (if GRUB doesn't show up, we have seen in the chapter about the boot process how GRUB works and how we can enable it)
- ▶ **Otherwise, if we don't have access to GRUB:**
 - ▶ We can always boot a live Linux (from a USB drive / backend of our server)
 - ▶ And then check the filesystem from there

Checking for errors: fsck (automatically)

Automatically checking

- ▶ **By default:**

- ▶ Ubuntu does not automatically check the filesystem on boot

- ▶ **By default:**

- ▶ Ubuntu will only check the filesystem when the dirty bit has been set
 - ▶ This happens, when the volume has not been unmounted properly
 - ▶ We can prevent this through the last column in /etc/fstab (<pass>)

- ▶ **But we can also check if the drives should be automatically checked - even if always unmounted properly**

- ▶ **We can see if this is enabled:**

- ▶ Time based: `tune2fs -l /dev/sdb2 | grep -i -F 'check'`
 - ▶ Mount based: `tune2fs -l /dev/sdb2 | grep -i -F 'mount'`

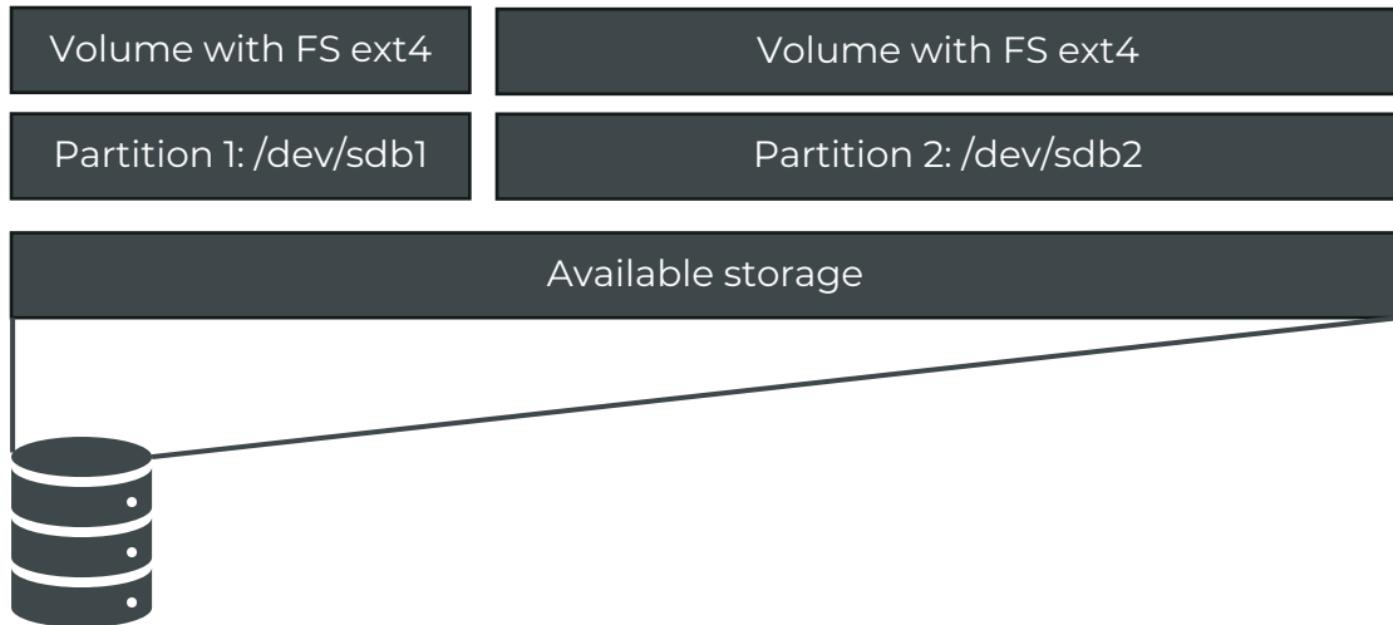
- ▶ **We can enable this:**

- ▶ This would check the volume every 30 mounts:
 - ▶ `tune2fs -c 30 /dev/sdb2`
 - ▶ Or every 6 months:
 - ▶ `tune2fs -i 6m /dev/sdb2`

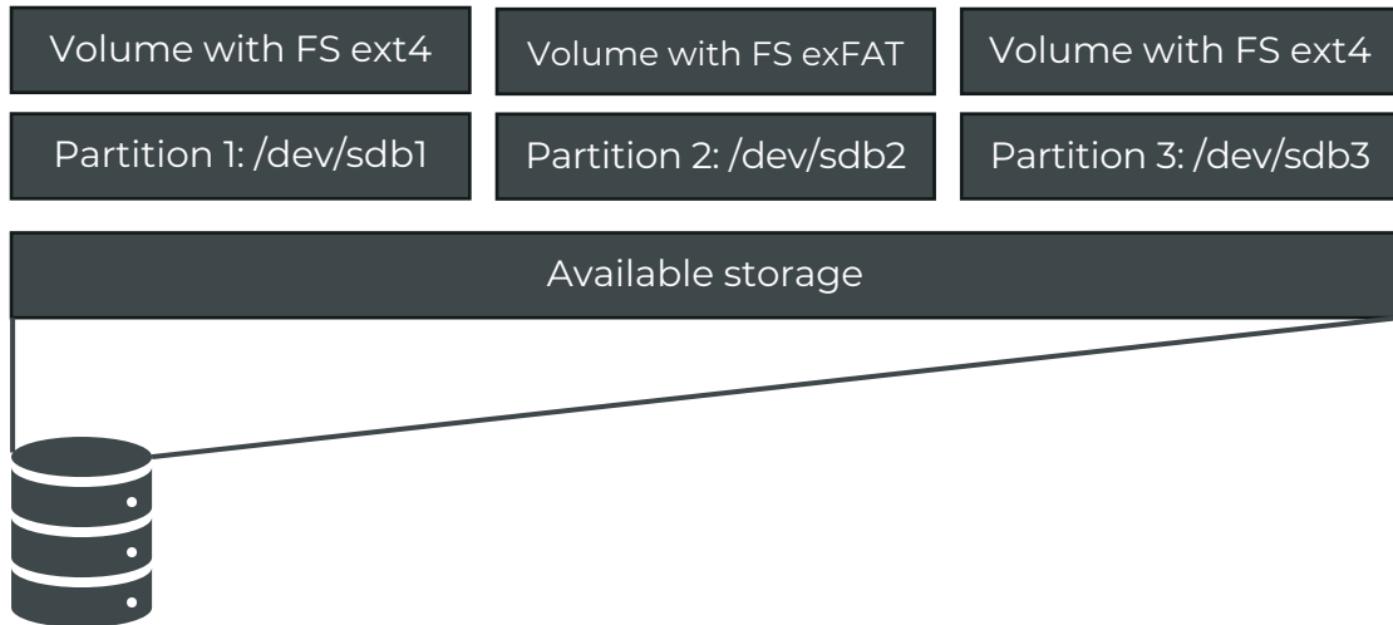
Bash & Linux CLI

Resizing a filesystem

Our partition(s) so far



Our new desired structure



How do we reduce the size of a partition?

- ▶ **How do we reduce the size of a partition?**

- ▶ First, we need to reduce the size of the filesystem:

- ▶ **Usually, the filesystem needs to be unmounted:**

- ▶ `umount /dev/sdb2`

- ▶ **Then (best practice), we run a filesystem check:**

- ▶ `fsck /dev/sdb2` or `fsck.ext4 /dev/sdb2`

- ▶ **Then, we can resize the ext4 filesystem**

- ▶ `resize2fs /dev/sdb2 10G`

- ▶ For other filesystems, this command might be different - or they may not support shrinking at all!

- ▶ **After this, we can resize the partition:**

- ▶ `parted /dev/sdb`

- ▶ **And in this program:** `resizepart`

Bash & Linux CLI

Resizing a filesystem (growing)

How do we increase the size of a partition?

- ▶ When we want to increase the size of a partition, the steps are similar, but in different order:
 - ▶ First, we need to resize the partition to the larger size
 - ▶ And then we can resize the filesystem:
 - ▶ Many filesystems offer online resizing => in that case, we don't have to umount the filesystem

Bash & Linux CLI

Bonus: Breaking the filesystem and fixing it

Let's break the volume and repair it

- ▶ **In this lecture:**

- ▶ We will first break the filesystem
- ▶ And then, we will try to repair it

- ▶ **Be extremely careful here:**

- ▶ A wrong command can cause permanent data loss

- ▶ **Even if you run this command in a virtual machine:**

- ▶ Maybe an external drive has been connected to the virtual machine
- ▶ And you might overwrite some data
- ▶ You might be accidentally in another shell of another computer and realize it too late
- ▶ I can't recommend following me along in this lecture
- ▶ **If you do, it's at your own risk!**

Bash & Linux CLI

Overview: Logical Volume Manager

Our setup so far



Overview: LVM

- ▶ How can we increase the size of the volume on top of /dev/sda2?
- ▶ It is limited by the size of our SSD / HDD
- ▶ **Why can't a volume span over multiple drives?**
 - ▶ Then we could just add an additional drive to our setup
 - ▶ And adjust the size of the filesystem
 - ▶ And we could access all that storage in one drive!

Outlook: LVM

- ▶ **At the end of this chapter:**

- ▶ Let's say you got 3 drives at 50GB each
- ▶ You will be able to create one partition with 150GB...
- ▶ Or 2 partitions with 75GB each on them!
- ▶ **Btw:** I'm aware drives are usually significantly larger. But the concepts work the same, no matter if a drive is 50TB or 50GB 😊



Way more features

- ▶ LVM would also support way more features:
 - ▶ Thin provisioning
 - ▶ Software RAID
- ▶ We will not have a look at those in this course

Bash & Linux CLI

Preparing the Virtual Machine

Preparing the VM

- ▶ **For this chapter, we need to add the following drives:**

- ▶ 3x 50GB
- ▶ 1x 150GB

- ▶ **After this, we might have to install LVM:**

- ▶ **Ubuntu:**

- ▶ `sudo apt install lvm2`

- ▶ **CentOS:**

- ▶ `sudo dnf install lvm2`

Bash & Linux CLI

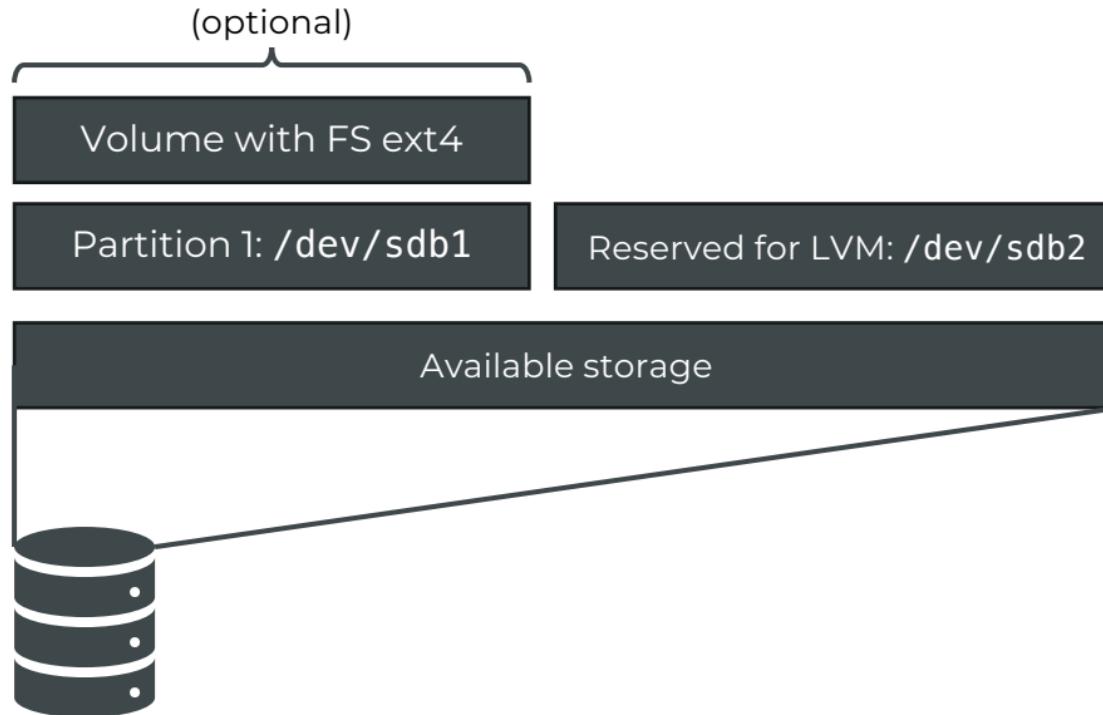
LVM: Logical Volume Manager

LVM: Logical Volume Manager

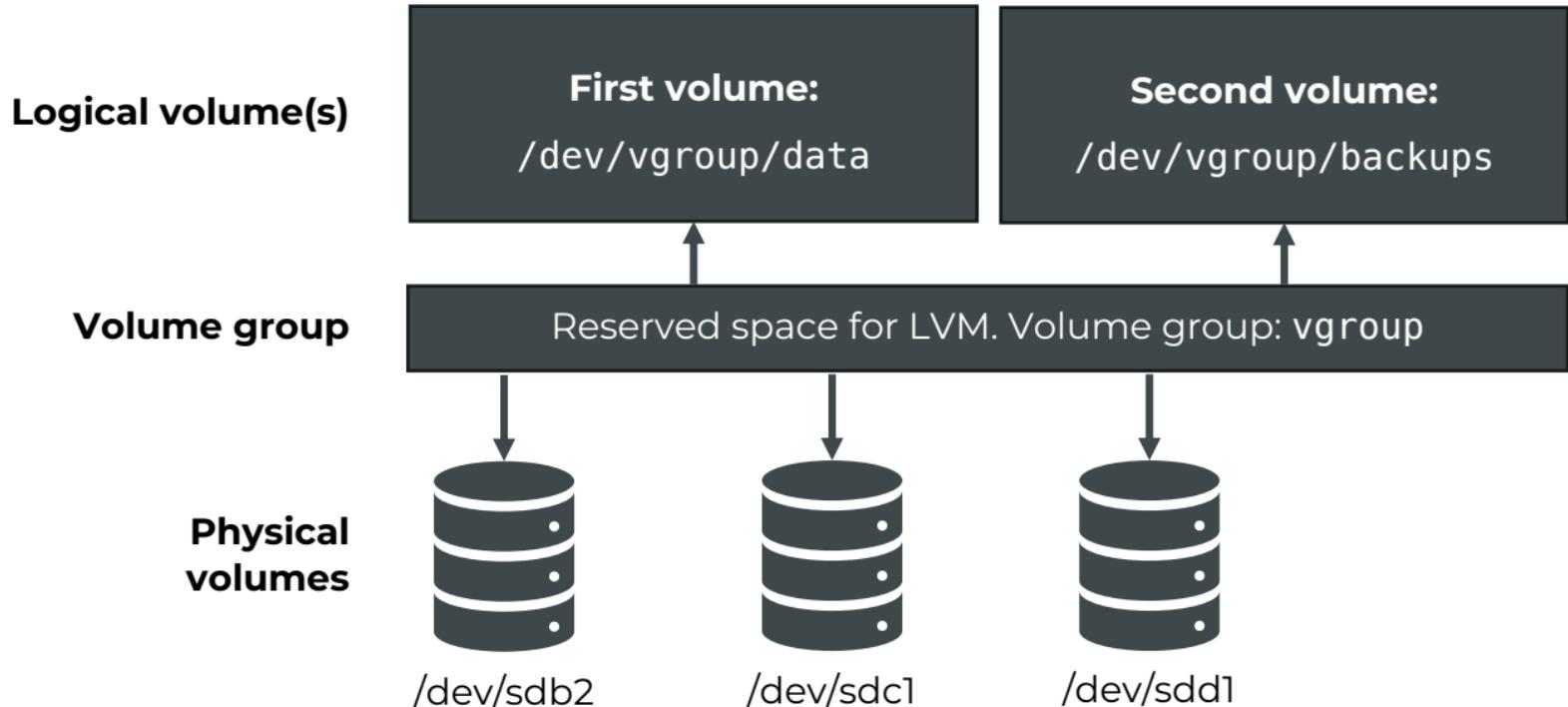
- ▶ **The idea:**

- ▶ Instead of placing our volume directly on disk...
- ▶ ... we place a device mapper in between
- ▶ This allows us to combine the space from multiple disks
- ▶ This space is then provided as a "volume group"
- ▶ On top of this, we can then create our partitions

The setup with LVM



The setup with LVM



Bash & Linux CLI

LVM: Initializing physical volumes

LVM: Let's initialize physical volumes

- ▶ We now want to recreate the setup from the lecture before
- ▶ The first step is that we initialize the physical volumes
- ▶ **For this, we have 2 options:**
 - ▶ **Initialize a part of a disk:**
 - ▶ First, we launch parted for our disk: `parted /dev/sdb`
 - ▶ Then, we create a new partition there: `mkpart primary 0 100%`
 - ▶ And then we set the type to LVM: `set [number] lvm on`
 - ▶ After this, we can exit parted and run `pvcreate` on this partition:
 - ▶ `pvcreate /dev/sdb1`
 - ▶ **Or we could initialize the full disk:** `pvcreate /dev/sdb`
 - ▶ It will refuse to overwrite an existing GUID partition table
 - ▶ We should avoid this - other programs might overwrite the beginning of the drive
 - ▶ **Once initialized, we can list the physical volumes:**
 - ▶ `pvs` / `pvdisplay`

Troubleshooting

- ▶ If the physical volumes don't show up:
 - ▶ `pvscan`

Bash & Linux CLI

LVM: Creating volume group

LVM: Creating volume group

- ▶ Now that we've initialized the physical volumes...
- ▶ ... we can create a volume group on top:
 - ▶ `vgcreate vgroup /dev/sdb1 /dev/sdc1 /dev/sdd1`
 - ▶ We can then list volume groups with the following commands:
 - ▶ `vgs`
 - ▶ `vgdisplay`
 - ▶ Sometimes, we might have to scan the drives for volume groups:
 - ▶ `vgscan`

Bash & Linux CLI

LVM: Creating logical volumes

LVM: Creating logical volumes

- ▶ Once we have a volume group, we can create logical volumes on top of it:
 - ▶ We can specify the size through -L:
 - ▶ `lvcreate -L 10G -n data vgroup`
 - ▶ Or we can create a logical volume with a percentage of the available free space of the volume group:
 - ▶ `lvcreate -l 100%FREE -n backups vgroup`
- ▶ If we want to display all available volumes:
 - ▶ `lvs / lvdisplay`
- ▶ If they cannot be found:
 - ▶ `lvscan`

LVM: Using logical volumes

- ▶ **Once we have a logical volume, we can use it just as a partition:**

- ▶ We can create a filesystem on top of it (`mkfs`)
- ▶ And then mount it
- ▶ Let's have a look at this!

Bash & Linux CLI

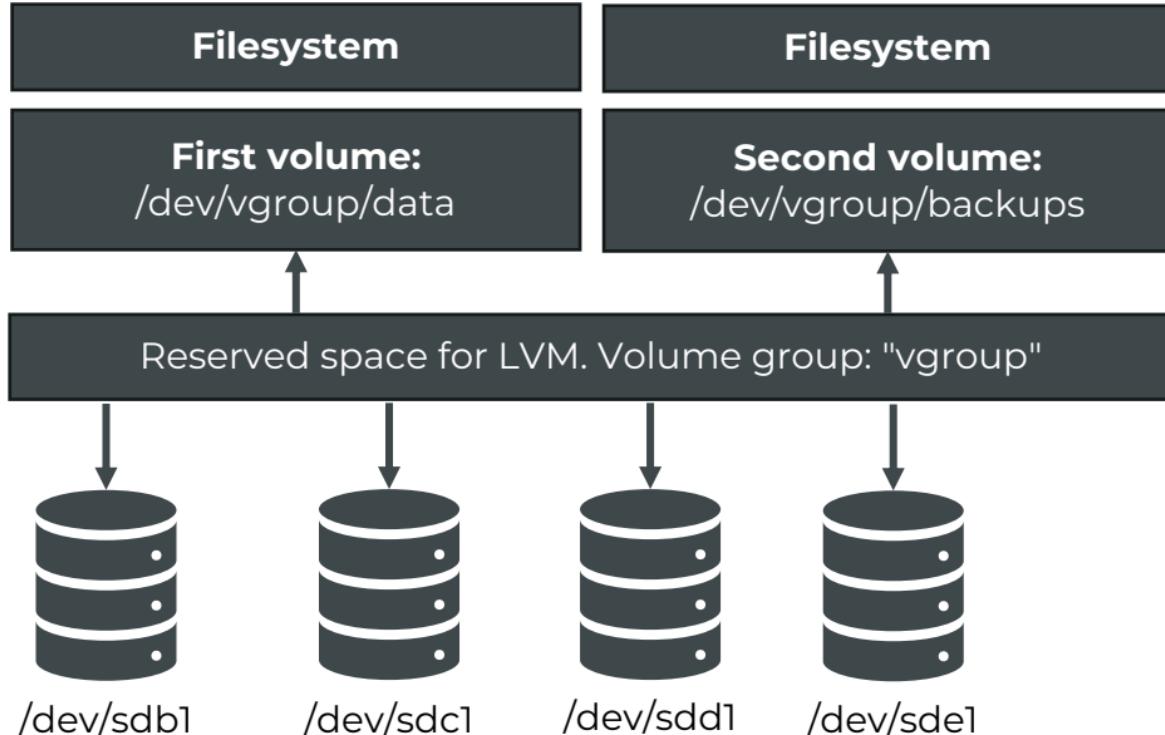
Modifying physical volumes

Filesystem on top of volume

Logical volume(s)

Volume group

Physical volumes



Adding physical volume to volume group

► How can we add an additional physical volume?

- ▶ First, we need to initialize a physical volume (just as before)
- ▶ We can then extend the volume group by this additional volume:
 - ▶ `vgextend vgroup /dev/sde1`

Removing a physical volume

- ▶ **How can we remove a physical volume?**
- ▶ **First, we need to move its data:**
 - ▶ `pvmove -v /dev/sdc1`
 - ▶ **-v:**
 - ▶ Verbose, more output will be printed to the terminal
- ▶ **After this, we need to remove this volume from the volume group:**
 - ▶ `vgreduce vgroup /dev/sdc1`
- ▶ **And after this, we can finally remove the physical volume:**
 - ▶ `pvremove /dev/sdc1`

Bash & Linux CLI

We need more space... how can we do this?

Filesystem on top of volume



Logical volume(s)

Volume group

Physical volumes

How can we increase the size of a logical volume

- ▶ **How do we increase the size of the volume?**

- ▶ Requirement: We need to have enough space available within our volume group
(unless we want to use "thin volumes" => not part of this course)

- ▶ **To extend the size of the "data" volume to 20GB:**

- ▶ `lvextend -L 20G --resizefs /dev/vgroup/data`

- ▶ **Or to increase it by 1GB:**

- ▶ `lvextend -L +1G --resizefs /dev/vgroup/data`

- ▶ For certain filesystems (such as ext4), `lvextend` can trigger the expansion of the filesystem for us: `--resizefs`

- ▶ **If this does not work:**

- ▶ We will manually have to increase the size of the filesystem
- ▶ Our filesystem must support this

How to decrease size of logical volume

- ▶ If we want to decrease the size of a logical volume:
 - ▶ First, we need to reduce the size of the filesystem:
 - ▶ In our case (ext4), we need to unmount the filesystem first - ext4 currently needs to be unmounted before the filesystem size can be reduced
 - ▶ `umount /dev/vgroup/data`
 - ▶ `resize2fs /dev/vgroup/data 15G`
 - ▶ After this, we can decrease the size of the logical volume to 15GB:
 - ▶ `lvreduce -L 15G /dev/vgroup/data`



Bash & Linux CLI

Deleting everything



LVM: Deleting everything

► How can we delete a logical volume?

- ▶ Be sure to umount it first and to remove entries in /etc/fstab
(if you have created them)!
- ▶ `lvremove /dev/vgroup/data`

► How can we delete a volume group?

- ▶ `vgremove vgroup`

► How can we delete physical volumes?

- ▶ `pvremove /dev/sdb1`

Bash & Linux CLI

CentOS - LVM for the main drive

LVM for the main drive

- ▶ **LVM for boot volume?**

- ▶ We can only use certain bootloaders (GRUB 2)
- ▶ Our kernel must support it
- ▶ Not all features are supported
 - ▶ (mostly affects advanced features of LVM, not part of this course)

- ▶ **Thus, currently, the best way is:**

- ▶ We can boot from a normal volume
 - ▶ Our kernel (with LVM support) is stored on this volume
 - ▶ And from there on, we can use LVM for the rest of our system
- ▶ **Let's see how CentOS is doing this right now!**

Bash & Linux CLI

LVM: Additional features

Important

- ▶ I want to give you an overview about LVM and how it works
- ▶ If you want to use it in production, there're way more features available
- ▶ Way too much to present it in this course
- ▶ Still, I would like to mention a few additional features, so you're aware of them

LVM: Additional features

- ▶ Also, LVM supports various other features:

- ▶ RAID (Redundant array of independent disks)

- ▶ LVM can be run on top of a software RAID
 - ▶ It supports RAID 0, 1, 4, 5, 6 & 10
 - ▶ Raid 0: Striping, we increase storage, but no redundancy
 - ▶ Raid 5: 1 drive can fail
 - ▶ Raid 6: 2 drives can fail

LVM: Additional features

► **Thin volumes:**

- We can assign 100GB to a volume, despite only having 50GB of storage
- If our volume needs more data, we can add additional drives later

► **Snapshot:**

- We can create a snapshot from our drives
- It only needs storage space, if data is changed after the snapshot
- Thus, we could create a snapshot, copy the snapshot as a backup, and remove the snapshot again (consistent backup solution)

Bash & Linux CLI

Bonus: Full software upgrade + Troubleshooting

Full software upgrade

- ▶ In this chapter, we will do a full software upgrade of our system
- ▶ During this process, the system broke, and could no longer boot
- ▶ **You will watch me fix the system, step by step:**
 - ▶ We will investigate the problem
 - ▶ And then fix it by booting a live system
- ▶ **This chapter is important:**
 - ▶ You will learn important knowledge that will give you hints about how you can deal with a system that doesn't boot anymore

Bash & Linux CLI

Bonus: Full software upgrade

Bonus: Full software upgrade



- ▶ **First, some remainders, we should have:**

- ▶ ... a complete backup
- ▶ ... enough disk space (at least several GB)
- ▶ ... enough time, also to fix anything in case something goes wrong
- ▶ ... at least waited a week or two after the release of a new version
- ▶ ... made sure all additional repositories support our new ubuntu version
- ▶ ... have a backup drive available to boot from in case anything goes wrong (for example: a bootable USB drive)
- ▶ ... and of course: evaluated if we want to upgrade!

Bonus: Full software upgrade



- ▶ **How do we update an ubuntu to a new software version?**
 - ▶ **First step: Install all normal updates:**
 - ▶ sudo apt update / sudo apt-get update
 - ▶ sudo apt full-upgrade / sudo apt-get dist-upgrade
 - ▶ **Reboot the system:**
 - ▶ sudo reboot
 - ▶ **Make sure the update manager is installed:**
 - ▶ sudo apt install update-manager-core
 - ▶ **And then we trigger the actual update:**
 - ▶ sudo do-release-upgrade
 - ▶ You can only upgrade one version at the time
 - ▶ (LTS => LTS)
 - ▶ (22.04 => 22.10 => 23.04)

Bash & Linux CLI

Troubleshooting (optional)

Troubleshooting

- ▶ Our system doesn't boot up
- ▶ What do we do?
- ▶ **First:**
 - ▶ At least for me: I need to stay calm, and just accept that the system will be down for a while. If I'm stressed, I can't think clearly.
 - ▶ Now we need to trouble shoot. What could the problem be?
- ▶ **The problem:** The system doesn't boot
 - ▶ Reasoning: Something during the boot process fails
 - ▶ This could be the bootloader or the kernel
 - ▶ Let's see if we can analyze this

Troubleshooting

- ▶ If I had not enabled the GRUB menu before
- ▶ How would I fix this computer?
- ▶ **The idea:**
 - ▶ We download a "live linux" system (usually a .iso image)
 - ▶ We burn this on a DVD, or copy it to USB drive with special programs (just copying the file is not enough)
 - ▶ We must make sure that the USB drive is bootable
 - ▶ All contents of the USB drive will be overwritten
 - ▶ We then boot from this external drive
 - ▶ From this Linux, we can then fix the Linux system on our hard drive
- ▶ **If you're a sysadmin of a server:**
 - ▶ In the backend, there should usually be an option to boot a live system to your existing server
 - ▶ It might support CLI only though

Bash & Linux CLI

Troubleshooting: Outro

Optional exercise

- ▶ Delete an important file that is needed during boot
- ▶ And try to restore it with a live DVD / recovery system

Troubleshooting

► If you're running an actual server:

- ▶ If possible, try to make a backup first (especially for virtual servers, this might be possible even if the system doesn't boot)
- ▶ The live system might be terminal based without a nice user interface
- ▶ You might not be able to see the bootloader, so debugging is a little bit more difficult
- ▶ Maybe the server provider can offer you some help

What are common problems

- ▶ **Problems during boot process:**

- ▶ Problems with kernel
- ▶ Problems with mounts
- ▶ Additional packages can cause problems
- ▶ Hardware problems

- ▶ **Problems while the system is running:**

- ▶ Services don't start
- ▶ Services might crash
- ▶ High CPU / memory load
- ▶ Wrongly configured software (for example: firewall)

Bash & Linux CLI

Bonus chapter: Cronjobs

What is a cronjob?

- ▶ Cronjobs allow us to repeatedly execute programs on our computer

- ▶ **For example:**

- ▶ We want to execute a backup script every day at 3am
 - ▶ We want to trigger an update script every minute

- ▶ **Can't we do this already?**

- ▶ **We can:** systemd, timer units with OnCalendar

- ▶ **Why do we need cronjobs then?**

Why this chapter

- ▶ Cronjobs are still important, software is being used on many systems
- ▶ They're more portable than systemd (run on Unix, not just on Linux)
- ▶ We can automatically send the output from a cronjob by email
- ▶ It's built-into many finished products, and won't be replaced any time soon
- ▶ **For example, in backends for website hosting... I had to write files like these quite often:**

```
▶ 00 */2 * * * '/usr/bin/flock' -n -E 0
  '/usr/home/codingij/.tmp/047ee190a657e4ae5d6d84057e95fa84.lck'
  /usr/bin/php80  /usr/www/users/codingij/my-project/artisan
  schedule:run

  15 */2 * * * '/usr/bin/flock' -n -E 0
  '/usr/home/codingij/.tmp/047ee190a657e4ae5d6d84057e95fa84.lck'
  /usr/bin/php80  /usr/www/users/codingij/my-project/artisan
  schedule:run
```

Bash & Linux CLI

Heads up: Different cron implementations

Different cron implementations

- ▶ Unfortunately, there're different cron implementations
- ▶ They all share the same configuration files
- ▶ But support different features
- ▶ **Some of the most popular implementations:**
 - ▶ **vixie-cron:**
 - ▶ Used in Ubuntu as the cron implementation (package: cron)
 - ▶ **anacron:**
 - ▶ Used in Ubuntu to support jobs that should run at regular intervals
 - ▶ If the system is shut down, they will be run during the next boot
 - ▶ Different configuration files than normal cron
 - ▶ **cronie:**
 - ▶ The cron implementation on CentOS
 - ▶ Fork of **vixie-cron**, but slightly different features
 - ▶ **anacron** is integrated into it

Bash & Linux CLI

Cronjobs: Overview

The cron daemon

► **What is the cron daemon (crond)?**

- Background process that manages scheduled tasks
- Executes commands at pre-specified intervals
- It will wake up every minute and check if something needs to be executed
- Named after Greek word "chronos" (time)

► **It reads in crontab files:**

- User-specific: /var/spool/cron/crontabs or /var/cron/tabs
- System-wide: /etc/crontab
 - The file must be owned by root, and must not be group or other-writable

► **On Debian / Ubuntu systems:**

- crond will also check the contents of /etc/cron.d
- But we should not use this folder: "*In general, the system administrator should not use /etc/cron.d/*"

(<https://manpages.debian.org/stretch/cron/cron.8>)

How do we create a cronjob (user-specific)?

- ▶ **We can edit the crontab file with the following command:**
 - ▶ `crontab -e`
 - ▶ During first launch, we might be asked which editor we'd like to use, otherwise we might also set the `EDITOR` variable
 - ▶ This will open the crontab file for the currently logged in user
 - ▶ We can now define user specific cronjobs
- ▶ **Important:**
 - ▶ We should only use this command to change user specific cronjobs
- ▶ **We can also just list the cronjobs:**
 - ▶ `crontab -l`
- ▶ **Let's have a look at this!**

Bash & Linux CLI

The format of crontab

What is the format?

- ▶ **We can define the entries as follows:**

- ▶ First, we can define some environment variables
(only possible in most implementations of cron):
- ▶ We might want to set the **SHELL** variable to bash (commands will otherwise be executed by **/bin/sh**)
- ▶ And we should consider specifying a PATH, otherwise **/usr/bin:/bin** will be used

- ▶ **And then the actual cronjobs:**

- ▶ [Minute] [Hour] [Day] [Month] [Day-of-Week] [Command]

- ▶ **Example:**

- ▶

```
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/local/sbin:/sbin:/usr/sbin:/bin:/usr/bin
5 3 * * * ping -c 10 google.com
```

- ▶ This will execute the ping command at 3:05am every day

Format of the time

- ▶ **How exactly does the format work?**
 - ▶ [Minute] [Hour] [Day] [Month] [Day-of-Week] [Command]
- ▶ **We can:**
 - ▶ **Just specify the value directly:**
 - ▶ 5 3 * * * command
 - ▶ **Specify multiple values:**
 - ▶ 0,15,30,45 * * * * command
 - ▶ **Specify a range (every full hour, between 8 and 20):**
 - ▶ 0 8-20 * * * command
 - ▶ **Every 5th minute:**
 - ▶ */5 * * * * command
 - ▶ **Every 2nd hour:**
 - ▶ 0 */2 * * * command
 - ▶ **Every 2nd hour, starting with 1am:**
 - ▶ 0 1-23/2 * * * command
 - ▶ **Use weekdays (here: every Monday) :**
 - ▶ 0 0 * * 1 command

Bash & Linux CLI

Ubuntu: What happens if we don't redirect the output?

What happens if a cronjob generates output?

- ▶ **If a cron generates output:**

- ▶ Cron will try to send this output by mail to the user it belongs to
- ▶ If we want the output to be sent to any other address, we can specify another variable in the crontab file:
 - ▶ MAILTO="adress@domain.com"

Bash & Linux CLI

CentOS: What happens if we don't redirect the output?

What happens if a cronjob generates output?

- ▶ **If a cron generates output:**

- ▶ Cron will try to send this output by mail to the user it belongs to
- ▶ If we want the output to be sent to any other address, we can specify another variable in the crontab file:
 - ▶ MAILTO="adress@domain.com"

Bash & Linux CLI

Bonus tip: flock

The complicated commands from the beginning

- ▶ Maybe you remember those complicated commands from the beginning:
- ▶

```
00 */2 * * * '/usr/bin/flock' -n -E 0
'/usr/home/codingij/.tmp/047ee190a657e4ae5d6d84057e95
fa84.lck' /usr/bin/php80 /usr/www/users/codingij/my-
project/artisan schedule:run
```
- ▶ **Why did I needed them to be this complicated?**

Bonus tip: flock

- ▶ We can use the flock command to lock a file
- ▶ `flock file.txt ping google.com`
 - ▶ As long as the program ping is running, file.txt is locked
 - ▶ At the same time, a file can only be locked by one program
 - ▶ => The next flock command will wait
- ▶ `flock -n -E 0 file.txt ping google.com`
 - ▶ => If the file.txt is locked already, the next flock command will just exit (-n) with error code 0 (=everything okay)
- ▶ **Let's have a look at this!**

Bash & Linux CLI

System-wide cronjobs

System-wide cronjobs

- ▶ **We can also specify system-wide cronjobs:**

- ▶ We need to specify those in the /etc/crontab file
- ▶ We are allowed to edit this file directly
- ▶ We can run cronjobs as any user!

- ▶ **The format there is thus slightly different:**

- ▶ [Minute] [Hour] [Day] [Month] [Day-of-Week] [User] [Command]
- ▶ Let's have a look at how those work ☺



Ubuntu

Bash & Linux CLI

Bonus: anacron (Ubuntu)

Bonus: anacron

- ▶ **Cronjobs had a few limitations:**

- ▶ They're only being executed, when the system is running
- ▶ They're being executed, no matter if a laptop is plugged in or not

- ▶ **anacron solves this:**

- ▶ By default, anacron jobs are only run when the system is plugged in
(due do its default systemd configuration)
- ▶ If the system is shut down, jobs that should've been executed during
the last days, will be executed
- ▶ How does anacron work?

How does anacron work?

- ▶ **The easiest way:**

- ▶ We can just add an executable file in the following directories:
 - ▶ /etc/cron.daily
 - ▶ /etc/cron.weekly
 - ▶ /etc/cron.monthly

- ▶ **Important:**

- ▶ The filename can only contain a-z, A-Z, 0-9, _ and - symbols:
 - ▶ Regex would be: ^[a-zA-Z0-9_-]+\$

- ▶ **We can also place jobs into the following file:**

- ▶ /etc/anacrontab

- ▶ **The format is as follows:**

- ▶ [period in days] [minutes after system boot]
[identifier for the last execution time] [command]

Bash & Linux CLI

Bonus: anacron (CentOS)

Bonus: anacron

- ▶ **Cronjobs had a few limitations:**

- ▶ They're only being executed, when the system is running
- ▶ They're being executed, no matter if a laptop is plugged in or not

- ▶ **anacron solves this:**

- ▶ By default, anacron jobs are only run when the system is plugged in (due to its default configuration)
- ▶ If the system is shut down, jobs that should've been executed during the last days, will be executed

Bonus: anacron on CentOS

- ▶ On CentOS, anacron is part of cronie
- ▶ **But we need to have installed the correct version of cronie:**
 - ▶ **The following package must be installed:**
 - ▶ cronie-anacron
 - ▶ **We can ensure this:**
 - ▶ `dnf install cronie-anacron`

How does anacron work?

- ▶ **The easiest way:**

- ▶ We can just add an executable file in the following directories:
 - ▶ /etc/cron.daily
 - ▶ /etc/cron.weekly
 - ▶ /etc/cron.monthly

- ▶ **Important:**

- ▶ The filename can only contain a-z, A-Z, 0-9, _ and - symbols:
 - ▶ Regex would be: ^[a-zA-Z0-9_-]+\$

- ▶ **We can also place jobs into the following file:**

- ▶ /etc/anacrontab

- ▶ **The format is as follows:**

- ▶ [period in days] [minutes after system boot]
[identifier for the last execution time] [command]

Bash & Linux CLI

Cronjobs: Best practices

Best practices with cronjobs

► **Planning and scheduling:**

- ▶ Distribute tasks evenly, avoid peak hours
- ▶ Avoid executing the same task multiple times (flock)

► **Logging and Error handling:**

- ▶ Log errors and analyze them regularly
- ▶ Implement error checks within your scripts

► **Security:**

- ▶ Run tasks with the least privileges possible
- ▶ Keep scripts and commands secure with proper permissions
- ▶ Avoid storing sensitive information in crontab files

► **Testing:**

- ▶ Test scripts thoroughly before adding to crontab
- ▶ Make sure they work with the PATH variable that cron provides
(or set your own)
- ▶ Monitor initial runs to ensure proper execution

► **And in general:** Be aware of different cron implementations

Bash & Linux CLI

Networking: Overview

Overview

- ▶ **In this chapter:**

- ▶ We will look at networking on Linux (client-side)

- ▶ **After this chapter, you will:**

- ▶ Have a deeper understanding of networking
 - ▶ Be able to troubleshoot network problems
 - ▶ Debug and identify slow connections
 - ▶ Understand how IP addresses are assigned in a network through DHCP
 - ▶ Understand what DNS is and troubleshoot problems
 - ▶ This will help you in your job, but also for configuring your home network

Bash & Linux CLI

What is the internet?

What is the internet?

