Linux, command line & MetaCentrum

Use of Linux command line not only for CESNET's MetaCentrum

Vojtěch Zeisek

Department of Botany, Faculty of Science, Charles University, Prague
Institute of Botany, Czech Academy of Sciences, Průhonice
https://trapa.cz/, zeisek@natur.cuni.cz

January 15 to 18, 2024

Outline I

1. Introduction
   - Learning machine
   - What it is a “UNIX”
   - Licenses and money
2. Linux
   - Generally about Linux
   - Choose one
   - Differences
3. UN*X
   - Disks and file systems
   - Types of users
   - Directory structure
   - Files and directories
   - Permissions

Outline II

Text

4 Command line
- BASH and other shells (“command lines”)
- Screen
- SSH — secure shell and screen
- BASH
- Directories
- Archives
- Searching
- Globbing, wildcards, quotes
- Variables
- Input, output and their redirecting
- Information and processes

Outline III

Network
Parallelization
Timing

5 Text
Reading
Extractions
AWK
Manipulations
Compressed text
Comparisons
Editors
Regular expressions

6 Scripting

Outline IV

Basic skeleton
BASH variables
Functions
Reading variables
Branching the code
Loops

7 Software
Packages
Compilation
Java
Windows applications
Scientific applications
Games

Outline V

8 MetaCentrum
- Information
- Usage
- Tasks
- Graphical connection
- Archive data storage
- More services

9 Git
- Git principle
- Git basics

10 Administration
- System services

11 The End

Outline VI

Resources
The very end

Introduction

First steps in the world of Linux and open-source software

1 Introduction
Learning machine
What it is a “UNIX”
Licenses and money

The course information

- The course page: https://trapa.cz/en/course-linux-command-line-2024
  - Česky: https://trapa.cz/cs/kurz-prikazove-radky-linuxu-2024
- Subject in SIS: https://is.cuni.cz/studium/eng/predmety/index.php?do=predmet&kod=MB120C23
  - Česky: https://is.cuni.cz/studium/predmety/index.php?do=predmet&kod=MB120C23
  - For students having subscribed the subject, requirements are on next slide
- Working version of the presentation is available at https://github.com/V-Z/course-linux-command-line-bash-scripting-metacentrum — feel free to contribute, request new parts or report bugs

Requirements to exam ("zápočet")

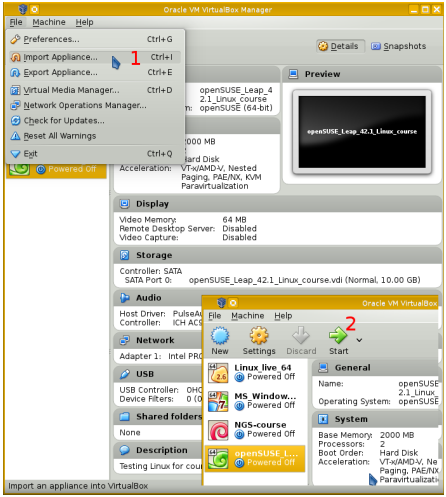
1. Be present whole course.
2. Be active — ask and answer questions.
3. Write short script solving any task the student is (going to be) solving. E.g. prepare script to process student's data on MetaCentrum. Or short script to do anything the student needs to do. This will be very individual. According to topics and interests of every student. Students can of course discuss with anyone, use Internet, manuals, etc. The aim is to learn how to solve real problem the student has/is going to have.
4. Write at least one page (can be split into multiple articles) on Wikipedia about any topic discussed during the course. Again, this is very open, students can write about any topic they like. I prefer native language of the student (typically to make larger non-English Wikipedia).

Materials to help you...

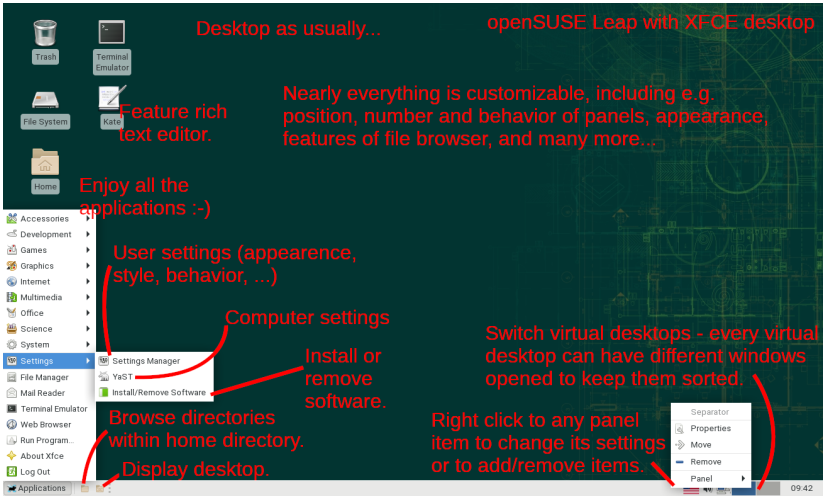
- Download the presentation from https://soubory.trapa.cz/linuxcourse/linux_bash_metacentrum_course.pdf
- Download the scripts and toy data from https://soubory.trapa.cz/linuxcourse/scripts_data.zip
  - Note: Open the scripts in some good text editor (slide 159) — showing syntax highlight, line numbers, etc. (NO Windows notepad); the files are in UTF-8 encoding and with UNIX end of lines (slide 76; so that too silly programs like Windows notepad won't be able to open them correctly)
  - Never ever open any script file in software like MS Word — they destroy quotation marks and other things by "typographical enhancements" making the script unusable

Virtual machine for learning

- If you do not have Linux installed, download and install VirtualBox from https://www.virtualbox.org/
- Download openSUSE Leap 15.3 Linux distribution for this course from https://botany.natur.cuni.cz/zeisek/openSUSE_Leap_courses.ova (∼4.6 GB)
- Launch VirtualBox and go to menu **File | Import appliance...** to import it. When done, launch it (**Start**)
- See also https://trapa.cz/en/vm-for-courses

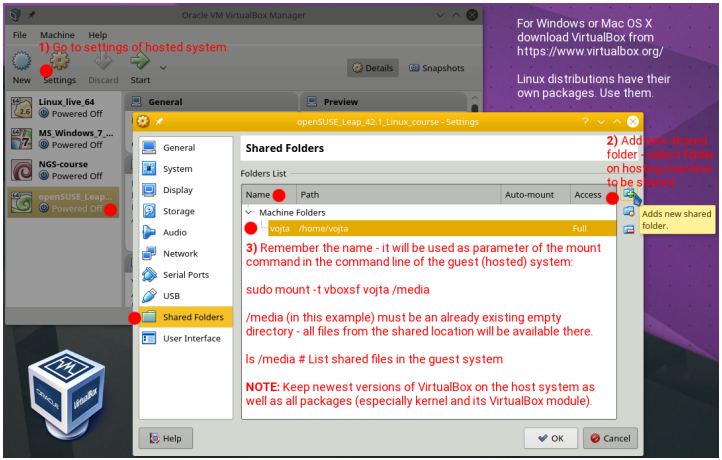


Enjoy learning virtual machine for the course



VirtualBox shared folder I

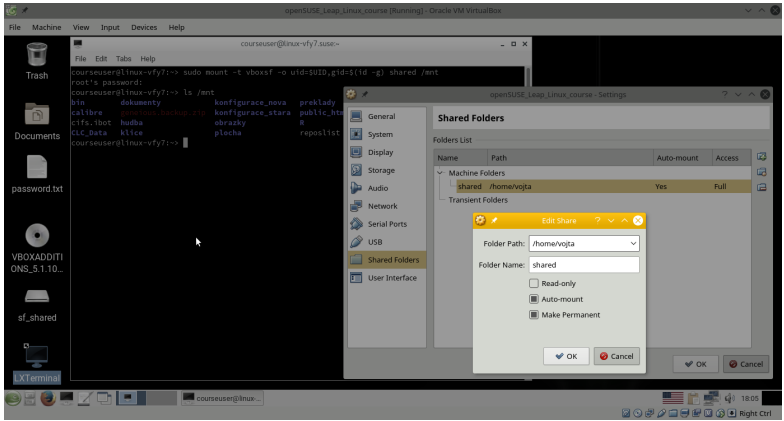
VirtualBox can be configured to share folder with host operating system



VirtualBox shared folder II

Go to menu "Devices | Shared Folders" and set pair of folders

```
sudo mount -t vboxsf -o uid=$UID,gid=$(id -g) shared /mnt
```



What it is UNIX, Linux and GNU I

- UNIX
  - Originally developed in Bell labs of AT&T in 1969, written in C, since then rich evolution...
  - Trademark — only systems passing certain conditions (paid certification) can be called "UNIX" — Solaris, HP-UX, AIX, etc. (commercial systems for big servers; nowadays rare)
  - **Main principles:** simple, multitasking, hierarchical structure, network, for more users (takes care about permissions etc.), configuration written in plain text files, important relationships among applications (generally one application = one task — they are chained), work primarily with text, has kernel and API (interface to communicate with the rest of the system)
- UNIX-like (UN*X, *nix)
  - Systems compatible with UNIX (**Linux**, BSD and its variants, **macOS**, ...)
  - Mainly open-source (UNIX is commonly commercial — source code is not publicly available, but its specification is)
  - Nowadays prevailing over "old" UNIX systems, used in many devices from tiny embedded toys to huge data centers
  - Try to provide same quality as paid systems, but (mostly) for free

What it is UNIX, Linux and GNU II

- Many courts about copyrights, parts of code, patents (USA allow software patents, EU not) — GNU, Linux, BSD, etc. try to ensure to have only code not covered by any patents to avoid possible legal issues
- GNU
  - "GNU's Not Unix!" — but it is compatible, respects its principles
  - System written from scratch, following ideas of UNIX
  - Since 1984 Richard Stallman (founder of Free Software Foundation) tried to make new kernel (Hurd — not finished yet...)
  - Generally set of basic system tools — working with many kernels (Linux, BSD*, macOS, ...), also present in many commercial paid UNIX systems
  - Source code is free and open — anyone can study it (Security!), report bugs, contribute, modify, share it, get inspiration, ...
  - GNU General Public License (GPL) — free spirit of open-source — license, idea, how to share software to benefit everyone

What it is UNIX, Linux and GNU III

- Inspired open public software development — crucial for our usage of Linux & al.

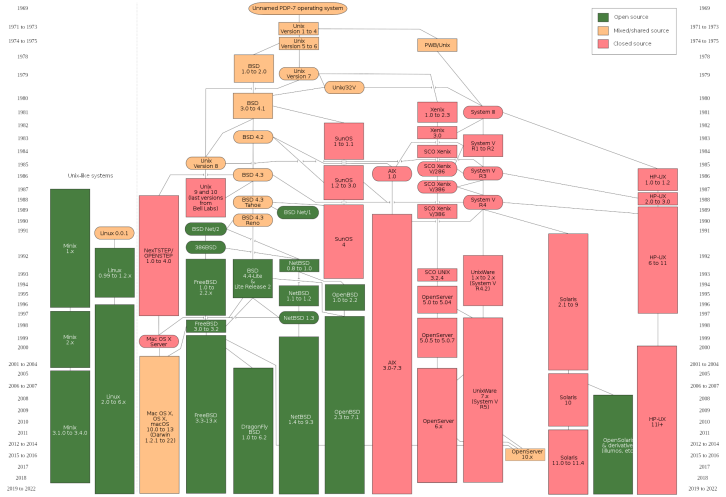
- Linux
  - First version of kernel (core of the system) written by Linus Torvalds in Helsinki in 1991
  - Kernel was in principle inspired by various UNIX systems and using GNU tools for standard work
  - Quickly became popular — anyone can take it and use for any needs, adopt (modify it), etc.
  - Used in small embedded (commonly network) devices, mobile devices (book readers, Android, ...), personal computers, servers (from home level to biggest data centers), ...
  - Nowadays powering most of the Internet
  - Anyone can contribute — not only code, also documentation, design, translations, ...
  - Most of people working with UNIX are using Linux (or macOS)
  - Powering everything from little IoT toys and devices to 100% of top 500 supercomputers

- **GNU tools** and **Linux kernel** are two important (but not sole) pieces of building set forming modern operating system — **"Linux distribution"**

What it is UNIX, Linux and GNU IV

- "Linux" nowadays usually mean Linux kernel + GNU basic tools (thus correctly "GNU/Linux") + thousands of various another applications (= "Linux distribution") — ecosystem of (free) open-source GNU/Linux applications of various origin forming together what we usually call simply "Linux"
- Term "Linux" has various meaning (see further)
  - If we mean only Linux kernel or whole distribution
  - If we talk about little embed Internet of Things (IoT) device, some mobile device, personal computer, server, supercomputer, ...
  - We can talk about particular Linux distribution (see further)/use case, or generally...

Extremely simplified UNIX phylogeny



Most common UNIX-based systems (except Linux)

- Apple macOS (previously Mac OS X) — system kernel is based on older BSD and uses plenty of GNU tools (although mostly older outdated versions)
- BSD — one of the oldest operating systems, still developed in many independent variants
  - Still very popular especially on servers, for special purposes, etc.
  - License allows closing of the code — used by Apple macOS kernel, PlayStation firmware, ...
  - Installation and management is for beginners usually harder than Linux, everything must be done manually, not so common as Linux anymore
  - E.g. FreeBSD, DragonFly BSD, OpenBSD, TrueNAS (for storage servers), ...
- Solaris — commercial, not very common anymore
  - Mainly special servers, paid
  - Several community-based variants freely available
- Command line usage is nearly same across UN*X systems — all follow same standards, use more or less same set of tools

Cathedral vs. market place

What is principal difference between free open-source and commercial software

- Commercial software is like a cathedral
  - Pay big money and get it in the state which the architect like
  - User can not modify it (or it is terribly expensive)
  - Might be you don't need everything — but still paying whole set
- Free open-source software (FOSS) is like a market place
  - Find there many producers of same tools — pick up those you like — freedom of choice
  - Take exactly the tools you need — any combination is possible
  - Much cheaper to shop there
- Both have pros and cons — depends what you wish...
- Consider e.g. difference between usage of full-featured and expensive Geneious (theoretically everything you need to process genetic data) vs. searching for hundreds of free tools and pipelines on the Internet (GitHub, R packages, fora, ...)
- According to book by Eric S. Raymond

Free and open-source software — (F)OSS I

- **Free like freedom of speech, not like free beer!**
- Open-source — source code can be seen by the holder of the license (not necessary by everyone — can be paid)
- Not every OSS (generally less strict conditions) has to be **F**OSS (you can do with it (almost) whatever you like) — source code might be available under some circumstance (only to look), but modification and/or reuse of the code prohibited (and then it is not **free**)
- GNU GPL ("copyleft") — probably most common OSS license, strict, viral — derived code has to keep the license — surprisingly not fully "free" as it doesn't allow changes of license
- LGPL — Lesser GPL — more permissive
- BSD license — permissive — allow derived code to became closed-source (commonly used by e.g. Apple macOS, Safari browser, small electronics, ...)
- Apache or Mozilla licenses etc. — specific use in particular software

Free and open-source software — (F)OSS II

- Creative Commons (CC) — software licenses are not suitable for multimedia, text, etc. — CC has many options (including denial of reuse of the product), see `https://creativecommons.org/`
- And many more licenses...

Spirit of FOSS

- Orientation might be tricky, but practical output for users is same — **the software can be independently checked for bugs, backdoor, malware, can be improved and under some circumstances, new software can be derived, and usually, it is available for free**
- Aim is to "liberate" software to keep open sharing of ideas, mutual improve and security control
  - Although the point is clear, there are debates how to reach it...

Free and open-source software — (F)OSS III

- Practical output of using open-source licenses for the user
  - Software can be used anywhere
  - Software can be modified — including various fixes of older code to work properly, or for different systems (author worked e.g. on Linux and someone else will do version for macOS or Windows)
  - User can learn from the software (from the code), new software can be developed on top of it
  - Easier to find and trace bugs
  - Security — no backdoors, possibility to do various audits, screenings, etc.
  - Bugs (problems) in the code can fixed by nearly anyone
  - Often available for free (e.g. on services like GitHub or SourceForge)
  - Easier distribution of the software (can be copied into various repositories, like those of Linux distributions, Conda or Homebrew)

How to make money with free open-source software?