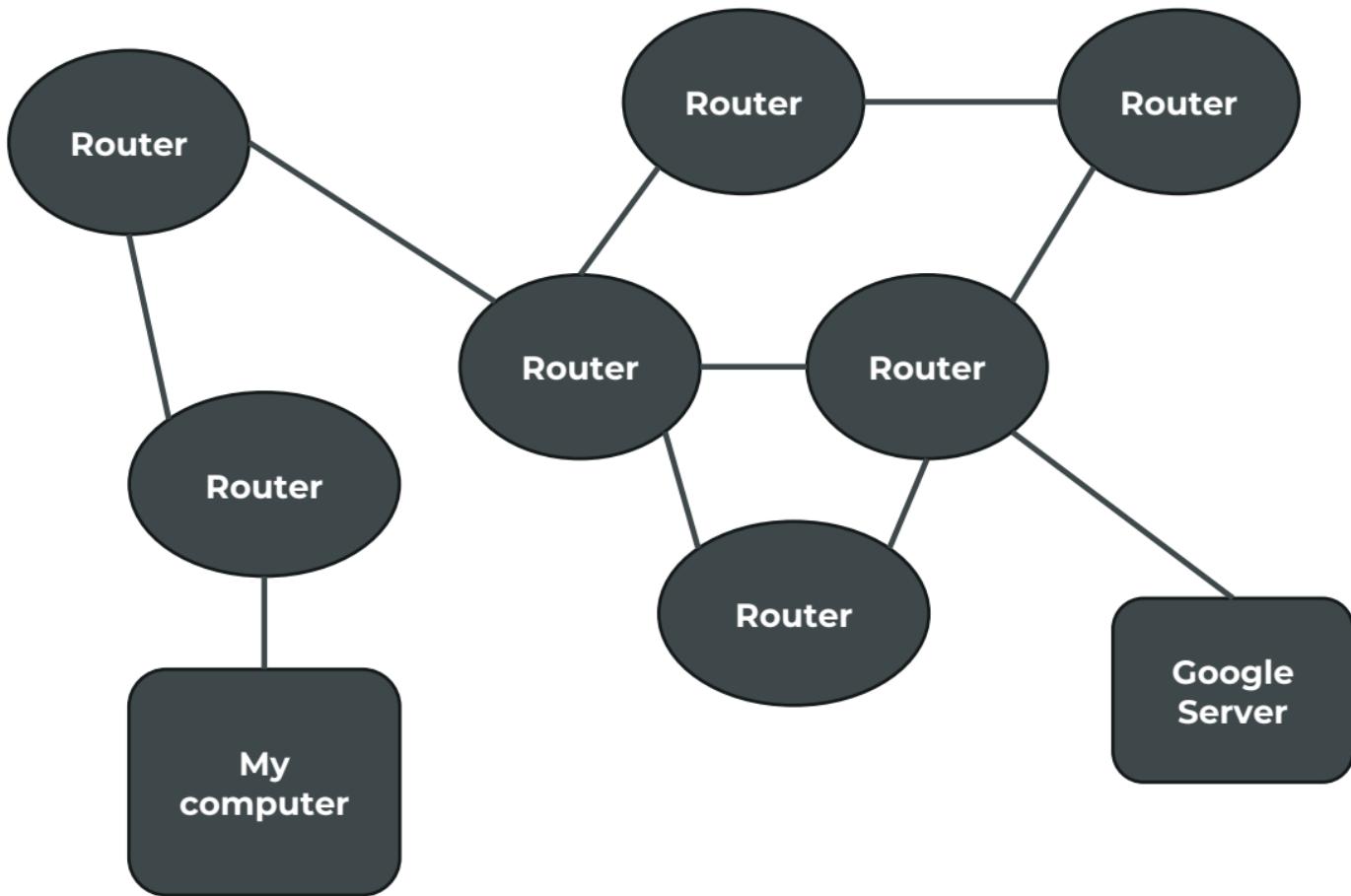
- ▶ **The internet:**

- ▶ Interconnected nodes, that form a mesh
- ▶ The nodes can communicate to each other

- ▶ **The goal:**

- ▶ We can build a connection with pretty much any other computer in the network... without having a dedicated connection to it!
- ▶ If one connection gets cut, we can seamlessly find a way around

- ▶ **Let's have a look at a visualization!**



Bash & Linux CLI

The program: ip

Introduction: The ip command

- ▶ **The ip command:**

- ▶ A versatile and powerful command-line tool for managing and diagnosing network configurations on Linux systems
- ▶ Replaces the older `ifconfig`, `route`, and `netstat` commands
- ▶ Commonly used to inspect and modify IP addresses, routes, and network interfaces

- ▶ **We can use it to show our network configuration:**

- ▶ `ip addr show`

Introduction: The ip command

► For macOS users:

- ▶ If we're on a mac, we can't use the `ip` program (at least as of now)
- ▶ But we can install `iproute2mac`
- ▶ This is a wrapper script, that will call the available tools from macOS
- ▶ And it will provide an emulated `ip` tool to us - though not all possibilities are supported, and the output might be slightly different than on Linux
- ▶ We can install it through HomeBrew:
 - ▶ `brew install iproute2mac`



Bash & Linux CLI

The program: Wireshark

Legal disclaimer

► **Important: This is not legal advice!**

- Wireshark is a powerful network tool
- Wireshark can intercept, monitor and analyze network traffic
 - This might be illegal if you do this to network traffic that is not yours
- Wireshark might capture sensitive information that is transmitted through the network
 - If this is data from others, it might be illegal
- Laws might be different depending on your jurisdiction

► **In my case:**

- Using Wireshark for ethical use cases is legal (such as learning, teaching, ethical hacking) - as long as I'm not logging private information of others
- I'm living alone, so all the internet devices in my network are mine

► **In your case:**

- You might need to ask your network administrator / other network users if it's okay to use Wireshark
- And should consider checking the laws of your country or state

The software: Wireshark

- ▶ In this chapter, we will use the software Wireshark to inspect network traffic
- ▶ This is a GUI application, but it really allows us to visualize what's happening on our network
- ▶ **Before we continue, let's install this application on our system:**
 - ▶ `apt install wireshark`
- ▶ **Let's have a look at Wireshark now!**

Bash & Linux CLI

Networking: OSI model

The OSI model

- ▶ **Definition:**

- ▶ Open Systems Interconnection (OSI) model
- ▶ A conceptual framework for understanding and designing computer networks

- ▶ **Purpose:**

- ▶ Standardizes the functions and protocols used in network communication
- ▶ Divides network communication into seven distinct layers, each with specific responsibilities

- ▶ **The idea:**

- ▶ We want competition, but they should be interoperable
- ▶ By learning the OSI model, we will be able to communicate more clearly about potential networking problems
- ▶ We could say: "*We got a problem on the data link layer*", and the problem is more clearly defined

The OSI model

► Application Layer (Layer 7)

- ▶ Provides the interface for applications to communicate over the network (e.g., HTTP, FTP, SMTP)

► Presentation Layer (Layer 6)

- ▶ Translates, encrypts, and compresses data for transmission between applications and the network (e.g., TLS/SSL)

► Session Layer (Layer 5)

- ▶ Manages and controls the establishment, maintenance, and termination of sessions between applications

► Transport Layer (Layer 4)

- ▶ Ensures reliable data transfer between hosts (e.g., TCP, UDP)

► Network Layer (Layer 3)

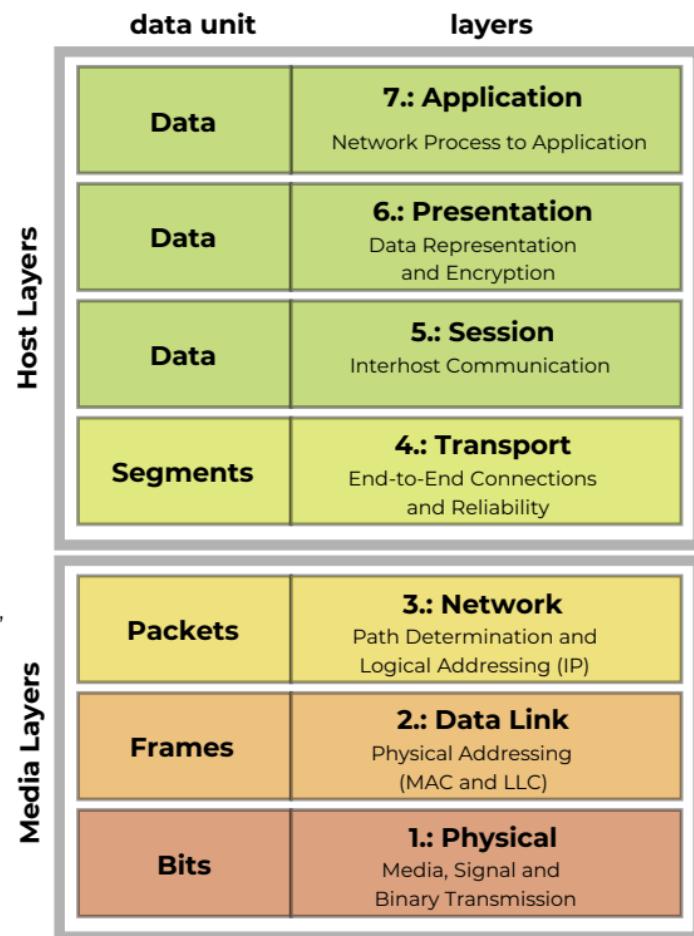
- ▶ Manages data routing and forwarding between networks (e.g., IP addresses, routers)

► Data Link Layer (Layer 2)

- ▶ Provides error-free data transfer between adjacent network nodes (e.g., Ethernet, MAC addresses)

► Physical Layer (Layer 1)

- ▶ Handles the transmission of raw data bits over a physical medium (e.g., cables, switches)



The OSI model

► **Modularity:**

- ▶ Allows changes and improvements in one layer without affecting others

► **Interoperability:**

- ▶ Facilitates communication between devices and systems from different vendors

► **Troubleshooting:**

- ▶ Simplifies the process of diagnosing and resolving network issues by isolating problems within specific layers

Bash & Linux CLI

Layer 1: Physical layer



Physical layer of the OSI model

- ▶ **Physical Layer (Layer 1) of OSI model**

- ▶ **Physical medium:**

- ▶ Copper cables, fiber-optic cables, wireless

- ▶ **Data transmission:**

- ▶ Convert digital data to signals

- ▶ **Signaling:**

- ▶ Voltage levels, modulation, synchronization

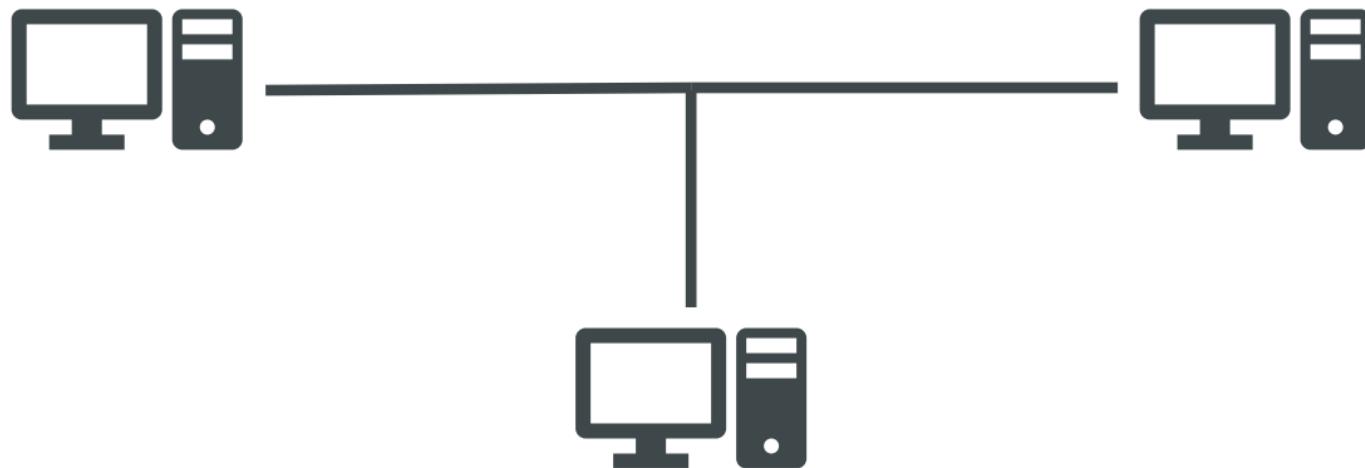
- ▶ **Error detection:**

- ▶ Basic methods, e.g., parity bits

- ▶ **TLDR:**

- ▶ We now have a physical connection, through which we can send bits

Ethernet splitter (Layer 1)



Bash & Linux CLI

Layer 1: Enable / Disable device

Layer one: Disable device

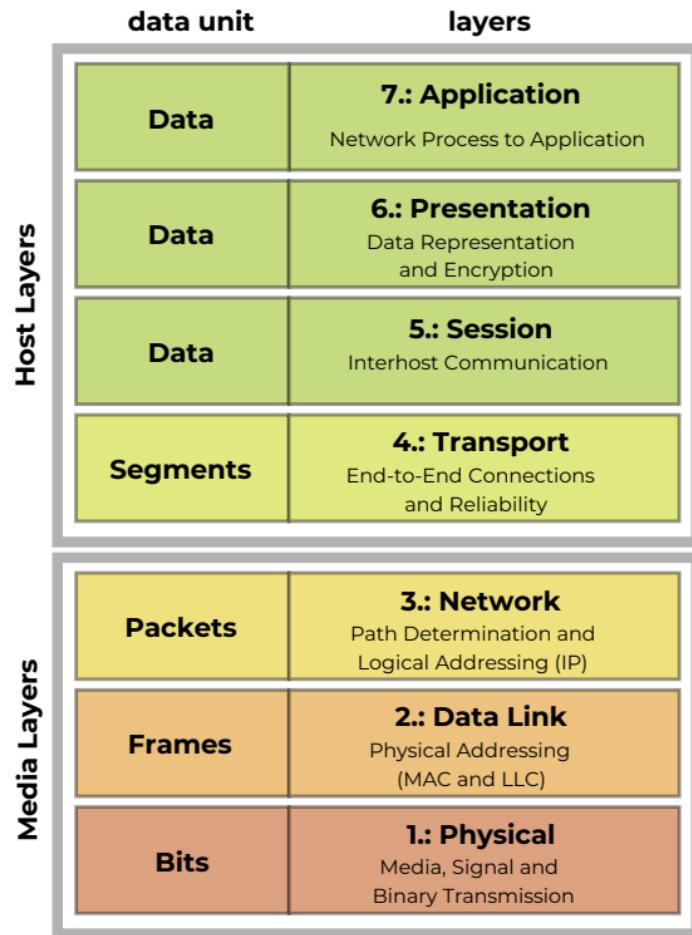
- ▶ We can also disable a device through software
- ▶ We first need to get the name of our device:
 - ▶ `ip addr show`
- ▶ **After this, we can enable / disable the device:**
 - ▶ `ip link set dev <interface> up`
 - ▶ `ip link set dev <interface> down`
- ▶ **Example:**
 - ▶ `ip link set dev enp0s5 down`
- ▶ **Be careful:**
 - ▶ This will disable this network device
 - ▶ If you're using it to connect to the remote machine... it will drop our connection!

Bash & Linux CLI

Layer 2: Data Link Layer

Data Link Layer

- ▶ Now, we have established a physical connection
- ▶ We now need a way to define who should receive the data
- ▶ On the data link layer, the units we send through it, are called *frames*
- ▶ **Example:**
 - ▶ How do we detect errors?
 - ▶ How do we make sure the receiver knows a frame is for him?
 - ▶ How do we make sure they don't send too much data that the receiver can't handle?
 - ▶ This is handled on the data link layer
 - ▶ But how exactly does it work?



Layer 2: Data Link layer

- ▶ **The goal of layer 2:**

- ▶ Reliable communication between devices on the same network

- ▶ **Divided into two sublayers:**

- ▶ **Logical Link Control (LLC):**

- ▶ Flow control and error detection
 - ▶ This is the interface between Data Link and Network Layers

- ▶ **Media Access Control (MAC):**

- ▶ Unique hardware addresses (MAC addresses)
 - ▶ Gives us a way to identify the sender and receiver of a frame

- ▶ **Frame encapsulation:**

- ▶ Organize data into frames for transmission
 - ▶ Add source and destination MAC addresses

- ▶ **Media access protocols:**

- ▶ Ethernet (IEEE 802.3)
 - ▶ Wi-Fi (IEEE 802.11)

Layer 2: What is ethernet?

- ▶ **What is Ethernet (IEEE 802.3):**

- ▶ A family of network technologies
- ▶ Mostly used in LAN (local area networks)
- ▶ Originally developed in the 1970s
- ▶ Ethernet splits the data into "frames"

- ▶ **An ethernet frame:**

- ▶ Is usually max 1.5kb in size
- ▶ Contains the actual data that we want to transfer (more on that later)
- ▶ Contains a checksum so the destination can verify if it has received the data correctly

- ▶ **Also contains addressing information:**

- ▶ MAC of the source ethernet interface
- ▶ MAC of the destination ethernet interface

Layer 2: What is WiFi?

- ▶ **What is WiFi (IEEE 802.11)?**

- ▶ A wireless networking protocol
- ▶ The frames that are being sent are slightly different than the ones from Ethernet - they need to contain additional data responsible for wireless connections

- ▶ **Still, they're highly compatible with each other:**

- ▶ Both protocols use MAC-addresses
- ▶ A wireless access point can convert Ethernet frames into WiFi frames

What is a MAC address?

- ▶ **What is a MAC address?**
 - ▶ **Media Access Control address**
 - ▶ Unique identifier for network interfaces
 - ▶ 48-bit (6-byte) address
 - ▶ **Address format:**
 - ▶ 6 groups of 2 hexadecimal digits, separated by colons
(e.g., 01:23:45:67:89:AB)
 - ▶ Assigned by manufacturers:
 - ▶ First 3 bytes: Organizationally Unique Identifier (OUI)
 - ▶ Last 3 bytes: Device-specific identifier
 - ▶ Static by default but can be changed (spoofed) in software
- ▶ **We can show information about our network devices with the following command:**
 - ▶ `ip addr show`

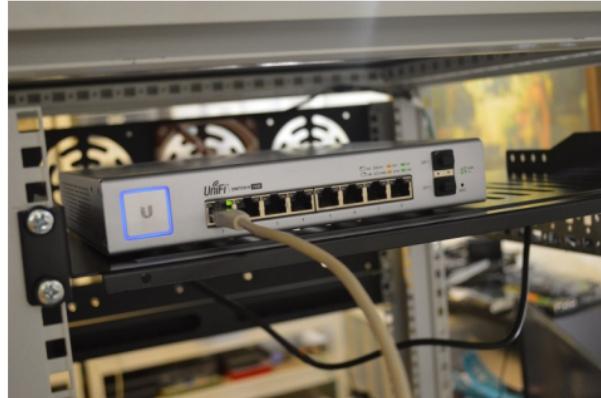
Bash & Linux CLI

Layer 2: Hardware for layer 2

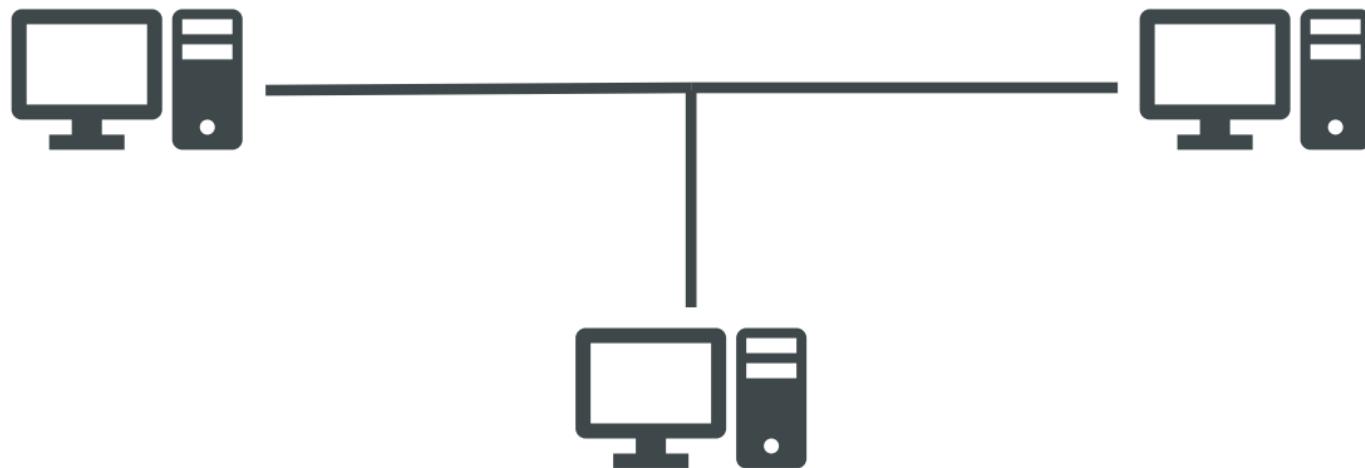
Data Link Layer

- ▶ **Typical hardware for this layer:**

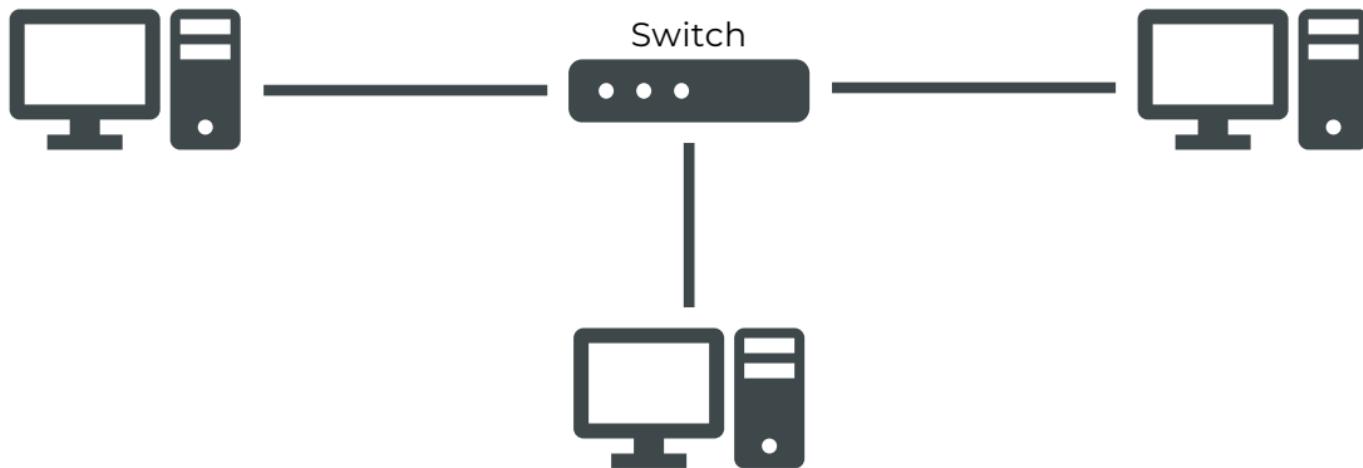
- ▶ Bridge
- ▶ Switch* (Ethernet)
- ▶ * through some switches might also operate on level 3
- ▶ Wireless Access Point



Ethernet splitter



What does a switch do?

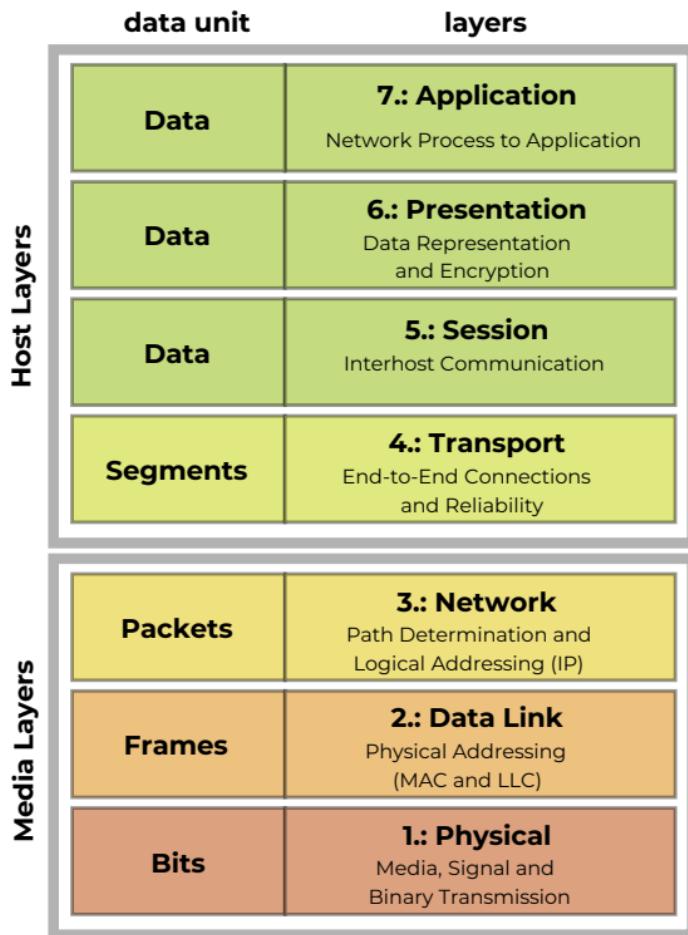


Bash & Linux CLI

Layer 3: Network layer

Layer 3: Network layer

- ▶ On layer 2, we could only send frames from one computer directly to another one
- ▶ If computers are not connected directly, we can't send messages through layer 2
- ▶ **Thus, we need the network layer:**
 - ▶ Here, we send "packets", that can be routed
 - ▶ Routing means, that a packet is forwarded
- ▶ **Idea here:**
 - ▶ We no longer work with MAC addresses, but with IP addresses
 - ▶ We send "packets", which are wrapped into frames at each step
 - ▶ Those packets can be forwarded
 - ▶ Thus, our router could forward a packet that is meant to be sent to the internet
- ▶ **How does a network work?**



First: What is a network?

- ▶ **Definition:**

- ▶ A network is a group of interconnected devices that can communicate and share resources.
- ▶ Devices can include computers, servers, printers, routers, switches, and more.

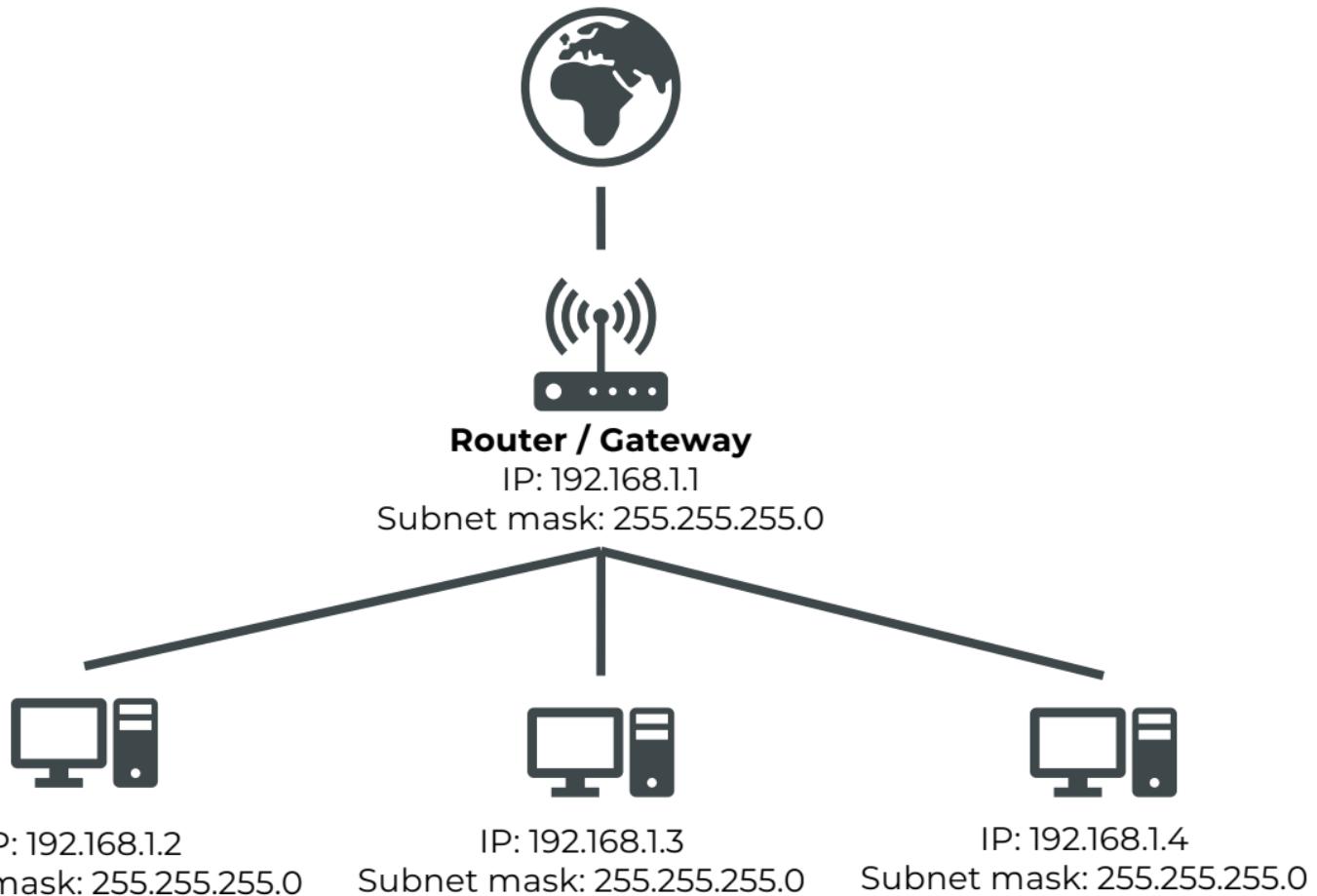
- ▶ **Types of networks (that are important for us now):**

- ▶ **Local Area Network (LAN):**

- ▶ Covers a small geographic area, such as a home or office

- ▶ **Wide Area Network (WAN):**

- ▶ Spans larger distances (country, continent, global), often connecting multiple LANs



How can we inspect this information?

- ▶ We can use the ip program again!
 - ▶ `ip addr show`
- ▶ This will show us our IP address in our LAN
- ▶ **But we can also use the program to show routes:**
 - ▶ `ip route show`

Bash & Linux CLI

Layer 3 (Networks layer): Subnets

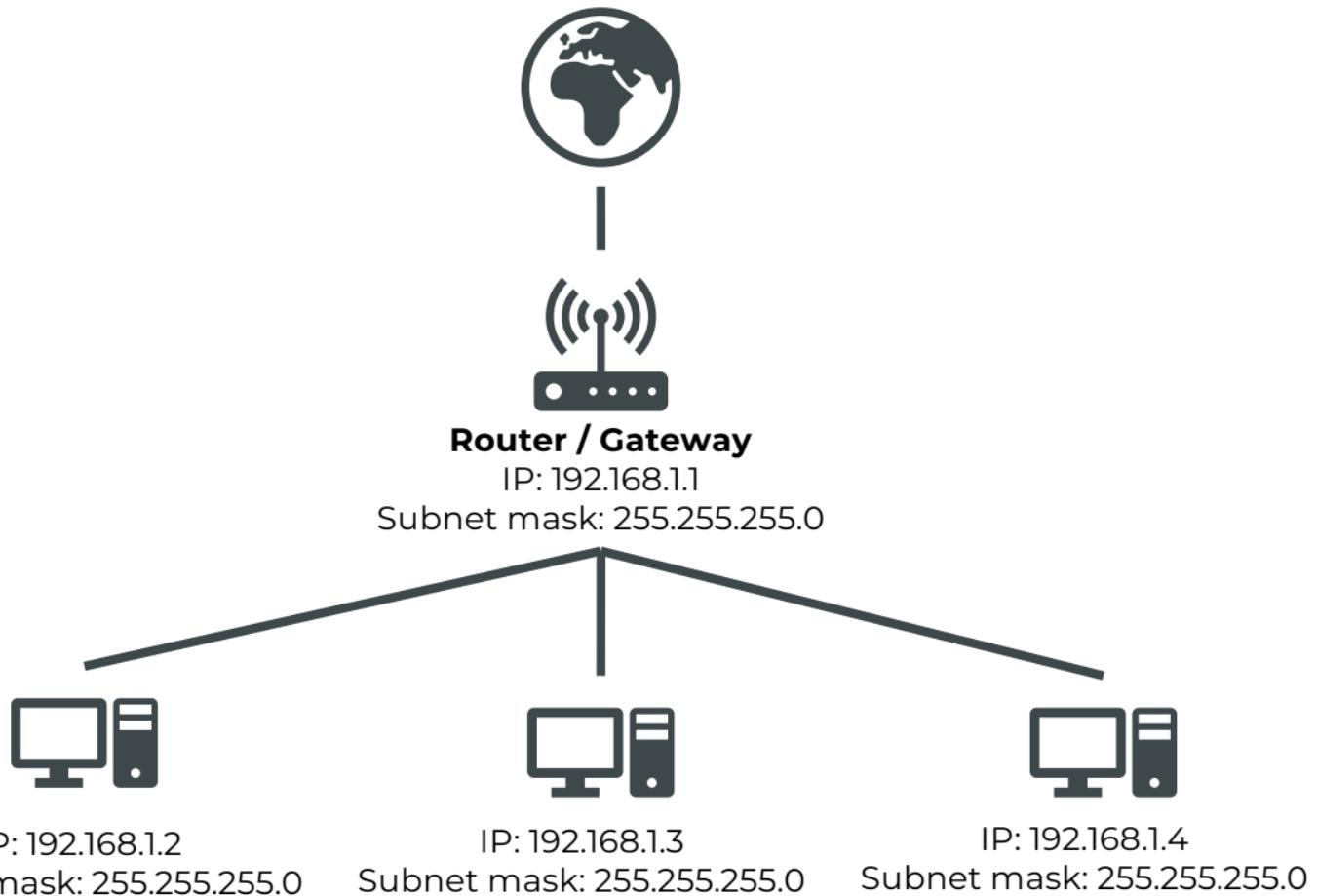
What is a subnet?

- ▶ A subnet is a network within a network
- ▶ This allows us to manage more computers
- ▶ And they make (large) networks more efficient
- ▶ **At home:**
 - ▶ Technically speaking, our home network is a subnet of the internet
- ▶ **In a corporate setting:**
 - ▶ We can split our corporate network into multiple subnets, and increase the efficiency

The problem right now

► **The problem:**

- ▶ How do we know where to send a frame to?
- ▶ If we want to send a packet to the internet... we need to send the frame to the router
- ▶ If we want to send a packet to another computer in our network... we can directly send the frame to it

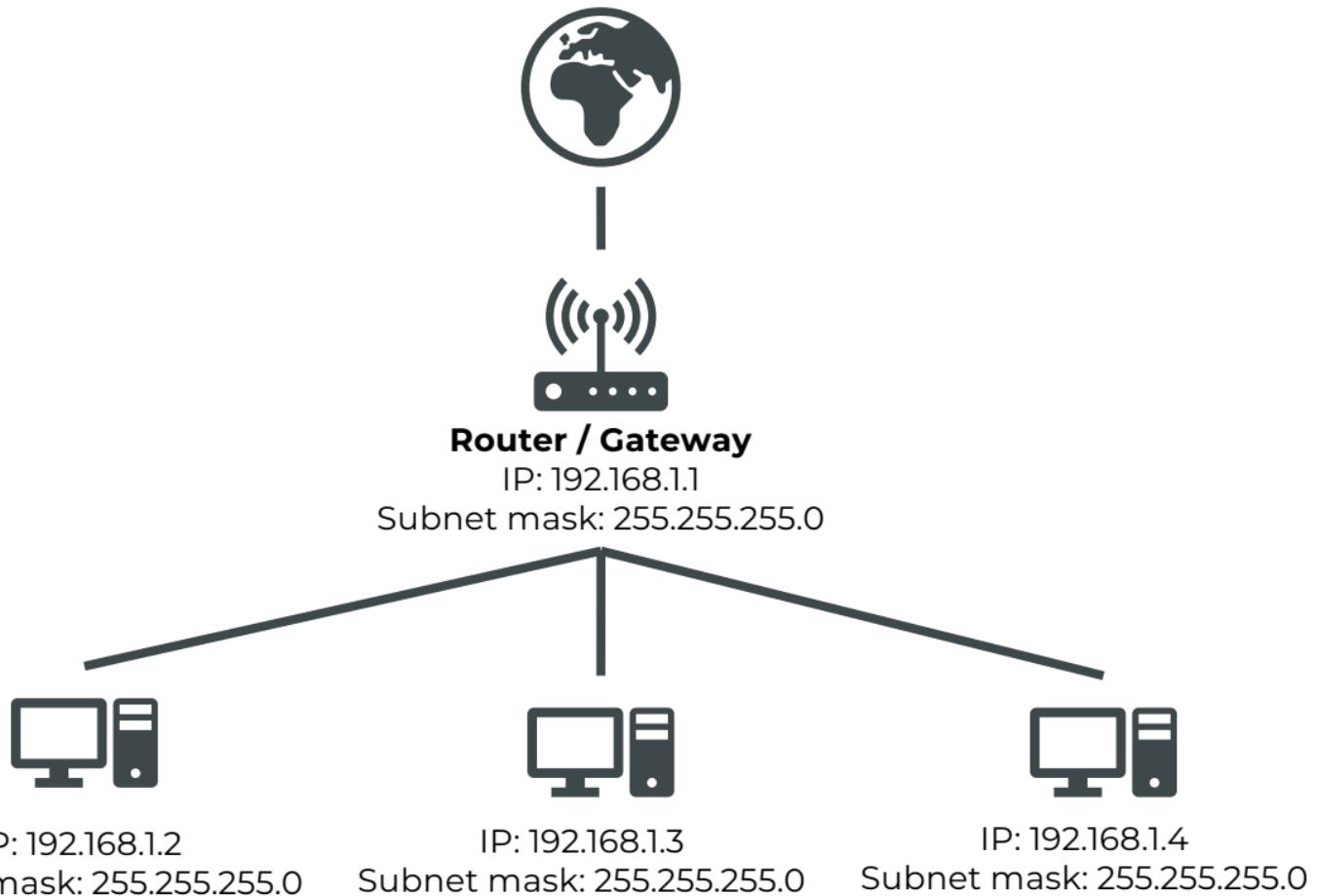


What does the subnet mask do?

- ▶ If we want to send a packet to another IP, we need to know where we want to send it to
- ▶ **Example 1:**
 - ▶ We want to send a packet to another computer in our local network
 - ▶ **We can send the frame directly to it:**
 - ▶ Btw, this is always the case, even if one computer connects through WiFi, and the other one through a LAN cable
 - ▶ In this case, the router part of the router will not be involved - the wireless access component of the router will be responsible for sending this packet to the WiFi connected device
- ▶ **Example 2:**
 - ▶ We want to send a packet to the internet
 - ▶ In this case, we need to send a packet to the router
 - ▶ Which will forward it for us to the internet

Bash & Linux CLI

Layer 3 (Networks layer): Subnet mask



Example: Subnet mask

- ▶ **So, if we have the following address:**

- ▶ IP: 192.168.1.2
- ▶ Subnet Mask: 255.255.255.0

- ▶ **Both will be represented as binary:**

- ▶ IP: 11000000.10101000.00000001.00000010
- ▶ Subnet Mask: 11111111.11111111.11111111.00000000

- ▶ **If we now want to send a packet to another computer in our LAN:**

- ▶ **Destination IP: 192.168.1.3 (11000000.10101000.00000001.00000011)**

- ▶ **Can we send the packet directly?**

- ▶ **We can apply a logical AND the subnet mask to our IP and the destination:**

- ▶ Applied to our IP: 11000000.10101000.00000001.00000000
- ▶ Applied to the destination IP: 11000000.10101000.00000001.00000000

- ▶ **Both are the same, so we can send the packet in the local network!**

Example: Subnet mask

- ▶ So, if we have the following address:

- ▶ IP: 192.168.1.2 (11000000.10101000.00000001.00000010)
- ▶ Subnet Mask: 255.255.255.0 (11111111.11111111.11111111.00000000)

- ▶ And we want to send a packet to:

- ▶ IP: 8.8.8.8 (00001000.00001000.00001000.00001000)

- ▶ We then apply the logical AND with the subnet mask to both IP addresses:

- ▶ Applied to our IP:

- ▶ 11000000.10101000.00000001.00000000

- ▶ Applied to 8.8.8.8:

- ▶ 00001000.00001000.00001000.00000000

- ▶ The result is different:

- ▶ => Thus, we need to send the packet to the gateway (our router)!

Shorter notations for subnet mask

- ▶ A subnet mask always starts with "1"s in binary form, usually followed by "0"s
- ▶ 255.255.255.0 => **11111111.11111111.11111111.00000000**
- ▶ Thus, we can also indicate the number of 1 immediately after our IP address
- ▶ **Example, our computer has the IP of:**
 - ▶ 192.168.1.3/24
 - ▶ **This means:**
 - ▶ IP: 192.168.1.3
 - ▶ **Subnet mask (24x 1):**
 - ▶ 255.255.255.0 => **11111111.11111111.11111111.00000000**
- ▶ **Of course, now this network can only use the following IP addresses:**
 - ▶ 192.168.1.1 -> 192.168.1.254
 - ▶ Broadcast-IP: 192.168.1.255

How can we inspect this?

- ▶ We can show information about our network devices with the following command:
 - ▶ ip addr show
- ▶ This will also show our IP address and subnet mask:
 - ▶ Example output:
 - ▶ inet 192.168.1.27/24

Bash & Linux CLI

Layer 3 (Networks layer): How we know where we need to send the frame to?



Router / Gateway

IP: 192.168.1.1

Mac: 94:a6:7e:5d:9a:3c

Subnet mask: 255.255.255.0



IP: 192.168.1.2

Mac: 00:1c:42:b5:66:42

Subnet mask: 255.255.255.0

IP: 192.168.1.3

Mac: b8:27:eb:2b:56:0d

Subnet mask: 255.255.255.0

IP: 192.168.1.4

Mac: a5:00:eb:2b:56:0e

Subnet mask: 255.255.255.0

Ethernet frame
To mac: b8:27:eb:2b:56:0d

IP packet
To IP: 192.168.1.3

The data the IP packet is
transferring

Bash & Linux CLI

Layer 3: How can change the IP address?

Layer 3: How to set IP address

- ▶ We can also use the tool `ip` to manage IP addresses:
 - ▶ List IP address(es):
 - ▶ `ip addr show`
 - ▶ Add an IP address to an interface:
 - ▶ `ip addr add <ip_address>/<prefix_length> dev <interface>`
 - ▶ Remove an IP address from an interface:
 - ▶ `ip addr del <ip_address>/<prefix_length> dev <interface>`
 - ▶ Example:
 - ▶ `ip addr add 192.168.1.10/24 dev enp0s5`

Important

► **Please note:**

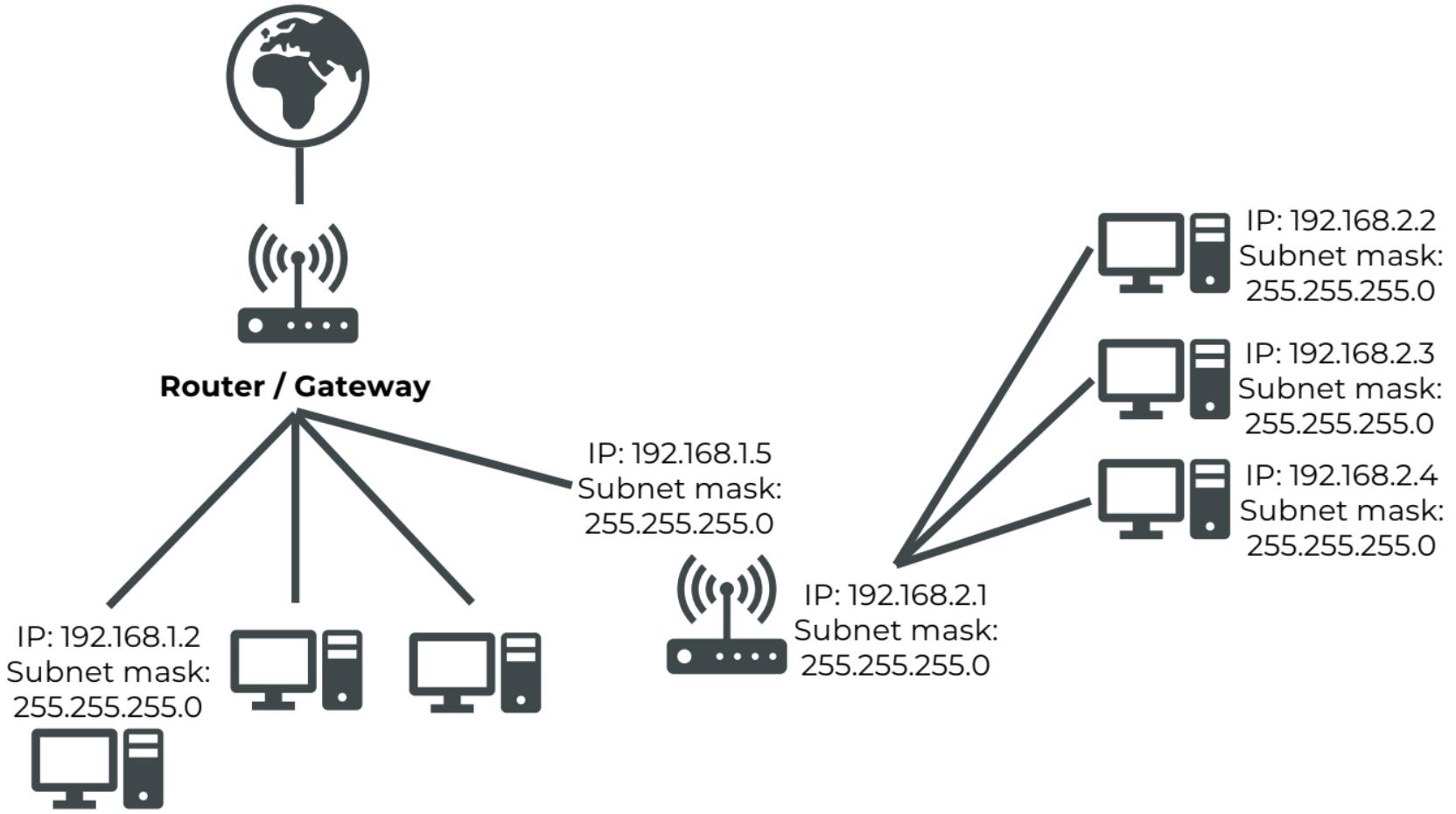
- ▶ In most LAN networks, the router is managing the IP addresses
- ▶ This process is called DHCP - more on that later
- ▶ It might be better to configure a MAC -> IP mapping in our router backend (might be called something like: LAN Setup -> "Reserved addresses")

Bash & Linux CLI

Layer 3: How can we inspect and modify routes?

Layer 3: Inspecting routes

- ▶ **How to we list routes?**
 - ▶ **Show the routing table:**
 - ▶ `ip route show`
 - ▶ **Display details for a specific route:**
 - ▶ `ip route get <destination>`
 - ▶ **Example:**
 - ▶ `ip route get 8.8.8.8`
- ▶ **Changing routes:**
 - ▶ **Add a route:**
 - ▶ `ip route add <destination> via <gateway> dev <interface>`
 - ▶ **Remove a route:**
 - ▶ `ip route del <destination> via <gateway> dev <interface>`
 - ▶ **Example:**
 - ▶ `ip route add 10.0.0.0/24 via 192.168.1.1 dev enp0s5`



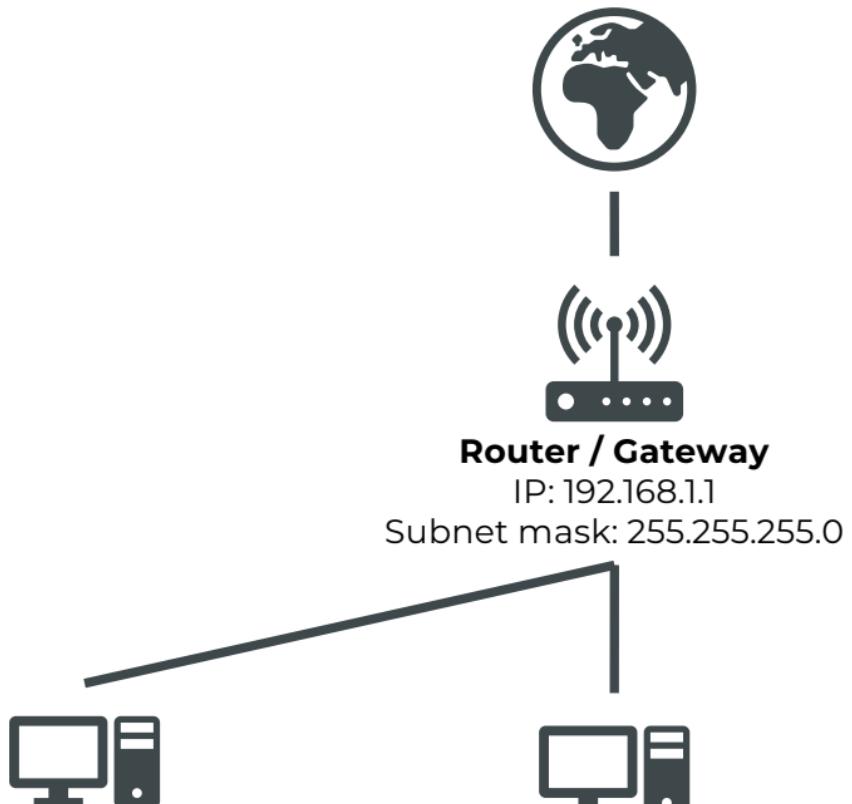
Bash & Linux CLI

Layer 3: How does the client get an IP address? => DHCP!