- Traditional model — user rents right ("buys a license") to use the software (and sometimes for support — usually for extra money)
- Common mistake — software is not "bought" — only license is rented ("permission to use it" with many limitations)
- Software as service
  - (F)OSS is available for free — user can use it as it is or buy a support — help of any type
  - No vendor lock-in — user has the code, so he can modify the software himself, change provider of the services, ...
  - Cheap for user as well as company — company specialized for one task, let's say server database, doesn't have to take care about the rest of the system — someone else does; user pays only what he needs
  - Our faculty is using Plone system for web pages — anyone can use it for free, someone (like we) asked a company to help, and if we'd decided, we could keep Plone and maintain it ourselves or find another company to help us with it

Linux

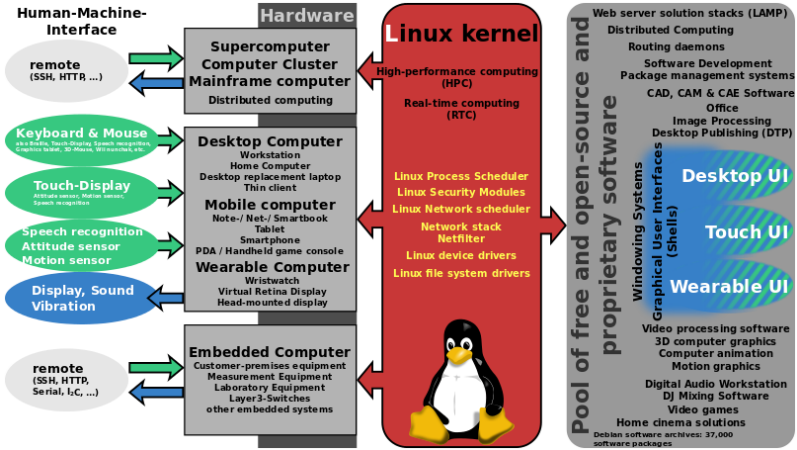
Generally about Linux

2 Linux
Generally about Linux
Choose one
Differences

What it is a “Linux”

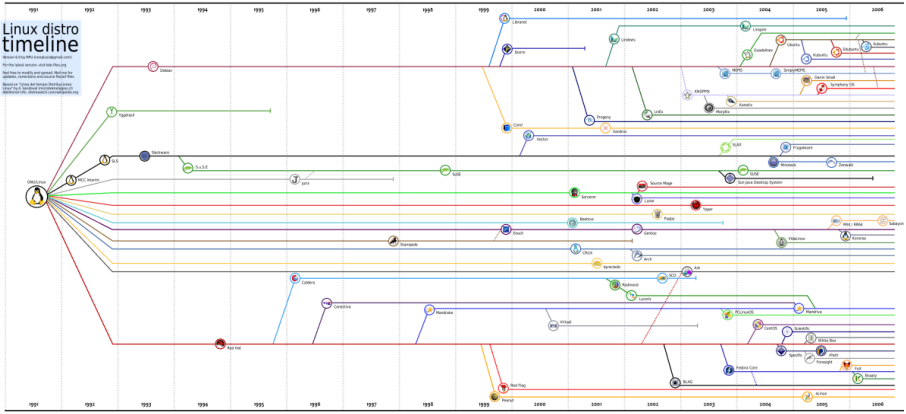
- Operating system respecting principles of UNIX
- Components
  - Linux **kernel** — basic part of the system responsible for hardware and very basic low-level running of the system (“**Linux *sensu stricto***”)
  - GNU core utilities — basic applications
  - Graphical user environment (GUI) — many choices
  - Many other applications — according to use — whatever imaginable
- Linux **distribution (“Linux *sensu lato*”)**
  - Somehow assemble Linux kernel, basic tools and some applications
  - Optionally add some patches and extra tools and gadgets
  - Make your own design! (very important;-)
  - If lazy, remake existing distribution (using e.g. web service)
  - Still surprised there are hundreds of them?
  - It is like Lego — pieces are more or less same across distributions, but result is very variable
  - From “general” for daily use (pick up whatever you like) to very specialized — special hardware devices, network services, rescue, ...

Linux kernel and other parts around it



https://en.wikipedia.org/wiki/Linux_distribution

Extremely simplified adaptive radiation of Linux distributions



https://en.wikipedia.org/wiki/List_of_Linux_distributions and https://distrowatch.com/ (∼900 distributions, ∼250 active)

Most common Linux distributions

- Debian (DEB) based
  - Debian — one of oldest and most common, especially on servers
  - Ubuntu (nowadays probably the most popular on PCs and notebooks) and derivatives — Kubuntu, Xubuntu, Lubuntu, ... (according to GUI used — most of the system is same)
  - Mint — Based on Ubuntu as well as Debian, user-friendly, popular
  - Kali, KNOPPIX, elementaryOS, Pop!_OS, MX Linux, Zorin OS, ...
- Red Hat (RPM) based
  - Red Hat — probably the most common commercial
  - Fedora — "playground" for Red Hat — very experimental
  - Centos — Clone of Red Hat
  - openSUSE — SUSE is second largest Linux company, openSUSE is community distribution (free) companion of SUSE Linux Enterprise
  - Mageia, PCLinuxOS, ...
- Manjaro, Solus, ...
- Android (practically Linux kernel, touch interface & Java)
- For experienced users: Arch, Slackware, Gentoo, ...

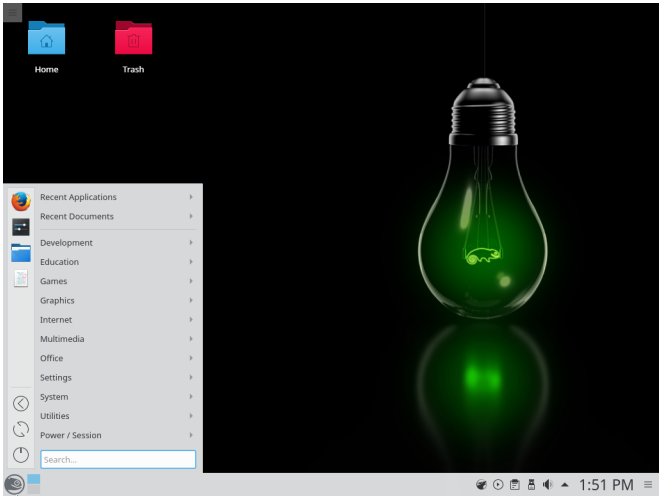
Graphical User Interfaces (GUI)

More like “Mac-style”, “Windows-style” or something else? Feature rich or minimalistic?

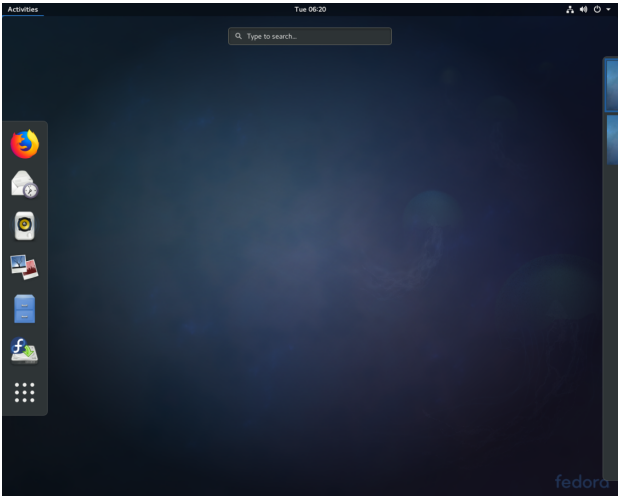
- Most of GUIs are available for most of common distributions — one is picked as default and “only” color style is different
- KDE — one of the most common, feature extremely rich, basically “Windows-like” (can be changed), extremely versatile
- GNOME — one of the most common, relatively simplistic interface, but still feature rich, “Mac-like”
- XFCE — lightweight version of older GNOME — for older computers or users not willing to be disturbed by graphical effects, basically “Mac-like” looking, but panels can be moved to “Windows style”
- Cinnamon — remake of GNOME to look more like Windows...
- Unity — originally developed by Ubuntu (discontinued, now community based), “Mac-style”
- And much more...
- Choose what you like — doesn’t matter much which one...

Ubuntu with GNOME

openSUSE with KDE — Kubuntu is same, but blue...

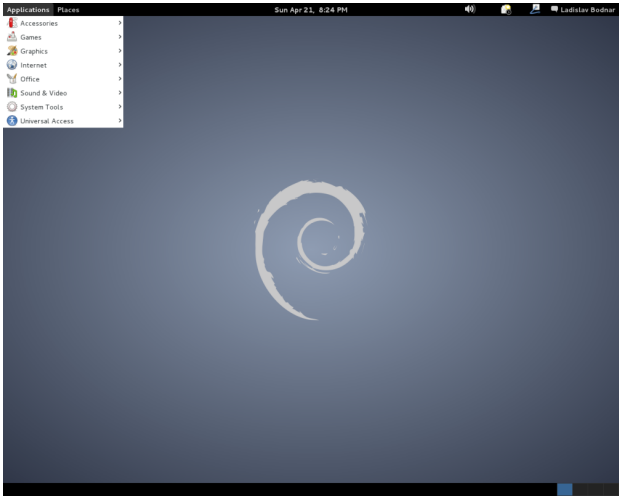


Fedora with GNOME — GNOME is always almost same

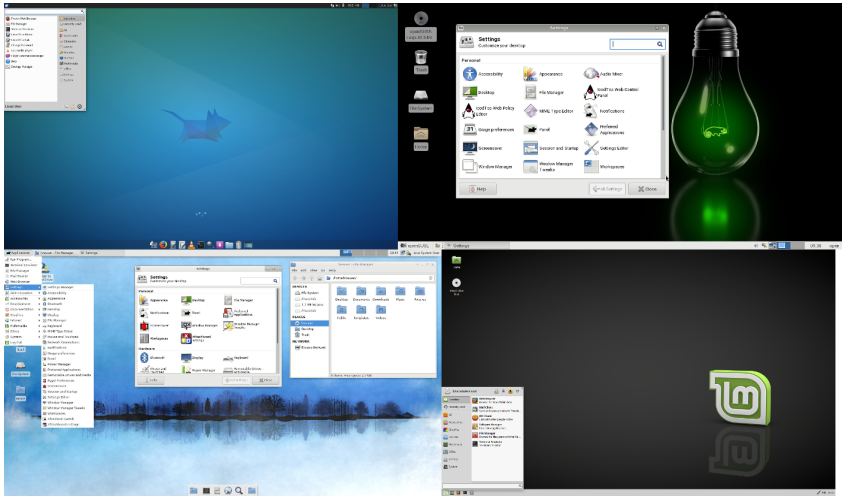


Linux Mint with Cinnamon

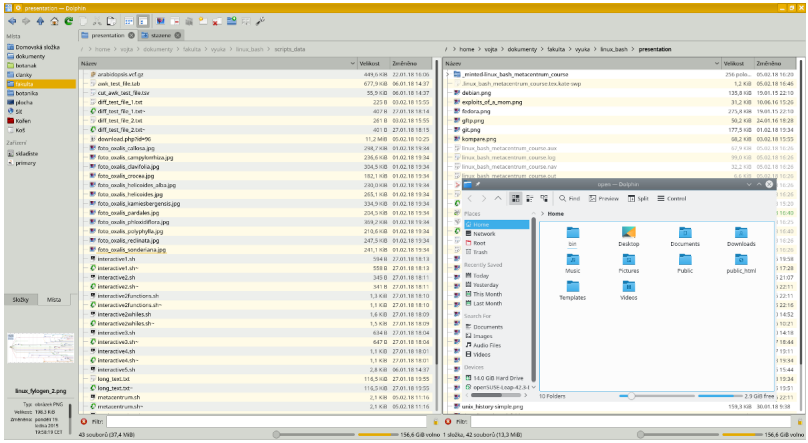
Debian with XFCE — Xubuntu has more “modern” design



XFCE in Xubuntu, openSUSE, Fedora and Linux Mint



Dolphin (KDE file manager) — default for openSUSE (inset) and after tuning in the same distribution...



How to try Linux? I

- If unsure what to do, ask someone for help. ;-)
- Install it on some computer together with or instead of Windows
  - If you can use whole disk, just boot from CD/USB and click “Next”...
  - If you don’t have whole disk, you need at least one (commonly more) disk partition(s) — if you don’t know how to manage them, ask someone skilled...
  - Major updates of Windows 10 or Windows reinstall use to destroy bootloader and it is not possible to start Linux anymore (although it can be usually easily fixed)
- Live CD/USB
  - The most easy — burn ISO image of CD from web of almost any Linux distribution or use for example UNetbootin to prepare bootable flash
  - You only have to know how to boot from CD/USB (usually press `ESC`, `DEL`, `F2`, `F10`, `F12`, ... when starting computer — varies according to manufacturer)
  - Commonly done to rescue data from broken Windows installations... ;-)
- Virtualization (slide 12)
  - Requires powerful computer — Intel i7 or AMD Ryzen, over 12 GB of RAM and hundreds of GB

How to try Linux? II

- Install virtual machine (probably the most easy is VirtualBox) — allows install and run another operating system inside host as an ordinary application — very easy and comfortable

- Linux subsystem in MS Windows Store (Windows 10 and 11)
  - To install follow `https://docs.microsoft.com/windows/wsl/about`
  - Version 1 only for command-line applications (it has some problems with paths, text files, ...)
  - Version 2 works well for most of tasks and allows GUI (experimental)
- Cygwin
  - Download and install from `https://www.cygwin.com/`
  - It is not native Linux, it is collection of GNU and open-source utilities compiled to work on Windows as more or less standard Windows applications
  - Follows POSIX standards (i.e. it works like normal UNIX command line, with all features)
  - Every application must be specially compiled to be able to work under Cygwin (it is sometimes complicated)
  - Collection is large, include also GUI and DE, but not everything is easy working
  - Installation of custom software might be very hard...

The Linux diversity...

- Try several distributions and just choose one you like...
- If selecting among the most common, it doesn't matter much which one you pick up
- Which design do you like?
- Which distribution is your friend or colleague using? To have someone to ask for help...
- You can change GUI (or its design) without change of distribution, it use to be highly configurable
- Applications are still same — no difference in Firefox across distributions — keep your settings when changing distribution
- Everyone using Android is using Linux — install some terminal emulator there :-)
- Special use — TrueNAS for home as well as business file servers, Parted Magic and/or SystemRescue to repair broken system (disk failure) and save data, ...

Differences among (common) Linux distributions

- Design and colors ;-)
- Default GUI (others can be installed)
- Applications available right after installation
- Default settings (not much)
- Package management — especially in command line
- Development model — conservative or experimental, fast or slow
- Management of system services (how to start/stop certain services like database or web server) — not important for daily usage for most of users
- Sometimes in location of some system files (can be customized) — also not important in daily usage of most of users
- Kernel is almost same, applications are same and used in same way...
- Command line is almost same across Linux, and almost same as in other UNIX systems (macOS, ...)

UN*X

Theory and principles of UNIX-based operating systems

3 UN*X
- Disks and file systems
- Types of users
- Directory structure
- Files and directories
- Permissions
- Text

Short overview of hard disk layout

- Physical disk (piece of hardware) has at least 1 partition — division seen in Windows as "disks" (`C` , `D` , ...) and mounted directory in UNIX
- MBR — older description of disk division, up to 4 primary partitions (OS typically requires at least one to run), one can be extended and contain more partitions, disks up to 2 TB
- GPT — newer, no relevant limits, requires UEFI (replacement of BIOS — responsible of start of nowadays computers)
- If unsure what to do, high probability to break it...
- Blank new partition has to be formatted to desired file system according to use and target operating system
- Linux distributions have easy graphical tools to manage disk partitions (e.g. GParted)
- Always have backup before such management!

Comparison of file systems (limits and compatibility)

FS name	Name length	Signs in name	Path length	File size	Partition size	Supported systems
FAT32	255	Unicode	No limit defined	4 GiB	2 TiB	Any
exFAT	255	?	No limit defined	16 EiB	64 ZiB	Any
NTFS	255	Variable	Variable	16 TiB	16 EiB	Windows (UN*X)
HFS+	255	Unicode	Unlimited	8 EiB	8 EiB	macOS
ext4	255	Any, not /	No limit defined	16 TiB	1 EiB	Linux (UN*X)
XFS	255	Any	No limit defined	9 EiB	9 EiB	Linux (UN*X)
Btrfs	255	Any	No limit defined	16 EiB	16 EiB	Linux (only?)
ZFS	255	Unicode	No limit defined	16 EiB	256 ZiB	UN*X

- FAT32 (including extensions) is old-fashioned and not reliable FS, but still common in various flash disks and memory cards
- NTFS (basic Windows FS) and FAT do not support UNIX permissions, so they can't be used as system partition in Linux; see also full comparison
- Btrfs, ext4, XFS and ZFS are not accessible from Windows at all (Linux mainly uses ext4)
- Btrfs, XFS and ZFS are the most advanced FS in common use

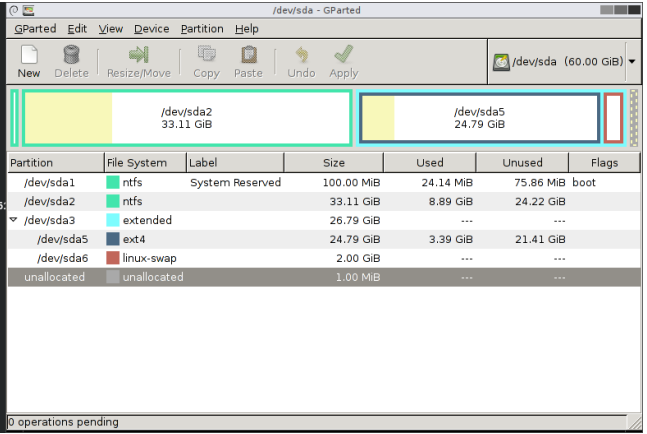
Mounting and unmounting disks and removable media