DHCP: Dynamic Host configuration protocol

- ▶ **How do our clients get their IP configuration?**

- ▶ DHCP (Dynamic Host Configuration Protocol)
- ▶ Network management protocol
- ▶ Automatically assigns IP addresses



IP: 192.168.1.2
Subnet mask: 255.255.255.0

**How does this Computer
get an IP?**

DHCP: Components

- ▶ **DHCP Server**

- ▶ Stores IP address pool
- ▶ Manages IP address leases
- ▶ Assigns and reclaims addresses
- ▶ **In a home network, this is usually the router**

- ▶ **DHCP Client**

- ▶ Requests IP address and configuration
- ▶ Renews or releases leases

- ▶ **Optional: DHCP Relay Agent**

- ▶ Forwards requests between subnets

- ▶ **Important technical hint:**

- ▶ DHCP helps us to manage layer 3, meaning that we can easily obtain the network configuration that we need
- ▶ Technically, DHCP is running on top of layer 4 (UDP). More on UDP later!

DHCP: The process

► 1.: Discover

- ▶ Client broadcasts DHCP discover message
(to the whole network, to the address: 255.255.255.255)
- ▶ Searches for DHCP server
- ▶ Other clients will just receive, but ignore this message

► 2.: Offer

- ▶ DHCP server responds with DHCP offer message
- ▶ This message contains IP address and lease information

► 3.: Request

- ▶ Client sends DHCP request message
- ▶ Accepts IP address and lease terms

► 4.: Acknowledge

- ▶ Server sends DHCP acknowledge message
- ▶ Confirms IP address assignment and lease duration

Bash & Linux CLI

Inspecting DHCP (systemd-networkd)

DHCP: Can we inspect this?

- ▶ On systems that use systemd to also manage their network, we can just show the logs for this unit:
 - ▶ `journalctl -u systemd-networkd`
- ▶ This will allow us to inspect potential DHCP related problems on our system
- ▶ **Let's have a look! 😊**

Bash & Linux CLI

Inspecting DHCP (NetworkManager)

DHCP: Can we inspect this?

- ▶ On systems that use **NetworkManager** to also manage their network, we can just show the logs for this unit:
 - ▶ `journalctl -u NetworkManager`
- ▶ This will allow us to inspect potential DHCP related problems on our system
- ▶ **Let's have a look! ☺**

Bash & Linux CLI

Layer 3 application: Ping

Layer 3: Ping

- ▶ Sometimes we want to inspect our connection to another computer
- ▶ **We can use the program ping for this:**
 - ▶ `ping 8.8.8.8`
 - ▶ The protocol is called ICMP (Internet Control Message Protocol)
 - ▶ Our program will send a package (ICMP: "Echo request") to the destination
 - ▶ If the destination supports the ICMP protocol, it should reply with an ICMP "Echo reply" packet
 - ▶ This allows us to measure the roundtrip time to a remote server

Bash & Linux CLI

Layer 3 application: Traceroute

Traceroute

► **Traceroute:**

- ▶ Network diagnostic tool
- ▶ Determines the path taken by data packets from source to destination
- ▶ Identifies network latency and potential routing issues

▶ **Let's have a look at how it works:**

- ▶ `traceroute google.com`

▶ **Output:**

- ▶ Hop number: Indicates the position of the router in the path
- ▶ Router IP address/hostname: Identifies the intermediate router
- ▶ RTT values: Latency measurements for each packet sent

▶ **This allows us to identify potential issues:**

- ▶ High latency: Indicates potential network congestion or problems
- ▶ Asterisks (*): Indicates packet loss or unresponsive router
- ▶ Routing loops: Repeated appearance of the same router in the output

Bash & Linux CLI

How does Traceroute work?

Traceroute

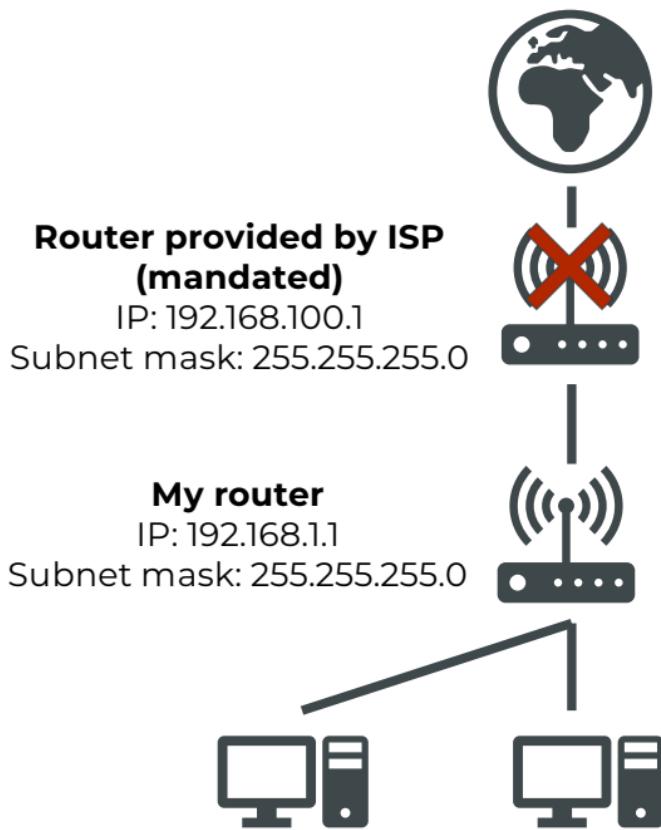
► **The idea, at first:**

- We send out an IP packet with a TTL of 1 to the destination
- Our router will decrease the TTL by 1. It's now 0
- Because it's now 0, it will discard the packet, and reply with a ICMP "Time Exceeded" packet
- We now have the IP of our router

► **The next step:**

- We now send out another IP packet with a TTL of 2
- Our router will decrease its TTL, and forward it to the next router
- The next router will decrease it, it's now 0
- Thus, the packet will be discarded, and we get another time exceeded packet back

► **Rinse and repeat!**



My network layout

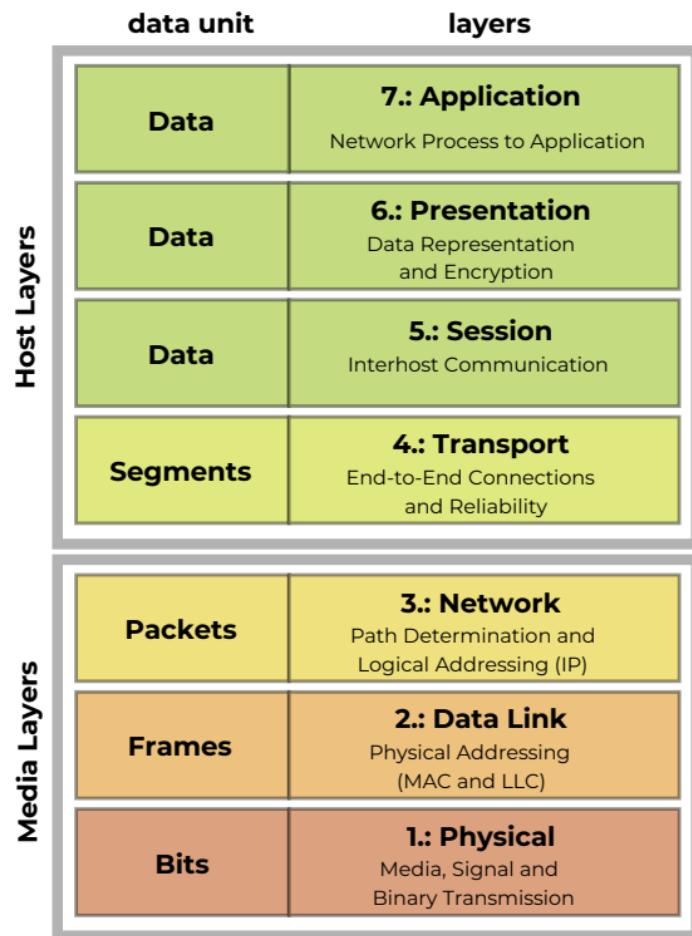
- ▶ My ISP (Internet Service Provider) provides an optical internet router, that I am mandated to use
- ▶ It would support WiFi... but I don't want my ISP to have control over my home network
- ▶ So how does my network look like?

Bash & Linux CLI

Layer 4: Transport layer

Layer 4: Transport layer

- ▶ Until layer 3, we still had various problems:
 - ▶ Packets might get lost or dropped
 - ▶ Dropped?
 - ▶ If a router is overloaded, it will just drop packets
 - ▶ We need to find a way how we want to deal with it
- ▶ There're 2 main ways to deal with it:
 - ▶ UDP:
 - ▶ We want our application to handle out-of-order / retransmissions
 - ▶ Sometimes it's better to accept data loss, and not re-transmit the data
 - ▶ Example: Video call!
 - ▶ TCP:
 - ▶ This will be handled on Layer 4
 - ▶ Packets will be ordered by the receiver, and re-transmitted



Layer 4: TCP

- ▶ **TCP: Transmission Control Protocol**

- ▶ Connection-oriented protocol
- ▶ Provides reliable, ordered, and error-checked data transmission
- ▶ Operates at Transport Layer (Layer 4) of OSI model

- ▶ **It allows us to write applications more easily:**

- ▶ They can see the connection as a data stream, that contains the data in a correctly ordered way, without any parts missing

- ▶ **In reality:**

- ▶ TCP is managing this
- ▶ It orders the data for us, and handles re-transmissions

- ▶ **It also ensures we utilize the bandwidth that the receiver can handle**

- / our connection can handle**

- ▶ Flow control (what the receiver can handle)
- ▶ Congestion control (what the connection can handle)

Bash & Linux CLI

Layer 4: Ports in TCP / UDP

Ports in TCP & UDP

► Ports in TCP:

- ▶ 16-bit numbers assigned to specific processes or services
- ▶ Range: 0-65535
- ▶ Differentiate between multiple connections on a single device

► Types of TCP Ports:

- ▶ Well-known ports (0-1023)
- ▶ Reserved for standard services and protocols
- ▶ Examples: HTTP (80), HTTPS (443), FTP (21), SSH (22)

► Registered ports (1024-49151):

- ▶ Assigned to specific applications by IANA
(Internet Assigned Numbers Authority)
- ▶ Examples: MySQL (3306), PostgreSQL (5432), VNC (5900)

► Dynamic or private ports (49152-65535):

- ▶ Not controlled by IANA
- ▶ Available for any application to use on an as-needed basis

TCP: Port communication

- ▶ **Source port:**

- ▶ Randomly assigned from dynamic/private port range
- ▶ Identifies the sending application on the client

- ▶ **Destination port:**

- ▶ Identifies the receiving application on the server
- ▶ Typically, this is a well-known or registered port number

- ▶ **Port combination:**

- ▶ Unique combination of source IP, source port, destination IP, and destination port
- ▶ Differentiates multiple connections between the same devices
- ▶ Allows multiple connections to coexist without conflicts

Bash & Linux CLI

Layer 4: Most used ports

Most common ports

► Most used TCP ports:

- ▶ HTTP (80): Web server communication
- ▶ HTTPS (443): Secure web server communication
- ▶ FTP (20, 21): File Transfer Protocol
- ▶ SSH (22): Secure Shell remote access
- ▶ Telnet (23): Remote terminal access (unencrypted)
- ▶ SMTP (25): Simple Mail Transfer Protocol
- ▶ IMAP (143): Internet Message Access Protocol
- ▶ POP3 (110): Post Office Protocol version 3

Most common ports

► Most used UDP ports:

- ▶ DNS (53): Domain Name System
- ▶ DHCP (67, 68): Dynamic Host Configuration Protocol
- ▶ SNMP (161, 162): Simple Network Management Protocol
- ▶ TFTP (69): Trivial File Transfer Protocol
- ▶ NTP (123): Network Time Protocol
- ▶ RTP (5004, 5005): Real-time Transport Protocol
(used for audio/video streaming)

Bash & Linux CLI

Layer 4: TCP: The handshake process

TCP: Three-way handshake

- ▶ How does a TCP handshake work?

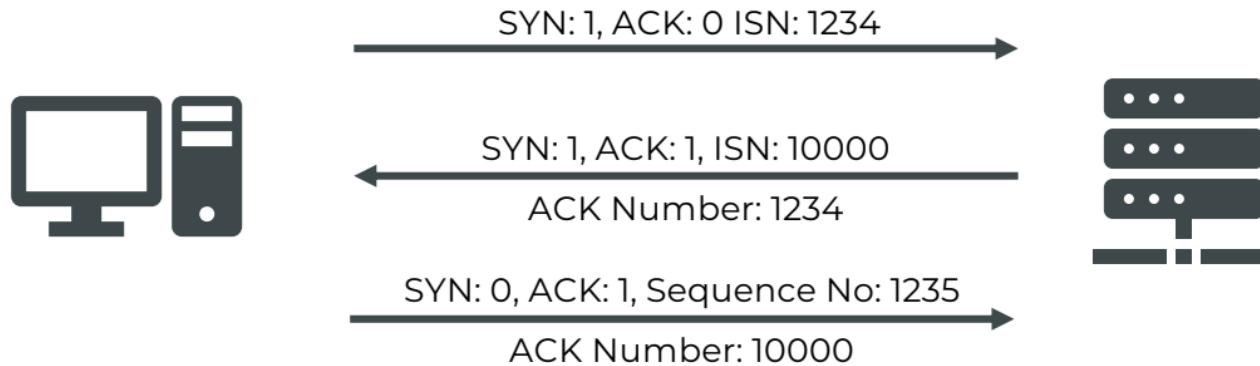
- ▶ **The goal:**

- ▶ Both computers need to know that the other side responds
- ▶ Both will later need to know how much data has already been received by the other side
- ▶ Thus, we need to exchange our "sequence numbers"

- ▶ **How does the handshake work?**

- ▶ First, our computer sends a SYN packet to the remote computer
- ▶ The remote computer will send a SYN-ACK packet back to us
- ▶ And then, we will reply with an ACK packet - the connection is successfully established now

Three-way handshake (TCP)



Bash & Linux CLI

Layer 4: The port scanner Nmap

The port scanner Nmap

- ▶ What is a port scanner?
- ▶ We want to find out which ports are open on another system
- ▶ **The idea:**
 - ▶ We can try to connect through all possible ports
 - ▶ If a port is open, we will get a message back from the server
 - ▶ This allows us to identify open ports and services that are available on the target system

Legal implications

- ▶ Port scanners are used quite often by attackers
- ▶ But also, by security experts ("ethical hackers"), to help them find potential vulnerabilities
- ▶ In my opinion, you should know how a port scanner works - it will allow you to detect open ports on your system
- ▶ And you will be able to test your firewall
- ▶ **But:**
 - ▶ Unauthorized port scanning can be illegal
 - ▶ Intrusion detection systems might flag you
- ▶ **Also:**
 - ▶ Laws in your jurisdiction might be different than in mine
 - ▶ Thus, before installing and running a port scanner, check your local laws!

The port scanner Nmap

► What is Nmap?

- Nmap is a powerful, flexible open-source tool
- We can use it for network discovery and security auditing
- It can also efficiently scan single hosts, or even large networks

► How to install Nmap:

► Debian / Ubuntu:

- `apt install nmap`

► CentOS / Red Hat:

- `dnf install nmap`

► Windows:

- You can download it from <https://nmap.org>

How to use Nmap

- ▶ **To scan a specific host:**

- ▶ By default, it will scan the most common 1000 TCP ports

```
nmap [hostname/IP]
```

- ▶ **We can also specify the port(s) manually:**

- ▶ nmap -p [port] [hostname/IP]
 - ▶ nmap -p [port1],[port2] [hostname/IP]
 - ▶ nmap -p [port1]-[port2] [hostname/IP]

- ▶ **To scan all ports:**

- ▶ nmap -p - [hostname/IP]

- ▶ **To scan a range of IP addresses:**

- ▶ nmap [first IP]-[last IP (last number)]
 - ▶ nmap 192.168.1.1-100

Bash & Linux CLI

Layer 4: Scan types in Nmap

Scan types in Nmap

- ▶ **We can choose between different types of scans:**

- ▶ **-sS (TCP-SYN-Scan):**

- ▶ The default scan method, if available
 - ▶ Nmap builds the packets and sends them
 - ▶ Relatively fast
 - ▶ Sends a SYN packet to establish a TCP connection, but doesn't follow through with the full connection
 - ▶ But we might need additional privileges (root)

- ▶ **If we receive a:**

- ▶ SYN/ACK: This port is open
 - ▶ RST (reset) message: This port is closed
 - ▶ No message: This port is probably blocked / filtered

Scan types in Nmap

- ▶ We can choose between different types of scans:

- ▶ **-sT (TCP-connect-scan):**

- ▶ The default, if SYN-scan is not possible:
 - ▶ User has not enough permissions to do a SYN-scan
 - ▶ We're trying to scan an IPv6 network
- ▶ Here, Nmap is using functionality of the operating system to create a connection
- ▶ **This means we do the full handshake with the remote server:**

- ▶ SYN, SYN-ACK, ACK
- ▶ And then just close the connection

- ▶ **This is especially prone to:**

- ▶ Be slower
- ▶ Can cause crashing at the other side
- ▶ Might cause logs at the other side

Scan types in Nmap

- ▶ **We can choose between different types of scans:**

- ▶ **-sU (UDP-scan):**

- ▶ We can use this to scan for open UDP ports
 - ▶ UDP works differently than TCP though

- The idea here:**

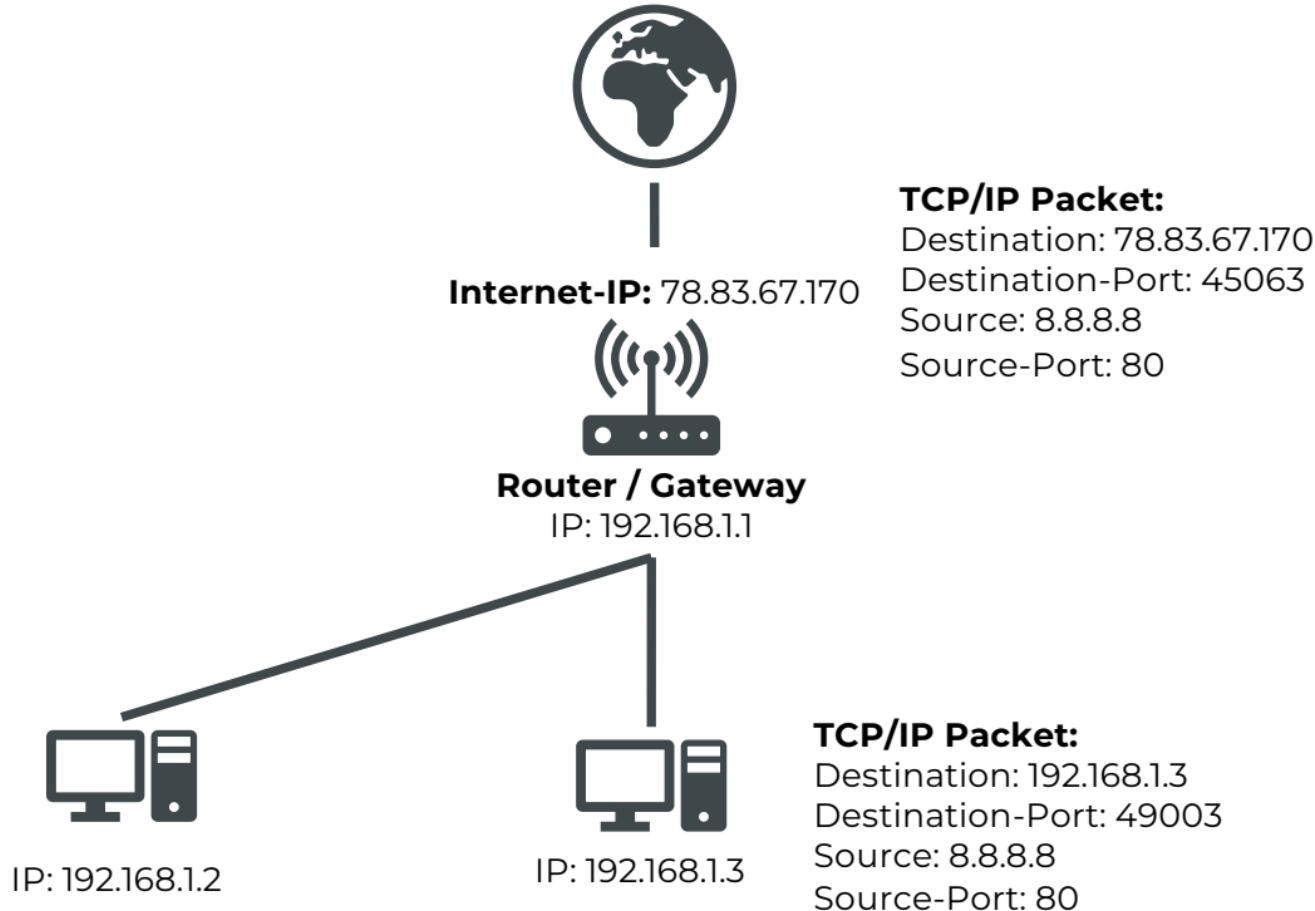
- ▶ We send a UDP packet to the remote destination
 - ▶ And if we get a reply, the port will be marked as "open"
 - ▶ If we get an error back, we will mark this port as "filtered"
 - ▶ If we don't get a reply, we will mark the port as "open|filtered"

- Important:**

- ▶ UDP port scanning is extremely slow
 - ▶ And we need to test each port multiple times as our request might have gotten lost on the way

Bash & Linux CLI

Layer 4: Network Address Translation (NAT)



Network Address Translation

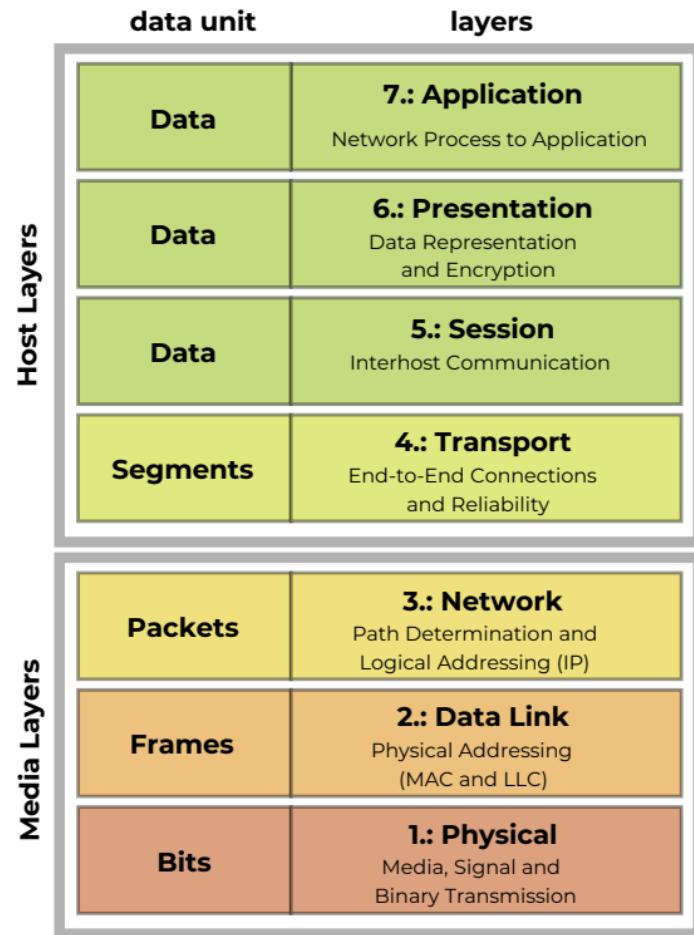
- ▶ Why is this important for you?
- ▶ Let's say you want to make a service available to the internet
- ▶ And you're behind a NAT
- ▶ **How do you do this?**
 - ▶ You need to tell your router, that when receiving requests on port 80 (or any other port), they should be forwarded to one of the devices in the network
- ▶ **Example:**
 - ▶ Port 80 should be forwarded to 192.168.1.3 on Port 8080
 - ▶ You should then reserve IP <-> MAC address combination for the IP 192.168.1.3 (so that no other DHCP client will get assigned this IP)
 - ▶ **If your home IP address changes:** You might need to use a service for accessing dynamic IPs (such as "dyndns" or others)
 - ▶ **Then you could use a domain such as:**
 - ▶ myhost123.dyndns.com to access your local computer from the internet

Bash & Linux CLI

Layer 5: Session layer

OSI model: Layer 5-7

- ▶ The layer 5-7 are often handled completely by the application
- ▶ Thus, the distinction between those layers is no longer as clear as before
- ▶ Sometimes they also use specific features of the layers below
- ▶ Still, those layers give us an overview about how an application can be structured



Layer 5: Session Layer

► **Session Layer**

- ▶ Establishes, maintains, terminates connections
- ▶ Supports communication between applications

► **Examples of Layer 5 protocols**

▶ **Network File System (NFS)**

- ▶ Remote file access over a network

▶ **Remote Procedure Call (RPC)**

- ▶ Request services from remote devices

▶ **Session Control Protocol (SCP)**

- ▶ Manages sessions between devices

Bash & Linux CLI

Layer 6: Presentation layer

Presentation layer

► **Presentation Layer**

- ▶ Deals with data representation
- ▶ Ensures data compatibility and security

► **Functions at Layer 6**

- ▶ Data conversion
- ▶ Transform data formats (e.g., ASCII, EBCDIC, Unicode)

► **Encryption/decryption**

- ▶ Secure data transmission (e.g., SSL/TLS)
- ▶ Data compression
- ▶ Reduce data size (deflate, brotli compression,...)

Example protocols

- ▶ **SSL/TLS (Secure Sockets Layer/Transport Layer Security):**
 - ▶ Provide secure communication over a network
 - ▶ Originally designed for securing HTTP connections
 - ▶ We might also consider it as operating at both level 6 and 7
- ▶ **MIME (Multipurpose Internet Mail Extensions):**
 - ▶ Defines how E-Mails support attachments, character encodings
(öäüß,...) and other features

MIME: Example

Delivered-To: info@example.com
Received: by <mac address> with SMTP id wh5csp414397pxb
Return-Path: <sender@example.com>
Received: from mail-sor-f41.google.com (mail-sor-f41.google.com.
[209.85.220.41]) by mx.google.com with SMTPS
Received-SPF: pass (google.com: domain of sender@example.com designates
209.85.220.41 as permitted sender) client-ip=123.45.67.89;
From: Jannis Seemann <sender@example.com>
Date: Fri, 5 May 2070 16:25:30 +0300
Message-ID: <CABARR2LYeqqoY-UMjisukB3yXW4cHP3wiy6bX=wPg-
rXmbws0A@mail.gmail.com>
Subject: Test
To: Jannis Seemann <info@example.com>
Content-Type: multipart/alternative; boundary="000000000003cee3205faf23bee" --
000000000003cee3205faf23bee Content-Type: text/plain; charset="UTF-8" *Hello
World!* -- Sent from my iPhone --000000000003cee3205faf23bee Content-Type:
text/html; charset="UTF-8" Content-Transfer-Encoding: quoted-printable <div
dir=3D"ltr"><div class=3D"gmail_default" style=3D"color:#000000">He= llo
World!</div><div><div dir=3D"ltr" class=3D"gmail_signature" data-sma=
rtmail=3D"gmail_signature"><div dir=3D"ltr"><div><div dir=3D"ltr"><div>
=</div><div>--</div><div>Sent from my iPhone</div></div></div></div></div></div> --000000000003cee3205faf23bee--

Hello World!

--

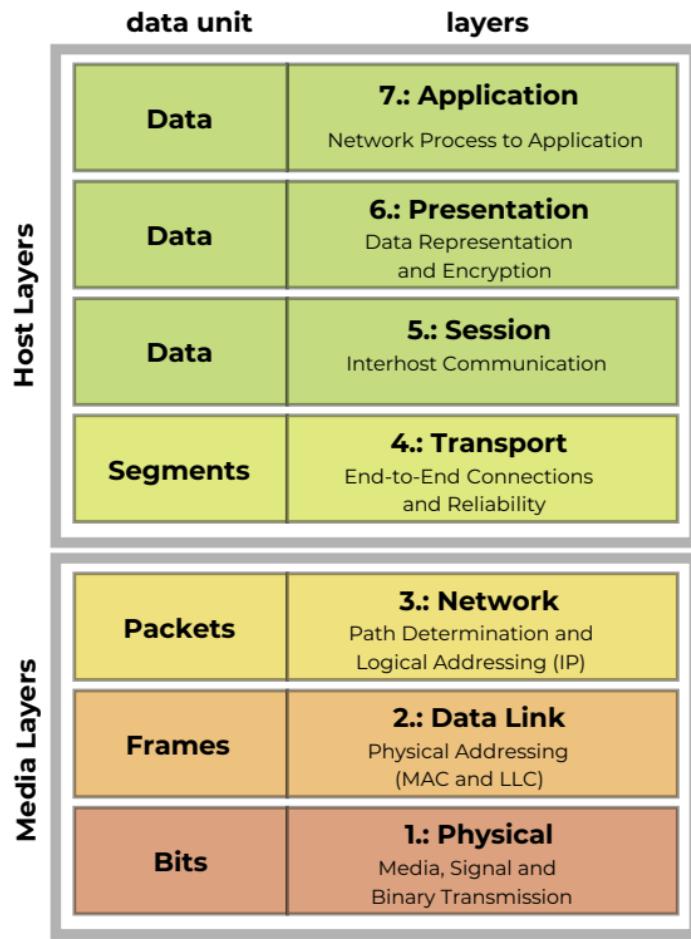
Sent from my iPhone

Bash & Linux CLI

Layer 7: Application layer

Application layer

- ▶ The application layer are the protocols that our application can then use
- ▶ **Examples would be:**
 - ▶ HTTP / HTTPS (for websites)
 - ▶ IMAP (for accessing emails on a remote server)
 - ▶ SSH (accessing a remote shell / scp,...)
 - ▶ POP3 (downloading e-mails)
 - ▶ Proprietary protocols (for example, custom VOIP implementations)



Bash & Linux CLI

Layer 7: The DNS protocol

DNS protocol

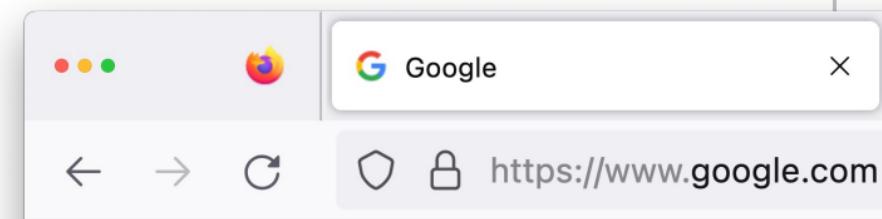
- ▶ **DNS: Domain Name System**

- ▶ Application Layer protocol
- ▶ Translates domain names to IP addresses
- ▶ Facilitates human-readable access to websites and services

- ▶ **The idea:**

- ▶ Instead of typing in the IP-address, we just want to type in "google.com", and our computer should do the rest

The DNS process



► The problem:

- ▶ How does our Browser know to which IP address it should connect to?

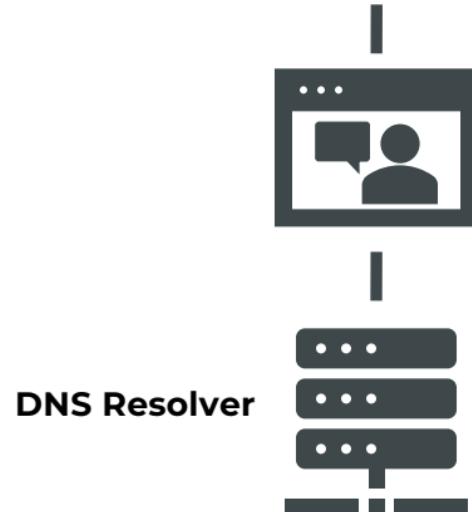
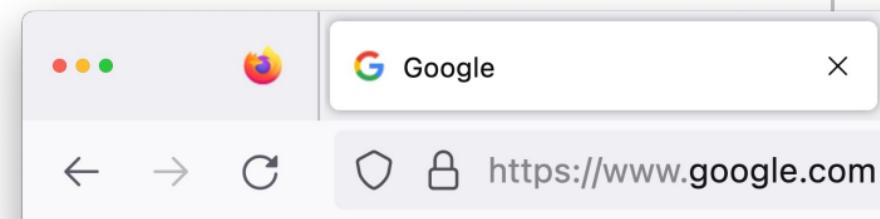
► Step 1:

- ▶ Local cache: The browser has its own cache and stores a mapping between Domains and their IP addresses
- ▶ If the IP can't be found there, the browser asks the operating system to resolve the domain name
- ▶ The operating system also has a local cache. If the IP can be found in it, we're done!
- ▶ If not, we need to continue the process

The DNS process

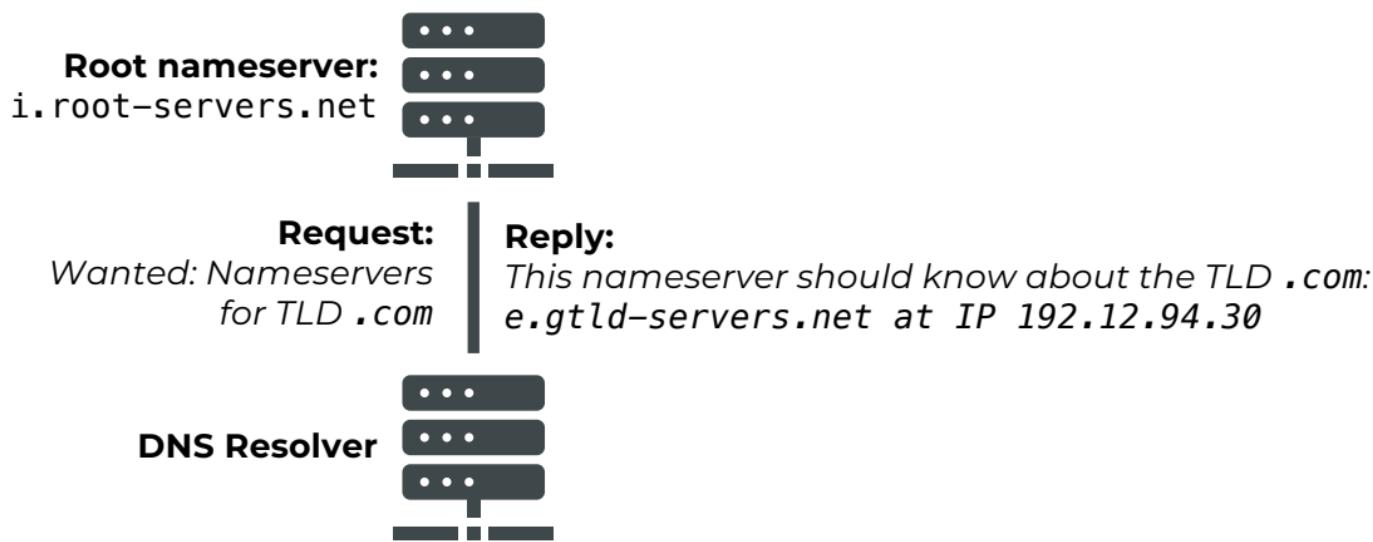
► Step 2: DNS resolver

- ▶ They help us resolve the DNS entries from a domain
- ▶ Usually provided by our Internet Service Provider (ISP)
- ▶ It will check its cache (resolver cache)
- ▶ If it can't find the IP address in its cache, it needs to resolve it
- ▶ How does it do it?



DNS: Resolving

- ▶ If the DNS resolver needs to resolve a Domain name, it will reach out to one of 13 root nameserver (at random)
- ▶ They are labeled with letters from A to M



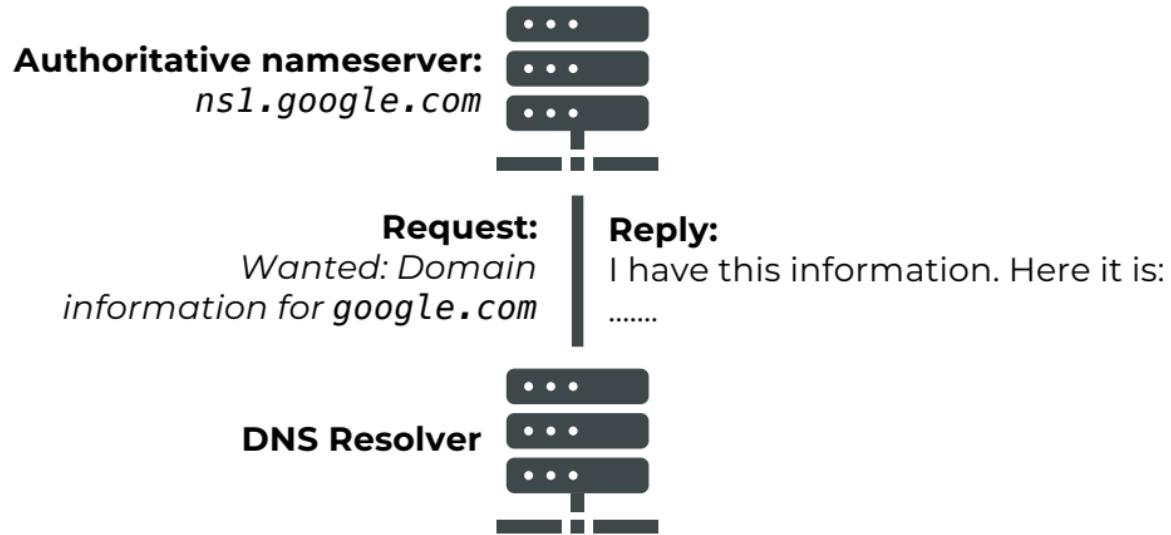
DNS: Resolving

- ▶ So, now we have the TLD nameserver...
- ▶ How does the resolving continue?



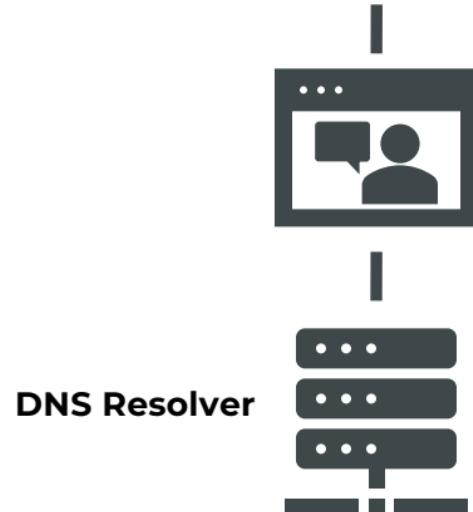
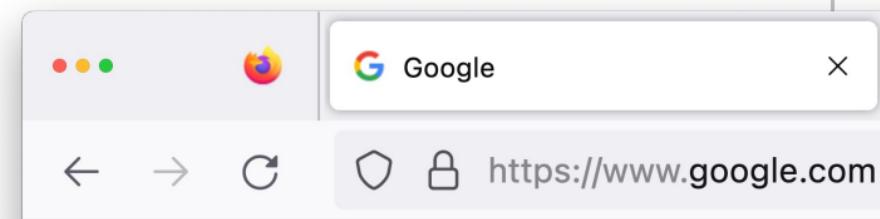
DNS: Resolving

- ▶ Now we got the authoritative nameserver for google.com
- ▶ Let's now continue!