- Mounting and unmounting of devices require root privileges
- In modern desktop Linux distributions, mounting is done automatically and media are visible mostly in `/media` or `/run/media` (and show some icon, widget or so in desktop environment)
- In Linux, physical disks are named from `sda` to `sdz`, each disk has partitions (at least one) numbered from `1`, (`sda1`, `sda2`, `sdb1`, ...), all are in `/dev` (`/dev/sdc3`)

```
eject # Open CD/DVD drive
mount # Which FS (disk partitions) are mounted
findmnt # See mounted devices in tree-like structure
mkdir /mnt/point # Empty directory must exist prior mounting into it
# mount usually recognize FS of mounted device, if not, add '-t FS_type'
mount /dev/sdXY /mount/directory # Mount disk sdXY to /mount/directory
mount -t iso9660 -o loop file.iso /mnt/iso # Mount CD/DVD ISO file
umount /dev/sdXY # Unmount disk sdXY, alternatively use below command
umount /mount/directory # Unmount disk from /mount/directory
```

Creation and control of FS I

```
$ sudo fdisk -l
[sudo] password for root:
Disk /dev/sda: 894.26 GiB, 960197124096 bytes, 1875385008 sectors
Disk model: Corsair Force LE
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disklabel type: gpt
Disk identifier: 55C22C41-DBE7-4E05-8139-FC47E5E342F4

Device       Start        End    Sectors   Size Type
/dev/sda1     2048     819199     817152   399M Microsoft basic data
/dev/sda2   819200    1124351     305152   149M EFI System
/dev/sda3  1124352 1875384319 1874259968 893.7G Linux LVM


Disk /dev/mapper/cr_ata-Corsair_Force_LE_SSD_162980210001055502 00-part3: 893.73 GiB, 9596
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/mapper/pocitac-odkladiste: 8.72 GiB, 9349103616 bytes, 18259968 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/mapper/pocitac-korinek: 885 GiB, 950261514240 bytes, 1855979520 sectors
Units: sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes


Disk /dev/sdb: 931.53 GiB, 1000204886016 bytes, 1953525168 sectors
```



Creation and control of FS II

- Work in command line or use graphical tool like GParted...
- All commands require root privileges (slide 54)
- `fdisk -l` lists disks and partitions
- To manage disk partitioning use `fdisk /dev/sdX` or `gdisk /dev/sdX`
- When hard drive is partitioned, partitions must be formatted in next step
- Commands `mkfs.*` create various FS, common syntax is `mkfs.XXX -parameters /dev/sdXY`, where sdXY is particular disk partition
- Parameters can set label and various settings of behavior of the disk partition, check `man mkfs.XXX`
- To check FS for errors use `fsck.XXX /dev/sdXY` (according to respective FS)
  - The filesystem must be unmounted when checking it
  - XFS uses `xfs_repair /dev/sdXY`

Creation and control of FS III

- Btrfs uses `btrfs check /dev/sdXY`, if it is unmountable,
  `btrfs-zero-log /dev/XY` use to help, last instance is
  `btrfs check --repair /dev/sdXY` (dangerous operation)
- If Btrfs is mountable, but there are various FS errors and/or performance issues,
  `btrfs scrub start -Bdf /mount/point`,
  `btrfs filesystem defragment -r -v /mount/point` and
  `btrfs balance start -v /mount/point` — manual running can take long time and strongly slow down the computer

- `tune2fs -parameters /dev/sdXY` can set various parameters to influence behavior of disk (labels and more) partition
- `hdparm -parameters /dev/sdX` can set advanced hardware parameters of hard drive

Creation and control of FS IV

- The most convenient is using graphical tools available in all distributions...
  - In openSUSE there is YaST administrative module — from command line launch `yast --qt partitioner` for graphical or `yast disk` for text-based version
  - All distributions have graphical tools like GParted where it is possible to comfortable manage disks
- `df -h` shows available/occupied space on disks/partitions, but because of special features of Btrfs it doesn't show every time correct values for this FS — it is better to use `btrfs filesystem df /mount/point` (`/mount/point` use to be the most commonly `/`)
- On UNIX FS, defragmentation and another maintenance tasks use to be done in background when computer is idling — unless there is at least ∼20% of free space on the device, this is not any problem and there are no performance issues
- Users must be sure what they are doing, otherwise system can be damaged

Another manipulations and information

- `dd` is powerful, but potentially dangerous tool used to backup or write disks or partitions (commonly to create bootable USB media)
- If writing disk image to the disk (`sdX`), disk's partition table is discarded and the disk is covered by whatever is in the ISO image

```
dmesg # Recent entries in main system log - filter with grep, tail, ...
dmesg | grep sd | tail # Get information about recently plugged media
# dd produces physical copy of whole device - including empty space
dd if=/dev/sdX of=image.iso # Backups disk sdXY to imago.iso
dd if=image.iso of=/dev/sdX # Used to write e.g. image of Linux live
                            # media to USB flash disk (Check sdX!)
lnav # Comfortably browse recent logs, quit by "q"
```

- If there are encrypted partitions, they are in `/dev/mapper/...`
- If LVM (slide 53) is used, see `lvscan` and `pvscan` to find correct location in `/dev/`
- Disks are also accessible through `/dev/disk/by-<TAB><TAB>`

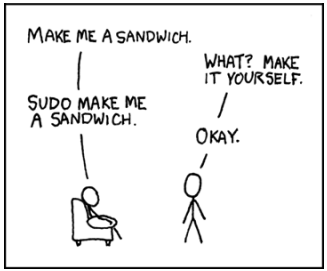
Put together more disks

Extend space and get higher data security

- RAID — Redundant Array of Inexpensive/Independent Disks
- RAID 0 — stripping, no redundancy, no security, speed up (two or more disks joined into one, files divided among disks)
- RAID 1 — mirroring — even number of disks of same size — resulting capacity is half, very fast, secure
- RAID 5 — at least three disks, one is used for parity control (in RAID 6 two disks are used for parity control), little bit slower, popular in cheaper storage servers (NAS)
- Combinations (RAID 10, ...)
- LVM — Logical Volume Management — built over several partitions/disks — seen by OS as one continuous space, can be dynamically managed
- Functionality of RAID and LVM (and more) is more or less covered by Btrfs and ZFS

Root vs. “normal” user

- Root is administrator — more than God (of the server) — can do anything
- Other users have limited permissions
  - System users providing particular service (web server, database, networking service) are as restricted as possible to do the task — security
  - “Human” users don’t have access to system files (at least not for modification), homes of users are separated



https://xkcd.com/149/

Becoming root

- Root privileges are required for any administrative task (install of software package, change of system settings, ...)
- Word “root” is used as name for system administrator user, and also for top of filesystem directory hierarchy (`/`)

```
# Gain root privileges
su # Requires root password (stay in current directory)
su - # Requires root password (go to /root)
su -c "some command" # Launch one command with root permissions
su USER # Became USER (USER's password is required)
sudo -i # For trusted users, became root (asks for user's password)
        # User (or group) has to be listed in /etc/sudoers
sudo somecommand # Launch somecommand with root's privileges - can be
                 # restricted for particular commands; /etc/sudoers can
                 # contain special settings for particular users/groups
cat /etc/passwd # See all users (including system users)
```

Directory structure in Linux I

- It is similar also in another UN*X systems
- Directory structure in Linux is similar to macOS (but there it is bit hidden, users usually don't see everything), but very different from Windows "logic"
- Top directory " `/` " — "root"
- Everything else (including disks and network shares) are mounted in subdirectories (`/...`)
- `/bin` — very basic command line utilities
- `/boot` — bootloader responsible for start of system
- `/dev` — devices — representations of disks, CD, RAM, USB devices, ...
- `/etc` — system configuration in plain text files — edit them to change system-wide settings (read documentation and comments there)
- `/home` — users' homes

Directory structure in Linux II

- `/lib`, `/lib64` — basic system libraries (32 and 64bits)
- `/lost+found` — feature of FS, after crash and recovery of FS, restored files are there
- `/media` — attached disks (USB flash, ...) usually appear there (might be in `/var/run/media` or `/var/media`) — subdirectories are automatically created when device is plugged and disappears when unplugged
- `/mnt` — usually manually mounted file systems (but they can be mounted elsewhere according to needs)
- `/opt` — optional, usually locally compiled software
- `/proc` — dynamic information about system processes
- `/root` — root's (admin's) home
- `/run` — temporal ID files (locks) of running processes

Directory structure in Linux III

- `/sbin` — basic system utilities
- `/selinux` — SELinux is security framework
- `/srv` — FTP and WWW server data (can be in `/var/srv`)
- `/sys` — basic system
- `/tmp` — temporary files — users have private dynamically created spaces there, used automatically by applications according to need
- `/usr` — binaries (executable applications) and libraries of installed applications
- `/var` — data of most of applications and services, including e.g. database data, system logs, ...
- `/windows` — if on dual boot, Windows disks are commonly mounted here

Directory structure in Linux IV

- In newest Linux distributions, most of `/bin`, `/etc`, `/lib`, `/lib64` and `/sbin` are mostly in `/usr` (original locations are just links)
- Can be altered, modified — especially on various servers
- E.g. MetaCentrum has storage servers in various locations accessible from frontends and calculation nodes in `/storage`
- Usually, work only in your home, anywhere else modify files only if you are absolutely sure what you are doing
- Normal user doesn't have permission to modify files outside his directory (with exception of plugged removable media, etc.)
- Try `man hier` for details

Configuration in `/etc` (examples)

- Configuration of system services (servers, ...) and behavior
  - Apache web server, database, FTP server, networking, basic system settings, ...
- `cron*` — cron automatically repeatedly scheduled tasks
- `fstab` — description of FS mounted during startup
- `group` — list of users and groups
- `passwd` — basic settings for users (home directory, default shell, ...)
- `resolv.conf` — DNS settings (part of basic networking)
- `shadow` — users passwords in encrypted form
- `skel` — basic directories and configuration for new users
- Much more according to software installed...

Example of configuration in `/etc/ssh/sshd_config`

```
# This is the ssh client system-wide configuration file.  See
# ssh_config(5) for more information.  This file provides defaults for
# users, and the values can be changed in per-user configuration files
# or on the command line.
# ...
Host *
# If you do not trust your remote host (or its administrator), you
# should not forward X11 connections to your local X11-display for
# security reasons: Someone stealing the authentification data on the
# remote side (the "spoofed" X-server by the remote sshd) can read your
# keystrokes as you type, just like any other X11 client could do.
# Set this to "no" here for global effect or in your own ~/.ssh/config
# file if you want to have the remote X11 authentication data to
# expire after twenty minutes after remote login.
ForwardX11Trusted yes
# ...
```

Everything is (text) file

- In UNIX specifications, everything is (text) file
  - Technically, directory is "just" a file listing its content
  - Text files are easy to read, parse, manipulate
    - Very easily editable (easy to change configuration)
    - Easy to transfer to another system
    - Easily comparable among users/versions/systems
  - UNIX command line tools are the most powerful when processing text files (of any sort, e.g. also genetic data)
- When transferring to and from Windows, be aware of EOL and encoding (slide 76)!
- FAT32 (commonly used for USB flash disks) has limits to maximal file size (~ 4 GB) and range of characters allowed in file name is limited (slide 46), it doesn't support UNIX permissions (breaks executability of scripts)
  - Avoid large single files (or use archives to split large files)
  - In file names keep only alphanumerical characters, dots and underscores (omit spaces and accented characters)
  - Use archives to keep needed permissions (e.g. executability of scripts), see further

File names

- **Space** serves as separator of parameters
- Linux allows any character in file name, except slash (`/`), so including anything on keyboard as well as line break (!) — be conservative...

```
mkdir My New Directory # Produces THREE directories (mkdir creates dirs;
                       # spaces separate parameters). Solutions:
mkdir "My New Directory" # (you can use single quotes '...' as well) or
mkdir My\ New\ Directory # "\" escapes following character
rmdir My\ New\ Directory # Same problem and solution when removing it
touch \* # Creates new empty file named just * (yes, asterisk)
rm * # What would be removed? :-) Solution: rm \* or rm '*'
```

- Files and directories starting by dot (`.` — `.xxx...`) are hidden by default (typically user settings and application data in user home)

```
touch .hiddenfile # Let's make empty text file hidden by default
ls # We will not see it (ls lists only "visible" files/directories)
ls -a # We will see it ("-a" to see all - also hidden - files/dirs)
```

Task: Try everything on this slide, also with different file names and characters.

Types of files

```
ls -la ~
-rw-r--r-- 1 vojta users 1685 Feb 25  2019 .bashrc
drwxr-xr-x 1 vojta users 1040 Jan  8 09:08 bin
lrwxrwxrwx 1 vojta users   28 Jan 11 18:00 .lyxpipe.in -> /tmp/kile-...
...
```

- Regular file — ordinary file, marked by dash (`-`) on beginning
- Directory — in UNIX special type of file, marked by `d` on beginning
- Symbolic link (symlink, “soft link”) — points to another place, marked by `l`, slide 65
- Hard link — just another name for existing file, no special symbol, slide 65
- Block and character device — in `/dev`, representations of devices (hard disks, terminals, ...), marked by `b` or `c` respectively
- Named pipe — pipe can be saved (by `mkfifo`), looks like a file, more at slide 125
- Socket — for communication among processes, also bidirectional, available on network

Links

- **Symbolic (soft) links (symlinks)** — like links on the web — short-cut to another place
  - When we delete link, nothing happens, when target, non-working link remains

```
ln -s target link_name # Link points to target (existing file/directory)
ls -l bin/cinema5
lrwxrwxrwx 1 vojta users 42 5. dub 2014 cinema5 -> # "l" marks link
  /home/vojta/bin/cinema5-0.2.1-beta/cinema5*      # "->" points to target
```

- **Hard links** — only second name for file already presented on the disk (available only for files): `ln target link_name`
  - If any one of the two files is deleted, the second remains to be fully working

```
ln .bashrc .bashrcX
ls -l .bash* # Numbers in first column show links pointing to it
             # For directories - number of items, for files = 1
-rw------- 1 vojta users 7298 21. jan 16.43 .bash_history # One link
-rw-r--r-- 2 vojta users 2707 29. nov 16.21 .bashrc       # Same as below
-rw-r--r-- 2 vojta users 2707 29. nov 16.21 .bashrcX      # Two links
```

Owner and group

- Every file has an owner and group — for finer setting of rights
- Group can have just one member — the user
- System usually shows names of groups and users, but important are IDs (numbers): GID and UID
- Commands `chown` to change owner requires root privileges
- Commands `chgrp` to change group often requires root privileges — user has to be member of particular group to be able to change ownership to it (if not, `root` must do it)
- Information about users and groups and their IDs are in `/etc/group` and `/etc/passwd`
- Ownership (and permissions, slide 68) are important especially on servers with plenty of users (e.g. on MetaCentrum)
- It is not possible to add particular permissions for particular user on one file/directory — there must be special group or ACL must be used (slide 71)

Change owner and/or group

```
ls -l # Shows also owner and group (columns 3 and 4):
drwxr-xr-x 1 vojta users    80  5. jan 16.12 linuxcourse
drwxr-xr-x 1 vojta users  1648 31. jan 10.15 presentation
-rw-r--r-- 1 vojta users  1944  5. jan 15.18 README
drwxr-xr-x 1 vojta users   822 29. jan 10.12 scripts_data
-rwxr-xr-x 1 vojta users  1126  5. jan 15.22 web_update.sh
l # Common alias for 'ls -l' or 'ls -la' (according to distribution)
ll # Common alias for 'ls -l' or 'ls -la' (according to distribution)
id # Display UID and GIDs of current user
# New owner or group can be defined as name or ID
chown newowner:newgroup files # Change owner and group
chown -R newowner files # Recursively (with subdirectories) change owner
chgrp -R newgroup files # Recursively (with subdirectories) change group
chown --help # Or 'man chown' for more options
chgrp --help # Or 'man chgrp' for more options
```

- Equally important is to have correct permissions (especially on server) — next slides

File and directory permissions

- Combination of permissions to read/write/execute for user(owner)/group members/others

Permission	Number	Directory	File
r	4	Read directory content	Read file content
w	2	Write into it (add/remove items)	Write into it, modify, delete it
x	1	Enter it	Launch application (script)

- `rwxrw-r--` — 3×3 characters for permissions for owner of the file/directory, group it is belonging to, and other users (`d` on beginning marks directories, `l` links, `+` ACL, slide 71)
- `764` — same as above — numbers for each role are summed — first one is for owner, second for group and last for others
- Executable scripts and binaries **require** executable permission (`x` , e.g. `chmod +x` or `chmod u+x`) — not supported on FAT drives (USB sticks, memory cards, ...)