The DNS process

- ▶ Now, the DNS resolver of our ISP has the DNS information
- ▶ And can now send this to our operating system
- ▶ And the operating system will give this information to our browser
- ▶ We can now finally send the request to google.com 😊



Bash & Linux CLI

Layer 7: DNS: Types of records

Type of DNS records

- ▶ **Common DNS record types**

- ▶ A: Maps a domain name to an IPv4 address
- ▶ AAAA: Maps a domain name to an IPv6 address
- ▶ CNAME: Provides an alias for another domain name
- ▶ MX: Specifies mail servers for a domain
- ▶ NS: Lists authoritative name servers for a domain

- ▶ **We can list the received DNS entries with the following command:**

- ▶ `host <server>`
- ▶ `host -a <server>`

- ▶ **Example:**

- ▶ `host -a google.com`

- ▶ **Let's have a look at this!**

Bash & Linux CLI

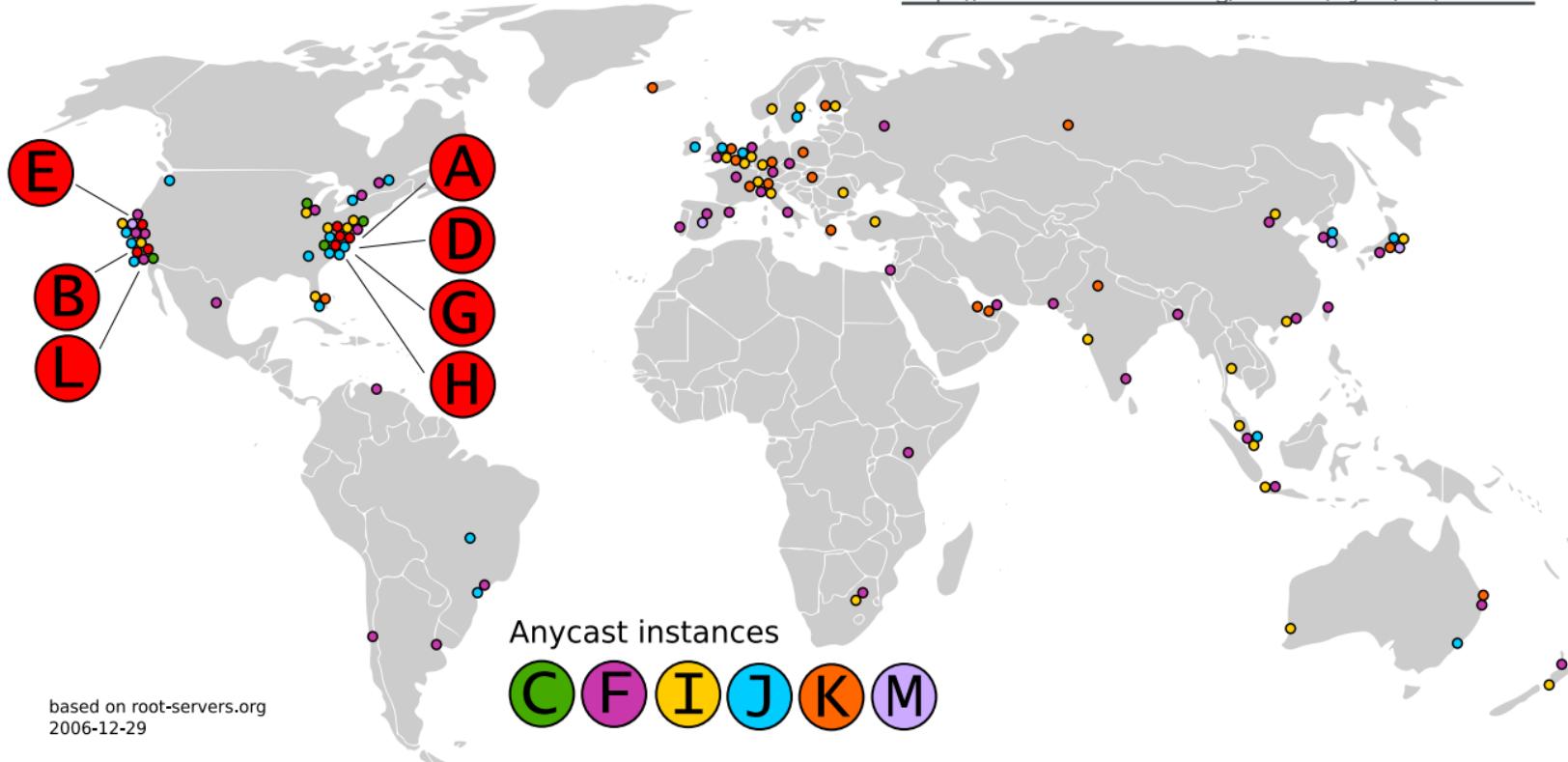
Layer 7: Bonus: DNS by hand

DNS by hand

- ▶ We can also manually send out DNS queries
- ▶ This is a nice way to experience how a DNS query is resolved
- ▶ For this, we can use the dig program
- ▶ Let's try to resolve google.com!
- ▶ **We will first have to pick a root server that we want to query:**
 - ▶ `dig @a.root-servers.net com NS`

Root nameservers

Copyright notice: No machine-readable author provided. Matthäus Wander assumed (based on copyright claims). CC BY-SA 3.0
<https://commons.wikimedia.org/wiki/File:Root-current.svg>
<https://creativecommons.org/licenses/by-sa/3.0/deed.en>



DNS by hand

- ▶ We can also manually send out DNS queries
- ▶ This is a nice way to experience how a DNS query is resolved
- ▶ For this, we can use the dig program
- ▶ Let's try to resolve google.com!
- ▶ **We will first have to pick a root server that we want to query:**
 - ▶ `dig @a.root-servers.net com NS`
- ▶ **And then, we can query the TLD nameserver of the .com domain:**
 - ▶ `dig @e.gtld-servers.net google.com NS`
 - ▶ `dig @ns1.google.com google.com any`

Bash & Linux CLI

The problems with DNS, DNSSEC

DNS: Problems

► **The problem:**

- DNS has been designed when the internet was significantly smaller, and we were still able to trust everyone
- This is no longer the case

► **DNS vulnerabilities:**

- DNS spoofing: Attacker redirects traffic by altering DNS records
- Cache poisoning: Attacker inserts malicious DNS entries into a DNS resolver's cache
- Man-in-the-middle attacks: Attacker intercepts DNS queries and provides false responses
- We can mitigate some consequences of that, by transferring data (not the DNS queries) via encrypted protocols such as https (and refusing to accept invalid certificates)

► **We could also mitigate the problem on a DNS level:**

- DNSSEC (DNS Security Extensions)
- But this is not part of this course

Bash & Linux CLI

Manually assigning an IP to a domain

Manually resolve an IP address

- ▶ We can also manually define the IP address that should resolve:

- ▶ **We can edit the following file:**

- ▶ /etc/hosts

- ▶ **The syntax is as follows:**

- ▶ <ip> <host>

- ▶ **Example:**

- ▶ 127.0.0.1 localhost

- ▶ **But we can also specify any other domain that should resolve to another IP:**

- ▶ 127.0.0.1 my-project.project

- ▶ 127.0.0.1 google.com

- ▶ 192.168.1.4 backup-server

Refreshing the local DNS server

- ▶ We might have to refresh our local DNS server
- ▶ **Mac:**
 - ▶ `sudo dscacheutil -flushcache`
- ▶ **Linux:**
 - ▶ **We can find out if we're using systemd-resolved or dnsmasq:**
 - ▶ `sudo lsof -i :53`
 - ▶ **If we're using systemd-resolved (most common choice):**
 - ▶ `sudo resolvectl flush-caches`
 - ▶ **We can also show the status:**
 - ▶ `sudo resolvectl status`
 - ▶ `sudo resolvectl statistics`
 - ▶ **Otherwise, if we're using dnsmasq... we can just restart this service:**
 - ▶ `sudo systemctl restart dnsmasq`

Bash & Linux CLI

Hostname & mDNS (.local)

Hostname

► **What is the hostname?**

- ▶ In our local network, we can use the hostname to easily access other computers
- ▶ It will be used during the DHCP negotiation...
- ▶ ... but also to allow easy access from other computers in our network

► **How do we display the hostname?**

- ▶ We can use the following command: `hostname`

► **How do we change our hostname?**

- ▶ We need to edit the following file: `/etc/hostname`
- ▶ And then also replace our existing hostname in the `/etc/hosts` file
- ▶ We should now reboot the computer to apply the changes

How does a local hostname get resolved?

- ▶ We usually can access other devices in our network through their hostname:
 - ▶ `ping ubuntu.local`
- ▶ Why do we sometimes need to add a `.local` at the end?
- ▶ This is called mDNS (Multicast DNS Standard)
- ▶ It's different from normal DNS
- ▶ **The idea:**
 - ▶ We designate a special domain (defined in mDNS: `.local`) just for local networks
 - ▶ Then, we can send a mDNS query to our whole network, and the target computer will reply with its IP address
- ▶ **Important:**
 - ▶ Both computers must be configured correctly
 - ▶ It's implemented in macOS by default (Bonjour zeroconf)
 - ▶ On Linux, the avahi daemon must be running and configured
 - ▶ **On CentOS, be sure to install `nss-mdns` and reboot the system after:**
 - ▶ `dnf install nss-mdns`
 - ▶ Windows also supports some features of mDNS

Bash & Linux CLI

How does HTTP work?

The program: telnet

- ▶ Sometimes, we want to manually create a TCP connection to a destination

- ▶ **Example:**

- ▶ We want to test the network connectivity on a certain server
- ▶ We want to see how the server behaves in unexpected situations (for example: violations to the protocol standard)
- ▶ For this, we can use the tool `telnet`

- ▶ **In our case:**

- ▶ We will use telnet to create an HTTP connection to a remote server
- ▶ All by ourselves, without a browser!
- ▶ How can we do it?
 - ▶ `telnet [host] 80`

The program: OpenSSL

- ▶ **If our connection is using SSL (such as HTTPS, HTTP over SSL):**
 - ▶ Then we need to use a program that handles the SSL handshake and encryption for us
- ▶ **For this, we can use the openssl command:**
 - ▶ `openssl s_client -connect [host]:[port]`

Bash & Linux CLI

IPv6

IPv6: Internet

► IPv4: Internet Protocol version 4

- ▶ Most widely used version
- ▶ 32-bit address space:
- ▶ 4.3 billion unique addresses
- ▶ Dotted-decimal notation (192.168.1.1)

► IPv6: Internet Protocol version 6

- ▶ Successor to IPv4
- ▶ 128-bit address space
- ▶ 3.4×10^{38} unique addresses
- ▶ 8 groups of 4 hexadecimal digits

(example: 2002:c0a8:640f:0000:0000:42ff:feb5:6642)

IPv6: Structure of the addresses

- ▶ **IPv6: Internet Protocol version 6**
- ▶ **Example:**
 - ▶ 2002:0000:640f:0000:0000:42ff:feb5:6642
- ▶ **We can shorten IPv6 addresses, by replacing 0000 with just a 0:**
 - ▶ 2002:0:640f:0:0:42ff:feb5:6642
- ▶ If multiple blocks with only zeros follow each other, we can combine them to a ":" (but only 1x in the whole address):
 - ▶ 2002:0:640f::42ff:feb5:6642

IPv6: Simplified networking

► IPv4:

- ▶ Not enough addresses, we had to be quite conservative with addresses
- ▶ Quite often, we must rely on NAT (Network-Address-Translation)
- ▶ This hides multiple computers behind one external address

► IPv6:

- ▶ IPv6 fixes this
- ▶ `2002:0000:640f:0000:0000:42ff:feb5:6642`
- ▶ The first half of the address identifies our network
- ▶ And the second half our computer
- ▶ We no longer need to use NAT

IPv6: Differences

- ▶ **However, our applications need to be compatible**
- ▶ **Example:**
 - ▶ **DHCP:**
 - ▶ Only supports IPv4
 - ▶ We need to also use DHCPv6, to get an IPv6 address as well
 - ▶ **A lot of other applications needed add compatibility:**
 - ▶ Code had to be adjusted
 - ▶ Luckily, this is done by now!
- ▶ **But usually, we can just use both at the same time:**
 - ▶ **Dual stack:**
 - ▶ IPv4 and IPv6 addresses used simultaneously
 - ▶ **Tunneling:**
 - ▶ IPv6 packets encapsulated within IPv4 packets for transmission
 - ▶ **Translation:**
 - ▶ Converting IPv6 packets to IPv4 packets, and vice versa

IPv6: The way to go

► **For internal networks:**

- Dual Stack for the whole network, preferably also with an IPv4 and an IPv6 external IP

► **For servers:**

- IPv4 & IPv6 IP: We want to maximize compatibility!

Bash & Linux CLI

Connecting to a remote server: ssh

Connecting to a remote server

- ▶ We can use the SSH protocol to connect to a remote server
- ▶ This allows us to control a remote server
- ▶ **This chapter is important for you:**
 - ▶ If you are or will be the sysadmin of a server
 - ▶ If you are a web developer and want to deploy your application on your own server
 - ▶ If you use tools like docker (or Kubernetes) to manage your cluster (and then a problem occurs that you need to manually debug)
 - ▶ Or want to use a small computer without a display (for example a Raspberry Pi)
- ▶ **This chapter is more than just theory:**
 - ▶ You will learn how you can secure your server and connect with a private key instead of a password
 - ▶ You will also learn how to prevent annoying disconnects (for example during a coffee break)

Bash & Linux CLI

What is SSH?

SSH: Secure Shell

► **Definition:**

- ▶ A cryptographic network protocol for securely accessing and managing network devices and servers
- ▶ Encrypts all data transmitted between the client and server, ensuring confidentiality and integrity

► **Use Cases:**

- ▶ Remote command execution
- ▶ Remote system administration
- ▶ Secure file transfer (using SCP or SFTP)
- ▶ Secure tunneling for other network protocols (e.g., X11, VNC)

SSH: Terminal

- ▶ The most common use case for ssh is that we use it to control a remote server through the terminal
- ▶ **For this, we need the following:**
 - ▶ **We need an SSH server:**
 - ▶ This is what we need to install on the server
 - ▶ It will accept incoming connections and allow us to connect to it
 - ▶ **We also need an SSH client:**
 - ▶ Most operating systems already ship with one
 - ▶ Even on Windows, we don't need to install anything!

SSH: Terminal

- ▶ **How will we use SSH in this course?**

- ▶ SSH server and SSH client could also just run on the same computer
- ▶ But that would be rather useless

- ▶ **Thus, in the next 2 lectures, we'll have a look at possible network configurations:**

- ▶ **Method 1:**

- ▶ Allows you to connect from your host (Windows / Mac / Linux) to your virtual machine through ssh

- ▶ **Method 2:**

- ▶ We will have 2 virtual machines, and we can connect from one of them to the other through ssh

Bash & Linux CLI

Let's create our network

Bash & Linux CLI

SSH server

SSH server

- ▶ **First, if we want to use SSH:**

- ▶ We need to install an SSH server on the computer we want to control

- ▶ **On Ubuntu:**

- ▶ `apt install openssh-server`
 - ▶ Depending on how we installed Ubuntu, it might already be active and installed

- ▶ **On CentOS:**

- ▶ `dnf install openssh-server`
 - ▶ Also here, it might already be installed

SSH server

- ▶ **This will then allow us to connect to this server:**

- ▶ We can use the ssh command for this
- ▶ We will then be asked for our password
- ▶ `ssh user@server`
- ▶ The user must exist on the remote server!

- ▶ **Example:**

- ▶ `ssh jack@localhost`
- ▶ `ssh jack@192.168.1.15`
- ▶ `ssh jack@ubuntu1.local`
- ▶ `ssh jack@example.com`

- ▶ **Important:**

- ▶ We might be shown a fingerprint warning. For now, we will just confirm this with "yes"
- ▶ This is an important security feature - more on that later!

Bash & Linux CLI

SSH server: Security (part 1, ssh port)

Bash & Linux CLI

SSH server: Security (part 1, ssh port, CentOS)

SSH server: Security

- ▶ **Security implications of SSH:**
 - ▶ SSH is an encrypted protocol, so usually the connection should be secure
- ▶ **We should still take quite a few precautions:**
 - ▶ Our password must be long enough, so an attacker cannot guess it
 - ▶ We should never leave a terminal open and leave our laptop in a public space - an attacker could just use that terminal then
- ▶ **But we should still do some additional changes to the configuration**

SSH server: Change port

- ▶ **SSH runs on port 22 by default**
- ▶ **The problem:**
 - ▶ There're many servers on the internet, that just scan other computers to see if SSH is open
 - ▶ We don't want our computer to be scanned - it's just polluting our log files.
- ▶ **The way to go:**
 - ▶ We should change the port to something else
 - ▶ We can edit the sshd config in the following file:
 - ▶ `/etc/ssh/sshd_config`
 - ▶ There, we can uncomment the line "#Port 22", and replace it with something else, for example: Port 222
- ▶ **Then, we need to restart sshd:**
 - ▶ First, test the configuration: `sshd -t`
 - ▶ And then restart sshd: `systemctl restart sshd`
- ▶ **To now connect to the server:**
 - ▶ `ssh janniss@ubuntu.local -p 222`
- ▶ **But be careful:**
 - ▶ Sometimes, firewalls of your internet provider / company / ... might just block "uncommon" ports

Monitoring the log file

- ▶ **If you want to monitor the log file:**
 - ▶ All of this is stored in /var/log/auth.log
 - ▶ You can grep for "sshd" in this file

Bash & Linux CLI

SSH server: Security (part 2)

SSH server: Disable root login

► The problem:

- ▶ By default, all normal users with password can use SSH
- ▶ Depending on the configuration, sometimes this also applies to the user root
- ▶ We should check, and consider disabling root login:
 - ▶ `PermitRootLogin no`
- ▶ This helps a little bit, as an attacker now has to first login as a user, and only then can elevate his privileges with sudo

SSH server: Whitelist users

- ▶ **The problem:**

- ▶ By default, all normal users with password can use SSH
- ▶ We might also want to whitelist users, so we have to explicitly think about who is able to access our server through ssh:
 - ▶ `AllowUsers [username]`
- ▶ **Example:**
 - ▶ `AllowUsers janniss`

Bash & Linux CLI

SSH: How to not lock ourselves out

How to not lock ourselves out

- ▶ When we are the admin of a remote server
- ▶ We never want to lock ourselves out of the server
- ▶ Fixing this would be rather difficult
- ▶ **My tips:**
 - ▶ Whenever something has changed with the sshd service
 - ▶ Try to connect from a second terminal to the same server
 - ▶ The first connection will usually still work, even if the configuration of the service has changed
- ▶ **Also of course:**
 - ▶ Always use `sudo sshd -t` to test the configuration first!
- ▶ **Let's have a look at what I mean!**

Bash & Linux CLI

SSH: public / private key

SSH: Key auth

- ▶ Another way to connect to an ssh server is through public / private key auth
- ▶ **The idea:**
 - ▶ **We generate a key pair:**
 - ▶ **Private key:** This key is private. We should never share it with anyone
 - ▶ **Public key:** This key is public. Anyone could theoretically have it, and it can be used to verify our private key
 - ▶ **Then, we can use this keypair for authentication!**

SSH: Key auth

- ▶ To generate a key pair, we can use the command:

- ▶ `ssh-keygen -t rsa -b 4096`

- ▶ This will generate a key pair:

- ▶ Private key: `~/.ssh/id_rsa`

- ▶ Public key: `~/.ssh/id_rsa.pub`

- ▶ We now need to transfer the public key into the file
`~/.ssh/authorized_keys` on the remote server

- ▶ We could do this manually:

- ▶ Connect to the server, create the file (if needed)

- ▶ And copy the public key into that file

- ▶ We also should set the permissions quite restrictive:

- ▶ `chmod 700 ~/.ssh`

- ▶ `chmod 600 ~/.ssh/authorized_keys`

- ▶ The easiest way:

- ▶ `ssh-copy-id -i ~/.ssh/id_rsa.pub user@server`

- ▶ `ssh-copy-id -i ~/.ssh/id_rsa.pub janniss@ubuntu.local`

Bash & Linux CLI

SSH: Disabling password auth

SSH: Disable password auth

- ▶ Now that key auth has been enabled, we can disable password auth
- ▶ This further reduces the attack surface - the key is way longer than any password could ever be
- ▶ **An attacker then need to know both:**
 - ▶ The private key
 - ▶ And the password of our user (for everything that needs sudo)
- ▶ **We can disable password auth with the following option:**
 - ▶ **File:** /etc/ssh/sshd_config
 - ▶ PasswordAuthentication no
- ▶ **Of course we need to restart sshd after:**
 - ▶ **Test config:** sshd -t
 - ▶ **Restart sshd:** systemctl restart sshd

Bash & Linux CLI

SSH: Prevent connection drops

SSH: Prevent connection drops

► By default:

- ▶ SSH builds a connection between a client and a server
- ▶ If no data is transmitted through this connection, after a certain time, the connection is dropped
- ▶ If we then try to use the connection again, it will fail
- ▶ This means, that during a lunch / coffee break, our connection might drop
- ▶ And we have to start from scratch again!

SSH: Prevent connection drops

- ▶ **How do we fix this?**
 - ▶ We can configure "keep-alive" packets, either on server side, or on client side
 - ▶ Best-practice: We configure this for the client side
- ▶ **We need to open one of the config files for the ssh program (not sshd!):**
 - ▶ **User config:** `~/.ssh/config` (should have `chmod 600`)
 - ▶ **System wide config:** `/etc/ssh/ssh_config`
- ▶ **There, we add the following content:**
 - ▶ Host *
 - `ServerAliveInterval 60`
 - `ServerAliveCountMax 3`
- ▶ **This will:**
 - ▶ Send an empty packet every 60 seconds to keep the connection open
(keep-alive packet)
 - ▶ Allow up to 3 of those keep-alive packets to fail, and the connection will remain open

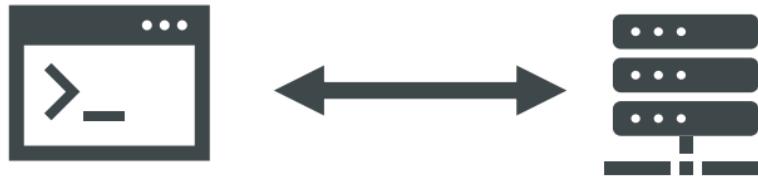
Bash & Linux CLI

SSH: Invalid key fingerprint

SSH: Invalid key fingerprint

- ▶ If we access a server for the first time, we store its key fingerprint on our system
- ▶ Each system has its own fingerprint(s)
- ▶ If we try to connect to a remote server...
- ▶ **... and we get a fingerprint warning:**
 - ▶ **Do not ignore this!**
 - ▶ We might have a connection to the wrong server (or the domain name might be wrong and resolve to another server)
 - ▶ We might be victim of a man-in-the-middle attack!

Man in the middle?



Bash & Linux CLI

SSH: SFTP

SSH: SFTP



- ▶ We can also use SSH to transfer files through it
- ▶ This protocol is called SFTP
- ▶ It is enabled on most servers that support ssh
- ▶ This is the perfect way to transfer data from / to a remote server
- ▶ **On Linux:**
 - ▶ **We could either use graphical programs for this:**
 - ▶ Usually through the protocol names sftp://, ssh://, fish://
 - ▶ **Or we can also use the program scp:**
 - ▶ `scp -r user@server:folder [destination]`
- ▶ **On Mac / Windows:**
 - ▶ I personally use cyberduck:
 - ▶ <https://cyberduck.io/>
 - ▶ Let's have a look at how this works!

Bash & Linux CLI

Useful utility: screen

Useful utility: screen

- ▶ **The utility screen:**

- ▶ Its a terminal multiplexer
- ▶ For us here:
 - ▶ It allows us to multiplex a single terminal to several processes
 - ▶ It does this by starting a virtual terminal
 - ▶ And will then just forward everything to the processes

- ▶ **We can start a new session:**

- ▶ `screen`

- ▶ **And detach from this session:**

- ▶ `CTRL + A, CTRL + D`

- ▶ **Or list existing sessions:**

- ▶ `screen -l`

- ▶ **And then join an existing session:**

- ▶ `screen -x [session]`

Bash & Linux CLI

Introduction: Setting up a Webserver



Why this chapter

- ▶ In my existing web development courses, students sometimes want to see a full setup of a webserver
- ▶ But this needs quite a bit of Linux knowledge, so it's a bit difficult to provide there
- ▶ **I'm thus providing this in this course:**
 - ▶ However, this chapter might require some web development knowledge
 - ▶ This chapter is only for existing web developers
 - ▶ Feel free to skip it!

Setting up a Webserver

- ▶ In this chapter, we will create a webserver based on:

- ▶ Linux
- ▶ Apache (httpd)
- ▶ MySQL
- ▶ PHP

- ▶ Unfortunately:

- ▶ The setup of this is quite different on Ubuntu / CentOS

- ▶ CentOS:

- ▶ Uses the upstream way to configure Apache
- ▶ upstream = the way it's meant from the original authors (Apache Foundation)

- ▶ Debian / Ubuntu:

- ▶ They have their own management tools on top
- ▶ Those tools greatly simplify the management of Apache

How do we solve it in this chapter?

- ▶ In this chapter, we have 4 different types of lectures:

- ▶ **Theory lectures:**

- ▶ In those lectures, I will explain the concepts and how everything works

- ▶ **CentOS:**

- ▶ After the theory, we will have a look at how we put the theory into practice on CentOS

- ▶ **Ubuntu:**

- ▶ And after that, how the same thing works on Ubuntu

- ▶ **Centos / Ubuntu:**

- ▶ Those lectures apply to both operating systems

- ▶ **Important for you:**

- ▶ Even if you use Ubuntu, I highly recommend you watch the CentOS lectures as well (without following along)
 - ▶ This will give you a better understanding of Apache

Adding a firewall,...

- ▶ In a later chapter, we will be adding a firewall (`firewalld`) to our server
- ▶ If you plan to use this server in production, I more than highly recommend you to also watch this chapter

Bash & Linux CLI

Theory: LAMP

LAMP

- ▶ In this lecture, we will analyze the LAMP setup
- ▶ L: Linux - we already know that
- ▶ But we should have a close look at the rest

LAMP: Apache HTTP Server



► **What is the Apache HTTP Server?**

- ▶ An open-source web server software
- ▶ Developed and maintained by Apache Software Foundation

► **What are the key features?**

- ▶ It's highly configurable, and we can configure it for our needs
- ▶ It can be extended with additional modules
- ▶ Supports various server-side scripting languages
- ▶ SSL and TLS support for encrypted connections
- ▶ It powers a significant percentage of the world's websites



LAMP: MySQL / MariaDB

► MySQL:

- ▶ MySQL is a popular, open-source relational database management system (RDBMS)
- ▶ It is being developed by Oracle nowadays (since 2010)

► Key Features:

- ▶ Supports standard SQL (Structured Query Language)

► The idea with SQL:

- ▶ We declare our schema ahead
(example: one user can have a username, a password and an email)
- ▶ And then we can easily store and query entries with SQL:
 - ▶ `SELECT username, email FROM users`
- ▶ Provides multi-user access to databases

LAMP: MySQL / MariaDB



► **MariaDB:**

- A fork of MySQL, created by original MySQL developers
- A community-developed, open-source relational database management system

► **Key Features:**

- Compatible with MySQL and offers more storage engines
- Includes GIS and JSON features

MySQL or MariaDB?

- ▶ For most applications, the differences between those database systems are minor
- ▶ We can even use a python MySQL connector to connect to a MariaDB server
- ▶ Here in this course, we will focus on MySQL

LAMP: PHP

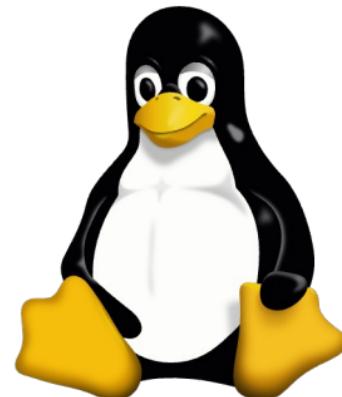


► PHP:

- PHP is an open-source scripting language
- Particularly suited for web development
- It can easily be embedded in HTML

► Key features:

- Gentle learning curve
 - We can easily write bad code
 - But it also has extensive object-oriented capabilities
- => we can also create large applications, and structure our code properly



Bash & Linux CLI

[CentOS]: Installing LAMP

[CentOS]: Installing LAMP

- ▶ On CentOS, we need to install the following tools:

- ▶ **Apache:** httpd

- ▶ **MySQL:**

- ▶ mysql: The MySQL client for the command line
 - ▶ mysql-server: The MySQL server

- ▶ **PHP:** php

- ▶ **So the command we need to execute is:**

- ▶ dnf install httpd mysql mysql-server php

- ▶ **How to launch Apache (httpd):**

- ▶ httpd will install a unit file (`httpd.service`)
 - ▶ But we need to enable and launch it manually:
 - ▶ `systemctl enable --now httpd.service`

- ▶ **We should now be able to access Apache:**

- ▶ <http://localhost/>

Bash & Linux CLI

[Ubuntu]: Installing LAMP

[Ubuntu]: Important

- ▶ If you had installed php through a third-party repository (PPA) in the chapter about package management on Ubuntu:
 - ▶ **Be sure to remove:**
 - ▶ This repository (should be a file in `/etc/apt/sources.list.d`)
 - ▶ And remove all installed PHP from your system:
 - ▶ `apt remove --purge 'php*'`
 - ▶ `apt remove --purge 'libapache2-mod-php*'`

[Ubuntu]: Installing LAMP

- ▶ On Ubuntu, we need to install the following tools:
 - ▶ **Apache:**
 - ▶ The name of this package is: apache2
 - ▶ **MySQL:**
 - ▶ mysql-server: Installs the MySQL server
 - ▶ mysql-client: Allows us to connect to the MySQL server through the command line.
 - ▶ **PHP:**
 - ▶ The name of the package is: php
 - ▶ Also, it might be necessary to also install libapache2-mod-php
 - ▶ **Thus:**
 - ▶ `apt install apache2 mysql-server mysql-client php libapache2-mod-php`

[Ubuntu]: How Apache is launched

- ▶ Apache2 installs a systemd unit file (`apache2.service`)
- ▶ This should be enabled by default

- ▶ **We can inspect this service by checking its status:**
 - ▶ `systemctl status apache2.service`

- ▶ **If it is not enabled, we can do this through systemctl:**
 - ▶ `systemctl enable --now apache2.service`

- ▶ **We should now be able to access Apache:**
 - ▶ <http://localhost/>

Bash & Linux CLI

[Theory]: How does Apache (httpd) work?

How does Apache (httpd) work?

- ▶ Apache HTTP Server (httpd) is a web server that serves HTTP requests from clients

- ▶ **Main operations:**

- ▶ Listens for incoming requests from clients (web browsers)
- ▶ Interprets the request, often mapping it to a file in its directory
- ▶ Sends the requested file or an error message back to the client

- ▶ **Module-based system:**

- ▶ Apache's functionality can be extended using modules
- ▶ Modules can enable features like URL rewriting, PHP support, SSL, etc.

- ▶ **VirtualHosts:**

- ▶ Supports serving multiple websites from a single Apache server
- ▶ Each site appears to have its own domain and configuration

The architecture of apache / httpd

- ▶ In order to serve multiple clients at once, apache / httpd starts several worker processes
- ▶ **Thus, we got:**
 - ▶ A main process to coordinate all of them
 - ▶ And several worker processes that do the actual work of serving the website

How can we configure apache / httpd?

- ▶ **The idea on both distributions:**

- ▶ We have one main configuration file
- ▶ And this one will include all the other required files
- ▶ By splitting up the configuration into multiple files, the configuration remains more clear
- ▶ And additional packages (apt / dnf) can just add additional configuration files

- ▶ **On CentOS, the configuration can be found in several files:**

- ▶ `/etc/httpd/conf/httpd.conf`
- ▶ `/etc/httpd/conf.modules.d/*.conf`
- ▶ `/etc/httpd/conf.d/*.conf`

- ▶ **On Ubuntu, the configuration can be found in several files:**

- ▶ `/etc/apache2/apache2.conf`
- ▶ `/etc/apache2/conf-enabled/*.conf`
- ▶ `/etc/apache2/sites-enabled/*.conf`

Bash & Linux CLI

[CentOS]: How is httpd configured?

Bash & Linux CLI

[CentOS]: Changing / Adding port

Bash & Linux CLI

[Ubuntu]: How is apache2 configured?

Bash & Linux CLI

What is a VirtualHost?

What is a VirtualHost?

► The problem:

- We would like to be able to serve multiple pages from the same apache / httpd instance
- Thus, the webserver needs to detect which website was requested
- And then serve the corresponding website

► The solution:

- We can create a "VirtualHost" for each of them
- And use them to override certain configuration
- Such as the folder from where to serve files
- This allows us to serve multiple domains!

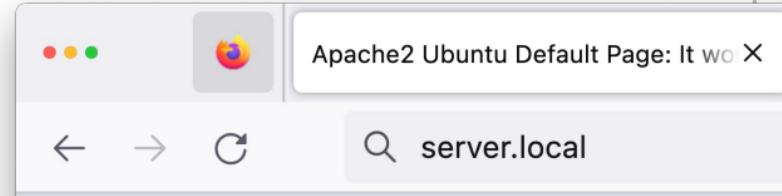
VirtualHost?

- ▶ Let's say Firefox requests the following website:

- ▶ <http://server.local>

- ▶ **In that case:**

- ▶ First, the IP is resolved to the corresponding computer
 - ▶ In this case: server.local is the hostname of a virtual machine in my local network, and we can just connect to it
 - ▶ Firefox will then send an HTTP request to port 80 on this server
 - ▶ The server will see Firefox has requested the site of "server.local"
 - ▶ And if we find a VirtualHost for this website, we can serve a different website
- ▶ Let's have a look at how Firefox tells this to the apache2 / httpd server!



How can we configure a VirtualHost?

- ▶ A VirtualHost will inherit all configuration of the main configuration file

- ▶ But we can override many features

- ▶ **Example of a VirtualHost definition:**

- ▶

```
<VirtualHost *:80>
    ServerName server.local
    ServerAdmin webmaster@localhost
    DocumentRoot /var/www/html/server.local
    ErrorLog /var/log/httpd/server.local.error.log
    CustomLog /var/log/httpd/server.local.log combined
</VirtualHost>
```

- ▶ **Important:**

- ▶ VirtualHost just defines the mapping from a request that is received by apache to an override of the default configuration

Bash & Linux CLI

[CentOS]: Let's create a VirtualHost

Bash & Linux CLI

[Ubuntu]: Let's create a VirtualHost

Bash & Linux CLI

The format of the logfile



Bash & Linux CLI

How can we use PHP?

How can we use PHP with apache?

- ▶ **How can we use PHP with apache?**

- ▶ For many websites, we want to use PHP to execute scripts on our server
- ▶ How can we connect Apache with PHP?
- ▶ Usually, this should happen automatically
- ▶ But it happens differently on CentOS than on Ubuntu!

How to use PHP with apache (CentOS)

- ▶ **On CentOS:**

- ▶ PHP will be configured through `php-fpm` (FastCGI)
- ▶ It's a separate service that is just responsible for running PHP

- ▶ **We can see this:**

- ▶ `systemctl status php-fpm.service`

- ▶ **The advantage:**

- ▶ It integrates better into SELinux (more on that later)
- ▶ It's a separate process, different user / group possible

- ▶ **But:**

- ▶ Additional overhead for the communication between 2 processes



Bash & Linux CLI

How can we use PHP? (Ubuntu)

How to use PHP with apache (Ubuntu)

- ▶ **On Ubuntu:**

- ▶ By default, it will be configured as an Apache module

- ▶ **This means:**

- ▶ PHP is being executed by the Apache worker processes
 - ▶ The performance is slightly better, slightly less overhead



Bash & Linux CLI

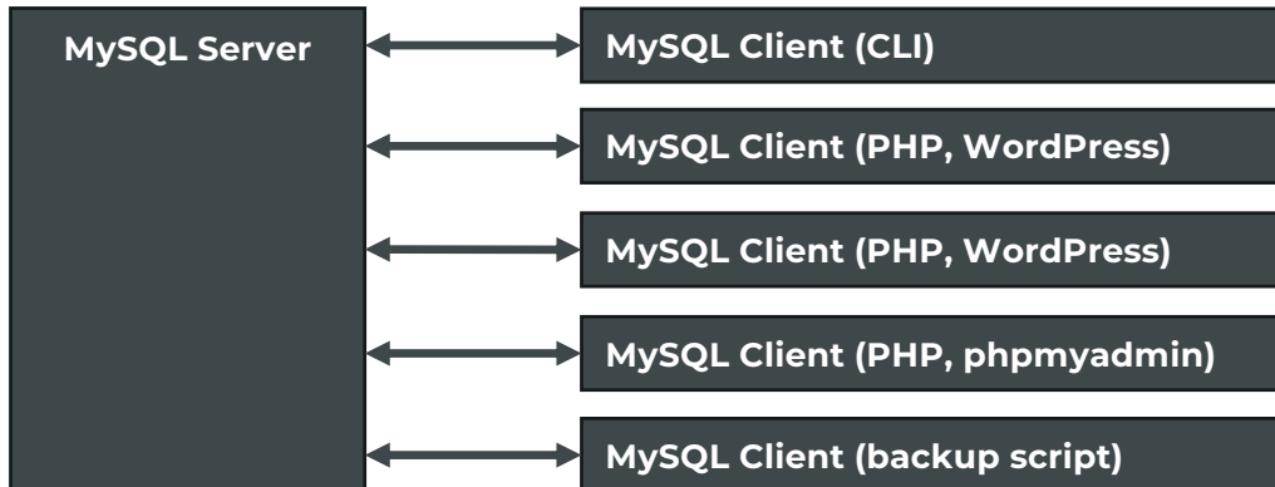
[CentOS, Ubuntu]: Setting up MySQL



MySQL™

MySQL architecture

- ▶ MySQL Server
- ▶ MySQL Client that connects to the MySQL server
- ▶ This allows several clients to share the same database



Setting up MySQL

► The next step is:

- We need to setup our MySQL server
- So that we can run phpmyadmin
- This luckily works (almost) the same on CentOS vs. Ubuntu

► First:

► We need to make sure MySQL is launched:

► CentOS:

- `systemctl status mysqld`
- `systemctl enable --now mysqld`

► Ubuntu:

- `systemctl status mysql`
- Usually, there should be no need for a `systemctl enable`



Bash & Linux CLI

[CentOS, Ubuntu]: Setting up a user

Setting up MySQL

- ▶ We now want to create an admin user for our database
- ▶ We can do this with the following commands:
 - ▶ USE mysql;
 - ▶ CREATE USER 'admin'@'localhost' IDENTIFIED BY 'password';
 - ▶ GRANT ALL PRIVILEGES ON *.* TO 'admin'@'localhost' WITH GRANT OPTION;
 - ▶ FLUSH PRIVILEGES;



Bash & Linux CLI

[CentOS]: Installing phpmyadmin



Bash & Linux CLI

[Ubuntu]: Installing phpmyadmin

Bash & Linux CLI

[CentOS, Ubuntu]: Let's create the DB



Bash & Linux CLI

Installing WordPress

What is WordPress?

- ▶ **WordPress:**

- ▶ A free, open-source content management system (CMS)
- ▶ Primarily written in PHP & MySQL

- ▶ **Key Features:**

- ▶ Highly customizable with themes and plugins
- ▶ User-friendly interface for managing content
- ▶ Supports various media types

- ▶ **Uses:**

- ▶ Widely used for creating blogs, websites, e-commerce platforms, etc.

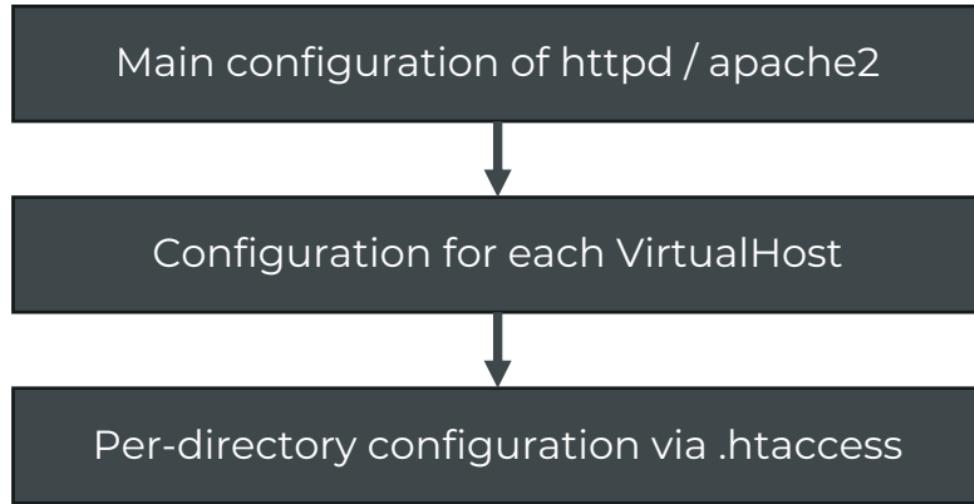
- ▶ **We can download WordPress here:**

- ▶ <https://wordpress.org>

Bash & Linux CLI

Changing configuration via .htaccess

How does configuration work



What is a .htaccess?

- ▶ A .htaccess file allows us to override certain options of our apache configuration for a specific directory (and its sub-folders)
- ▶ **Example .htaccess:**
 - ▶ Deny from all
 - ▶ This will deny all requests to this directory
- ▶ **We might have to configure our webserver to allow .htaccess overrides**
 - ▶ We can do this by adding the following entry to our VirtualHost declaration:
 - ▶

```
<Directory /var/www/html>
    AllowOverride All
</Directory>
```

Bash & Linux CLI

How to password-protect a directory

Password-protected directory

- ▶ For this, we first need to create a password file
- ▶ This file will contain our passwords in an encrypted format
- ▶ **We can do this through the following command:**
 - ▶ `htpasswd [file] [user]`
 - ▶ **Best practices:**
 - ▶ Filename: `.htpasswd`
 - ▶ File is in the same folder as the `.htaccess`
- ▶ **After this, we can create a `.htaccess` with the following content:**
 - ▶

```
AuthType Basic
AuthName "Restricted area"
Require valid-user
AuthUserFile [full-path-to-htpasswd]
```

Bash & Linux CLI

How to access phpmyadmin through an SSH tunnel

SSH Tunnel

- ▶ **SSH Tunnel:**

- ▶ It allows us to redirect a port on our machine and send it through the SSH connection

- ▶ **We can use it the following way:**

- ▶ `ssh -L 8088:localhost:80 ubuntu.local -p 222`
- ▶ This will forward our port 8088 through the SSH connection
- ▶ The destination is "localhost:80" - according to the remote server

Bash & Linux CLI

Firewall: firewalld

Firewall: firewalld

- ▶ **In this chapter:**

- ▶ We will have a look at how we can setup a firewall
- ▶ And how we can allow external incoming traffic

- ▶ **For this, we will have a look at firewalld:**

- ▶ It's architecture
- ▶ How we can use it
- ▶ How we can add additional rules

- ▶ **Important:**

- ▶ This is just a quick introduction, to give you an overview about how everything works
- ▶ If you want to use it in production, be sure to do some further research!

Heads up: SSH

► **If you are connected through SSH:**

- Be extra careful when you change the settings of the firewall
- You might accidentally lock yourself out

► **Best practice for this chapter:**

- If you have changed the SSH port, you might want to change the SSH port to the default port (22) for now
- Otherwise, the default configuration might will lock you out and prevent you from connecting

Bash & Linux CLI

Why / when do we need a firewall?

What is a firewall?

- ▶ A firewall helps us to protect our system
- ▶ It does this by filtering the network traffic
- ▶ **The most important task is packet filtering:**
 - ▶ Analyzes packets against a set of rules
 - ▶ We can then decide what to do with those packets
 - ▶ We could block them or allow them according to our rules
 - ▶ Or forward those through another network interface
- ▶ **There're also other types of firewalls:**
 - ▶ Heuristic firewalls
 - ▶ Deep packet inspection

So why / when should we use a firewall?

- ▶ **Example:**

- ▶ We can block all incoming network traffic
- ▶ Except ssh, because we want to allow a remote computer to still be able to connect to our server
- ▶ Except http, because we want to run a webserver that should be reachable

- ▶ **The result:**

- ▶ Even if another service would be configured in a way to accept incoming connections, we would still be unable to connect from anywhere else
- ▶ This decreases our attack surfaces

Common myths about firewalls

► Does a firewall protect my PC from the internet?

- ▶ No: A firewall just allows us to define custom rules for network traffic
- ▶ Usually, we only define rules for incoming network traffic
- ▶ Aside from minor exceptions (such as Filesharing,...), most applications don't allow incoming traffic anyway, and are not responding to it
- ▶ Also, we're usually behind a NAT without port forwarding (IPv4), or incoming traffic is blocked by our router anyway. Thus, others from the internet can't connect to us anyway

► Can we use a firewall to block all incoming traffic?

- ▶ We shouldn't do this. Certain servers (such as FTP or IRC servers) might try to send packets to us, to confirm our identity (ident protocol, running on TCP port 113)
- ▶ If they don't hear back from us, we might not be able to connect to the remote server at all

Let's have a look at open ports

- ▶ Let's inspect our system and have a look at the opened ports:
 - ▶ IPv4:
 - ▶ sudo ss -4nap
 - ▶ IPv6:
 - ▶ sudo ss -6nap
- ▶ To inspect the parameters, we can use --help:
 - ▶ sudo ss --help

Bash & Linux CLI

Why don't we learn iptables?

Why don't we learn iptables?

iptables

Kernel subsystem:
netfilter

- ▶ **iptables:**

- ▶ User-space program, that allows us to configure the netfilter of the kernel

- ▶ **The problem:**

- ▶ Technically, iptables is not yet deprecated
(as of recording of this course)

- ▶ **But (RHEL):**

- ▶ "*The ipset and iptables-nft packages have been deprecated in RHEL.*

The iptables-nft package contains different tools such as iptables, [...]"

https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/9/html/9.0_release_notes/deprecated_functionality

- ▶ **But (gentoo linux documentation):**

- ▶ "*It should be replaced with its successor nftables*"

<https://wiki.gentoo.org/wiki/Iptables>

Why is iptables in the process of being deprecated?