Permissions examples

```
ls -l # Shows permissions, links, owner, group, size, date, name
# Only owner can read and write the file; 600:
-rw-------   1 vojta users   38211 20. jan 09.23 .bash_history
# Owner can write read and write the file, others read; 644:
-rw-r--r--   1 vojta users    2707 29. nov 16.21 .bashrc
# Owner can enter, read and write directory, others can read and enter it;
# 755:
drwxr-xr-x  41 vojta users    4096 27. pro 09.55 bin
# Only owner can read, write and enter the directory, others nothing; 700:
drwx------  58 vojta users    4096 17. pro 15.45 .config
# Link, everyone can seemingly do everything; 777:
lrwxrwxrwx   1 vojta users      37 20. jan 09.33 .lyxpipe.in ->
  /tmp/kde-vojta/kilemj7d3E/.lyxpipe.in # Check permissions of target!
# Executable (application) - everyone can launch it, but only owner can
# write into the file (change or delete); 755:
-rwxr-xr-x   1 vojta users    2187 27. nov 13.10 strap.sh*
```

- Permission to “write” also means permission to **delete** it

Check and modify permissions

```
ls -l # Long list - file names and attributes
ls -a # All, including hidden files (starting with dot)
ls -F # Add on the end of name "/" for directories and "*" for executable
ls -h # Human readable size units (use with -l or -s)
ls --color ## Colored output
ls -laFh --color # Combine any parameters you like
chmod u/g/o/a+/-r/w/x FILE # For respective user/group/others/all adds
                           # /removes permission to read/write/execute
chmod XYZ FILE # Instead of XYZ use number code of permission
chmod -R # Recursive (including subdirectories)
chmod +x script.sh # Make script.sh executable for everyone
chmod o-r mydir # Remove read permission from others on mydir
chmod 600 FILE1 FILE2 # Make both files R/W only by their owner
chmod 000 FILE # No one can do anything - owner or root must add
               # some permissions before any other action...
chmod 777 * # All permissions for everyone on everything (no recursive)
chmod --help # Or 'man chmod' for more options
```

Extending permissions — ACL (Access control list) I

- By default, it is not possible to give specific permission to the user who is not owner, nor member of group owning the file
- In ext4 FS it has to be turned on manually (usually it is by default), it is part of Btrfs, XFS and ZFS — it is not available on every computer/server
- Command `getfacl` lists those extra permissions, `setfacl` sets
- When in use, "basic" tools listing permissions (e.g. `ls -l`, ACL in use is marked by `+` after permissions — next slide) sometimes do not show correct result and permissions may work unexpectedly
- Important especially in network environment with many users
- If intensively used, `ls -l` sometimes doesn't show correct permissions (see next slide), it can be confusing and lead into various issues

Extending permissions — ACL (Access control list) II

- If not in use on server (like e.g. on CESNET data storage in Ostrava, `du4.cesnet.cz`), relatively high number of groups is required to be able to correctly setup sharing permissions
- MetaCentrum has storages and clusters connected via NFSv4 protocol (see also slide 141) — commands `getfacl` and `setfacl` do not work there, use `nfs4_getfacl`, `nfs4_setfacl` and `nfs4_editfacl` instead (usage is very similar), see also `https://wiki.metacentrum.cz/wiki/Access_Control_Lists_on_NFSv4`
- Permissions ("classical" as well as ACL) require some time to practice and master it...

Importance of permissions

Too permissive (wrong) permissions can lead to data loss, or to data inaccessibility!

ACL examples

```
getfacl FILE # get ACL for FILE:
# file: dokumenty
# owner: zeisek # Correct
# group: zeisek # Correct
user::rwx       # Correct
user:nasik:r-x  # This is not seen from 'ls -l' output below!
group::r-x      # This group has only one member, this is fine
mask::r-x
other::---      # Correct
ls -l FILE # Compare this and previous output (This might be wrong!):
drwxr-x---+  2 zeisek zeisek     6 17. zář 20.40 dokumenty/
setfacl -m u/g:USER/GROUP:r/w/x FILE # Add for USER/GROUP r/w/x right
# E.g. recursively add read permission to user 'arabidopsis_data' to
# folder 'dokumenty/arabidopsis' (no extra group is required)
setfacl -mR u:arabidopsis_data:r dokumenty/arabidopsis
setfacl -R ... # Recursive (including subdirectories)
setfacl -b FILE # Remove all ACL from FILE (combine with -R ...)
```

Set default permissions for new files

- `umask` sets implicit permissions for newly created files for user
- Syntax is similar to `chmod`, but reverse (e.g. `027` keeps all rights for owner, for group removes writing and nothing left for others)
  - `umask` number **removes** certain permissions
- `umask 027` (or other number) is typically set in file `~/.bashrc`
  - `~` means user's home directory
  - `.bashrc` is user's configuration for BASH (see slide 92)
- Typically used in network environment
- Set with care — new permissions will have plenty of consequences — different are typically needed for web pages, private files, shared files, ...
- `umask` work recursively for all new files in user home directory — it is not possible to set new implicit rules for particular directory

Other permissions

- `sticky bit` — new directory/file in shared directory (where everyone can write) will be deletable only by owner (typically in `/tmp`)

```
chmod +t somedirectory
ls -la /
drwxrwxrwt 22 root root 800 21. jan 18.20 tmp # "t" marks it
```

- `setgid` — application can have root permission even it was launched by normal user

```
chmod u+s someapplication
ls -al /bin/passwd
-rwsr-xr-x 1 root shadow 51200 25. zář 08.38 /usr/bin/passwd # Note "s"
```

- `chattr` — change of advanced attributes on Linux FS
- Mostly, there is no need to modify them

```
chattr -RVf -+=aAcCdDeijsStTu files
man chattr # See explanation of attributes
lsattr # List extended attributes
```

Text and text — differences among operating systems

- Windows and UNIX have different internal symbol for end of line (new line) — EOL
  - UNIX (Linux, macOS, ...): LF (`\n`)
  - Windows/DOS: CR+LF (`\r\n`)
  - Mac v. < 9: CR (`\r`) (Mac up to 9 wasn't UN*X, since OS X it is)
- Good text editor (slide 159) can open correctly any EOL, but for example execution of script written in Windows will probably fail on Linux if it has wrong EOL
- Different systems use different encoding
  - UNIX: mainly UTF-8 (Unicode, universal), UTF-16 for Asian languages
  - Windows: win-cp-125X (variants according to region)
  - Older UNIX: ISO-8859-X (variants according to region)
  - Other much less common (historical) types...
  - Important mainly for accented characters
- Text editors can usually open any encoding, but automatic detection commonly fails — set it manually, see also slide 160

Command line I

Practical usage of command line tools

4 Command line
- BASH and other shells (“command lines”)
- Screen
- SSH — secure shell and screen
- BASH
- Directories
- Archives
- Searching
- Globbing, wildcards, quotes
- Variables
- Input, output and their redirecting

Command line II

Practical usage of command line tools

Information and processes
Network
Parallelization
Timing

Launching commands and scripts

- Parameters of commands are separated by space and preceded by one or two dash(es)
- Parameter `-h` or `--help` usually gives help for particular command
- Getting help with `man` command
  - `man somecommand`
  - Arrows to move up and down, `q` to quit
  - Type `/` and then type text to search and hit Enter to search — next hit by `n`
  - Command `info` more advanced — type `?` for help
- Parameters can be combined, order doesn't matter (same variants: `ls -la`; `ls -al`; `ls -a -l`; `ls -l -a`)
- "Long" parameters (`--XXX`) must stay separated
- Commands (applications) must be in PATH (slide 118) — actual directory isn't
  - If the script is is current directory (out of PATH), use `./script.sh` or full path
- Custom scripts must have execute permission (`chmod +x script.sh`)

macOS and Homebrew

- macOS contains outdated versions of many command line utilities with limited functionality comparing to what we are going to use (what is available in modern Linux distributions)
- Several projects provide Linux style way of installation and update of various (not only) command line tools, probably the best is Homebrew
- Xcode must be installed prior installation of Homebrew
- Homebrew contains also plenty of scientific packages, there is also specialized similar source for bioinformatics (and another sciences)
- Tools installed via Homebrew are installed into `/usr/local` not to interact with system packages
- Derived project is Linuxbrew (works also on Windows subsystem for Linux) useful especially for installation of some special (scientific) software unavailable in main Linux repositories (software resources)