- ▶ **iptables:**

- ▶ Complex syntax
- ▶ Less performant (each rule needs to be processed separately)
- ▶ Lack of atomic rule set updates
- ▶ Different tools for IPv4 and IPv6 (iptables vs. ip6tables)
- ▶ In the process of being deprecated from Linux distributions

My suggestion

► **I would like to make a bold statement:**

- ▶ If you're not trying to pass a Linux exam that requires knowledge in iptables
- ▶ Or you will work with systems that still use iptables
- ▶ There's no reason to still learn iptables nowadays

Bash & Linux CLI

The architecture of firewalld

The architecture of firewalld

Tools to interact with
firewalld (incomplete)

`firewall-cmd`

`firewall-config`

Daemon and service

`firewalld`

Firewalld backends

`iptables`
(deprecated backend)

`nftables`
(more modern backend)

Linux kernel

Kernel subsystem: `netfilter`

By default: firewalld

- ▶ **By default:**
 - ▶ All incoming TCP and UDP connections will be blocked
(except the ones allowed)
- ▶ **A few services are enabled by default:**
 - ▶ ssh (port 22)
 - ▶ dhcpcv6-client (udp port 546)
- ▶ **Also:**
 - ▶ IPv6 is enabled by default
 - ▶ Certain ICMP requests are blocked to prevent common network attacks
- ▶ **We can have a look at the current configuration of firewalld:**
 - ▶ `firewall-cmd --state`
 - ▶ `firewall-cmd --list-all`

Bash & Linux CLI

[Ubuntu]: Installing firewalld

Ubuntu: How to install firewalld

- ▶ **First, we should ensure that ufw ("uncomplicated firewall") is disabled:**
 - ▶ `ufw disable`
- ▶ **We then need to install firewalld:**
 - ▶ `apt update`
 - ▶ `apt install firewalld`
- ▶ **After this, we should enable the firewall:**
 - ▶ `systemctl enable --now firewalld`
- ▶ **And we can confirm this with the following command:**
 - ▶ `firewall-cmd --state`
 - ▶ `firewall-cmd --list-all`

Bash & Linux CLI

Services in firewalld

Services in firewalld

- ▶ **The idea:**

- ▶ Instead of specifying the port (or multiple ports) manually,
we can just enable a service
- ▶ This allows us to more easily understand what our firewall is doing

- ▶ **List all available services:**

- ▶ `firewall-cmd --get-services`

- ▶ **Get info for a service:**

- ▶ `firewall-cmd --info-service http`

Services in firewalld

- ▶ **Btw, services are all defined as .xml files shipped as part of firewalld:**
 - ▶ We can have a look into the following folders:
 - ▶ /usr/lib/firewalld/services
 - ▶ /etc/firewalld/services
- ▶ **But we should not edit those .xml files directly:**
 - ▶ `firewall-cmd --help`

Bash & Linux CLI

Opening and closing the firewall

Open the firewall

- ▶ **Temporarily (until next reboot):**
 - ▶ `firewall-cmd --add-service=[service-name]`
 - ▶ `firewall-cmd --add-port=[port]/[protocol]`
- ▶ **Example:**
 - ▶ `firewall-cmd --add-service http`
 - ▶ `firewall-cmd --add-port 80/tcp`
- ▶ **Permanently:**
 - ▶ `firewall-cmd --permanent --add-service=[service-name]`
 - ▶ `firewall-cmd --permanent --add-port=[port]/[protocol]`
- ▶ **Follow up with a reload to update the running configuration:**
 - ▶ `firewall-cmd --reload`

Closing ports for a service

- ▶ **Temporarily (until next reboot):**
 - ▶ `firewall-cmd --remove-service=[service-name]`
 - ▶ `firewall-cmd --remove-port=[port]/[protocol]`
- ▶ **Example:**
 - ▶ `firewall-cmd --remove-service http`
 - ▶ `firewall-cmd --remove-port 80/tcp`
- ▶ **Permanently:**
 - ▶ `firewall-cmd --permanent --remove-service=[service-name]`
 - ▶ `firewall-cmd --permanent --remove-port=[port]/[protocol]`
- ▶ **Follow up with a reload to update the running configuration:**
 - ▶ `firewall-cmd --reload`

Bash & Linux CLI

Zones in firewalld

What are zones in firewalld?

► What is the idea?

- ▶ A zone defines the level of trust for network connections or interfaces
- ▶ Network interfaces are assigned to a specific zone
- ▶ Through those, we can give them a certain level of trust
- ▶ An interface can only be in one zone at a time
- ▶ But a zone could be used by multiple interfaces

► Why do we need them?

▶ Example:

- ▶ We're on a laptop, and want to have different firewall rules for Ethernet and WiFi
- ▶ We're a server with 2 network cards, one public, and one for our internal network. We might want to have different firewall rules then

Build-in zones in firewalld

- ▶ **Zone levels:**

- ▶ firewalld provides several predefined zones, ranging from the most trusted (trusted) to the least trusted (drop)

- ▶ **Examples:** trusted, home, work, public, drop

- ▶ **We can list the zones with the following command:**

- ▶ `firewall-cmd --get-zones`

- ▶ **To list the rules of a zone, we can just add --zone:**

- ▶ `firewall-cmd --zone=work --list-all`

- ▶ **The zones are configured through the following folders:**

- ▶ `/usr/lib/firewalld/zones/`

- ▶ `/etc/firewalld/zones/`

- ▶ (we should not edit them manually)

Build-in zones in firewalld

- ▶ **By default, we are usually in the public zone:**
 - ▶ `firewall-cmd --get-default-zone`
- ▶ **But we can change it:**
 - ▶ `firewall-cmd --set-default-zone=work`

Changing the rules for a specific zone

- ▶ **All the ways to enable / disable a port / service can also specify the zone**
 - ▶ If we don't specify the zone parameter, it will use the default zone
 - ▶ **Temporarily (until next reboot):**
 - ▶ `firewall-cmd --zone=[zone] --add-service=[service-name]`
 - ▶ `firewall-cmd --zone=[zone] --add-port=[port]/[protocol]`
 - ▶ **Permanently:**
 - ▶ `firewall-cmd --zone=[zone] --permanent --add-service=[service-name]`
 - ▶ `firewall-cmd --zone=[zone] --permanent --add-port=[port]/[protocol]`

Assigning an interface to a zone

- ▶ **Assign an interface to a zone:**

- ▶ **Temporarily (until next reboot):**

- ▶ `firewall-cmd --zone=zone_name --change-interface=[interface]`

- ▶ **Permanently:**

- ▶ `firewall-cmd --permanent --zone=[zone] --change-interface=[interface]`

- ▶ We might need to reload the current configuration after:

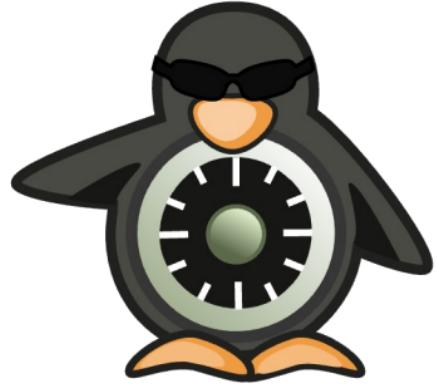
- ▶ `firewall-cmd --reload`

Bash & Linux CLI

Outlook: firewalld

Outlook: firewalld

- ▶ We only had a quick look at firewalld
- ▶ **Firewalld could do way more:**
 - ▶ We can block outgoing connections and filter those as well
(with additional rules)
- ▶ **Masquerading and NAT support:**
 - ▶ We can use firewalld to create our own router and define
NAT forwarding rules
 - ▶ We can forward a port to another port
(even on another machine)
 - ▶ We can assign IP ranges to rules and allow access to a specify
port only to certain IPs / IP ranges



Bash & Linux CLI

SELinux: Security-Enhanced Linux

SELinux: Security-Enhanced Linux

► What is SELinux?

- SELinux is a security module for the Linux kernel
- So far, we only had Discretionary Access Control (DAC)
- SELinux provides Mandatory Access Control (MAC)
- It allows us to enforce access control security policies
- Thereby, it reduces vulnerability to privilege-escalation attacks

► Why SELinux?

- Additional layer of security / access control
- Even if our application is flawed, it can be prevented from causing harm
- It gives us fine-grained control over system resources
- It can even limit the access rights of the root user!
- **In this chapter, you will learn why SELinux is so important and why you should not disable it!**

SELinux: Security-Enhanced Linux

- ▶ **History of SELinux**

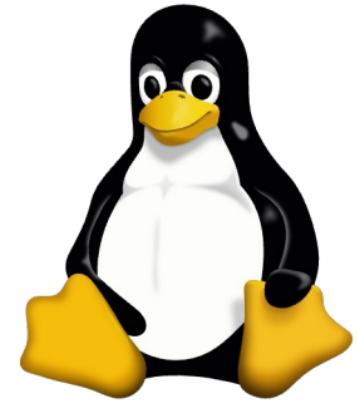
- ▶ Initially developed by the United States National Security Agency (NSA)
- ▶ Made open source for the benefit of the wider community
- ▶ Adoption into Linux kernel as of version 2.6

- ▶ **Included in various major Linux distributions:**

- ▶ Fedora, CentOS, Red Hat Enterprise Linux, AlmaLinux, Rocky Linux
- ▶ Oracle Linux
- ▶ Scientific Linux

- ▶ **But we can also install it on others (such as Ubuntu or Debian)**

- ▶ => **Though this didn't work!**



Bash & Linux CLI

Discretionary access control

Discretionary access control

► What is discretionary access control?

- ▶ It's a security concept for IT systems
- ▶ It allows us to define the access to a resource based on the identity (user / group) of the actor
- ▶ Access rights are defined on a per-user / per-group basis
- ▶ If a process is running as a specific user, and this user has access to certain files... the process can access those files

Discretionary access control

- ▶ We have used discretionary access control already:

- ▶ Files belonged to a user and a group
- ▶ Files had permissions associated to them:

- ▶ `ls -al`

- ▶ **Output:** `-rwxr-xr-x janniss janniss`

- ▶ Also, processes had a user and a group associated to them:

- ▶ `ps -eo pid,user,group,comm`

PID	USER	GROUP	COMMAND
1	root	root	systemd
2	root	root	kthreadd
[...]			
3915	jseemann	jseemann	gnome-terminal-
3936	jseemann	jseemann	bash

Discretionary access control

- ▶ **Our kernel thus knows:**

- ▶ Who a file belongs to (user + group)
- ▶ Who is running a process (user + group)

- ▶ **Then, the kernel can implement access control:**

- ▶ The kernel can deny access to files we are not supposed to access
- ▶ And this allows us to at least somewhat control who can access what on our system



Bash & Linux CLI

The problem with discretionary access control

The problem with discretionary access control

- ▶ **If our application only executes code in a ways we thought about:**
 - ▶ Everything is okay
 - ▶ Discretionary access control is enough
- ▶ **But we should not expect this to be the case:**
 - ▶ There might be bugs in our code
 - ▶ There might be bugs in libraries that we use
- ▶ **Thus, we should assume:**
 - ▶ People will be able to break in, and execute arbitrary code on our system
 - ▶ How can we prevent them even after that?

Example: Running a Webserver

- ▶ **Let's have a look at an example:**

- ▶ Let's say we're running a webserver (nginx)
- ▶ And through an incorrectly placed symlink, an attacker could suddenly access the whole filesystem

- ▶ **The attacker could then access all files that the Unix user / group of the webserver has access to:**

- ▶ Database configuration (if installed)
- ▶ Configuration of the mailserver (if installed)
- ▶ Cronjob configuration (/etc/crontab)
- ▶ Maybe this even involves more sensitive files, such as confidential data / files on external drives / files of our database,...

- ▶ **Let's have a look at this!**

The problem with discretionary access control

- ▶ **Bugs like these happen easily:**

- ▶ **Path or symlink attacks:**

- ▶ Manipulating files or symbolic links to gain unauthorized access or escalate privileges
- ▶ This is what we had seen before

- ▶ **Privilege escalation attacks:**

- ▶ A lower privilege user or process trying to gain higher-level permissions

- ▶ **Buffer overflow attacks:**

- ▶ When an application writes more data to a buffer than it can handle, potentially allowing arbitrary code execution

- ▶ **Directory traversal attacks:**

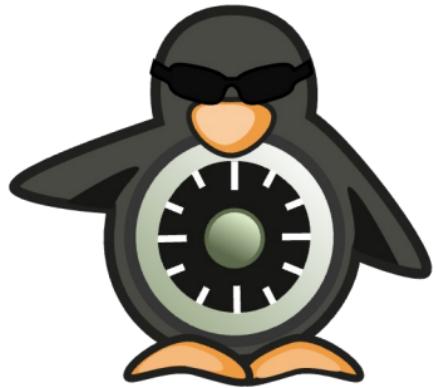
- ▶ Manipulating a web application to access directories that should be off-limits

The problem with discretionary access control

- ▶ **Bugs like these happen easily:**
 - ▶ **Command injection attacks:**
 - ▶ Injecting and executing commands to compromise an application or system.
 - ▶ **Local and remote file inclusion (LFI/RFI):**
 - ▶ Including files from local or remote servers for execution, often used in web application attacks
 - ▶ **Example:**
 - ▶ Unsecured upload form on a PHP website
 - ▶ User can upload their own .php files - which then can be executed by the webserver

The problem

- ▶ We should expect our application to be broken into or to do things we did not wanted it to do
- ▶ **The idea:**
 - ▶ Even in that case, how can we limit the consequences of that?



Bash & Linux CLI

Mandatory access control

Mandatory access control

- ▶ **Mandatory access control:**

- ▶ In addition to the rules from the discretionary access control...
- ▶ ... access is granted based on additional rules and properties

- ▶ **The idea:**

- ▶ MAC rules (policy) is adjusted to our system and its applications
- ▶ It's significantly more fine-grained than discretionary access control

- ▶ **Example:**

- ▶ According to the permissions (discretionary access control),
a program could just access `/etc/crontab`

- ▶ **But, with SELinux:**

- ▶ We can enforce, that this access is still being denied

- ▶ **The result:**

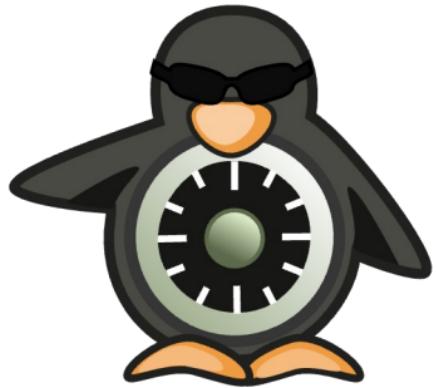
- ▶ A process can only access the files it is supposed to access

Mandatory access control with SELinux

- ▶ So, first the DAC access control rules are being applied
- ▶ **And if those allow access to a resource / file:**
 - ▶ Then, SELinux will apply its mandatory access control policy to determine if access should be granted
- ▶ **Thus:**
 - ▶ MAC with SELinux is not exclusive to DAC
 - ▶ It further restricts DAC and makes our system more secure

[Ubuntu]: Mandatory Access Control

- ▶ Ubuntu uses a different security framework to accomplish similar security to SELinux: AppArmor
- ▶ The goal is to have an easier way to configure it
- ▶ **We can check if AppArmor is enabled:**
 - ▶ `systemctl status apparmor.service`
 - ▶ `aa-enabled`
- ▶ **Important:**
 - ▶ Certain versions of Ubuntu (especially those for embedded systems) might not run AppArmor by default



Bash & Linux CLI

SELinux: Checking and changing the status

SELinux: Checking the status

- ▶ **To check if SELinux is running:**
 - ▶ `getenforce`
- ▶ **To see the current status of SELinux:**
 - ▶ `sestatus`
- ▶ **We can temporarily set the current status of SELinux:**
 - ▶ **`setenforce 0 / setenforce permissive:`**
 - ▶ Changes SELinux to permissive mode
 - ▶ Policy violations are logged, but otherwise ignored
 - ▶ **`setenforce 1 / setenforce enforcing:`**
 - ▶ Changes SELinux to enforcing mode
 - ▶ Policy violations are logged, and prevented

Important

- ▶ **setenforce 0 / setenforce disabled:**

- ▶ If we run into problems (policy violations) due to SELinux, this command would always solve them
- ▶ We should never do this for this reason!
- ▶ This command should only be used for collecting data in order to adjust SELinux permissions and / or our policy (more on that later)

SELinux: Permanently changing status

- ▶ The SELinux configuration can be changed in the file `/etc/selinux/config`

- ▶ **Here, we can set if SELinux should be enabled:**

- ▶ `SELINUX=enforcing`:

- ▶ SELinux should enforce the policy (and log violations)

- ▶ `SELINUX=permissive`:

- ▶ SELinux should only log policy violations

- ▶ `SELINUX=disabled`:

- ▶ SELinux is disabled

- ▶ We should consider if we really want to do this

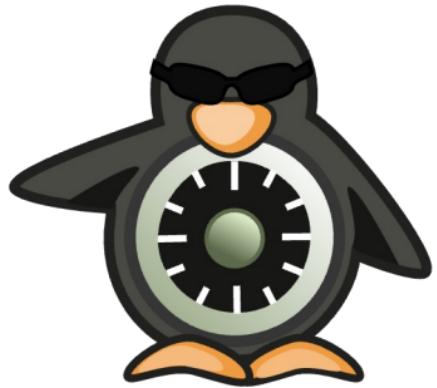
- ▶ **We can also change which policy should apply (`SELINUXTYPE`):**

- ▶ This (and the possible options) might vary between distributions

- ▶ On CentOS, the default policy is `targeted`:

- ▶ SELinux policy is applied to system services

- ▶ But not to normal users



Bash & Linux CLI

File contexts in SELinux

File contexts

- ▶ File contexts are a way of labeling files
- ▶ This will allow SELinux to enforce its policy and to check if access to a certain file / resource is allowed
- ▶ We will first have a look at how the labeling works
- ▶ Then we will explore an example how this labeling protects us
- ▶ And then we will see how we can change the labeling

File contexts in SELinux

- ▶ Files on SELinux systems are labeled
- ▶ Those labels are completely independent from the normal unix file permissions
- ▶ **We can have a look at those labels with the ls command:**

- ▶ `ls -Z`

- ▶ **Example output:**

```
[jseemann@centos ~]$ ls -Z  
unconfined_u:object_r:user_home_t:s0 Desktop  
unconfined_u:object_r:user_home_t:s0 Documents  
unconfined_u:object_r:user_home_t:s0 Downloads  
unconfined_u:object_r:audio_home_t:s0 Music  
unconfined_u:object_r:user_home_t:s0 Pictures  
unconfined_u:object_r:user_home_t:s0 Public  
unconfined_u:object_r:user_home_t:s0 Templates  
unconfined_u:object_r:user_home_t:s0 Videos
```

File contexts in SELinux

- ▶ **What does it mean?**

- ▶ `unconfined_u:object_r:user_home_t:s0 Desktop`

- ▶ **The syntax is as follows:**

- ▶ `[SELinux user]:[SELinux role]:[SELinux type]:[SELinux level]`

- ▶ **Here in this case:**

- ▶ The SELinux user is: `unconfined_u`

- ▶ The SELinux role is: `object_r`

- ▶ The SELinux type is: `user_home_t`

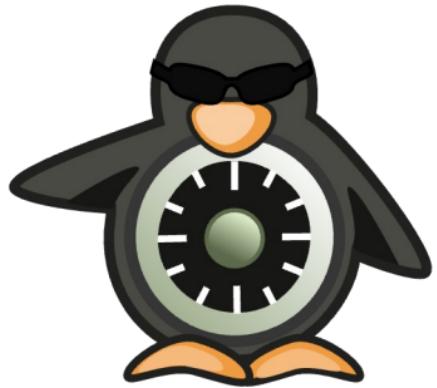
- ▶ And the SELinux level is: `s0`

- ▶ **In SELinux, best-practice is as follows:**

- ▶ All SELinux users end with `_u`, roles end with `_r`, and types end with `_t`

- ▶ Based on the Context, SELinux will decide if access should be granted

- ▶ For us, the type will be the most important part of the SELinux context

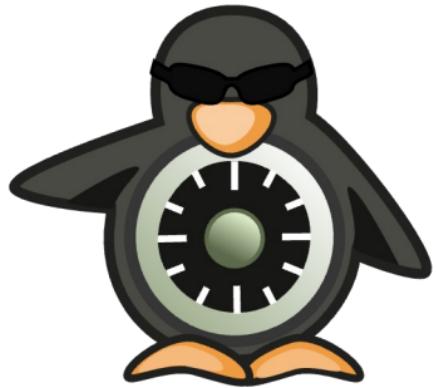


Bash & Linux CLI

How does SELinux protect us? Example: nginx

How does SELinux protect us?

- ▶ The type allows the SELinux policy to allow access to just certain parts of the filesystem
- ▶ **Example:**
 - ▶ A webserver should only be allowed to access `httpd_sys_content_t, httpd_config_t, etc_t` (and a few others)
 - ▶ If it tries to access any file with another type - this access is not allowed and thus prevented (if SELinux is enforcing its policy)



Bash & Linux CLI

Changing the file context

What happens when we create a new file?

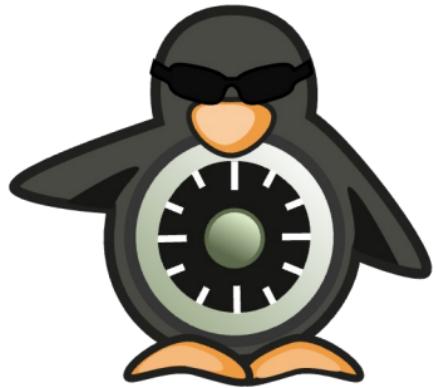
- ▶ **By default:**
 - ▶ If we create a file, it will inherit the SELinux context from its parent directory
 - ▶ Except when the SELinux policy includes a rule that overrides this default behavior (type transition)
 - ▶ Let's have a look at this!

Changing the context (temporarily)

- ▶ We can change the SELinux labels with the chcon command:
 - ▶ chcon [-u USER] [-r ROLE] [-l RANGE] [-t TYPE] [-R] [-v] FILE
 - ▶ -R: Recursive
 - ▶ -v: Verbose (print which files have been changed)
- ▶ Example:
 - ▶ chcon -t user_home_t -R -v folder
 - ▶ This is meant for temporary debugging

Resetting the context

- ▶ If we want to reset the file context to the default, we can use the following command:
 - ▶ `restorecon -R -v folder/file`
 - ▶ This will reset the SELinux type for a given file or folder
- ▶ Important options:
 - ▶ **-R:** Recursive
 - ▶ **-v:** Print all changes
 - ▶ **-F:** By default, only the SELinux type is being reset. By adding this option, it will also reset the SELinux user & role



Bash & Linux CLI

How do default contexts work?

How do default contexts work?

- ▶ SELinux somehow used a default context for many files
- ▶ How does this work?
- ▶ Those defaults are defined in the following folder:
 - ▶ /etc/selinux/[policy]/contexts
 - ▶ We can have a look at those, but we should never edit those files directly
- ▶ It's best to use the following command:
 - ▶ semanage fcontext -l
- ▶ But how can we change the default context?

How do default contexts work?

- ▶ If we want to change the default context:

- ▶ `semanage fcontext -a -t httpd_sys_content_t '/public(/.*)?'`

- ▶ **-a:**

- ▶ We want to add a default context rule (`-d` would remove one)

- ▶ **-t httpd_sys_content_t:**

- ▶ This is the SELinux type that we want to be applied

- ▶ **'/public(/.*)?' :**

- ▶ Absolute path, with regular expression to define which files should be affected

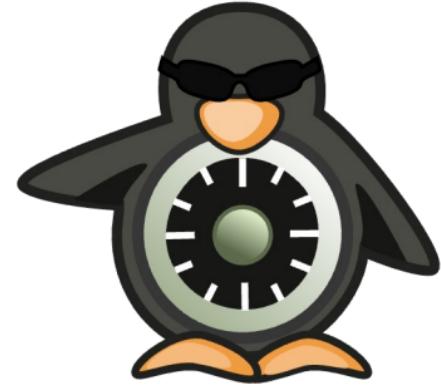
- ▶ Or we create a mapping to an existing context of another folder:

- ▶ `semanage fcontext -a -e /usr/share/nginx/html /public`

- ▶ **-e:** Defines the mapping

Bash & Linux CLI

SELinux: Processes

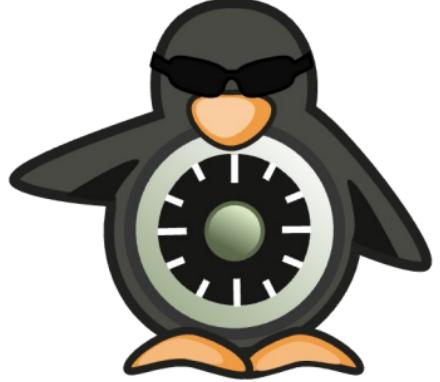


SELinux: Processes

- ▶ All processes on our system run with a specific SELinux security context
 - ▶ **This includes:**
 - ▶ A SELinux user
 - ▶ A SELinux role
 - ▶ A SELinux type
 - ▶ This security context is used to define which files a process can access
 - ▶ This provides an additional layer of security
-
- ▶ **When a process is started:**
 - ▶ **The policy defines the transition into a different SELinux context:**
 - ▶ **Example:**
 - ▶ `systemd` (as SELinux type `init_t`) can start `nginx` as
`system_u:system_r:httpd_t`

SELinux: Processes

- ▶ **How can we see the SELinux security context of a process?**
 - ▶ We could use the System Monitor on a Desktop system to show it
 - ▶ **Or, on the command line, we can use the program ps:**
 - ▶ `ps -Z`
 - ▶ `ps -efZ`
- ▶ **For htop, we need to enable the following column:**
 - ▶ `SECATTR`
 - ▶ Please note that for this, the program `htop` must be installed
 - (and the repository `epel-release` might have to be enabled for CentOS)



Bash & Linux CLI

Bonus: How does the policy look like?

SELinux: How does the policy look like?

- ▶ **How does the policy look like?**

- ▶ **We can have a look at the upstream policy:**

- ▶ Upstream = The version provided by the SELinux project
 - ▶ The actual policy that is running on our machine might be different!
 - ▶ <https://github.com/SELinuxProject/refpolicy>

- ▶ **The file types:**

- ▶ **.fc:**

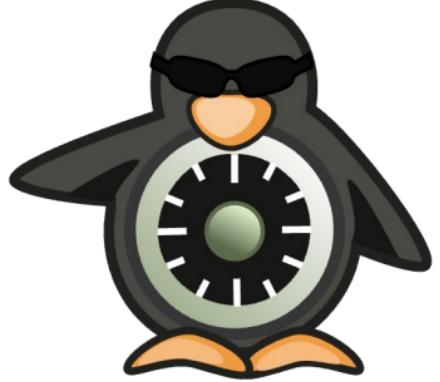
- ▶ File context definitions

- ▶ **.te:**

- ▶ Type enforcement rules, defines transitions

- ▶ **.if:**

- ▶ Contains interfaces / functions that can be used in the .te file



Bash & Linux CLI

SELinux: The targeted policy

SELinux: The targeted policy

- ▶ **By default (targeted policy):**
 - ▶ SELinux is only enforcing its typing policy for system processes
- ▶ **The goal here:**
 - ▶ SELinux should protect our services
 - ▶ But it should not protect us against a compromised user

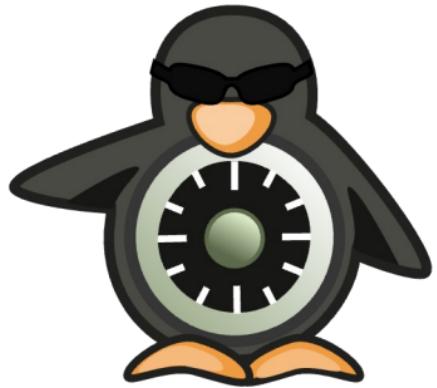
SELinux: Policy

- ▶ **How can we find out our users SELinux domain?**
 - ▶ **We can use the ID program for this:**
 - ▶ `id -Z`
 - ▶ **Or through the semanage command:**
 - ▶ `semanage login -l`
 - ▶ In my case, it's `unconfined_u`
 - ▶ Thus, the SELinux rules allow pretty much everything for our normal user

Consequences of unconfined_u

► Thus:

- Most processes that our interactive user starts will be running as the SELinux user `unconfined_u`
- The SELinux policy can override the SELinux context for certain processes, even if we start them as `unconfined_u`



Bash & Linux CLI

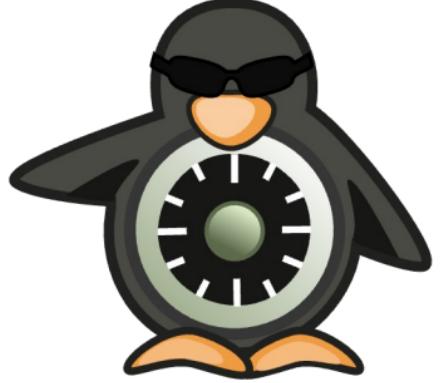
SELinux: Booleans

SELinux: Booleans

- ▶ The SELinux policy needs to be adjusted to our applications needs
- ▶ This it what makes a system with SELinux more secure
- ▶ For adjusting this, we want to have a simple way to configure our system
- ▶ Luckily, we can use SELinux Booleans for this
- ▶ Through them, we can toggle policy rules
- ▶ **To have a look at the available booleans:**
 - ▶ `getsebool -a`
- ▶ **To also get an explanation about them:**
 - ▶ `semanage boolean -l`

SELinux: Enabling / Disabling a boolean

- ▶ If we want to enable a boolean, we can use the `setsebool` command:
 - ▶ Temporarily:
 - ▶ `setsebool httpd_read_user_content on`
 - ▶ Permanently:
 - ▶ `setsebool -P httpd_read_user_content on`



Bash & Linux CLI

SELinux: Auditing policy violations

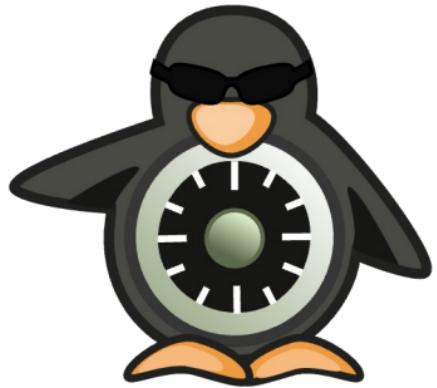
SELinux: Auditing policy violations

- ▶ Whenever a SELinux policy violation happens:
 - ▶ It will (usually) be recorded in the audit log of SELinux:
 - ▶ `/var/log/audit/audit.log`
 - ▶ We can query this log file with the `ausearch` command:
 - ▶ `ausearch -ts recent`
 - ▶ By default, `ausearch` would display all violations of today
 - ▶ But with `-ts recent`, only the last 10 minutes will be shown
 - ▶ Once we found an entry, we can query `journalctl`:
 - ▶ `journalctl -t setroubleshoot --since=14:20`
 - ▶ And then we can run `sealert` as shown in the `journalctl` output to display additional info

Policy violations

► **Usually, this is because:**

- ▶ Incorrectly applied SELinux labels for files or directories
- ▶ We have changed the configuration of the application
- ▶ Confined processes act in a different way than their policy
- ▶ Our application or the policy is faulty



Bash & Linux CLI

SELinux: Managing ports

SELinux: Managing ports

- ▶ If an application wants to listen on a specific port, SELinux must allow this too

- ▶ **We can have a look at the mapping with the following command:**

- ▶ `semanage port -l`

- ▶ **We can add an additional port:**

- ▶ `semanage port -a -t http_port_t -p tcp 8888`

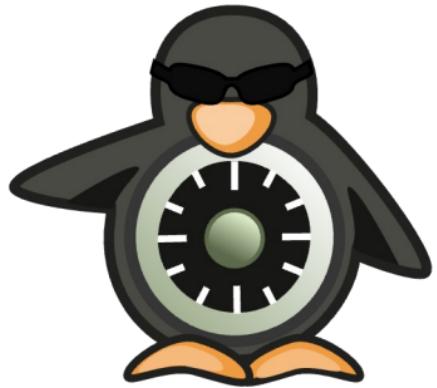
- ▶ **The parameters:**

- ▶ `-a`: We want to add a port

- ▶ `-t http_port_t`: For the type `http_port_t`

- ▶ `-p tcp`: For the protocol TCP

- ▶ `8888`: The port we want to add (8888)



Bash & Linux CLI

SELinux: Disabling during boot

SELinux: Disabling during boot

- ▶ Sometimes, enforcing strict policy rules might prevent our system from booting up
- ▶ **If the GRUB boot menu is enabled:**
 - ▶ We can disable SELinux for a single boot:
 - ▶ `selinux=0`
 - ▶ Or set SELinux to permissive for a single boot:
 - ▶ `selinux=1 enforcing=0`
- ▶ **If GRUB is disabled:**
 - ▶ We would need to change our GRUB configuration
 - ▶ We have had a look at this in the chapter about the boot process and systemd

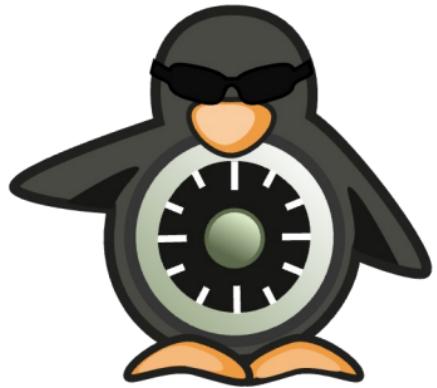
Security implications

► **Of course, this also means:**

- If a potential attacker has direct access to the GRUB menu
- The attacker could just disable SELinux

► **To prevent this:**

- You might want to consider disabling the GRUB menu from showing up
- Also, we might want to disable booting from external drives and protecting our BIOS / UEFI with a password
- This will make recovery more difficult though



Bash & Linux CLI

SELinux: Summary & Outlook

SELinux: Summary

- ▶ You can now work with SELinux, and configure the server properly
- ▶ You can confidently identify problems and adjust the SELinux configuration
- ▶ **Thus:**
 - ▶ Your server / system will be more secure
 - ▶ And SELinux protect it in case your application gets compromised
- ▶ **And:**
 - ▶ You can now leave SELinux enabled!

SELinux: Outlook

► **Writing your own policy:**

- ▶ You can also write your own policy for your application
- ▶ This could make sense if you want to deploy a large-scale application
- ▶ This is also not part of this introduction
- ▶ We can also let our application run and turn this into a policy:
 - ▶ `audit2allow`

► **Confining users:**

- ▶ Through this, we can even limit what interactive users can do
- ▶ This is not part of this introduction

► **Multi-Level Security (MLS) for files:**

- ▶ We could use it to implement security levels, even for users:
- ▶ Unclassified (s0), classified (s1), secret (s2), top secret (s3)
- ▶ Users in secret could then not access top secret
- ▶ And we could disallow modifying of classified and unclassified files
- ▶ MLS is not recommended for systems that run X window system

Bash & Linux CLI

Linux distributions

Why do we need so many?



Why do we need so many?

- ▶ **Why do we need so many distributions?**

- ▶ They usually cater to a slightly different audience
- ▶ Their contribution model is different, some are community driven, others are driven by a company
- ▶ For some, we can buy additional support packages
- ▶ They have different update schedules / update durations

- ▶ **But they all have similarities:**

- ▶ We have a shell available that allows us to work with the system
- ▶ We usually have tools to facilitate package management
(such as `apt`, `dnf`, `emerge`, `pacman`...)
- ▶ Most of them use `systemd` for the system startup
- ▶ Many other tools are shared as well

In this chapter

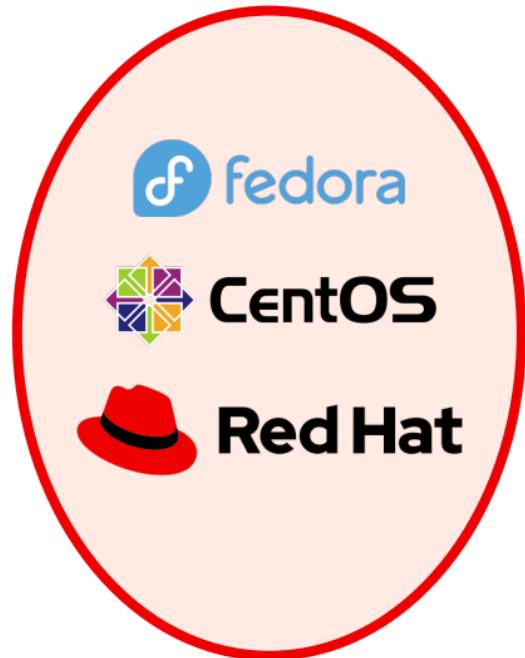
- ▶ We want to gain an overview over common Linux distributions
- ▶ And explore their advantages / disadvantages, target audience,...
- ▶ Also, this chapter can give you ideas if you want to explore additional distributions
- ▶ **Important:**
 - ▶ I've tried to be as accurate as possible
 - ▶ This topic can be highly subjective
 - ▶ I'll try my best to be as objective as possible

Bash & Linux CLI

The Red Hat family

Red Hat Family

- ▶ The Red Hat family refers to distributions derived and / or influenced by Red Hat Enterprise Linux
- ▶ **They all share certain features:**
 - ▶ **RPM Package Management:**
 - ▶ Utilizes the RPM (Red Hat Package Manager) for software distribution and management
 - ▶ **SELinux:**
 - ▶ Security Enhanced Linux for advanced system security
 - ▶ **Extensive Documentation:**
 - ▶ Comprehensive and well-maintained documentation for system administrators and users
 - ▶ **Ecosystem:**
 - ▶ A rich ecosystem of software, tools, and utilities optimized for or compatible with the Red Hat family
 - ▶ **But what are the differences?**



Fedora



► **Cutting-Edge Technology:**

- ▶ Often includes the latest features and technologies
- ▶ This makes it appealing to developers and tech enthusiasts

▶ **However, because of this:**

- ▶ Not as stable as RHEL / CentOS Stream

► **Relationship with Red Hat:**

- ▶ Community-driven project, but direct initiative from Red Hat
- ▶ Fedora serves as a testing ground for new features that might be included in RHEL in the future



CentOS Stream

- ▶ **Cutting-Edge Technology:**

- ▶ Bridges the gap between the bleeding-edge Fedora and the stalwart RHEL
- ▶ Offers a rolling preview of what the next minor RHEL release will look like

- ▶ **As a result:**

- ▶ More stable than Fedora
- ▶ But not as stable as RHEL's major releases

- ▶ **Relationship with Red Hat:**

- ▶ A direct initiative from Red Hat
- ▶ CentOS Stream provides insights into the next version of RHEL, allowing the community and Red Hat to shape it together

Red Hat Enterprise Linux (RHEL)



Red Hat

- ▶ **Enterprise Focus:**

- ▶ Targeted at the business environment
- ▶ Red Hat offers support and certifications
- ▶ This is often required by organizations

- ▶ **Subscription Model:**

- ▶ Source code is freely available (thanks to GPL license)
- ▶ But we still need to pay for a subscription to get access to the software

Bash & Linux CLI

Extended Red Hat family: Alma Linux, Oracle Linux,...

Extended Red Hat family

- ▶ Previously, CentOS was downstream of Red Hat
- ▶ This is no longer the case
- ▶ **Luckily, other distributions fill this gap:**
 - ▶ Rocky Linux
 - ▶ Alma Linux
 - ▶ Oracle Linux
- ▶ **What do they do?**
 - ▶ They try to maintain 100% binary compatibility with RHEL
 - ▶ This means they take the source code of RHEL, remove Red Hat related code, and ship it as their own
 - ▶ This is normal and allowed, according to the GPL License



Bash & Linux CLI

Debian & Debian family

The Debian family

- ▶ **Community-Driven:**

- ▶ Entirely community-driven without a dominant commercial entity

- ▶ **Release When Ready:**

- ▶ Debian's release philosophy is "when it's ready", ensuring high-quality releases

- ▶ **DPKG & APT:**

- ▶ Uses the dpkg package management system, with the apt frontend for user-friendly package operations

- ▶ **Derived Distributions:**

- ▶ Serves as a base for many other distributions, including Ubuntu and its many derivatives

- ▶ **Universal Operating System:**

- ▶ Aims to be the "universal operating system", suitable for various applications from desktops to servers

The branches of Debian

Experimental

- Highly unstable packages and concepts
- Used for early testing

Unstable ("sid")

- Active development branch with the latest software
- Not recommended for critical use

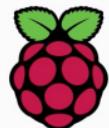
Testing

- Pre-release packages bound for the next stable version
- More updated but might have minor bugs

Stable

- Production-ready
- Thoroughly tested
- Older but reliable software.

The Debian family



Raspberry Pi OS



Ubuntu



Bash & Linux CLI

Ubuntu

Ubuntu

► **Operating System:**

- ▶ Ubuntu is a popular, open-source Linux operating system based on Debian
- ▶ Maintained by Canonical Inc.

► **User-Friendly:**

- ▶ Known for its ease of use and beginner-friendly approach

► **Regular Releases:**

- ▶ Every 6 months

► **Support:**

- ▶ Supported for nine months
- ▶ Users are encouraged to update

► **Audience:**

- ▶ Suitable for those wanting the newest software
- ▶ Users should be comfortable with more frequent updates

Ubuntu LTS (Long Term Support)

- ▶ **Stable Version:**

- ▶ Meant for users who prefer stability

- ▶ **Release Cycle:**

- ▶ Occurs every two years
 - ▶ Provides a more thoroughly tested & stable platform

- ▶ **Support Period:**

- ▶ Supported for five years with security updates / bug fixes

- ▶ **Extended Support:**

- ▶ Support can be extended even more (paid)

- ▶ **Audience:**

- ▶ Often preferred by businesses, institutions
 - ▶ Or users who value stability over the very latest features

Ubuntu variants (flavors)



► **Ubuntu Server:**

- ▶ No GUI by default -> command line only
- ▶ Intended for use on servers in data centers, cloud computing, or hosting web services

► **Kubuntu**

- ▶ Uses K Desktop Environment (KDE)

► **Xubuntu**

- ▶ Uses the XFCE Desktop Environment

► **Lubuntu**

- ▶ Uses the LXQT Desktop
- ▶ For older computers and devices with limited resources

► **Important:**

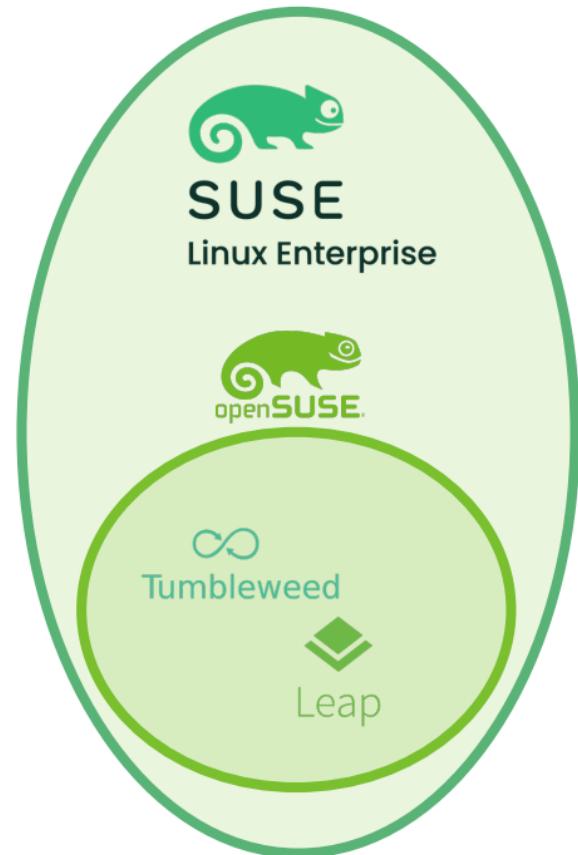
- ▶ They're all the same Linux distribution
- ▶ Just a different selection of default packages!

Bash & Linux CLI

SUSE Linux

SUSE family

- ▶ **SUSE Linux Enterprise (SLE)**
 - ▶ Aimed at businesses and enterprise environments
 - ▶ Offers *long-term support, stability, and certified compatibility*
 - ▶ Commercially supported with a *subscription model*
- ▶ **openSUSE**
 - ▶ openSUSE is a community-driven project sponsored by SUSE
 - ▶ Provides a free and open-source platform
 - ▶ Acts as a testing ground for new ideas
- ▶ **openSUSE Leap:**
 - ▶ Geared towards users who want more stability
- ▶ **openSUSE Tumbleweed:**
 - ▶ Rolling release version of openSUSE, suitable for users who want the latest software and updates





Bash & Linux CLI

Arch Linux

Arch Linux



► **Rolling release model:**

- Always providing the latest software versions
- This leads to continuous updates

► **Not made for production use**

► **DIY philosophy:**

- Arch allows you to build your system from the ground up
- This makes it highly customizable
- But more complex to set up

► **Software management:**

- Utilizes the `pacman` package manager
- Allows simple install of user made packages

Bash & Linux CLI

Gentoo Linux

Gentoo



- ▶ **Highly customizable:**

- ▶ Gentoo is renowned for its flexibility
- ▶ It's highly customizable for a user's particular needs and hardware

- ▶ **Portage package management:**

- ▶ Uses Portage (`emerge`), a unique package management system
- ▶ Software is compiled from source code
- ▶ This allows users to get the maximum performance for a specific system

- ▶ **Learning curve:**

- ▶ Many different control and customization options
- ▶ Steeper learning curve
- ▶ More suitable for experienced users

Bash & Linux CLI

How to choose a Linux distribution?

How to choose a Linux distribution

- ▶ **First:**

- ▶ **Write down the requirements:**

- ▶ Do you need commercial support?
 - ▶ For how many years?
 - ▶ Are you willing to pay for a license?
 - ▶ Do you need SELinux?
 - ▶ Do you use specific software that requires a specific distribution?
 - ▶ Are there any additional requirements?

- ▶ **After this, you can evaluate:**

- ▶ You can try to run the same program on two different distributions
 - ▶ Most of them will feel quite similar
 - ▶ But maybe you like one more?
 - ▶ What do your colleagues think?

Bash & Linux CLI

Part 4: Bash scripting

Part 4: Bash scripting

► You now know:

- ▶ How to master the command line
- ▶ How to manage a Linux system

► In the next few chapters:

- ▶ We will further enhance your CLI knowledge
- ▶ You will learn how to write Bash scripts
- ▶ This will allow you to automate common tasks, write backup scripts,...

Part 4: Bash scripting

► What will you learn:

- ▶ How to create a Bash script
- ▶ Variables in Bash
- ▶ How to access a JSON API from Bash
- ▶ How to test with if conditions
- ▶ How to run parts of your code multiple times (for, while,...)
- ▶ How to create fancy dialogs
- ▶ How to use functions in Bash scripts
- ▶ How to utilize Arrays to store multiple values in a variable

Bash & Linux CLI

Bash scripting

Reminder: Bash



- ▶ Bash: **bourne again shell**
 - ▶ command-line shell and scripting language on Unix-like systems
- ▶ So far, we have used it as a command-line shell (CLI)
- ▶ **But now, we want to start using Bash as a scripting language:**
 - ▶ Automation of repetitive tasks (e.g., creating backups, monitoring servers, evaluating log data,...)
 - ▶ Instead of using additional programming languages (like Python)
- ▶ **In this course:**
 - ▶ You will learn to write Bash scripts
 - ▶ They are great for scripts of no longer than 100 lines
- ▶ **Why not longer?**
 - ▶ => Bash is not recommended for longer scripts
("Shell Style guide" by Google)

Bash scripting

► In this chapter, we will investigate:

- ▶ How do we create a Bash script?
- ▶ What is `#!/usr/bin/bash`?
- ▶ What are comments?
- ▶ How can we declare Bash variables?
- ▶ What's the difference between a Bash variable and an environment variable?
- ▶ How can we modify strings?

Bash & Linux CLI

Our first Bash script



Our first Bash script

- ▶ A Bash script is plain **text file** containing a sequence of commands
- ▶ We can create it with any text editor
- ▶ **We can then execute the script by invoking bash:**
 - ▶ `bash script.sh`
- ▶ **Best practices:**
 - ▶ File extension is not necessary
 - ▶ If we use a file extension, we should use something meaningful:
 - ▶ `.sh`, `.bash`
 - ▶ Do not use a name that also exists as a command name
 - ▶ Name should only contain alphanumeric characters and underscore:
 - ▶ `A-Z`, `a-z`, `0-9`, `_`
 - ▶ We should avoid any other characters

Bash & Linux CLI

Bash script with shebang

Bash script with shebang

- ▶ Bash scripts should begin with a **shebang** at the top, specifying the **interpreter** for executing the script
 - ▶ An interpreter reads a script line by line, interprets each command, and executes them in sequence
 - ▶ `# (sharp) + ! (bang) = shebang`
 - ▶ **This is a Unix feature:** The statement after `#!` is executed as an interpreter for the script
- ▶ **Example:**
 - ▶ `#!/usr/bin/env bash`
 - `echo 'hello from a bash script'`
- ▶ **Bash scripts can then be executed directly:**
 - ▶ `./script`
 - ▶ They need to be executable for this: `chmod +x ./script`



Which shebang to use?

- ▶ Quite often, you might also see different shebangs:

- ▶ Example: `#!/bin/bash`
- ▶ This would also be completely valid

- ▶ But in our course, we will be using the following:

- ▶ `#!/usr/bin/env bash`

- ▶ The reason:

- ▶ On macOS, the system version of Bash might be outdated
(due to licensing issues)
- ▶ Thus, on macOS, we might have to install a newer version of Bash
- ▶ But we can't override the system version of Bash (system integrity protection)
- ▶ `#!/usr/bin/env bash` picks the first one found by our PATH variable in our environment
- ▶ => Thus on macOS, the more recent version is picked

Bash & Linux CLI

Bash and macOS

Bash & Linux CLI

Bash scripting: Comments

Comments

- ▶ Comments are ignored during execution
- ▶ **They are used to:**
 - ▶ provide explanations
 - ▶ document code, or
 - ▶ temporarily disable certain parts of the script
- ▶ Comments in Bash start with a single #
- ▶ **Sometimes, you might see comments with :**
 - ▶ : I am a comment
 - ▶ Technically, : stands for NOP, meaning that we execute an empty command
 - ▶ Expansions are still being applied
 - ▶ This is not a real comment, and should not be used as such

Multi-line comments

- ▶ **Sometimes, you might see a multi-line comment as such:**

- ▶ : << 'my_comment'

- This here is the content of the comment. Hello world!

- my_comment

- ▶ This is also not a comment

- ▶ Here, we define the contents of a temporary file as a heredoc
(and we named it my_comment)

- ▶ And send this (as stdin) to the NOP command (:)

- ▶ The single quotes around my_comment define that we want to
disable expansions

- ▶ **But this is still not a comment**

- ▶ It's still an empty command that is being executed

- ▶ **We should avoid using this!**

Bash & Linux CLI

Bash scripting: Variables

Bash variables vs. environment variables

- ▶ We have already looked at environment variables
- ▶ In this lecture, we will have a look at Bash variables
- ▶ They work slightly differently depending on how we use them
- ▶ Thus, you should consider those as 2 different types of variables

Bash variables

► Bash variables are used to store and manipulate data:

- They are valid within a shell session
- Variables can be defined and used directly in the shell session
- Data is then stored temporarily within the current shell session

► Syntax: variable_name=value for declaration and assignment

► No spaces around the = operator!

- Variables are case-sensitive
(myVar and myvar are different variables)

► Important:

- There are no data types, all variables in Bash are strings

► To access a variable:

- We can use variable expansion:

```
var='content'  
echo "${var}"
```



variable_name

Bash variables

► Best practice:

- ▶ Avoid using special characters or reserved keywords
- ▶ Try to use Bash variables, unless you need to use an environment variable
- ▶ You can simplify complex commands, by using a variable to turn them into multiple, simpler commands
- ▶ Use lowercase characters, and separate words by an underscore:
 - ▶ `my_var`

Bash & Linux CLI

Bash variables vs. environment variables

Bash variables vs. environment variables

► **Environment variables:**

- ▶ A feature of the operating system
- ▶ They work independently of the program / programming language
- ▶ They are automatically inherited to child processes

► **Bash variables:**

- ▶ A feature of Bash
- ▶ They work within a single Bash context
- ▶ They are not inherited to child processes

Best practice

- ▶ Use Bash variables whenever possible
- ▶ Use environment variables when you want those to be inherited to child processes

Bash & Linux CLI

How to (not) define a variable

How to not declare a variable

- ▶ **`variable_name= value`**

- ▶ We create an temporary environment variable `variable_name` and set it to ''
- ▶ After that, we execute `value` with this environment variable

- ▶ **`variable_name =value`**

- ▶ Bash will execute `variable_name` as a command with the parameter `=value`
- ▶ Most likely, the command `variable_name` will not exist, thus yielding an error

- ▶ **`variable_name = value`**

- ▶ Bash will execute `variable_name` as a command with 2 parameters: `=` and `value`

Listing and declaring variables: declare

The command declare

- ▶ We can also create variables with declare:

- ▶ Syntax:

- ▶ `declare [options] variable_name=value`

- ▶ Options (examples):

- ▶ `-r`: declares a read-only variable that cannot be modified

- ▶ `-i`: Declares an integer variable

- ▶ `-x`: Declares an environment variable

- ▶ Example:

- ▶ `declare -r course_title='Linux'`

- ▶ This variable is now read-only, and cannot be changed

- ▶ We can also use declare to display all variables:

- ▶ `declare -p`

Bash & Linux CLI

Deleting / unsetting variables

Deleting / unsetting variables

- ▶ **Unsetting variables:**

- ▶ **Most often:**

- ▶ We never unset a variable
 - ▶ Bash scripts are not that long anyway
 - ▶ And once finished, our whole Bash process is removed from memory anyway

- ▶ **Method 1:** We just assign an empty value to a variable:

- ▶ `course_title=`
 - ▶ But this does not remove the variable
 - ▶ The variable still exists (it's just set to the empty string)

- ▶ **To really remove a variable:**

- ▶ `unset course_title`

Bash & Linux CLI

Accepting user input: read

Reading user input

- ▶ Use the `read` command to prompt the user for input and store it in a variable:
 - ▶ `read -p 'Enter your name: ' name`
- ▶ **We can also explore additional options:**
 - ▶ `read --help`
- ▶ **Important:**
 - ▶ If you're using a different shell (such as `zsh`), the `read` command will have different options
 - ▶ `read -p` might not even work then!

Bash & Linux CLI

Advanced read usage (IFS)

Using read for multiple variables

- ▶ **We can also read multiple values with just one read command:**
 - ▶ `read variable1 variable2`
- ▶ **Here, the input will be split into 2 segments:**
 - ▶ The first word will be written into the variable `variable1`
 - ▶ All the remaining words will be written into `variable2`
- ▶ **Let's have a look at this:**
 - ▶ `read -p 'Please enter your full name: ' firstname lastname`
- ▶ **But how does it work?**

Advanced read usage: IFS

- ▶ In Bash, the variable **IFS** controls how whitespace is being handled by certain commands
- ▶ The **IFS** variable is a Bash variable
- ▶ **By default, it contains the following characters:**
 - ▶ space, tab, newline
 - ▶ **IFS=\$' \t\n'**
 - ▶ Please note the **\$** at the beginning - only this will convert **\t** into a real tab!
- ▶ The input that we provide to the **read** command will be split along those characters (though the newline character still terminates the **read** command)
- ▶ **We can overwrite this variable though!**
 - ▶ **IFS=";" read firstname lastname**
 - ▶ We can even combine this with process substitution
 - ▶ **Let's have a look at this!**

Bash & Linux CLI

Reading files

File contents to variable

- ▶ **We can already read the contents of a file to a variable:**

- ▶ `file_content=$(cat file.txt)"`
- ▶ Here, we're using command substitution to execute the cat program
- ▶ Then, we collect its output and store it in the variable `file_content`

- ▶ **But there's a more efficient way to do it!**

- ▶ `file_content="$(< file.txt)"`
- ▶ < would normally be used for input redirection
(= using a file as standard input for a program)
- ▶ But in combination of command substitution and no program there,
< means that we want to read the contents of the file

Bash & Linux CLI

Bash script: Let's create a Bash script!

Let's create a Bash script

- ▶ **We now want to create a more complex Bash script:**
 - ▶ It should first ask us about our name
 - ▶ It should then greet us, and our name should be printed in green color (`tput`)
 - ▶ Then, it should list the number of files on our Desktop
- ▶ **It should then print out a horizontal line of dashes:**
 - ▶ -----

Bash & Linux CLI

Exercise: System monitor script

Exercise

- ▶ **Exercise: Create a system info script!**

- ▶ **So far:**

- ▶ We're still a bit limited in what we can do
- ▶ We don't have conditionals (if, else,...) - but we'll get to that soon ☺

- ▶ **Ideas for the script. It could...**

- ▶ ...print the CPU usage of the last few minutes (`uptime`)
- ▶ ...print the processes that take the most % CPU (`ps`):
- ▶ ...print the % of free disk space of the main drive (`df`)
- ▶ ...print how much memory is available on our system:
 - ▶ **Idea:** We can just print out a single line from the file `/proc/meminfo`
- ▶ ...print out if the webserver is running or not:
 - ▶ **Idea:** You can pipe the output of `systemctl status [unit]` into a grep
- ▶ ...ping a remote webserver a few times, and print out the whole result

Bash & Linux CLI

Bash scripting (part 2)

Bash scripting (part 2)

- ▶ **In this chapter, we'll have a more in depth look at Bash scripting:**

- ▶ We will have a look at the difference between:
 - ▶ `ls | wc -l`
 - ▶ `wc -l < <(ls)`
- ▶ We will split a script into multiple files (source)
- ▶ Also, we will make our scripts accessible through PATH
- ▶ We will have a look at the best practices for writing Bash scripts
- ▶ We will see how we can prevent potential errors through Shellcheck
- ▶ And we will setup an editor (Visual Studio Code), so we can write scripts more easily

Bash & Linux CLI

Bash: Pipe vs. process substitution

Remainder: Process substitution

- ▶ We had seen before, that process substitution can replace a pipe
 - ▶ **This is a normal pipe:**
 - ▶ `ls | wc -l`
 - ▶ The output of the `ls` command is being used as the standard input of the `wc -l` command
 - ▶ **This is process substitution <(command):**
 - ▶ `wc -l < <(ls)`
 - ▶ The command `ls` will be executed, it's result will be written to a temporary file
 - ▶ And this temporary file will be used as the input for the redirection (which will then load the file into the `stdin` of `wc -l`)
 - ▶ What's the difference between those commands?

Pipe vs. process substitution

- ▶ **Pipe:**

- ▶ `ls | wc -l`
- ▶ `ls` is executed in the main shell
- ▶ `wc -l` is executed in a subshell

- ▶ **Process substitution:**

- ▶ `wc -l < <(ls)`
- ▶ `wc -l` is executed in the main shell
- ▶ And `ls` is executed in a subshell

- ▶ **Here, it doesn't make any difference**

- ▶ **But what if the commands had side effects?**

Pipe vs. process substitution

- ▶ **Pipe:**

- ▶ `echo "Linux" | read -r topic`
- ▶ `read -r topic` is executed in a subshell
- ▶ Thus, the variable exists only in the subshell
- ▶ If we try to access this variable after, it will not exist in the main process

- ▶ **Process substitution:**

- ▶ `read -r topic < <(echo "Linux")`
- ▶ `read -r topic` is executed in the main shell
- ▶ Thus, this variable exists after the command!

Bash & Linux CLI

How to split Bash script into multiple files

How to split a Bash script into multiple files

- ▶ If we want to split a complex script into multiple files:
 - ▶ We can just launch the other script in a subshell:
 - ▶ bash other-script.sh
 - ▶ ./other-script
 - ▶ But this will launch the other script in a subshell
 - ▶ Meaning: We have a different scope for our variables
 - ▶ Only environment variables are inherited from the parent to the other script
 - ▶ We can also execute a Bash script in the same context as our current script:
 - ▶ source [file]
 - ▶ source ./other-script.sh
 - ▶ Let's have a look at this ☺

Bash & Linux CLI

Making our scripts accessible through PATH

Making our scripts accessible through PATH

- ▶ A Bash script is just an executable file
- ▶ If our PATH includes a folder with executable files (no matter of what type), those files can be accessed directly
- ▶ **For this, we can update our PATH variable:**
 - ▶ `PATH="${PATH}: [new-path]"`
- ▶ **Best practice:**
 - ▶ If a script should be accessed through PATH
- ▶ **Bash script must:**
 - ▶ Be executable (`chmod +x`)
 - ▶ Have a shebang (`#!/usr/bin/env bash`)
- ▶ **Bash script should:**
 - ▶ Not have an extension (`my_script` instead of `my_script.sh`)

Bash & Linux CLI

Relative & absolute paths in shell scripts

Relative paths in shell scripts

► **Important:**

- If you're using relative paths in a shell script
- They always refer to the current working directory

► **Meaning:**

- It does not matter where the executable file is stored
- Commands such as these will always refer to the current working directory (`pwd`):
 - `touch ./file.txt`
 - `cat ./file.txt`

► **One solution:**

- We can always try to use absolute paths
- Those should then always work

► **But can we also use relative paths that start at the path of our Bash script?**

Relative paths in shell scripts

- ▶ If we want to get the path to the executable shell script, we can use the following line:

- ▶ `SCRIPT_DIRECTORY=$(dirname -- "${BASH_SOURCE[0]}")`

- ▶ What happens here?

- ▶ Let's start inside: `${BASH_SOURCE[0]}` :

- ▶ This variable is an array (collection of multiple entries, more on those later)

- ▶ From this, we're accessing the first element: `[0]`

- ▶ It contains the full file paths to the code that we're executing

- ▶ This collection can have multiple entries if we'd be using functions
(more on those later)

- ▶ But the first entry (`[0]`) is always the path to the code that we're currently executing

- ▶ We're then using this as an argument to the `dirname` program:

- ▶ `dirname -- "${BASH_SOURCE[0]}"`

- ▶ The double `--` are just used to indicate that no matter what comes after - it will always be a path (and not an argument such as `-o`, `-f`,...)

Bash & Linux CLI

Best practices for shell scripts

Best practices for shell scripts

- ▶ There're different style guides available for Bash scripts
- ▶ **A style guide:**
 - ▶ A collection of rules and best-practices to follow
 - ▶ This allows multiple team members to have a common code style, and follow common rules
 - ▶ This makes collaboration significantly easier
- ▶ **I would like to especially mention the following guide:**
 - ▶ Shell Style Guide by Google
 - ▶ <https://google.github.io/styleguide/shellguide.html>
 - ▶ I personally try to follow this guide by 90%+

Unix best practices

- ▶ **Also, we should follow Unix best practices, such as:**

- ▶ We should print out a help menu when provided with the parameter "-h"
 - ▶ (more on how to accept parameters later in this course)
- ▶ We should use stdin and stdout to facilitate communication with other programs
- ▶ The program should only do one thing and do it well (single purpose programs)
- ▶ Choose portability over efficiency: We should try to avoid platform-specific code

Bash & Linux CLI

Finding and preventing bugs: Shellcheck

Finding and preventing bugs

- ▶ Shellcheck checks and analyzes your shell scripts
- ▶ And gives you tips how to prevent bugs
- ▶ **Example:**
 - ▶ **Syntax validation:** Detects and alerts us for syntax errors
 - ▶ **Code safety:** Finds common pitfalls, and reduces the chance of unexpected behavior
 - ▶ **Style recommendations:** It helps us to adhere to coding standards
 - ▶ **In general:** It helps us to optimize our script
- ▶ **Usually, we must install this tool first:**
 - ▶ **Ubuntu:** apt install shellcheck
 - ▶ **CentOS:** dnf install shellcheck (EPEL must be enabled)
 - ▶ **Mac:** brew install shellcheck
 - ▶ (Homebrew must be installed: <https://brew.sh>)



Bash & Linux CLI

Setting up Visual Studio Code

Setting up Visual Studio Code

- ▶ **Visual Studio Code:**

- ▶ A popular code editor that allows us to edit text files for multiple programming languages
- ▶ One of those languages is Bash
- ▶ We can just create a Bash file in it
- ▶ Also, we can install the Shellcheck extension - then, Visual Studio Code will give us feedback to our scripts!

- ▶ **Download:**

- ▶ <https://code.visualstudio.com/>

- ▶ **Important:**

- ▶ Visual Studio Code != Visual Studio
- ▶ Those are 2 separate tools!

Bash & Linux CLI

Math in Bash

Math in Bash

► In this chapter:

- ▶ You will learn how to do basic arithmetic calculations in Bash with integers
- ▶ You will learn how to create integer variables in Bash
- ▶ You will learn how to do basic arithmetic calculations in Bash with decimal numbers
- ▶ In addition, you will practice your knowledge in a quiz and an exercise

Bash & Linux CLI

How to calculate with Bash

How to calculate with Bash

- ▶ **So far, we have not been able to calculate with Bash:**

- ▶ `x=1; y=2; echo "${x} + ${y}"`
 - ▶ prints: 1 + 2

- ▶ **Can we run this calculation in Bash?**

- ▶ Bash can calculate with integers within double round brackets: `((...))`
 - ▶ Within those brackets, we do not need a \$ to access variables!

- ▶ **For example:**

```
x=1  
y=2  
(( result = x + y ))  
echo "${result}"
```

How do we output the result directly?

- ▶ How can we output the result directly?

- ▶ We can use \$((. . .)):

- ▶ x=1

- y=2

- echo "\$((x + y))"

Bash & Linux CLI

Declare integer variables in Bash

Declare integer variables in Bash

- ▶ **We can declare the type of a variable**

- ▶ `declare -i ivar`

- ▶ If we then assign a value to this variable, the expression will automatically be evaluated

- ▶ **However, be careful:**

- ▶ Integer variables will calculate math for us - it is possible to omit the round brackets in certain cases - but not always:

- ▶ `x=4; y=3;`
`ivar=x+y`

- ▶ **Thus, best practice:**

- ▶ `((ivar = x + y))`
 - ▶ `ivar=$((x + y))`

- ▶ **Let's have a look in the code!**

Bash & Linux CLI

How to read an integer variable

Bash & Linux CLI

Decimal numbers in Bash?

Decimal numbers in Bash

- ▶ **Just to clarify:**

- ▶ Bash can only calculate with integers!
- ▶ There's no way to run calculations on decimal numbers

- ▶ **So this will just return 2:**

- ▶ `echo "$((5 / 2))"`
- ▶ However, we could use the modulo operator (%) to get the remainder in the integer division:
 - ▶ `echo "$((5 % 2))"` returns 1

Because: $5/2=2$ (integer division),

and the remainder is $5-2*2=1$

Decimal numbers in Bash

- ▶ **However, there's the Unix command bc:**

- ▶ bc: (**b**asic **c**alculator):

- ▶ It's a whole numerical programming language
 - ▶ But usually, we only use it for floating point calculations in Bash
 - ▶ It takes a string, and then runs the commands from that string

- ▶ For example:**

- ▶ `echo "12.3 + 4.5" | bc`

- ▶ For division:**

- ▶ We need to set a scale first! This specifies the number of decimal places for our result:

- ▶ `echo "scale=4; 12.3 / 4.5" | bc`

Bash & Linux CLI

Bonus: Summarizing numbers with awk

The tool: awk

- ▶ awk is a programming language to analyze text files
 - ▶ It's especially useful for tabular data, or CSV files
 - ▶ **CSV:** Comma separated values
 - ▶ Because it's a separate programming language, we will not focus on awk in this course
 - ▶ But I still want to provide you with 2 examples
-
- ▶ **To sum up a file, we can use the following script:**
 - ▶ `awk '{ sum += $1 } END { print sum }' file`
 - ▶ The first code block (`{ sum += $1 }`) is executed for each line
 - ▶ The code block after the `END` is executed at the end
 - ▶ Thus, the sum is calculated and printed at the end

The tool: awk

- ▶ If you want to summarize a CSV file, you can configure the character by which columns are being split
- ▶ **Let's say the data was in the following format:**
 - ▶ Budapest; 1800000
 - ▶ New York; 8500000
 - ▶ London; 9000000
- ▶ **How can we summarize the numbers?**
 - ▶ `awk -F ';' '{ sum += $2 } END { print sum }' file`
 - ▶ **First, we change the delimiter by which the columns are detected:**
 - ▶ `-F ';'`
 - ▶ **And then, we can just access the second column:**
 - ▶ `$2`

awk

- ▶ As you can see, awk can be useful to allow for a quick summarization of data
- ▶ **However:**
 - ▶ For simple tasks, Bash might be enough
 - ▶ And for more complex tasks, Python is usually way more versatile
- ▶ **Thus, in practice:**
 - ▶ awk can be useful
 - ▶ But you can usually also just write the same logic in Bash or Python

Bash & Linux CLI

Fetch JSON from API

Project: Fetch JSON from API

► In this chapter:

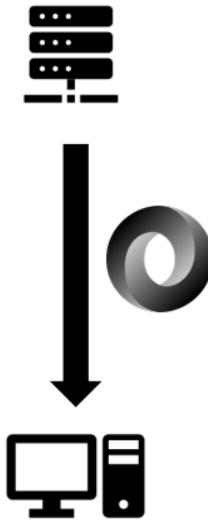
- You will learn how to fetch data with `curl`
- You will learn how to parse JSON with `jq`
- You will then write an interactive weather data fetcher as a bash script!
- Let's have a look at the project ☺

Bash & Linux CLI

Access data from the web

Access data from the web

- ▶ When we access data "from outside", we're able to write way more and way more powerful programs
- ▶ Thus, in this lecture, we will look at APIs, and how we can access them
- ▶ But what is an API?

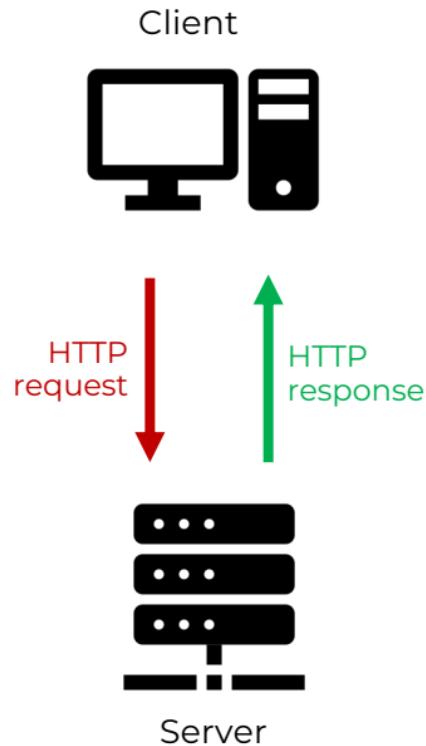


What is an API?

- ▶ API stands for **A**pplication **P**rogramming **I**nterface
 - ▶ It is a set of protocols, tools, and standards that enable different software applications to communicate and interact with each other
 - ▶ It allows developers to access and use functionalities of a system or application without needing to understand or modify the underlying code.
- ▶ **However, in this chapter:**
 - ▶ API means web-API
 - ▶ A server publishes data, that can be accessed by other programs
 - ▶ For example, through JSON

Access data through HTTP

- ▶ We can use HTTP (**Hypertext Transfer Protocol**) to access data from the internet:
 - ▶ 1.: Client sends a request (HTTP request) to the server
 - ▶ 2.: The server replies with a response (HTTP response)
- ▶ **Let's have a look at how this works!**



Bash & Linux CLI

The program: curl

The program curl: overview

► **curl:**

- ▶ It's a program that allows us to transfer data over multiple protocols (HTTP, HTTPS, FTP, SFTP,...)
- ▶ Data retrieval: Fetch and display web pages or API content
- ▶ It's especially useful to integrate it into shell scripts

► **How do we install it?**

- ▶ That depends on our system...
- ▶ **Mac:** brew install curl
- ▶ **Ubuntu:** sudo apt-get install curl
- ▶ **Let's have a look!**

Parameters of curl

► Most important parameters for curl:

► **-v / --verbose:**

- Displays detailed information about the request and response, useful for debugging

► **-o / --output:**

- Saves the response to a file instead of displaying it in the console

► **-L / --location:**

- Follows HTTP redirects (3xx) to retrieve the final resource

► **-s / --silent:**

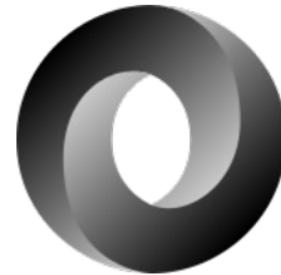
- Silent mode: Do not display progress meter / error data
- It still outputs the data we're accessing

Bash & Linux CLI

JSON: JavaScript Object Notation

What is JSON?

- ▶ **JSON** stands for *JavaScript Object Notation*
 - ▶ A format for the exchange of structured data between different applications, particularly on the web
 - ▶ File extension: **.json**
- ▶ Originally derived from the JavaScript programming language
 - ▶ However, .json files can be processed by many programming languages
- ▶ **Syntax:**
 - ▶ Elements consist of key-value pairs
 - ▶ `{"firstname": "Jannis"}`
 - ▶ Related entries are represented as objects in curly braces `{...}`
 - ▶ Objects and lists ("arrays" with the square braces: `[...]`) can be nested arbitrarily within each other



Bash & Linux CLI

The program: jq

The program: jq

./jq

- ▶ **How do we process JSON in our bash?**
 - ▶ Unfortunately, this is not build-in
- ▶ **But we can use a separate program: jq**
 - ▶ This program allows us to easily access JSON and parse it
 - ▶ It's a complete programming language, that allows way more than to just access an element in JSON
 - ▶ But we will use it mostly for that
- ▶ **We might have to install it:**
 - ▶ **Mac:**
 - ▶ brew install jq
 - ▶ **Ubuntu:**
 - ▶ sudo apt-get install jq
- ▶ **Let's have a look at how we use it!**

Bash & Linux CLI

Exercise: Fetch weather for any city

Project: Weather fetcher

- ▶ Your task is to write a program that fetches the weather from an API
- ▶ However, we will use a mock-API for this:
 - ▶ APIs change quite often...
 - ▶ ... and then this online course breaks
 - ▶ It may take a while until I hear about it as an instructor
 - ▶ And in the meantime, this course wouldn't work properly
- ▶ The solution:
 - ▶ I've created a mock-API for us
 - ▶ <https://downloads.codingcoursestv.eu/055%20-%20bash/api/api.php?city=Budapest>
 - ▶ This generates random data - but aside from that behaves just like a real weather API would
- ▶ However: Feel free to search for a free weather API - there're many out there. Feel free to adjust the code!

Project: Weather fetcher

► Your exercise:

- ▶ Create a program (bash-file), that asks the user to enter the name of a city ("London", "New York",...)
- ▶ It should then send a request to my weather API, to fetch the current temperature
- ▶ This temperature is in °C. Depending on your preferences, you might want to convert it to °F - for example with the program "bc".

► Important, before you start:

- ▶ We might have to use additional parameters for curl...
- ▶ `-G --data-urlencode 'city=New York'`
- ▶ **Let's have a look!**

Bash & Linux CLI

Solution: Fetch weather for any city

Bash & Linux CLI

Control Flows: if & conditions

Control flows: if & conditions

► In this chapter:

- We will have a look at how we can use `if` and conditions in Bash

► However:

- They work quite differently compared to other programming languages
- To fully understand it, we'll start at the foundation and work our way up
- Also, there're many ways to express the same if conditions
- We will thus also have a look at the other ways
(and why you should try to avoid them)

► The roadmap:

- Exit code of programs
- How to test values (`[[...]]`)
- How to use if statements
- Small project at the end

Details matter

- ▶ We will also look at important details, why do we have so many ways to write if conditions?
 - ▶ `if [[...]]; then`
 - ▶ `if [...]; then`
 - ▶ `if test; then`
 - ▶ `if ...; then`

Bash & Linux CLI

The exit code of programs

Exit code in Bash

- ▶ If a program exists, it provides an **exit code**, which indicates the **success** or **failure** of its execution.
- ▶ Exit code 0 is considered true, and any non-zero value is considered false
 - ▶ Exit code **0** -> **success**,
 - ▶ **non-zero value** (e.g., **1**, or anything else) -> **failure**
- ▶ You can access the **exit code of the last command** using the special variable \$?

Exit code in C++

- ▶ If you have programmed in C++ before, the exit code is the return value of the main function

- ▶ **Example:**

```
#include <iostream>

int main(int argc, const char * argv[]) {
    // insert code here...
    std::cout << "Hello, World!\n";
    return 0;
}
```

- ▶ **Important:**

- ▶ Exit code usually gets truncated to the lower 8 bit
(meaning: Integer range from 0-255)
- ▶ Certain shells might use exit codes from above 126 for signals
- ▶ **Thus:** Exit code should be between 0 and 125

Bash & Linux CLI

Chaining commands

Chaining commands

- ▶ We can combine multiple commands in the following way:

- ▶ [command1] && [command2]
- ▶ [command1]; [command2]
- ▶ This works on the CLI, but also in Bash scripts
- ▶ It is especially useful for CLI usage though

- ▶ But what is the difference?

- ▶ ;
 - ▶ Always executes the next command
- ▶ &&
 - ▶ Only executes the next command, if the first command's exit code is 0

Bash & Linux CLI

Chaining with OR: ||

Chaining with OR: ||

- ▶ We can also combine multiple commands with a logical OR:

- ▶ [command1] || [command2]

- ▶ In this case, a logical OR is being applied

- ▶ This means:

- ▶ If the first command executes successfully (exit code 0):

- ▶ The second command will not be executed

- ▶ If the first command executes unsuccessfully (exit code not equal to 0):

- ▶ The second command will be executed

- ▶ Let's have a look at this!

Bash & Linux CLI

Testing values

Testing values

- ▶ If we want to test values, we can use `[[...]]` for this:
 - ▶ `[[condition]]`
 - ▶ **This will then set the exit code:**
 - ▶ **0:** If the condition has been fulfilled
 - ▶ **1:** If the condition has not been fulfilled
- ▶ **Example:**
 - ▶ `[["hello" == "hello"]]`
 - ▶ Condition fulfilled, exit code will be 0
 - ▶ `[["hello" == "world"]]`
 - ▶ Condition not fulfilled, exit code will be 1
 - ▶ `[["hello" != "world"]]`
 - ▶ Condition fulfilled, exit code will be 0
 - ▶ `[["${name}" == "max"]]`
 - ▶ This depends on the contents of the variable `name`

The rules for [[...]]

► **Important:**

- ▶ Word splitting and pathname expansion are disabled within [[...]]
- ▶ But other expansions, such as variable expansion / command substitution are still working!

Bash & Linux CLI

If statements in Bash

if statements in Bash

- ▶ The if statement is used for conditional branching based on the result of a test or exit status

- ▶ **Syntax:**

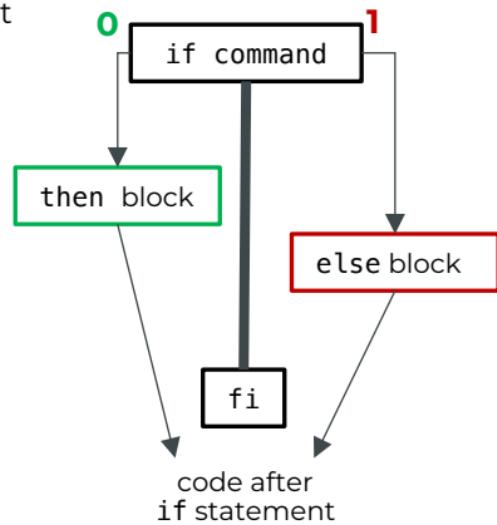
```
if command; then
    # Code to be executed if the exit code is 0
else
    # Code to be executed if the exit code is not 0
fi
```

- ▶ **Important:**

- ▶ It does not matter for the if how the exit code has been generated

- ▶ **Thus, we can just use:**

- ▶ `if [["${name}" == "Max"]]; then`



Bash & Linux CLI

The elif statement

The elif statement

- ▶ The `elif` statement allows you to evaluate multiple conditions in a series of `if` statements

- ▶ **Syntax:**

```
if command1; then
    # Code to execute if command1 is true (exit code = 0)
elif command2; then
    # Code to execute if command2 is true (exit code = 0)
else
    # Code to execute if no previous conditions are true
fi
```

- ▶ Conditions are evaluated from top to bottom
- ▶ Once a condition is found to be true, the corresponding code block is executed, and all remaining conditions are skipped

Bash & Linux CLI

Comparing strings with [[...]]

Testing strings

- ▶ To check if a variable is set to a specific string:

- ▶ `[[${filename} == 'data.csv']]`

- ▶ To check if a variable is of length zero:

- ▶ This one would work:

- ▶ `[[${filename} == '']]`

- ▶ But this is preferred:

- ▶ `[[-z ${filename}]]`

- ▶ To check if a variable is not of length zero:

- ▶ This one would work:

- ▶ `[[${filename} != '']]`

- ▶ But this one is preferred:

- ▶ `[[-n ${filename}]]`

Heads up: [[...]]

- ▶ Technically, [[...]] also supports =
- ▶ It is treated in the same way as ==
- ▶ **Thus, those would be equivalent:**
 - ▶ [["\${filename}" == 'data.csv']]
 - ▶ [["\${filename}" = 'data.csv']]
- ▶ But we should try to always use ==
- ▶ It's more clear

Bash & Linux CLI

Pattern matching in [[...]]

Pattern matching

- ▶ We can also check if a pattern matches
 - ▶ `[["file.txt" == *.txt]]`
- ▶ Important:
 - ▶ Filename expansion is disabled within `[[...]]`
 - ▶ Different syntax rules apply within the square brackets
 - ▶ The `*` means that we want to match any string, including the empty string
 - ▶ A `?` sign would match any single character
 - ▶ `[...]` matches any single character within the square brackets
- ▶ Important:
 - ▶ This is not a regular expression!
- ▶ Also, more complex pattern matching is available:
 - ▶ https://www.gnu.org/software/bash/manual/html_node/Pattern-Matching.html

Regular expressions

- ▶ **We can also check for a regular expression:**

- ▶ `[["file.txt" =~ \.txt$]]`
- ▶ Here, we check that the variable filename ends with .txt, but through a regular expression
- ▶ More on those later!

Bash & Linux CLI

Checking files with [[. . .]]

Checking files with [[. . .]]

- ▶ **We can also check for files:**

- ▶ `[[-e "${filename}"]]`
- ▶ Checks, if a file exists

- ▶ **What options do we have?**

- ▶ `-e`: File exists
- ▶ `-f`: Regular file
- ▶ `-d`: Directory
- ▶ `-r`: Readable
- ▶ `-w`: Writable
- ▶ `-x`: Executable

Bash & Linux CLI

Numeric tests

Numeric tests

- ▶ **To compare numbers, we have several options:**

- ▶ **We can use [[...]]** in combination with:

- ▶ -lt: Lower than
 - ▶ -le: Lower than or equal
 - ▶ -gt: Greater than
 - ▶ -ge: Greater than or equal

- ▶ **Example:**

- ▶ `[["${num_files}" -gt 15]]`

- ▶ **Or we can use an arithmetic expression:**

- ▶ `((num_files > 15))`

- ▶ **Remember:**

- ▶ Within arithmetic expression, different syntax rules apply
 - ▶ We can access variables without a \$

Important

► **This does not work:**

► `[["${num_files}" > 15]]`

► **The reason:**

► Within `[[...]]`, the `>` operator performs
lexicographical comparison, not numeric comparison

Bash & Linux CLI

More complex conditions

More complex conditions

- ▶ **Negation:**

- ▶ `!`: negate a condition

- ▶ **Example:**

- ▶ `[[! -e "${filename}"]]`

- ▶ **Logical Operators:**

- ▶ We can also combine conditions with logical operators:

- ▶ `&&` for a logical AND

- ▶ `||` for a logical OR

- ▶ **Example:**

- ▶ `num=6;`

- `[["${num}" -gt 5 && "${num}" -lt 10]]`

- `[["${num}" -gt 10 || "${num}" -lt 0]]`

- ▶ **But we could also write it like this (only for `&&`):**

- ▶ `[["${num}" -gt 5]] && [["${num}" -lt 10]]`

Bash & Linux CLI

Project: Downloading files only when needed

Downloading files only when needed

- ▶ **Project:**

- ▶ We want to analyze the eBook of Romeo and Juliet

- ▶ **The question:**

- ▶ How many lines contain the word "love"?

- ▶ **The project:**

- ▶ **We need a script:**

- ▶ Downloads the eBook file - but only when needed!
 - ▶ And then, calculates how many lines contain the word "love"
at least 1 time

- ▶ **Also:**

- ▶ It should print error messages to stderr
 - ▶ It should exit with the correct exit code on error

Bash & Linux CLI

How not to test: [, test

Other ways to test

- ▶ Unfortunately, there're way more ways to achieve (almost) the same result as with `[[...]]`
- ▶ **The reason:**
 - ▶ For an if block, it doesn't matter how the exit code is being generated:
 - ▶ `if command; then`
 - ▶ It can be generated by the shell construct `[[...]]`, but it can also be a real program

Programs to run tests

- ▶ **Those programs are especially:**
 - ▶ /bin/[
 - ▶ /bin/test
- ▶ **Because those programs are in our PATH, we can just call them:**
 - ▶ ["\${filename}" == 'data.csv']
 - ▶ test "\${filename}" == 'data.csv'
- ▶ **Or combine them with an if:**
 - ▶ if ["\${filename}" == 'data.csv']; then
 - ▶ if test "\${filename}" == 'data.csv'; then
- ▶ **Important:**
 - ▶ The options that [and test support are slightly different than
[[...]]

Programs to run tests

- ▶ **But we need to be careful:**

- ▶ Bash is calling an external program here
- ▶ Different rules apply than for [[...]]
- ▶ Also, [and test do not support regular expressions

- ▶ **Example:**

- ▶ [["\${filename}" == *.txt]]
 - ▶ Because we're using the shell expression [[...]], * does not mean for filename expansion, but for pattern matching
- ▶ ["\${filename}" == *.txt]
 - ▶ Here, we are calling the program called [
 - ▶ It just happens to require] as a last argument
 - ▶ *.txt is just a normal shell argument
- ▶ **Thus, the following are applied:**
 - ▶ Filename expansion
 - ▶ Word splitting

Best practice

- ▶ Always prefer `[[...]]` over `[...]` or `test`
- ▶ **I personally see it like this:**
 - ▶ `[` and `test` work for compatibility reasons
 - ▶ But for everything "new" (or written in the last 15+ years),
we should use `[[...]]`.

Bash & Linux CLI

The case statement

The case statement

- ▶ Quite often, we want to do multiple pattern matchings after each other
- ▶ For this, we could use an `if` with many `elif`
- ▶ **Let's have a look at this!**
- ▶ **This was not very convenient:**
 - ▶ If we only want to do pattern matching, we should prefer using `case`
 - ▶ `case` is optimized for this use case

The case statement

```
► case expression in
    pattern1)
        # Code to execute if expression matches pattern1
        ;;
    pattern2)
        # Code to execute if expression matches pattern2
        ;;
    pattern3)
        # Code to execute if expression matches pattern3
        ;;
*)
    # Code to execute if expression does not match any patterns
    ;;
esac
```

Bash & Linux CLI

while loops in Bash

In this chapter: while loops

- ▶ **We will have a look at while loops:**

- ▶ They allow us to repeatedly execute a block of code
- ▶ This can be useful if we want to repeat a certain part of our script

- ▶ **Also, we will have a look at a few examples:**

- ▶ How to read a text file line by line
- ▶ How to write a script to trigger multiple downloads

Bash & Linux CLI

Repeating code: while

while loops

- ▶ If we want to repeatedly execute a block of code, we can use a `while` loop

- ▶ **Syntax:**

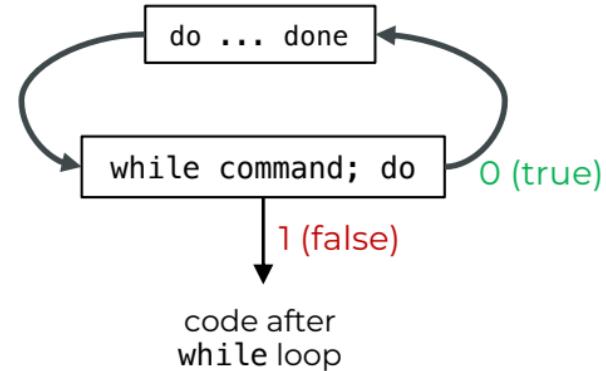
```
while command; do  
    # Code to execute while the command is true (= 0)  
done
```

- ▶ **Syntax (with condition):**

```
while [[ condition ]]; do  
    # Code to execute while the condition is true  
done
```

- ▶ **Command evaluation:**

- ▶ The condition is evaluated at the beginning of each iteration
- ▶ If the condition is true (exit code = 0), the code block is executed
- ▶ If the condition is false, the loop is terminated, and the program continues with the next line of code (after the `done`)



Bash & Linux CLI

The keywords: continue and break

continue and break

- ▶ **continue:**

- ▶ The `continue` statement is used to skip the rest of the code block for the current iteration and move to the next iteration
- ▶ Allows you to bypass certain iterations based on specific conditions

- ▶ **break:**

- ▶ The `break` statement is used to exit a loop prematurely
- ▶ When encountered, the program finishes the while loop immediately

- ▶ **Let's have a look at this!**

Bash & Linux CLI

Reading a file with while

Example: while loop

- ▶ We can use while loops to read a file line by line
- ▶ **We can do this with the following code:**
 - ▶

```
while IFS= read -r my_line; do
    echo "${my_line}"
done < file.txt
```
- ▶ **What does it all mean?**
 - ▶ **IFS=**
 - ▶ The IFS variable defines how whitespace is treated in Bash
 - ▶ For the read command only, we set it to empty - this preserves whitespace at the beginning / end of each line
 - ▶ **read -r my_line**
 - ▶ Reads a single line of the file and provides the contents in a variable called "my_line" (my_line is a variable name of our choosing)

Bash & Linux CLI

Exercise: Downloading files

Exercise: Downloading images

- ▶ **Example:**

- ▶ **We want to download all images from this folder:**

- ▶ <https://downloads.codingcoursestv.eu/055%20-%20bash/while/images/image-1.jpg>

- ▶ **We can just replace the "-1" with "-2" to access the next picture:**

- ▶ <https://downloads.codingcoursestv.eu/055%20-%20bash/while/images/image-2.jpg>

- ▶ **The task:**

- ▶ Can we do this automatically?
 - ▶ Can we also ignore if a few images can't be found, and keep trying
(for up to a few images, like 10 images or so)?

- ▶ **Feel free to do this on your own first**

- ▶ **We will do this together in the next lecture!**

Important

- ▶ I am providing the hosting of the server
- ▶ I have purposely slowed down the delivery of the image
- ▶ **Still:**
 - ▶ Please try to avoid endless loops
 - ▶ If your script is running for more than 1 minute, please quit it

Bash & Linux CLI

Solution: Downloading files

Bash & Linux CLI

for loops in Bash

In this chapter

- ▶ In this chapter, we will have a look at for loops
- ▶ They allow us to repeat a block of code
- ▶ **With while, we had a few problems:**
 - ▶ We had to make sure that we don't end up in an endless loop
 - ▶ We had to keep track of what the loop should do
- ▶ **For loops solve this:**
 - ▶ They are ideal if we know ahead of time how often we want the loop to be executed
 - ▶ They're optimized for this
- ▶ Also, we will have a look at common use cases for for loops
- ▶ **Let's have a look at how they work!**

Bash & Linux CLI

Bash: A first for loop

Bash: for loop

- ▶ The for loop allows us to iterate over a pre-defined number of elements

- ▶ **The syntax:**

```
▶ for [variable] in [elements]; do  
    # Code to execute  
done
```

- ▶ **First example:**

- ▶ Let's say we got 2 names: Lauren and Oliver
- ▶ If we want to execute a code block for each of them, we can use a for loop:

```
▶ for name in "Lauren" "Oliver"; do  
    echo "${name}"  
done
```

Bash & Linux CLI

Bash: For loop + sequence expression

Bash: for loop + sequence expression

- ▶ If we want to generate numbers, we can also use the sequence expression:
- ▶ What is a sequence expression?
 - ▶ {start..end}
 - ▶ {start..end..step}
 - ▶ This allows us to generate a sequence of elements
- ▶ Let's have a look at this first!
- ▶ How can we use it in a for loop?
 - ▶

```
for i in {1..5}; do
    echo "${i}"
done
```

Bash & Linux CLI

Bash: For loop + additional expansions

Bash: for loop

- ▶ **For loop and brace expansion:**

- ▶ We can also use the brace expansion to generate additional elements

- ▶ **Example:**

```
▶ for file in names{.csv,.txt,.docx}; do  
    echo "${file}";  
done
```

- ▶ **Important:**

- ▶ When using brace expansion, we're only generating elements / strings
 - ▶ It does not matter if the files exist or not
 - ▶ Brace expansion never checks for file existence

Bash: for loop & filename expansion

- ▶ In a for loop, we can also use the filename expansion
 - ▶ This one will check our filesystem and only provide the elements if the corresponding files exist
- ▶ Example (for with filename expansion):
 - ▶

```
for file in *.txt; do
    echo "${file}"
done
```
- ▶ Of course, we can also combine filename expansion with brace expansion:
 - ▶

```
for file in *.{txt,jpg}; do
    echo "${file}"
done
```
- ▶ Here, because we're using filename expansion, it will only expand to files that exist

Bash & Linux CLI

Bash: For loop + command substitution

Bash: for loop + command substitution

- ▶ We can also use a for loop in combination with command substitution

- ▶ Quite often, we then call the seq program:

- ▶ seq \$start \$end
 - ▶ This generates the range for us

- ▶ Example code:

- ▶ start=1; end=5
for i in \$(seq \$start \$end); do
echo "\$i"
done

- ▶ But we can use the command substitution with every command

- ▶ This would print out every word of a file:

- ▶ for word in \$(cat file.txt); do
echo "\${word}"
done

Bash & Linux CLI

Bash: For loop + arithmetic expression

Bash: for loop

- ▶ We can also use a different form of the for loop:

- ▶

```
for (( init; test; after )); do
    # Code to executed
done
```

- ▶ init: Executed one time at the beginning
- ▶ test: Executed before each iteration
- ▶ after: Executed after each iteration

- ▶ Example:

- ▶

```
for (( i = 0; i <= 4; i++ )); do
    echo "${i}"
done
```



Bash & Linux CLI

Project: Image processing with Image Magick and for loops

Project: Image Magick



► **In this chapter:**

- We will first have a quick look into Image Magick

► **This tool allows us to work with images:**

- Query image for information (such as width, height,...)
- Image conversion (such as: .png -> .jpg)
- Image rescaling (changing resolution, compression,...)
- Image manipulation (adding text, changing contrast,...)

► **Also, we will learn how to work with file paths / names in Bash!**

► **After this:**

- We will write a script will create thumbnails for all images in a folder

► **However:**

- We only want to create thumbnails that do not exist yet
- Also, if an image is sufficiently small (let's say 100x100 pixels max.), we do not want to create a thumbnail



Bash & Linux CLI

Let's install Image Magick

Image Magick



- ▶ Image Magick allows us to convert and manipulate images
- ▶ **First, we need to install it:**
 - ▶ **Ubuntu:**
 - ▶ `apt install imagemagick`
 - ▶ **CentOS:**
 - ▶ `dnf install ImageMagick`
 - ▶ (EPEL should be enabled)
 - ▶ **Mac:**
 - ▶ `brew install imagemagick`
 - ▶ (homebrew must be installed => <https://brew.sh/>)
- ▶ After this, we can confirm that imagemagick has been installed correctly:
 - ▶ `convert --version`



Bash & Linux CLI

Image Magick: Querying an image

Image Magick: identify

- ▶ **The identify program is part of the Image Magick suite**
 - ▶ It allows us to collect information about an image
 - ▶ **To see all available options:**
 - ▶ `identify -help`
 - ▶ **Example:**
 - ▶ `identify [image]`
 - ▶ `identify -verbose [image]`
 - ▶ `identify -format '%wx%h' [image]`
 - ▶ **This will print out the width and the height of an image:**
 - ▶ **1920x1080**



Bash & Linux CLI

Image Magick: Resizing

Image Magick: Resizing an image

- ▶ **How to resize an image:**

- ▶ For this, we can use the convert program

- ▶ **Example:**

- ▶ `convert input.jpg -resize 100x100 output.jpg`

- ▶ **input.jpg:**

- ▶ The image that we want to resize

- ▶ **-resize 100x100:**

- ▶ The maximum target image size. If the image is not a square, the maximum width / height is 100px

- ▶ **output.jpg:**

- ▶ The image that we want to create

Bash & Linux CLI

Working with filenames

Working with filenames

- ▶ For the next project, we will need to work with filenames
- ▶ Thus, in this lecture, we want to have a quick look at how we can work with file paths in Bash
- ▶ Let's have a look at this!

basename vs. dirname

- ▶ Let's say we have a long pathname, such as:
 - ▶ /home/janniss/Desktop/file.txt
- ▶ How can we get the folder?
 - ▶ dirname /home/janniss/Desktop/file.txt
- ▶ Will print:
 - ▶ /home/janniss/Desktop
- ▶ How can we get just the filename?
 - ▶ basename /home/janniss/Desktop/file.txt
- ▶ Will print:
 - ▶ file.txt

Extracting parts of the filename

- ▶ Let's say we got just a filename, in a variable:

- ▶ `filename='file.txt'`

- ▶ How can we get the extension (txt)?

- ▶ For this, we can use the Shell Parameter Expansion

- ▶ `echo "${filename##*.}"`

- ▶ Will print:

- ▶ `txt`

- ▶ **##:** Removes the longest match (from the beginning of the variable) of the pattern.

- Here, the pattern is `*.`, meaning it should match everything, but it must have a dot at the end.

- ▶ **#:**

- ▶ `echo "${filename#*.}"`

- ▶ This would just remove the shortest match of the pattern (and try to match from the beginning)

- ▶ The extension would not be detected correctly for `hello.world.txt`

Extracting parts of the filename

- ▶ Let's say we got just a filename, in a variable:

- ▶ `filename='file.txt'`

- ▶ How can we get the filename, without the extension (file)?

- ▶ For this, we can use the Shell Parameter Expansion

- ▶ `echo "${filename%.*}"`

- ▶ Will print:

- ▶ `file`

- ▶ %: Removes the shortest match of the pattern (from the end of the variable). Here, the pattern is `.*`, meaning it should match everything, but it must have a dot at the beginning.

- ▶ %%:

- ▶ `echo "${filename%%.*}"`

- ▶ This would just remove the longest match of the pattern

- ▶ The filename without the extension would not be detected correctly for

- `hello.world.txt`



Bash & Linux CLI

Exercise: Image processing

Exercise: Image processing

- ▶ **Your task:**

- ▶ Create an image processing script

- ▶ **What should it do?**

- ▶ **It should process all images in a directory:**

- ▶ It should check, if a thumbnail already exists - if it does, we can skip generating a new thumbnail
 - ▶ It should also check, if we need to generate a thumbnail at all (meaning that the width / height of the image is greater than 100 pixels)
 - ▶ Only then, a thumbnail should be generated



Bash & Linux CLI

Solution: Image processing

Bash & Linux CLI

Creating dialogs

Dialogs

Welcome to this chapter about dialogs!

< **OK** >



Important message



Are you sure? This can be **really** dangerous

No

Yes

Let's create fancy dialogs

- ▶ In this chapter, we want to have a look at how we can create fancy dialog flows in Bash
- ▶ **For this, we will have a look at the following:**
 - ▶ First, we will look at how to create simple menus within Bash (no external programs)
 - ▶ Then, we will mostly focus on the `dialog` program (CLI dialogues)
 - ▶ But we will also have a quick look at `zenity` (GUI dialogues)

Bash & Linux CLI

The select construct

The select construct

- ▶ To create simple menus, we can use the **select** construct in Bash:

```
▶ select selected_var in option1 option2 option3 option4 [...]; do  
    echo "${REPLY}: ${selected_var}"  
    break  
done
```

- ▶ What happens here?

- ▶ We can specify the options from which the menu should be generated
- ▶ The user will then be asked to choose an option
- ▶ The chosen option will be provided in the variable that we specified
(here: `selected_var`)
- ▶ The selected index will be provided in a variable called `REPLY`
- ▶ Usually, the dialog would show again. To prevent this, we need a `break`

Configuring the select

- ▶ **To configure the select, we can change the following variable:**

- ▶ PS3
- ▶ This is used to define the prompt, and how the menu asks the user for input

- ▶ **Example:**

- ▶ `PS3='Please select your option: '`

Bash & Linux CLI

Project: select

Bash & Linux CLI

Installing the tool: dialog

Installing the dialog program

- ▶ Depending on the system, the dialog program might not be installed
- ▶ **Thus, we need to install it:**
 - ▶ **Ubuntu:**
 - ▶ apt install dialog
 - ▶ **CentOS:**
 - ▶ dnf install dialog
 - ▶ **Mac:**
 - ▶ brew install dialog
 - ▶ (homebrew must be installed => <https://brew.sh/>)
- ▶ **After this, we can check if dialog is installed:**
 - ▶ dialog --help

Using the dialog program

- ▶ The `dialog` program offers many different dialog options
- ▶ We will not be able to explore all of them
- ▶ **We can get an overview:**
 - ▶ `dialog --help`
- ▶ In this course, we will have a look at the most important options

Bash & Linux CLI

Creating our first dialog

Important message



Creating a simple message box

- ▶ To create a simple message box, we can use the following command:

- ▶ `dialog --msgbox [message] 0 0`
- ▶ `dialog --msgbox 'Bash is fun' 0 0`

- ▶ What do the parameters mean?

- ▶ `--msgbox [message] 0 0`
 - ▶ We want to create a dialog of the type "`--msgbox`"
 - ▶ Then, we need to provide our message
 - ▶ `0 0`: The height and the width should be calculated automatically

Additional parameters: --msgbox

- ▶ To create a more advanced message box, we can add a few options:

```
▶ dialog --backtitle [backtitle] --title [title] --msgbox [message] 0 0  
▶ dialog --backtitle 'Important message' --title 'Be careful' --msgbox 'Bash is fun' 0 0
```

- ▶ What do the options mean?

- ▶ --backtitle:

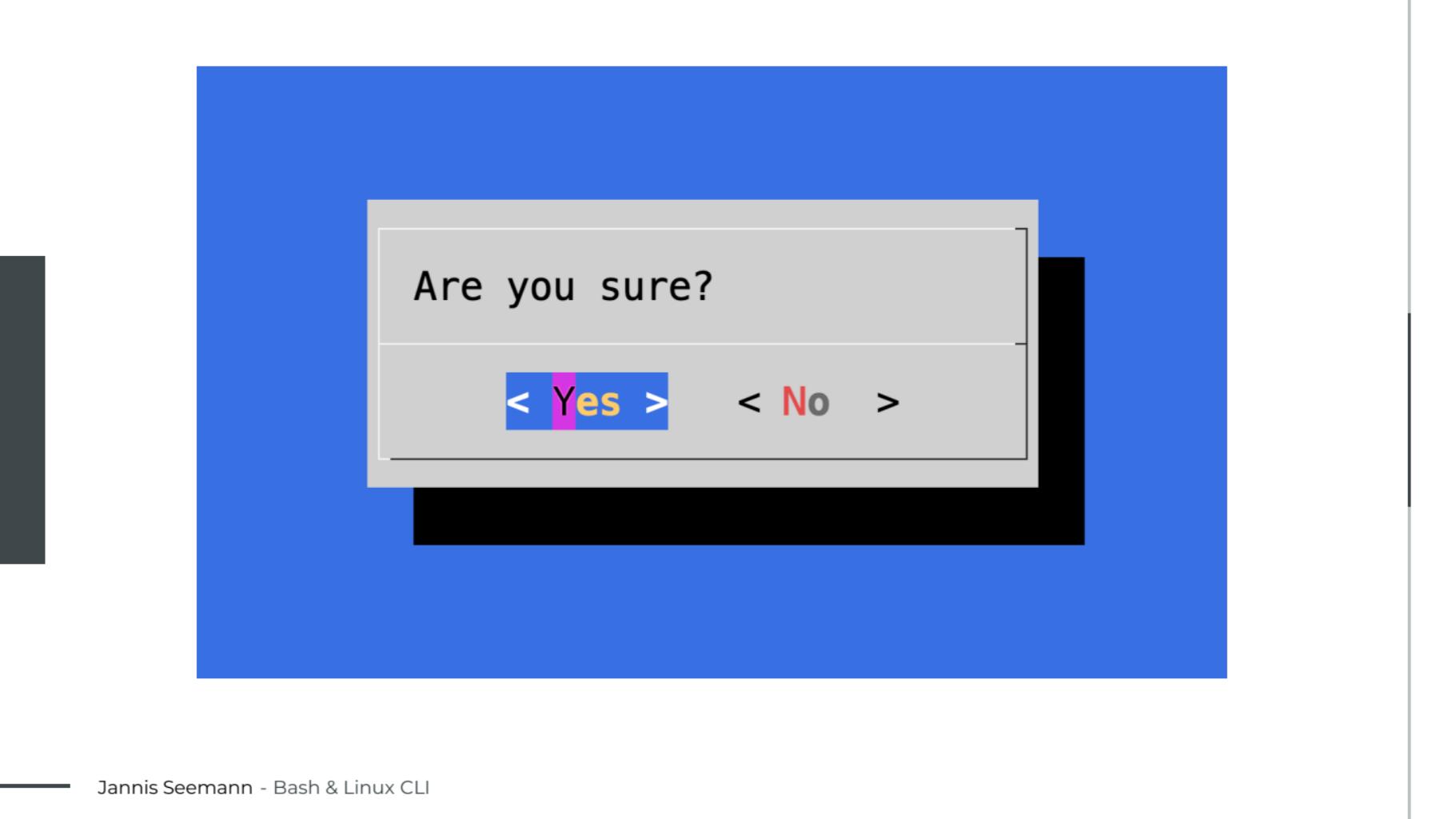
- ▶ We define the title of the background

- ▶ --title:

- ▶ We define the title of the dialog

Bash & Linux CLI

Creating yes/no dialog



Are you sure?

< Yes >

< No >

Creating a yes/no dialog

- ▶ To create a yes/no dialog, we can use the following command:

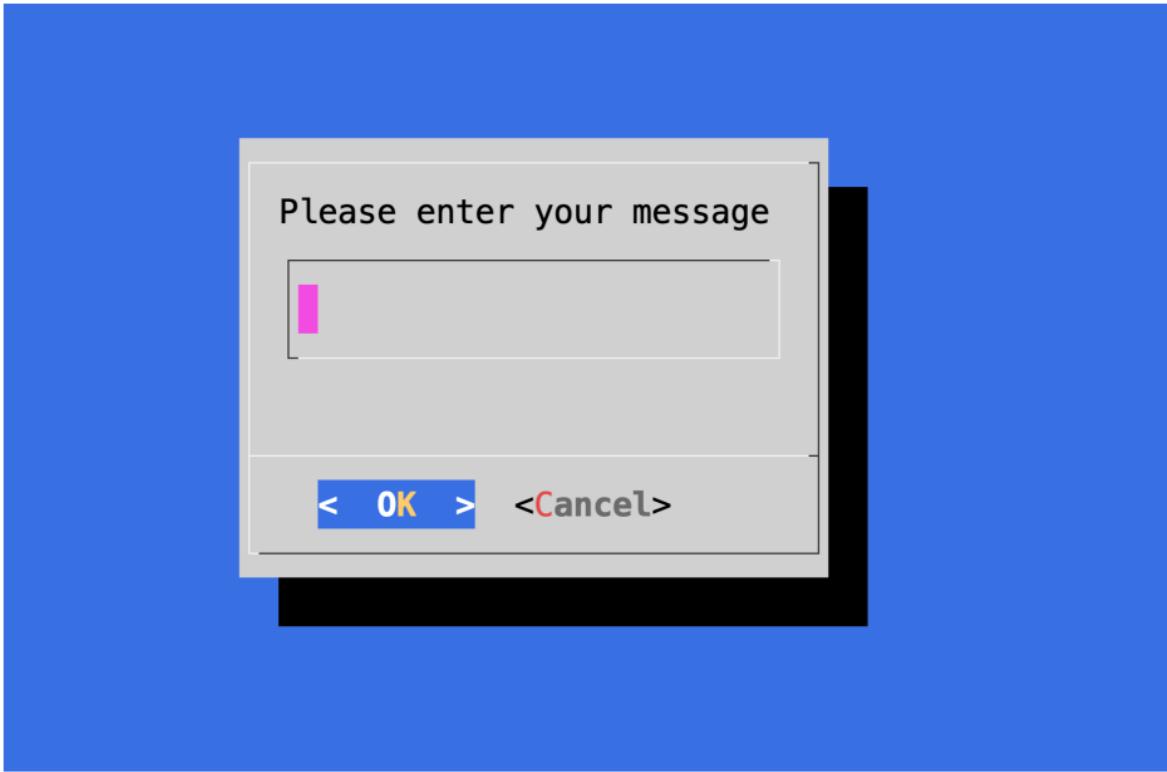
- ▶ dialog --yesno [message] 0 0
- ▶ dialog --yesno 'Are you sure?' 0 0
- ▶ 0 0:
 - ▶ Height and width should be calculated automatically

- ▶ But how do we get the value?

- ▶ Yes => Exit code of 0
- ▶ No => Exit code of 1
- ▶ This allows us to easily use this dialog in a shell script!

Bash & Linux CLI

Allowing input (--inputbox)



Allowing text input

- ▶ If we want to allow text input for the dialog, we can use the following option:
 - ▶ `dialog --inputbox [message] 0 0`
 - ▶ `dialog --inputbox 'Please enter your message' 0 0`
- ▶ **What about the input?**
 - ▶ Our input will then be written to stderr!

But how do we get the output?

- ▶ **Dialog writes output to two locations:**

- ▶ **stdout:**

- ▶ This is for the dialog itself, not our input

- ▶ **stderr:**

- ▶ This is for the input that we entered in the dialog

- ▶ **Thus, we need the following command:**

- `message=$(dialog --inputbox 'Please enter your message' 0 0 2>&1 >/dev/tty)`

- ▶ **What do the parts do?**

- ▶ `$(...)`: Creates a subshell in which the dialog is being executed

- ▶ `2>&1`: Stderr is being redirected to the current stdout (stdout of the subshell)

- ▶ `>/dev/tty`:

- ▶ `stdout` is being redirected to the current `tty`. The normal output is now no longer sent as `stdout` to the terminal, but written directly to it

Bash & Linux CLI

Creating a menu

Please choose your favorite
linux

Ubuntu Linux

CentOS

< OK >

<Cancel>

Creating a selectable menu

- ▶ To create a selectable menu, we can use the following command:

```
▶ dialog --no-tags --menu [menu description] 0 0 0 [tag1] [item1] [tag2] [item2] ...
```

- ▶ Example:

```
▶ dialog --no-tags --menu 'Please choose your favorite linux' 0 0 0 'ubuntu'  
'Ubuntu Linux' 'centos' 'CentOS'
```

- ▶ What do the options mean?

- ▶ --no-tags:

- ▶ Don't show tags in the menu

- ▶ --menu [menu description]:

- ▶ Defines the label for the menu

- ▶ 0 0 0:

- ▶ Auto height, auto width, auto menu height

- ▶ This menu also provides the selected tag as the stderr output

Bash & Linux CLI

Bonus: zenity



Important message



Are you sure? This can be **really** dangerous

No

Yes

Dialogs with zenity

- ▶ If we want to create GUI dialogs, we can use similar programs
- ▶ One of those is `zenity`
- ▶ **We need to install it:**
 - ▶ **Ubuntu:**
 - ▶ `apt install zenity`
 - ▶ **CentOS:**
 - ▶ `dnf install zenity`
 - ▶ **Mac:**
 - ▶ `brew install zenity`
 - ▶ (homebrew must be installed => <https://brew.sh/>)
- ▶ **After this, we can check if zenity is installed:**
 - ▶ `zenity --help`

Using zenity

- ▶ The options of zenity are different than for dialog
- ▶ **Thus, we should have a more in-depth look:**
 - ▶ `zenity --help-question`
- ▶ **Example:**
 - ▶ `zenity --question --text='Do you confirm?' --title='Message'`
 - ▶ **We can use Pongo for markup:**
 - ▶ `zenity --question --text='Are you sure? ' --title='Hello'`
- ▶ **Or, to create a text entry dialog:**
 - ▶ `zenity --entry`
- ▶ **Important difference to dialog:**
 - ▶ Because the dialog itself is shown through a GUI, the result can be written to stdout (instead of stderr)

GUI dialoges

► **Be careful though:**

- ▶ For this, we need a graphical user interface
- ▶ We might not have it when we connect through ssh to a remote server
- ▶ Thus, we might prefer to just use CLI dialogs

Bash & Linux CLI

Exercise: Interactive Bash script!

Exercise: Interactive program!

- ▶ We now want to use a dialog to allow for easy data entry
- ▶ **The idea:**
 - ▶ We create a Bash script to collect student information
 - ▶ It will ask us for the name of a student to enroll
 - (`dialog --inputbox`)
 - ▶ After this, this name will be written to a text file
 - ▶ We will be asked if we want to continue with entering the next name, or quit the program

Bash & Linux CLI

Solution: Interactive Bash script!

Bash & Linux CLI

Accepting arguments

Arguments in Bash

- ▶ **So far, we've already seen how we can pass arguments to other programs:**
 - ▶ `ls -al`
 - ▶ `cat filename.txt`
- ▶ **Can we also accept arguments?**
 - ▶ We will first see how to pass information to a Bash script
 - ▶ This will allow us to write something like this:
 - ▶ `./script data.csv`
- ▶ **Also, we will have a look at getopt:**
 - ▶ This will allow us to accept and parse custom arguments:
 - ▶ `./script -c -f filename.txt`

Bash & Linux CLI

Arguments for our script

Arguments for a script

- ▶ **We can call a script and pass arguments to it:**

- ▶ `./script argument1 argument2 argument3`

- ▶ **Accessing the arguments:**

- ▶ But how do we access those arguments?

- ▶ **For this, we can use special variables:**

- ▶ Program name (here: `./script`): `$0`

- ▶ First positional argument: `$1`

- ▶ Second positional argument: `$2`

- ▶ (and so on)

Important

- ▶ Expansions are still applied, before the script is executed
- ▶ This happens before our script is being executed
- ▶ **Example:**
 - ▶ `./script *`
 - ▶ This will search for all files in our current path
 - ▶ Those will then be passed as an argument to our script
 - ▶ The script itself will not be notified about the original command (before the expansion)!

Arguments: Additional variables

- ▶ We can also access additional variables that Bash provides for us

- ▶ **Additional variables:**

- ▶ `$#`

- ▶ Number of positional arguments

- ▶ `$@`

- ▶ Provides the positional arguments as an array
 - ▶ (more on arrays later)

- ▶ `$*`

- ▶ Provides the positional arguments as a single string

- ▶ `$$`

- ▶ Provides the current process ID (PID)

Bash & Linux CLI

The shift command

The shift command

- ▶ The shift command allows us to write programs that work with a variable number of arguments
- ▶ **How can we use it?**
 - ▶ `shift [number]`
 - ▶ If we omit number, it is assumed to be 1
 - ▶ Shift pretty much just shifts the positional arguments (`$1, $2, $3,...`)
- ▶ **Shift does the following:**
 - ▶ The first `[number]` arguments will be thrown away
 - ▶ The remaining arguments take their place then
 - ▶ Also, `$#` is updated
 - (the variable for the number of positional arguments)

Example: The shift command

- ▶ **Example:**

- ▶ shift 2
- ▶ Arguments \$1 and \$2 are thrown away
- ▶ \$3 is moved to \$1
- ▶ \$4 is moved to \$2
- ▶ \$5 is moved to \$3
- ▶ (and so on)
- ▶ \$# will be updated to the current number of positional arguments

Bash & Linux CLI

shift and while

We can combine shift with a while loop!

- ▶ Of course, we can combine shift with a while loop
- ▶ This allows our script to accept multiple arguments
- ▶ **Example:**

```
▶ while (( $# != 0 )); do
    echo "The argument is: $1"
    shift
done
```

Bash & Linux CLI

Accepting arguments: getopt

Accepting arguments: getopt

- ▶ To accept arguments, we can use the getopt command that Bash provides for us
- ▶ It helps us parse CLI options (such as: -a, -l, -h,...)
- ▶ **How does it work?**
 - ▶ It goes through the arguments, and checks for arguments that start with a "-" (and are not exactly "--" or "-")
- ▶ **Syntax:**
 - ▶ getopt [available-args] [var]
 - ▶ Each time we call it, it will try to find one more argument for our bash script and provide it in a variable
 - ▶ It will keep track in the OPTIND variable
 - ▶ We can call it multiple times to get multiple arguments

Accepting arguments: getopt

▶ Example:

- ▶ We want to accept the option "a" and "l" for our script:

- ▶ ./script -la
- ▶ ./script -l -a
- ▶ ./script -al
- ▶ ./script -a

▶ We can then use getopt:

- ▶ getopt "la" option
- ▶ getopt will now go through the options, get the first one, and provide it in a variable called "option"

Bash & Linux CLI

getopts and while

getopts and while

- ▶ If we want to accept multiple arguments:

```
▶ while getopts 'al' opt 2>/dev/null; do  
    echo "${opt}"  
done
```

- ▶ What happens here?

- ▶ getopts 'al' opt 2>/dev/null
 - ▶ We're trying to get the next option. If an option is not allowlisted in the pattern 'al', getopts prints an error
 - ▶ We redirect this error to /dev/null

getopts, while and case

- ▶ We can then use a case inside the while loop:

```
▶ while getopts 'al' opt 2>/dev/null; do
    case "${opt}" in
        a) echo "Option a"
           ;;
        l) echo "Option l"
           ;;
        ?) echo "An error occurred"
           ;;
    esac
done
```

Bash & Linux CLI

Accepting parameters: getopt

Accepting parameters

- ▶ We can also use getopt to accept parameters for our Bash script:
 - ▶ getopt 'af:' opt
 - ▶ The colon after the f means that we require a parameter after the -f option
- ▶ Important:
 - ▶ In each iteration, it will still only find one option
 - ▶ We should call it in a while loop with a case inside
- ▶ This enables the following ways to call our program:
 - ▶ ./script -a -f filename.txt
 - ▶ ./script -a
 - ▶ ./script -f filename.txt
- ▶ If the option -f has been found (meaning variable "opt" is equal to "f"):
 - ▶ The parameter (here: filename.txt) can be found in OPTARG

Bash & Linux CLI

Bash: Functions

Functions in Bash

- ▶ **Functions:**

- ▶ We will have a look at how functions work in Bash
- ▶ This will allow you to modularize your code

- ▶ **Though, be careful:**

- ▶ This would theoretically allow us to write more complex scripts
- ▶ But the main purpose of Bash scripts is that we have a simple language for **small** utilities
- ▶ Bash scripts are not meant for complex logic!
- ▶ Still, a little bit of modularization can be helpful

Important

- ▶ To me, it feels like Bash has its own concepts of how functions are being treated
- ▶ **If you have programmed before:**
 - ▶ Try to forget many concepts that you know from other programming languages (C++ / Python / JavaScript / ...)
 - ▶ And treat Bash as a separate language

Bash & Linux CLI

Functions in Bash

Functions in bash

- ▶ Functions in Bash allow reuse code blocks
- ▶ This helps improving code organization and maintainability

- ▶ **Syntax:**

```
function_name() {  
    # Code to be executed  
}
```

- ▶ **To invoke a function, we can simply write its name:**
 - ▶ `function_name`
- ▶ **Let's have a look at this!**

Alternative way to define a function

- ▶ **FYI:** We can also use the function keyword in front of the name:

```
▶ function function_name {  
    # Code to be executed  
}
```

- ▶ **But we will omit the function keyword**

Bash & Linux CLI

Important: Variables in functions

Important: Variables in Functions

- ▶ **Important:**

- ▶ Variables are global by default

- ▶ **This means, that this will print out "Max":**

- ```
▶ my_function() { name='Max'; }
my_function
echo "${name}"
```

- ▶ **To avoid this, we should use the local keyword:**

- ▶ Then the variable will be local and valid within the function only

- ▶ **This will print nothing:**

- ```
▶ my_function() {
    local name
    name='Max'
}
my_function
echo "${name}"
```

Important

- ▶ Sometimes, storing data in global variables is useful in Bash
- ▶ Otherwise, we might need a lot of additional code just to bring data from one function to another
- ▶ **If you do use global variables:**
 - ▶ Be sure to add a command above each function that uses them
 - ▶ And document their use!

Bash & Linux CLI

Getting data into the function

Passing stdin to a function

- ▶ We can pass data as stdin into a function

- ▶ **Example:**

```
▶ greet() {  
    local name  
    read name  
    echo "Hello, ${name}!"  
}  
echo "Olivia" | greet
```

Passing arguments to functions

- ▶ Functions can also accept arguments
- ▶ **But:**
 - ▶ There's no special syntax for this
 - ▶ It works just as the position arguments for bash scripts (\$1, \$2,...)
- ▶ **Example:**

```
▶ greet() {  
    echo "Hello, $1!"  
}  
greet "World"
```

How to provide default values

- ▶ There's no way in Bash to provide a default value for an argument

- ▶ **But we can use the default value expansion:**

- ▶ `${var:-default}`

- ▶ **In our case:**

- ▶

```
greet() {  
    local name  
    name=${1:-'World'}  
    echo "Hello, ${name}!"  
}
```

Bash & Linux CLI

Returning values from functions

Returning values

- ▶ A return value of a Bash function is equal to the return value of a program (=exit code)

- ▶ **Example:**

```
▶ greet() {  
    local name  
    name=${1:-'World'}  
    echo "Hello, ${name}!"  
    return 0  
}
```

- ▶ **Thus, returning 0 means:**

- ▶ Everything worked properly

- ▶ **And returning any other integer means:**

- ▶ An error has occurred

Returning values

- ▶ So how do we return actual values from a function?
- ▶ **Simple answer:** We don't
- ▶ **But:**
 - ▶ We can either write to a non-local variable, and use it in another function (extremely ugly)
- ▶ **Or we write to stdout (preferred):**

```
▶ greet() {  
    echo "Hello, $1";  
}  
  
message=$(greet 'Max')  
echo "${message}"
```

Bash & Linux CLI

Project: Trivia

Project: Trivia

- ▶ **We now want to start developing a real program:**

- ▶ We want to develop a trivia quiz, that runs completely within Bash
- ▶ For this, we will use the data that comes from an open trivia DB:
 - ▶ <https://opentdb.com/>

- ▶ **The goal of this project:**

- ▶ We will examine how we can utilize functions to structure our code
- ▶ We can see how we get data into and out of a function
- ▶ We will explore important Bash concepts

Legal

► **Important:**

- The questions and answers of the quiz are published under the following terms:
- Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)
- <https://creativecommons.org/licenses/by-sa/4.0/>
- I have not done any changes to the quiz questions
- The questions here in this course are published under the same terms

Bash & Linux CLI

Project: Trivia (part 1)

Exercise: Project trivia (part 1)

► Exercise:

- ▶ Create a Bash script and write a function `load_question`
- ▶ When invoked, `load_question` should load the .json file, determine how many questions are in it, and provide a random question
- ▶ The question should be provided in the following format:
 - ▶ [Question]
 - [Valid answer]
 - [Invalid answer 1]
 - [Invalid answer 2]
 - [Invalid answer 3]

Exercise: Project trivia (part 1)

- ▶ **Example value (all written to stdout):**

- ▶ What did the first moving picture depict?

- A galloping horse

- A woman in a dress

- A man walking

- A crackling fire

Bash & Linux CLI

Project: Trivia (part 2: solution)

Bash & Linux CLI

Project: Trivia (part 3)

Bash & Linux CLI

Project: Trivia (part 4)

Bash & Linux CLI

Project: Trivia (part 5)

Bash & Linux CLI

Project: Trivia (part 6, final touches)

Bash & Linux CLI

Bash: Arrays

Bash: Arrays

► In this chapter:

- ▶ We will have a look at arrays
- ▶ They allow us to easily manage a collection of entries
- ▶ Why can't we just use normal variables for this?
- ▶ **Let's have a look at this!**

Bash & Linux CLI

Arrays in Bash

Arrays in Bash

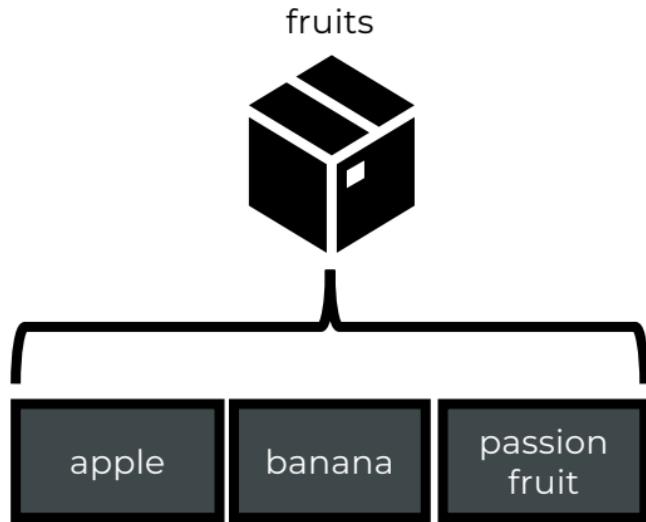
- ▶ With arrays, we can manage a collection of entries

- ▶ To create an array:

- ▶ `array_name=(value1 value2 value3 ...)`
- ▶ `fruits=("apple" "banana" "passion fruit")`

- ▶ We can also declare an array explicitly:

- ▶ `declare -a fruits=("apple" "banana" "passion fruit")`



Arrays in Bash

- ▶ By default, only the first element will be accessed:

- ▶ `echo "${fruits}"`

- ▶ Square brackets:

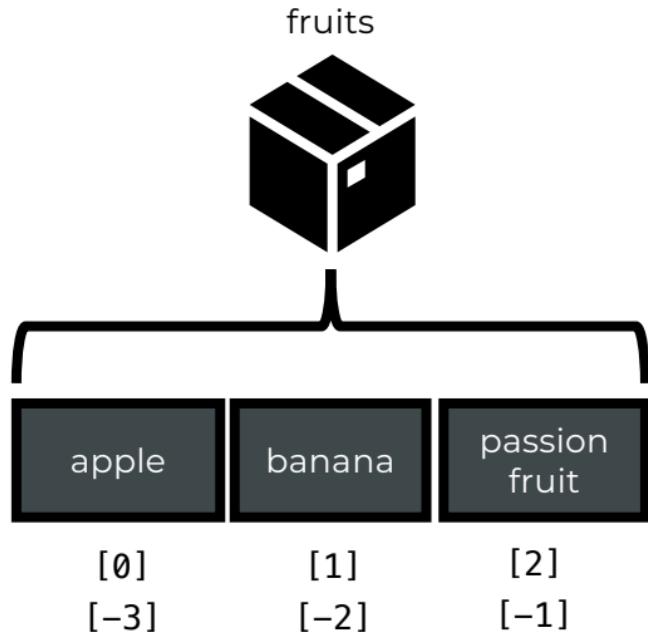
- ▶ To access a specific entry, we can use square brackets
 - ▶ Indexing starts at 0

- ▶ Example:

- ▶ `echo "${fruits[0]}"` -> "apple"

- ▶ As of Bash 4, negative indices are also possible

- ▶ `echo "${fruits[-1]}"`



Bash & Linux CLI

Accessing the whole array

Accessing the whole array

- ▶ `fruits=("apple" "banana" "passion fruit")`
- ▶ **To access the whole array:**
 - ▶ **`echo "${fruits[@]}"`**
 - ▶ This is the proper way to access the whole array
 - ▶ It will create one "word" for each entry of the array, meaning "passion fruit" will be one entry
 - ▶ **`echo "${fruits[*]}"`**
 - ▶ This also works, and will print the whole array
 - ▶ It will turn the whole array into a single entry (separated by the first character of the variable IFS, usually a space)
 - ▶ When using an echo, the result is the same
 - ▶ But it's best to get used to using [@]

Bash & Linux CLI

Overwriting elements

Overwriting elements

- ▶ `fruits=("apple" "banana" "passion fruit")`
- ▶ **How can we overwrite an element?**
 - ▶ For this, we can just use the normal assignment operator
- ▶ **Example:**
 - ▶ `fruits[2]='mango'`
- ▶ **Let's have a look at this!**

Bash & Linux CLI

Array operations

Array operations

- ▶ **Array length:**

- ▶ Syntax: \${#array_name[@]} or \${#array_name[*]}

- ▶ **Example:**

- ▶ `length=${#fruits[@]}`

- ▶ **Adding elements:**

- ▶ We can append elements by using the `+=` operator

- ▶ **Example:**

- ▶ `fruits+=("grape")`

- ▶ **Deleting elements:**

- ▶ You can remove elements through the `unset` command

- ▶ **Example:**

- ▶ `unset fruits[2]` → removes the element at index 2

Array slicing

- ▶ **Array slicing:**

- ▶ Extract a portion of an array using the slicing syntax
 `${array_name[@]:start_index:length}`

- ▶ **Example:**

- ▶ `sliced_array=("${fruits[@]:1:2}")`

Bash & Linux CLI

How to copy an array

Copying an array

- ▶ `fruits=("apple" "banana" "passion fruit")`
- ▶ **Can we copy this array into another variable?**
 - ▶ `fruits_copy=("${fruits[@]}")`
- ▶ **Remember:**
 - ▶ (...): We want to create an array
 - ▶ \${...}: This is just a way to access a variable
 - ▶ [@]: We want to get all individual elements of the array as individual elements
- ▶ **We can also use this to append / prepend elements:**
 - ▶ `fruits_new=("mango" "${fruits[@]}" "papaya")`

Bash & Linux CLI

Arrays and for loops, select

Arrays and for loop, select

- ▶ We can also use arrays in a for loop
- ▶ **For this, we need to access the array in the following way:**
 - ▶ "\${array[@]}"
- ▶ **Example:**
 - ▶ fruits=("apple" "banana" "passion fruit")
 - ▶ for fruit in "\${fruits[@]}"; do
 - echo "\${fruit}"
 - done
- ▶ **The same syntax also works in a select:**
 - ▶ select fruit in "\${fruits[@]}"; do
 - echo "\${fruit}"
 - done

Bash & Linux CLI

Common tasks: read & arrays

Bash: read & arrays

- ▶ **We can use the read program to read an array for us:**
 - ▶ `read -a fruits`
 - ▶ The read command will accept input, and each word will become an element of the array fruits
 - ▶ **Let's have a look at this!**
- ▶ **We can also combine this with a redirect of the output of a process substitution:**
 - ▶ `read -a uptime_data < <(uptime)`
 - ▶ **This is the way to go if we want to split the input**
 - ▶ **Why can't we write it in a simpler way?**
 - ▶ `uptime_data=($(uptime))`

Bash: Why do we need read?

- ▶ **Why can't we write it in a simpler way?**

- ▶ `uptime_data=($(uptime))`
- ▶ This one would indeed always work - the output of uptime is predictable and (most likely) won't contain any problematic characters

- ▶ **Problematic characters?**

- ▶ Let's say we want to execute a slightly different program in the command substitution:

- ▶ `data=($(echo '*'))`

- ▶ **When running this code, the following will be performed:**

- ▶ Command substitution
- ▶ Word splitting
- ▶ **Filename expansion (globbing)**
- ▶ Array assignment
- ▶ **Thus, filenames will be resolved!**

Bash & Linux CLI

Common tasks: Output to array

Output to array

- ▶ Quite often, we want to convert each line of an output into an element in an array
- ▶ This used to be rather complicated, especially if we wanted to do it properly
- ▶ **Luckily, we can use the readarray (or mapfile) command:**
 - ▶ `readarray -t data < file.txt`
 - ▶ **Important:**
 - ▶ We need to be running Bash in version 4.0 or higher
 - ▶ **Important options:**
 - ▶ `-t`: Remove a trailing newline from each line read
 - (this is often useful to avoid having each array entry ending with a newline)
 - ▶ `-n count`: Copy at most count lines. If count is 0, all lines are copied
 - ▶ `-0 origin`: Begin assigning to array at index origin. The default index is 0
 - ▶ `-s count`: Discard the first count lines read

Bash & Linux CLI

Arrays: Check if element exists

Check if an element exists

- ▶ A common array operation is that we want to check if an array contains a certain element
- ▶ How can we do this?
- ▶ **Unfortunately, we need to write this ourselves:**
 - ▶

```
fruits=("apple" "banana" "passion fruit")
search_for="banana"
found=0
for fruit in "${fruits[@]}"; do
    if [[ "$fruit" == "$search_for" ]]; then
        found=1
        break
    fi
done
```

Check if an element exists

- ▶ This one does not work properly:

```
▶ if printf "%s\n" "${array[@]}" | grep -q "^${search_for}$"; then  
    echo "$search_for exists in the array."  
else  
    echo "$search_for does not exist in the array."  
fi
```

- ▶ In case the `search_for` term contains special characters (that are part of regular expressions), the regular expression will be executed
- ▶ More on regular expressions later!

Bash & Linux CLI

Bonus: Associative arrays

Bonus: Associative arrays

- ▶ Starting with Bash 4.0+, Bash supports associative arrays
- ▶ Associative arrays allow you to store **key-value** pairs
- ▶ An associative array in Bash somewhat resembles a HashMap / unordered dictionary
- ▶ **To declare an associative array:**
 - ▶ `declare -A array_name`
 - ▶ `declare -A student_scores`
- ▶ **We can then set values:**
 - ▶ `array_name[key]=value`
 - ▶ `student_scores["John"]=95`
- ▶ **Or access values:**
 - ▶ `echo "${student_scores["John"]}"`
- ▶ **To get all keys:**
 - ▶ `echo "${!student_scores[@]}"`
- ▶ **We can also check for the existence of a key:**
 - ▶ `[[-v "${student_scores["John"]}"]]`

Bonus: Associative arrays

- ▶ **To get the value to a variable key:**
 - ▶ `echo "${student_scores[$key]}"`
- ▶ **We can also check for the existence of a key:**
 - ▶ `[[-v student_scores["John"]]]`
- ▶ **We can also create an associative array with some initial values:**
 - ▶ `declare -A student_scores=(
 ["John"]=95,
 ["Lauren"]=99
)`

Bash & Linux CLI

Exercise: Arrays

Exercise: Directory monitor (arrays)

- ▶ In this chapter, we want to write a shell script that allows us to easily monitor the size of directories
- ▶ For this, we want to declare the paths to those directories in an array
- ▶ **And for each of those directories, we want to:**
 - ▶ Calculate the total size of this directory (with the du program)
 - ▶ Then, check if this size is above a defined threshold
 - ▶ And if this is the case, we want to print out the size of this directory
- ▶ **Let's have a look at the result!**

Bash & Linux CLI

Solution: Arrays

Bash & Linux CLI

Bonus: Project arrays

Bonus: Project arrays

- ▶ You have now implemented this directory monitor script through arrays
- ▶ But can we make those paths configurable through CLI options?
- ▶ **Let's have a look at 2 use cases:**
 - ▶ **Method 1:**
 - ▶ `./solution_path_1.sh [path1] [path2] [path3]`
 - ▶ Here, we can read the paths directly, as all the arguments to the script are provided in the array in \$@
 - ▶ **Method 2:**
 - ▶ `./solution_path_2.sh -d [path1] -d [path2] -d [path3]`
 - ▶ For this, we will need to use the getopts program in order to be able to parse the paths
- ▶ **Let's have a look at both!**

Bash & Linux CLI

Project: Backup script

Project: Backup script

► In this chapter:

- We want to develop a backup script
- For this, we will first investigate different ways to compress data,
and how we can influence this
(`.tar`, `.tar.gz`, `.tar.bz2`,...)
- Also, for the backup script, we will see how we can utilize arrays
in a Bash script to easily configure the folders that need to be
backed up

Archiving and compression

- ▶ Quite often, archiving and compression is combined
- ▶ This is the case with `.zip` files - they archive and compress at the same time
- ▶ Here, we will investigate those concepts separately first
- ▶ And then see how we can combine archiving and compression

Bash & Linux CLI

Archiving

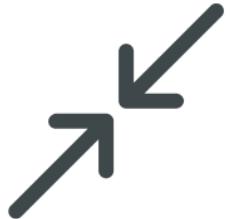
Archiving

- ▶ Archiving means that we have a bunch of files, and we want to combine them to a single file
- ▶ For this, we can archive them with the program tar
- ▶ **To create an archive:**
 - ▶ `tar -cf [archive] [files / folders to backup...]`
- ▶ **To list the contents of an archive:**
 - ▶ `tar -tf [archive]`
- ▶ **To extract an archive into the current working directory:**
 - ▶ `tar -xf [archive] [files to extract (optional)]`
- ▶ **To extract an archive to a different directory:**
 - ▶ `tar -xf [archive] -C [directory] [files to extract (optional)]`
- ▶ **If we want tar to be more verbose, we can also add the -v option:**
 - ▶ `tar -xvf [archive]`

Bash & Linux CLI

Compression

Compression



- ▶ **Compression:** Means that we have data, and we want to reduce its size without losing the original information.
- ▶ **Popular compression algorithms:**
 - ▶ **Using gzip (GNU zip):**
 - ▶ To compress: `gzip -k [file]` (produces `file.gz`)
 - ▶ `-k` stands for: Keep original file
 - ▶ To decompress: `gzip -d [file.gz]` or `gunzip [file.gz]`
 - ▶ **Using bzip2:**
 - ▶ To compress: `bzip2 -k [file]` (produces `file.bz2`)
 - ▶ To decompress: `bzip -d [file.bz2]` or `bunzip2 [file.bz2]`
 - ▶ **Using xz (uses LZMA compression):**
 - ▶ To compress: `xz -k [file]` (produces `file.xz`)
 - ▶ To decompress: `xz -d [file.xz]`

Which compression algorithm to use?

- ▶ **gzip:**

- ▶ Simple compression algorithm
- ▶ Widely adopted
- ▶ Safe choice for broad compatibility
- ▶ Weakness: Compression ratio is not that high

- ▶ **bzip2:**

- ▶ Usually offers a better compression ratio than gzip
- ▶ But more complex compression algorithm, thus slower

- ▶ **xz (LZMA compression):**

- ▶ High compression ratio (usually higher than bzip2)
- ▶ Decompression usually faster than bzip2
- ▶ **But:** Compression takes quite long
- ▶ Ideal when we're looking for the highest compression ratio

Bash & Linux CLI

Archiving and compression

Can we combine archiving and compression?

- ▶ **First try:**

- ▶ Let's manually try to create a .tar archive
- ▶ And add compression to it (let's say bz2)
- ▶ Then we end up at a .tar.bz2
- ▶ Let's have a look at this

- ▶ **But can't this be easier?**

Bash & Linux CLI

Archiving and compression: tar

Archiving and compression

- ▶ Because we often want to combine archiving and compression,
we can use build-in flags for tar for this
- ▶ **Using gzip with tar (-z flag):**
 - ▶ To create: `tar -czf [archive.tar.gz] [files/folders...]`
 - ▶ To extract: `tar -xzf [archive.tar.gz]`
- ▶ **Using bzip2 with tar (-j flag):**
 - ▶ To create: `tar -cjf [archive.tar.bz2] [files/folders...]`
 - ▶ To extract: `tar -xjf [archive.tar.bz2]`
- ▶ **Using xz with tar (-J flag):**
 - ▶ To create: `tar -cJf [archive.tar.xz] [files/folders...]`
 - ▶ To extract: `tar -xJf [archive.tar.xz]`

Bash & Linux CLI

Exercise: Creating a backup script



Exercise: Let's create a backup script

► Let's create a backup script:

- Let's now combine all of this into a backup script
- It should take several folders, and turn them into individual archives
- Those archives should also be tagged with the current date
- This will allow us to have daily snapshots of our data
- Also, we have a tool that generates a SQL dump of our database - we also need to backup this output
- **Let's have a look at this! 😊**

A screenshot of a macOS Finder window titled "shop". The window displays a list of files and folders with columns for Name, Date Modified, Size, and Kind. The "Name" column is sorted in ascending order. The "Date Modified" column shows times relative to the current date. The "Size" column indicates file sizes, and the "Kind" column identifies the file types. A "+" button is visible in the top right corner of the list area.

Name	Date Modified	Size	Kind
backup_db.sh	Today at 14:55	1 KB	Shell Script
> backups	Today at 14:50	--	Folder
customers_data	Today at 09:15	--	Folder
customers.csv	Today at 09:15	293 bytes	CSV Document
inventory_data	Yesterday at 18:17	--	Folder
inventory.csv	Yesterday at 18:11	208 bytes	CSV Document



Exercise: Backup script

- ▶ **Exercise: Create a script (`backup.sh`) that:**

- ▶ Backs up `customers_data` to `backups/customers-data-[month]-[day].tar.bz2`
- ▶ Backs up `inventory_data` to `backups/inventory-data-[month]-[day].tar.bz2`
- ▶ Backs up the output of the script to `backups/orders-[month]-[day].sql.bz2`:
 - ▶ `backup_db.sh`
 - ▶ Feel free to have a look at this script first, before executing
 - ▶ You might have to add a `chmod +x` in order to be able to execute this file
 - ▶ In practice, this would be a call to tools like `mysqldump` (instead of our own `.sh` file)

- ▶ **Also:**

- ▶ This script should also log each action into a log file: `backup.log`
- ▶ Also, it would be great if the folders (`customers_data`, and `inventory_data`) were configurable with an array at the beginning of the script
- ▶ If an error occurs, it should be logged to `stderr` (and to `backup.log`)!

Bash & Linux CLI

Creating a backup script (date)

Bash & Linux CLI

Solution: Creating a backup script (II)

Bash & Linux CLI

Solution: Creating a backup script (III)

Bash & Linux CLI

Solution: Creating a backup script (IV)

Bash & Linux CLI

The command-line tool grep



Motivation

► **Setting:**

- ▶ Let's say we want to search through a file and find all lines that contain a string
- ▶ How can we find those?
- ▶ How can we find out how often a pattern occurs in a text?

► **Later:**

- ▶ We will also be able to perform more complex searches with grep
- ▶ But for now, let's start with this simple search

Bash & Linux CLI

What is grep?

What is grep?

► What is grep?

- ▶ Command-line tool for processing plain-text data
- ▶ Searches and matches pattern within text files / streams to find relevant information

► Where does the name come from?

- ▶ grep stands for g/re/p (global/regular expression/print)
- ▶ g/re/p used to be a feature of the qed text editor - nowadays, its successor ed might still be installed on your system

Basic usage of grep

- ▶ The general usage of grep:

- ▶ Searching stdin:

- ▶ [command with stdout] | grep -F [pattern]

- ▶ Searching a file:

- ▶ grep -F [pattern] [file...]

- ▶ What are the parameters?

- ▶ **-F**: Disables regular expressions

- ▶ **[pattern]**: The pattern to search for

- ▶ **[file...]**:

- ▶ Files to search in for the pattern

- ▶ If omitted, grep will perform the search on stdin

- ▶ Best practice:

- ▶ Pattern in single quotes, we want to disable shell expansions here!

- ▶ Let's have a look at this!

How can we find help?

- ▶ grep has an enormous amount of available options
- ▶ Thus, in this course, I will limit it to the most important ones
- ▶ **If man files (build-in manuals) are installed on your system:**
 - ▶ `man grep`
- ▶ **Otherwise, you can always check the manual:**
 - ▶ <https://www.gnu.org/software/grep/manual/grep.html>

Be careful!

- ▶ grep is not equal to grep
- ▶ There're different implementations of grep
- ▶ They might work slightly differently or support different parameters / options
- ▶ This especially affects grep on macOS vs. Linux

Bash & Linux CLI

Important options for grep

Important options for grep

- ▶ **grep -F [pattern] / fgrep [pattern]:**
 - ▶ Disables regular expressions
 - ▶ For now, we want to set this option
- ▶ **grep -i [pattern]:**
 - ▶ Case-insensitive search
- ▶ **grep -n [pattern]:**
 - ▶ Prints the line numbers before each line
- ▶ **grep -o [pattern]:**
 - ▶ Prints only the matching part of each line, with an individual line for each matching part
- ▶ **grep -c [pattern]:**
 - ▶ Counts the number of lines in which a pattern occurs
 - ▶ **Important:** Each line is only counted once, even if the pattern can be found multiple times

Bash & Linux CLI

Additional options for grep

Important options for grep

- ▶ **grep -r [pattern]:**

- ▶ Recursive search: Searches in the current working directory and in all subdirectories
- ▶ But we can also specify custom paths after the pattern

- ▶ **grep -s [pattern]:**

- ▶ Suppress error messages about nonexistent or unreadable files

- ▶ **grep -l [pattern]:**

- ▶ Show only the names of files containing matches

- ▶ **grep -q [pattern]:**

- ▶ **Checks if a certain pattern is present:**

- ▶ If pattern is found => exit code 0
 - ▶ If not => exit code 1

- ▶ **grep --color [pattern]:**

- ▶ Highlight the match in color

Bash & Linux CLI

Introduction: grep & regular expressions

What are regular expressions?

- ▶ Regular expressions (often shortened to "regex") are tools that help us search for specific patterns within text
- ▶ **Unlike a basic search, we can search for patterns:**
 - ▶ Phone numbers
 - ▶ Email addresses
 - ▶ Anything we want
- ▶ Regular expressions are extremely versatile and commonly used
- ▶ You might have used them in other programming languages already
- ▶ **Let's have a look at an example:**
 - ▶ `grep -E '^[[:space:]]@+[[:space:]]+' [file]`
 - ▶ This would be a simple regular expression to find email addresses
(not extremely accurate, but might be good enough)
 - ▶ We will understand more of the syntax later

Types of regular expressions

- ▶ **BRE:**

- ▶ Basic regular expressions
- ▶ The default syntax in grep
- ▶ Defined through the POSIX standard

- ▶ **ERE:**

- ▶ Extended regular expressions
- ▶ We can switch to this syntax by using `-E`
- ▶ Also defined through POSIX

- ▶ **PCRE:**

- ▶ Perl compatible regular expressions
- ▶ Used in many other programming languages
- ▶ Supported by GNU grep, but not by BSD grep
- ▶ This means they work on Linux, but not on Mac
- ▶ We can switch to this syntax by using `-P` (if supported)

In this course

- ▶ In this course, we will focus on BRE and ERE
- ▶ **The reason:**
 - ▶ They are supported by `grep` on all platforms
 - ▶ Also, the tests from Bash (`[[...]]`) only support ERE
 - ▶ `if [["${email}" =~ [^@]+@[^@]+]]; then`
 - ▶ More on regex + tests in a later lecture!
- ▶ **Also:**
 - ▶ PCRE support almost the same features as ERE with only minimal syntax differences
 - ▶ The main difference is that PCRE supports additional features, that are not available in ERE

Bash & Linux CLI

Writing our first regular expression (BRE)

Writing our first regular expression (BRE)

- ▶ **We can already write simple regular expressions:**

- ▶ `grep 'romeo' shakespeare.txt`
- ▶ This already runs a regular expression!
- ▶ However, we're not using any special characters yet that influence the pattern search

- ▶ **Let's start using the first metacharacter (special character):**

- ▶ `.`:
- ▶ Matches any single character except a newline

Bash & Linux CLI

Regular expressions (BRE): Additional metacharacters

Additional metacharacters in BRE

- ▶ ^:
 - ▶ Matches the beginning of a line
- ▶ \$:
 - ▶ Matches the end of a line
- ▶ \;:
 - ▶ Used to escape a metacharacter so it's treated as a literal
(or to introduce special sequences)
- ▶ **Example:** \.
 - ▶ This would only match the normal "."

Bash & Linux CLI

Regular expressions (BRE): The * quantifier

The * quantifier

- ▶ Quite often, we want to allow repetitions
- ▶ How could we write a regular expression, that matches all of those?
 - ▶ hello
 - ▶ helllllo
 - ▶ helllllllllo
- ▶ For this, we need to allow the `I` to repeat
- ▶ We can do this by using a quantifier
 - ▶ `*`:
 - ▶ Matches zero or more occurrences of the preceding character (or group)
- ▶ **Important:**
 - ▶ Regular expressions are greedy
 - ▶ They try to match as much as possible!

Bash & Linux CLI

Regular expressions (BRE): Character classes

Character classes

- ▶ **We can define our own character class:**

- ▶ [aeiou]: Would match any vowel
- ▶ [abc]: Would match either the letter a, b, or c

- ▶ **Example:**

- ▶ echo "cpp" | grep 'c[+p][+p]'
- ▶ echo "c++" | grep 'c[+p][+p]'
- ▶ echo "c+p" | grep 'c[+p][+p]'

- ▶ **We can also combine it with quantifiers:**

- ▶ echo "cpp" | grep 'c[+p]*'
- ▶ Would also match: cccccccc
- ▶ Would not (completely) match: cccccccccc

Bash & Linux CLI

Regular expressions (BRE): Character ranges

Character ranges & negating character set

- ▶ We can also use the range operator to build our own character class:

- ▶ [0-9]:

- ▶ Would match any single character between 0-9
 - ▶ Meaning: It would match any single digit

- ▶ [a-zA-Z]:

- ▶ Would match any single character between a-z or A-Z

Bash & Linux CLI

Regular expressions (BRE): Negating a character class

Negating a character class

- ▶ We can also negate a character set:

- ▶ [^ab]:

- ▶ Would match any single character that is not an a or a b

- ▶ [^0-9]:

- ▶ Would match any single character that is not a digit

- ▶ [^0-9_]:

- ▶ Would match any single character that is neither a digit nor an underscore

Bash & Linux CLI

Regular expressions (BRE): Named character classes

Named character classes

- ▶ Luckily, we don't have to build the character classes ourselves
- ▶ We can also just utilize named character classes
- ▶ We can use them within the square brackets of a normal character class
- ▶ **Most important named character classes:**
 - ▶ [:digit:] -> [0-9]
 - ▶ [:lower:] -> [a-z]
 - ▶ [:upper:] -> [A-Z]
 - ▶ [:alpha:] -> [[:lower:][:upper:]] -> [a-zA-Z]
 - ▶ [:alnum:] -> [[:digit:][:lower:][:upper:]] -> [0-9a-zA-Z]
 - ▶ [:blank:] -> Space or tab character
 - ▶ [:space:] -> Blank character (tab, whitespace, line break, carriage return \r,...)

Bash & Linux CLI

Regular expressions (BRE): Character groups

Character groups

- ▶ We can also define character groups
- ▶ **In BRE, we can do this with the following syntax:**
 - ▶ `\(...\)`
- ▶ They allow us to more clearly specify what we want to match
- ▶ **Example:**
 - ▶ `grep '[cC]\(pp\)*\(\+\+\)*'`
- ▶ **All of those would match:**
 - ▶ C, cpp, c++, Cpp++, cccccccccccc
- ▶ **But those would not match:**
 - ▶ c+p
 - ▶ cpp++

Bash & Linux CLI

Regular expressions: ERE vs. BRE

ERE vs. BRE

- ▶ **So far, we only had a look at BRE (basic regular expressions)**
- ▶ **Now, we're ready to have a look at ERE:**
 - ▶ Extended regular expressions
 - ▶ They extend the features of POSIX BRE
- ▶ **If we want to enable ERE, we need to use the flag -E:**
 - ▶ `grep -E [ERE]`

Important: ERE vs. BRE

- ▶ **Important:**

- ▶ ERE enables a different syntax mode for our regular expression
- ▶ Also, additional features are supported
- ▶ Luckily, most of the syntax is identical

- ▶ **Though character groups are defined in a different way:**

- ▶ This is great, because the new syntax is simpler to read
- ▶ **BRE:** \(...\)
- ▶ **ERE:** (...)

- ▶ **Meaning:**

- ▶ **BRE:**

- ▶ `grep '[cC]\(pp\)*\(\+\+\)*'`

- ▶ **ERE:**

- ▶ `grep -E '[cC](pp)*(\+\+\+)*'`

Bash & Linux CLI

Regular expressions (ERE): Alternations

Regular expressions (ERE): Alternations

- ▶ **Alternation:**

- ▶ The | symbol allows you to match one of several alternatives
- ▶ This is extremely useful

- ▶ **Example:**

- ▶ `hello (world|mars)!`

- ▶ **Would match:**

- ▶ `hello world!`

- ▶ `hello mars!`

Bash & Linux CLI

Regular expressions (ERE): Additional quantifiers

Regular expressions (ERE): Additional quantifiers

- ▶ In ERE, we can use additional quantifiers, to specify how often the character class / group before should be repeated:
 - ▶ ?:
 - ▶ Matches **zero or one** repetition
 - ▶ +:
 - ▶ Matches **one or more** repetitions
 - ▶ {n}:
 - ▶ Matches exactly n repetitions
 - ▶ {n,m}:
 - ▶ Matches n to m repetitions
 - ▶ {n,}:
 - ▶ Matches n to more repetitions

Bash & Linux CLI

Regular expressions (ERE): Word boundaries

Word boundaries

► To match a word boundary:

- ▶ **\<:**
 - ▶ Matches the empty string at the beginning of a word
- ▶ **\>:**
 - ▶ Matches the empty string at the end of a word
- ▶ **\b:**
 - ▶ Matches the empty string at the edge of a word
 - ▶ Technically, this is a feature of the PCRE regular expressions
 - ▶ It also works in GNU grep though

Bash & Linux CLI

Regular expressions: Using ERE features in BRE

Using ERE features in BRE

- ▶ Luckily, in most implementations of grep, we can easily use ERE features in BRE as well
- ▶ **We just need to escape the metacharacters:**
 - ▶ **Example (ERE):**
 - ▶ `grep -E 'hello (world|mars) !?'`
 - ▶ **Example (BRE):**
 - ▶ `grep 'hello \(\world\|\mars\)\ !\?'`
 - ▶ **Here:**
 - ▶ `\(\)` always creates a character group in BRE
- ▶ **Optional features (usually supported, but not part of POSIX)**
 - ▶ `|` for the alternation is usually supported in BRE implementations if we put a backslash in front of it: `\|`
 - ▶ The quantifier `?` is not supported by BRE by default. We can also use it by putting a backslash in front of it: `\?`

Important

► **Important:**

- ▶ Technically, this BRE now contains features that are not specified in the POSIX standard for BRE
- ▶ Though in this case, most grep implementations will support this
- ▶ You should test this manually though and be aware of potential problems

Bash & Linux CLI

Bonus: Using ERE in Bash tests:

```
[[ ... =~ ... ]]
```

Regular expressions: ERE in Bash

- ▶ Btw, when testing in a Bash condition, we can also use an extended regular expression:

- ▶ **[["\${variable}" =~ pattern]]:**

- ▶ Exit code is 0, if the variable on the left matches the ERE pattern on the right. Otherwise, the exit code is 1

- ▶ **Example:**

```
▶ #!/usr/bin/env bash
    read -p "Enter an email address: " email
    if [[ "$email" =~ ^.*@.*\..*\$ ]]; then
        echo "Valid email address structure."
    else
        echo "Invalid email address structure."
    fi
```

Bash & Linux CLI

Bonus, regular expressions: PCRE

Regular expressions: PCRE

- ▶ We can also enable PCRE regular expressions in certain implementations of grep
- ▶ **PCRE:** Perl-Compatible Regular Expression
- ▶ **This is usually possible with the GNU implementation of grep:**
 - ▶ `grep -P [pattern]`
 - ▶ The pattern is then treated as a PCRE regular expression
 - ▶ Most of the syntax is identical to ERE
 - ▶ Though additional features are supported
(such as lookaheads, lookbehinds,...)
- ▶ **Also:**
 - ▶ We're now using a different regex engine (PCRE)
 - ▶ This can lead to slightly different result in certain cases
 - ▶ But we now can use PCRE regex patterns from other programming languages

PCRE: Additional named character classes

- ▶ In PCRE, we got additional character classes
- ▶ **The most important ones are:**
 - ▶ **\d:** Matches a digit, equivalent to [0–9]
 - ▶ **\D:** Matches any character that is not a decimal digit
 - ▶ **\s:** Matches any whitespace character. Typically, this is a space, tab, newline, carriage return, vertical tab, and the form feed
 - ▶ **\S:** Matches any character that is not a whitespace character
 - ▶ **\w:** Matches any "word" character, typically: [a-zA-Z0-9_]
 - ▶ **\W:** Matches any character that is not a word character
- ▶ **Important:**
 - ▶ These character sets might also work within an ERE or BRE
 - ▶ This depends on the implementation of grep

Bash & Linux CLI

Bonus: Why PCRE?

Lookaheads & lookbehinds

PCRE features: Lookaheads

► Positive lookahead:

- ▶ `(?=pattern)`

► Example:

- ▶ `echo "I have 5 apples and 3 oranges." | grep -oP '\d+(?= apples)'`
- ▶ This will match the number(s) that come before the string " apples".

► Negative lookahead:

- ▶ `(?!pattern)`

► Example:

- ▶ `echo "I have 5 apples and 3 oranges." | grep -oP '\d+(?! apples)'`
- ▶ This will match the number(s) that do not come before the string " apples".

PCRE features: Lookbehinds

► Positive lookbehind:

- ▶ (?=<pattern>)

► Example:

- ▶ echo "I have 5 apples and 3 oranges." | grep -oP '(?=<have)\d+'
 - ▶ This will match the number (and the number only) that comes after the word "have"

► Negative lookbehind:

- ▶ (?<!pattern)

► Example:

- ▶ echo "I have 5 apples and 3 oranges." | grep -oP '(?<!have)\d+'
 - ▶ This pattern will match numbers that aren't directly after "have "

Why different syntax?

- ▶ For lookbehinds, we need a different syntax - they're computationally quite expensive
- ▶ Thus, the regex engine needs to be able to differentiate between lookahead and lookbehind
- ▶ Also, additional constraints (such as a maximum length of lookbehind,...) might be imposed by the regex engine

Bash & Linux CLI

Project: Analyzing logfiles

Exercise / Project: Analyzing logfiles

- ▶ **Project:**

- ▶ We now want to practice regular expressions on some more "hands-on" data

- ▶ For this, we will analyze the logfile of a webserver*

- ▶ ***technically:**

- ▶ Due to privacy restrictions, I was unable to use a real-world dataset for this

- ▶ I thus have generated a log file with completely random data

- ▶ It is in the same format as the log format of the apache (httpd) webserver

- ▶ **Let's have a look at the logfile!**

Questions: Logfile

- ▶ **Questions:**
 - ▶ **How many requests have been answered with status "200" (successfully)?**
 - ▶ **How many GET requests have been issued to .zip files?**
 - ▶ For this, check for the pattern: GET [...] .zip
 - ▶ (you need to rewrite this into a proper regular expression)
 - ▶ **Can we extract the IP addresses only?**
 - ▶ Ensure that your code works with IPv4 and IPv6 addresses!
 - ▶ Does the same IP address ever send multiple requests to our server? Or do all requests come from a unique IP?
 - ▶ **What is the most requested URL from Firefox browsers?**
 - ▶ You can detect the Firefox browsers in the user agent
 - ▶ It needs to contain "Firefox"

Important

► **Important:**

- ▶ The goal is to only get a general overview over the data
- ▶ We do not want to fully parse the data
- ▶ Quite often it's enough to write a regular expression that works 99%+ of the time
- ▶ This should be also your goal in this exercise

► **Also:**

- ▶ Be sure to combine them! For some answers, you need more CLI tools than just `grep`

Bash & Linux CLI

Solution: Analyzing logfiles

Bash & Linux CLI

Downloads files

Downloading files

► In this chapter:

- You will learn how you can use the program `wget` to download files
- Even if you know `curl`...
- ...for certain download operations, `wget` is just way more convenient
- We will also learn about checksums, and how they can help you to verify your download

Bash & Linux CLI

The program: wget

The program: wget



- ▶ **The program GNU wget:**

- ▶ It's a utility to non-interactively download files from the internet
- ▶ Free & Open-Source Software (GPL license)
- ▶ We can use it to download files from the internet
- ▶ It can run in the background without user intervention
- ▶ We can pause and resume a download (the server we're downloading from must support this)
- ▶ We can even download entire websites

- ▶ **We might have to install it!**

- ▶ As usual, it depends on our system...

- ▶ **For example:**

- ▶ **Mac:** brew install wget
- ▶ **Ubuntu:** sudo apt-get install wget

Main differences: wget vs. curl

Feature	wget	curl
Main purpose	Downloading files non-interactively	Data transfer with various protocols
Protocols supported	HTTP, HTTPS, FTP	HTTP, HTTPS, FTP, SFTP, SMTP, POP,...
Recursive downloading	Yes, can download entire websites or directories	No, only single files or specified targets
Upload capability	No	Yes, can upload files using supported protocols
Command-line options	More than 100 options	Over 200 options
Library (irrelevant for bash)	No	libcurl, a multiprotocol transfer library
HTTP/2 support	No	Yes

Bash & Linux CLI

wget: CLI options

wget: CLI options

- ▶ **What are the most important CLI options?**

- ▶ **-O, --output-document:**
 - ▶ Specifies a local file for the downloaded file
- ▶ **-c, --continue:**
 - ▶ Resumes an incomplete or interrupted download from where it left off
- ▶ **-b, --background:**
 - ▶ Run in background, does create a log file by default
- ▶ **-q, --quiet:**
 - ▶ Do not generate any output

Bash & Linux CLI

Bonus: Batch download with wget

Batch download with wget

- ▶ With wget, we can also download all links from a .txt-file
- ▶ This is perfect for batch downloading many files
- ▶ **For example:**
 - ▶ You want to deploy your application(s) to a server
 - ▶ Or you just want to download multiple files with a single command
- ▶ **How do we do it?**
 - ▶ We can use the parameter `-i, --input-file=FILE`, in order to download a list of files!
 - ▶ **Let's have a look at how this works!**

Bash & Linux CLI

Bonus: Download complete website with wget

Download website with wget

► **Important:**

- ▶ wget does not understand JavaScript
- ▶ It can only parse the HTML code from the website
- ▶ Interactive JavaScript is not being executed!
- ▶ Modern, interactive websites might break or not download properly!

Download whole webpage with wget

- ▶ Let's download all images from a website!

- ▶

```
wget https://downloads.codingcoursestv.eu/055%20-%20bash/wget/folder/ -r -l 5 -np -A jpg -e robots=off
```
- ▶ **-r, --recursive:**
 - ▶ Enables recursive downloading, following links to download entire directories or websites.
- ▶ **-l, --level:**
 - ▶ Sets the maximum depth of recursion, limiting how deep wget will follow links (by default: 5)
- ▶ **-np, --no-parent:**
 - ▶ Restricts downloads to the specified directory and its subdirectories, avoiding parent directories.
- ▶ **-A, --accept=jpg:**
 - ▶ Comma-separated list of accepted extensions. Only those links will be followed / downloaded
- ▶ **-e robots=off:**
 - ▶ Ignores robots commands, that might prevent wget from following links

Download whole webpage with wget

- ▶ We can also download a whole webpage with wget!
 - ▶ wget <https://downloads.codingcoursestv.eu/055%20-%20bash/wget/gym/> -r -l 5 -np -e robots=off -p -k
 - ▶ -p, --page-requisites:
 - ▶ Downloads all page requisites, such as images, stylesheets, and scripts, for offline viewing.
 - ▶ -k, --convert-links:
 - ▶ Converts links in downloaded HTML files to local references, for offline browsing.

Bash & Linux CLI

Checksums

Checksums

- ▶ **Checksums:**

- ▶ They allow us to verify the download

- ▶ **One of the (technical) goals:**

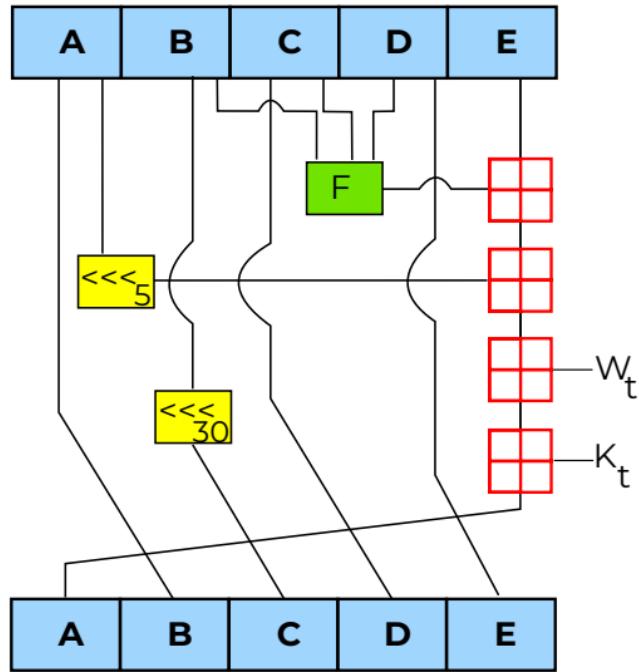
- ▶ If one bit in the input changes...
 - ▶ the whole output changes

- ▶ **There're several different checksum algorithms:**

- ▶ md5
 - ▶ sha-1
 - ▶ sha-2 (sha-224, sha-256, sha-384, sha-512)
 - ▶ sha-3

- ▶ **Attack vector:**

- ▶ Can we construct a change in a specific way, so that the algorithm still generates the same checksum?
 - ▶ md5, sha-1 are the most vulnerable



Bash & Linux CLI

Advanced Bash features

Advanced Bash features

- ▶ **In this chapter:**

- ▶ We will have a look at advanced Bash features

- ▶ **Such as:**

- ▶ Changing a directory without needing cd
 - ▶ Correction of errors
 - ▶ Extended globbing
 - ▶ Errors when globbing didn't succeed
 - ▶ How to work with the Bash history

Bash & Linux CLI

Bash: Changing the behavior of cd

Bash: Changing the behavior of cd

- ▶ In Bash, we can set 2 main options to influence the behavior of cd:

- ▶ **autocd:**

- ▶ Allows directory names to be entered without the cd command

- ▶ **To enable:**

- ▶ `shopt -s autocd`

- ▶ **cdspell:**

- ▶ Autocorrects minor spelling errors in the cd command

- ▶ **To enable:**

- ▶ `shopt -s cdspell`

- ▶ Let's have a look at those!

Bash & Linux CLI

Bash: Customizing globbing

Customizing globbing

- ▶ If we want to change the behavior of globbing, those options are especially useful:

- ▶ **globstar:**

- ▶ Enables the use of `**` in pathname expansions to match files and directories recursively

- ▶ **To enable:** `shopt -s globstar`

- ▶ **dotglob:**

- ▶ Include filenames beginning with a dot (`.`) in pathname expansion results

- ▶ **To enable:** `shopt -s dotglob`

- ▶ **failglob:**

- ▶ If set, patterns which fail to match during pathname expansion will result in an error

- ▶ This is especially useful, as otherwise the original value will be used

- ▶ **To enable:** `shopt -s failglob`

- ▶ **nocaseglob:**

- ▶ Case-insensitive pattern matching during pathname expansion

- ▶ **To enable:** `shopt -s nocaseglob`

Bash & Linux CLI

Bash: Extended globbing

Bash: Extended globbing

- ▶ Bash also supports extended globbing
- ▶ This further allows us to influence how globbing should behave

- ▶ **First, we need to enable extended globbing:**

- ▶ `shopt -s extglob`

- ▶ **After this, we can use it:**

- ▶ **?(pattern1|pattern2|...):**

- ▶ Matches zero or one occurrence of the given patterns

- ▶ ***(pattern1|pattern2|...):**

- ▶ Matches zero or more occurrences

- ▶ **+(pattern1|pattern2|...):**

- ▶ Matches one or more occurrences

- ▶ **@(pattern1|pattern2|...):**

- ▶ Matches one of the given patterns

- ▶ **!(pattern1|pattern2|...):**

- ▶ Matches anything except the given patterns

Bash & Linux CLI

Bash: Grouping commands

Bash: Grouping commands

- ▶ We can also group commands into blocks:

- ▶ { command1; command2; }:
 - ▶ Group commands without creating a subshell
- ▶ (command1; command2;):
 - ▶ Group commands, and execute them in a subshell

- ▶ Why is this important?

- ▶ We can combine it with other features from our shell

```
▶ {  
    read line1  
    read line2  
    echo "First Line: $line1"  
    echo "Second Line: $line2"  
} < file.txt
```

Bash & Linux CLI

Bash: Working with the history

Bash: Working with the history

- ▶ We can inspect our history with the `history` command:

- ▶ **`history`:**

- ▶ List the command history with line numbers

- ▶ **`history -c`:**

- ▶ Clear the entire command history

- ▶ **`history -d n`:**

- ▶ Delete the history entry at position n

Bash: Persistent history

- ▶ The history is persistent across sessions
- ▶ **How does this work?**
 - ▶ Commands are saved in `~/.bash_history` by default
 - ▶ We can change this in `HISTFILE`
 - ▶ To control the size of the history: `HISTSIZE`
- ▶ **Important:**
 - ▶ By default, this file will be overwritten
 - ▶ But we can also tell Bash to just append to this file:
 - ▶ `shopt -s histappend`

Changing what is stored in history

- ▶ **Controlling history behavior:**

- ▶ **export HISTCONTROL=ignoredups:**

- ▶ Avoids saving duplicated commands consecutively

- ▶ **export HISTCONTROL=ignoreboth:**

- ▶ Avoids saving duplicated commands and commands that start with a space

- ▶ **export HISTIGNORE="ls:cd:exit":**

- ▶ Ignores the specified commands (in this case: ls, cd, and exit)

How can we work with the history?

- ▶ **To access the history:**
 - ▶ **[Arrow key up] / [CTRL] + [P]:**
 - ▶ Recall the previous command in history
 - ▶ **[Arrow key down] / [CTRL] + [N]:**
 - ▶ Recall the next command in history
- ▶ **Let's have a look at this!**
- ▶ **But we can also use the history expansion for this!**

History expansion

- ▶ **History expansion:**

- ▶ **!!:**

- ▶ Execute the last command

- ▶ **!n:**

- ▶ Execute the command numbered n in the history list

- ▶ **!-n:**

- ▶ Execute the command that's n lines back

- ▶ **!string:**

- ▶ Execute the most recent command starting with "string"

- ▶ **!?string?:**

- ▶ Execute the most recent command containing "string"

- ▶ **^[search]^ [replace]:**

- ▶ Execute the last command again

- ▶ But replace [search] will be replaced with [replace]

Bash & Linux CLI

Bash: Reacting to signals (trap)

Bash: Reacting to signals (trap)

- ▶ If we write our own bash script
- ▶ Sometimes we want our script to react to signals

- ▶ **Example:**

- ▶ If our script receives a SIGINT ([CTRL] + C) signal or SIGTERM ([CTRL] + Z, kill), we want to execute a cleanup function

```
▶ cleanup() {  
    echo "Cleaning up..."  
}  
  
trap cleanup SIGINT SIGTERM  
  
while true;  
  
    do sleep 1  
  
done
```

- ▶ This can be useful for cleaning up temporary files

- ▶ **Let's have a look at this! ☺**



Bash & Linux CLI

Zsh: A great alternative to Bash

What is Zsh?



- ▶ Zsh, or the Z shell, is a Unix shell that was introduced in 1990

- ▶ **It's a great alternative to Bash:**

- ▶ Large overlap between Zsh and Bash
- ▶ Certain features are significantly improved
- ▶ Additional customization options
- ▶ Enhanced interactive features
- ▶ **Also:** Both shells can coexist on the same system

- ▶ **This means:**

- ▶ We can use Zsh as our main shell
- ▶ And still write Bash scripts for our shell scripts

The popularity of Zsh

- ▶ Zsh is becoming increasingly popular
- ▶ It's used by ~10-20% of all Unix shell users
- ▶ **It's also the default shell for a few operating systems:**
 - ▶ **macOS:**
 - ▶ Starting with macOS Catalina, Zsh became the new default shell
 - ▶ Due to licensing reasons: Zsh is released under the MIT license,
Bash is released under the *GNU General Public License (GPL)*
 - ▶ **Kali Linux, Parrot OS:**
 - ▶ Also, some Linux distributions made Zsh their default shell
 - ▶ **Most other Linux distributions:**
 - ▶ Zsh is usually available through the default repositories
 - ▶ We can install it through the package manager
 - ▶ We can also set it as a default shell



How do I use Zsh?

► In my opinion:

- ▶ I love the customization options of Zsh
- ▶ Also, I can recommend using "Oh My Zsh"
- ▶ The features for common, basic CLI usage are almost identical
(even slightly improved)
- ▶ Though more advanced features (variables, advanced
expansions,...) are quite different

Bash & Linux CLI

Zsh: Installation

Zsh: How to install

- ▶ To install Zsh, we can just use the corresponding install command of our Linux distribution:
 - ▶ Ubuntu:
 - ▶ apt install zsh
 - ▶ CentOS:
 - ▶ dnf install zsh
- ▶ We might want to manually configure the Zsh shell:
 - ▶ autoload -Uz zsh-newuser-install
 - ▶ zsh-newuser-install -f
 - ▶ If this does not work - be sure to maximize your terminal / zoom out!
- ▶ If we want to set it as a default shell:
 - ▶ chsh -s \$(which zsh)
 - ▶ I would recommend waiting with this
 - ▶ It's best to explore Zsh a bit more first

Bash & Linux CLI

A first look into Zsh

A first look into Zsh

- ▶ Now that we've installed Zsh
- ▶ Let's now explore some basic features
- ▶ Let's open the shell and navigate to some directories!

Bash & Linux CLI

Zsh: Configuration files

Configuration files



Configuration files

- ▶ **/etc/zshenv, .zshenv:**

- ▶ Always the first to be read
- ▶ Used for essential configuration (EDITOR, PATH...)
- ▶ Also loaded for scripts

- ▶ **/etc/zprofile, .zprofile:**

- ▶ Loaded for login shells

- ▶ **/etc/zshrc, .zshrc:**

- ▶ Loaded for interactive shells

- ▶ **/etc/zlogin, .zlogin:**

- ▶ Loaded for login shells (after zshrc)

- ▶ **.zlogout, /etc/zlogout:**

- ▶ Loaded in login shells on logout
- ▶ Allows us to unload configuration



Best practice

- ▶ I put all my configuration into the `~/.zshrc` file
- ▶ **Reasoning:**
 - ▶ I'm not using Zsh for scripts anyway
 - ▶ Then, my configuration is loaded for all interactive shells
 - ▶ This is enough for me!



Bash & Linux CLI

Zsh: Oh My Zsh

Oh My Zsh

- ▶ **Oh My Zsh is an open-source tool to enhance your Zsh experience:**
 - ▶ It's the main reason why I prefer Zsh over Bash as my shell
- ▶ **Designed to be easily customizable:**
 - ▶ **themes** for styling the terminal
 - ▶ **plugins** to provide enhanced support
- ▶ **Installation:**
 - ▶ You need to follow the install guide from Oh My Zsh
- ▶ **By default:**
 - ▶ Your existing .zshrc will be backed up
 - ▶ And a new one will be created



Bash & Linux CLI

Bonus: **Zsh vs. Bash: Breaking differences (echo)**

Different behavior of echo

- ▶ The echo command works differently!
- ▶ **echo "Hello\nworld":**
 - ▶ **Bash:**
 - ▶ **Prints:** Hello\nworld
 - ▶ **Solution 1:** echo -e "Hello\nworld"
 - ▶ **Solution 2:** echo \$'hello\nworld'
 - ▶ **Zsh:**
 - ▶ **Prints:**
 - ▶ Hello
 - ▶ world
 - ▶ **To disable this behavior:**
 - ▶ echo -E "Hello\nworld"
 - ▶ **Prints:** Hello\nworld

Bash & Linux CLI

Bonus: Zsh vs. Bash Breaking differences (Expansions)

Important

- ▶ The most common expansions work the same...
 - ▶ ... (at least most of the time)
- ▶ But some expansions work completely differently

Zsh vs. Bash: String expansions

- ▶ Example: Turning a string into uppercase:

- ▶ Bash:

```
▶ message='hello world'  
echo "${message^^}"
```

- ▶ In Zsh, we can just use an expansion flag for this (U):

```
▶ message='hello world'  
echo "${(U)message}"
```

Bash & Linux CLI

Bonus: Zsh vs. Bash: Breaking differences (Arrays)

Different Array Indexing

- ▶ Let's say we got an array:

- ▶ fruits=("apple" "banana" "passion fruit")
- ▶ echo "\${fruits[1]}"

- ▶ In Bash:

- ▶ Will print out: banana

- ▶ In Zsh:

- ▶ Will print out: apple

- ▶ The reason:

- ▶ In Bash, Array indexes start at [0]
- ▶ In Zsh, they start at [1]

Bash & Linux CLI

Bonus: Zsh: Additional command (repeat)

The command repeat

- ▶ The repeat command is a unique feature in Zsh

- ▶ **It allows us to easily repeat a command:**

- ▶

```
repeat [COUNT] {
    # commands to be executed
}
```

- ▶ It's not natively available in Bash

- ▶ **In Bash:**

- ▶ We would need to use a `for` loop

- ▶ **But repeat in Zsh is just way more convenient!**

- ▶ **Let's have a look at this! ☺**

Bash & Linux CLI

Bonus:

Zsh: Extended globbing

Zsh: Extended globbing

- ▶ Zsh further extends our globbing capabilities
- ▶ **We can enable them:**
 - ▶ `setopt EXTENDED_GLOB`
- ▶ **For example, we can use qualifiers in a glob:**
 - ▶ They are unique to Zsh, and not available in Bash
 - ▶ They allow us to qualify our globs with attributes of the files
 - ▶ **(.)**: Matches regular files
 - ▶ **(/)**: Matches directories
 - ▶ **(@)**: Matches symbolic links
 - ▶ **(U)**: Files owned by me
 - ▶ **(^U)**: Files not owned by me
 - ▶ **(*)**: Executable files
 - ▶ ... and many more
- ▶ **Example:** `print -- *.txt(.)`

Bash & Linux CLI

Bonus: Zsh: Shell scripts?

Zsh Scripts

- ▶ **We can execute a script with Zsh**
- ▶ **For this, we can just use a different shebang:**
 - ▶ `#!/bin/zsh`
 - ▶ `#!/usr/bin/env zsh`
- ▶ **Be careful:**
 - ▶ Bash is usually seen as the standard for shell scripts
 - ▶ Using Zsh for shell scripts is not considered best practice
- ▶ **Important: Ask yourself the question first:**
 - ▶ Do you really want to deviate from this?
- ▶ **Also:**
 - ▶ `bash script.sh` will of course always execute a script through Bash
 - ▶ And `zsh script.sh` will do the same for Zsh
 - ▶ No matter the shebang!

Bash & Linux CLI

Final words

Final words

► **You have now learned quite a bit:**

- ▶ You can use Bash on the command line
- ▶ You understand how Linux works under the hood
- ▶ You can automate tasks with Bash scripts

► **All in all:**

- ▶ You now have a great understanding of Linux
- ▶ You now can work with the command line - no matter the task!

**The goal has been reached:
You can now confidently use Linux**

Thank you

- ▶ I want to thank you for watching and buying this course
- ▶ This really means a lot to me and my team
- ▶ And it enables me to create such a comprehensive online course
- ▶ **Thank you for enabling me to do that!**