Working with Homebrew

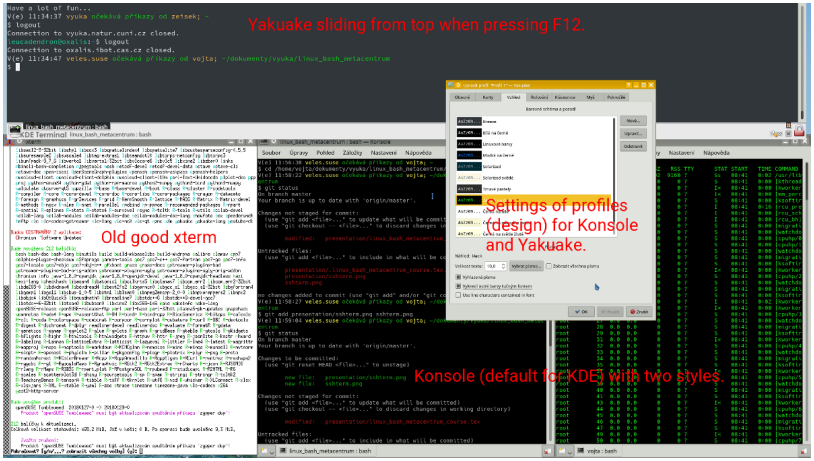
```
xcode-select --install # Install compilation tools
# Install Homebrew
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
  install/HEAD/install.sh)"
brew help # Basic help
# Install updated basic UNIX tools
brew install coreutils gnu-sed gawk grep bash gcc make wget dos2unix
brew list # List of installed packages (brew formulae)
brew info FORMULA # Information about particular formula (package)
brew search KEYWORD # Search for applications
brew update # Update Homebrew
brew upgrade # Update all packages installed by Homebrew
brew uninstall FORMULA # Remove Homebrew package (formula)
brew cleanup # Cleaning after uninstallation
# Completely remove Homebrew
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/
  install/master/uninstall.sh)"
```

The shell

- Many names, many ways how to get it, still the same thing
- Fish — friendly interactive shell — the command line interface
- Terminal (console)
  - Originally machine used for connection to remote server
  - System uses old fashioned terminal for inner purposes
    - From GUI available using `Ctrl+Alt+F1` to `F12`
    - Changing terminals using `Alt+F1` to `F12`
    - Return back to GUI using `Alt+F7`
    - Some are used for log outputs etc.
  - Nowadays used “indirectly” with special applications (“emulators”)
- **Terminal emulator**
  - Application used to get the “terminal” and work in command line
  - Every GUI has some — Konsole, Yakuake, XTerm, Gnome Terminal, Guake, XFCE Terminal, LxTerminal, ...
  - Commonly allow appearance customization — font, colors, background, style of notifications, ...
  - Launch as many copies as you need (usually allow tabs for easier work)

The command line can have various look and feel...

Change colors, font size, etc. for your terminal to like it more and work comfortably



Screen

Split terminal or keep task running after logging off

- When you log off or network connection is broken during work on remote server, running tasks for particular terminal usually crashes
- Sometimes number of connections to the server is limited
- `screen` is solution — virtual terminals
- Launch `screen` to start new screen terminal, read some info, confirm by **Space key** or **Enter**
- To detach from the screen press `Ctrl+A, D` (quickly press `Ctrl+A`, release, press `D`) — screen is still running in background — you can even log off
- To return back to running screen use `screen -r` — if only one screen is running, you get back to it
- If more screens are running, use `screen -r 1234` (the number is seen from `screen -r`)
- To cancel running screen press `Ctrl+D` (or type `exit` or `logout`)

Tmux

- More advanced (but not so common) alternative to `screen`

```
tmux # Start new tmux session
# Or name the new session (useful if there are more sessions)
tmux new -s SomeName
# Detach from the session by Ctrl+B, D
# List sessions on the server
tmux ls
# Attach to existing session by name
tmux attach-session -t SomeName
# Attach to existing session by its number
tmux attach-session -t 0
# Cancel running session by Ctrl+D or exit
```

- Get help by `Ctrl+B, ?` (`Q` to quit)
- Split window by `Ctrl+B, %` and navigate between them by `Ctrl+B, L/R arrow`
- It has plenty of options

Login to remote server

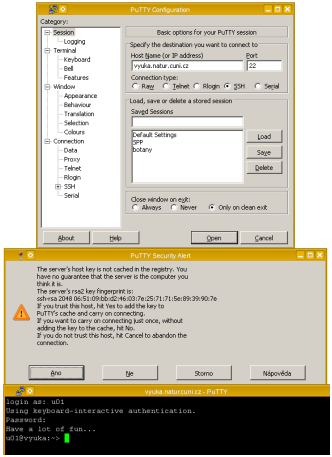
SSH — secure shell — encrypted connection

```
ssh remoteUser@remote.server.cz
# When logging first time, check
# and confirm fingerprint key
yes # And press Enter
# Type remote user's password
# (nothing is shown when typing)
# Confirm by Enter
```

- Toy server: user names `cusr01` – `cusr30`

```
ssh cusrXY@vyuka.natur.cuni.cz
```

- If fingerprint key changes, ssh complains a lot — possible man in the middle attack
- From Windows use Putty (see right figure)



SSH and screen practice

Tasks

1. Login via SSH to `vyuka.natur.cuni.cz` and launch `screen`.
2. Run commands `pwd`, `whoami` and `ls -la`. What do they show?
3. Detach from screen and logout from the server.
4. Login again to the server. Is it same or different process? Why?
5. Reattach to the screen — same state as before logout from server.
6. Open new screen and practice tasks with file names (slide 63).
7. Detach and re-attach to both screens.
8. Close all screen sessions and logout from the server.

BASH and others

- Shell (**sh**) — feature rich scripting programming language — general specification, several variants
- So called POSIX shell — Portable Operating System Interface — transferable among hardware platforms (and UNIX systems)
- **Interpreter of our commands inserted into command line**
- **BASH** — Bourne again shell
  - Probably the most common shell, based on original `sh`, respecting original specification, adding new features; probably the most common
  - We will use it
- Other variants: **csh** (syntax influenced by C), **ksh** (younger, backward compatible with bash), **zsh** (extended bash; probably the most advanced, not so common), **ash** (mainly in BSD)
  - There are some differences in syntax and features
- Language suitable for easy scripting and system tasks, not for “big” programming, neither for graphical applications

Nice BASH features for easier work (selection) I

- Arrows up and down list in the history of commands
- List whole history by command `history`
- `Ctrl+R` — reverse search in history — type to search last command(s) containing typed character(s) — repeat typing `Ctrl+R` to search deeper in history
  - When on correct entry, hit Enter or use L/R arrow keys to edit it
  - `Ctrl+J` — exit the search
- `TAB` — list command and files starting by typed characters
- `Home` / `End` (or `Ctrl+A` / `Ctrl+E`) — go to beginning/end of the line
- `Ctrl+L` — clear screen (like `clear` command)
- `Ctrl+Shift+C` / `V` — copy/paste the text from terminal emulator
- `Ctrl+C` — cancel running task

Nice BASH features for easier work (selection) II

- `Ctrl+D` — log out (like commands `exit` or `logout`)
- `Ctrl+U` — move text before cursor into clipboard
- `Ctrl+K` — move text after cursor into clipboard
- `Ctrl+Y` — paste selection from the above commands
- `Ctrl+left/right arrow` — skip words
- `Ctrl+S` — suspend output of long verbose commands (resume with `Ctrl+Q`)
- `Ctrl+T` — flip current and left character
- `!xx` — launch last command starting with `xx` (use with care)
- `Ctrl+X+E` — start text editor (default, defined in `~/.bashrc`) in current directory
- Can slightly differ (be limited) among various systems, terminal emulators, etc.

Places to store BASH settings for interactive work in shell

- `/etc/bash.bashrc` — System wide BASH settings — can be overridden by user's configuration
- `~/.bashrc` — File is loaded each time user creates new session (typically opens new terminal window)
- `~/.bash_profile` — Used specifically (not in every system) when user is using remote connection (e.g. SSH) — user can have different settings for local and remote work
- `/etc/profile` — System wide profile file — can be overridden by user's configuration
- `~/.profile` — Settings loaded when user logs-in (mainly for language settings), sometimes used by remote connections
- Note: BASH scripts are non-interactive shells — they do not read settings above — there are no aliases etc., but they inherit some settings (PATH, language, ...) and they can read global (not local) variables

BASH settings (popular examples)

Write them into BASH configuration file

- In any text editor open `~/.bashrc` (and/or `~/.bash_profile`) and edit it
- Behavior of BASH can be set to fit user's needs
- Terminal emulators allow to set custom fonts and colors, ...
- Prompt can be colored using custom values of `PS1` and `PS2` variables (`export PS1='...'`), see examples and search

```
eval "$(dircolors -b)" # More colors for outputs (typically ls)
# Colored GCC warnings and errors (useful for compilation)
export GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:
  locus=01:quote=01'
# Example of more colorful prompt (do the same for PS2)
export PS1='\[\033[0;34m\]\u\[\033[0m\]@\[\033[0;32m\]\h\[\033[0m\]:
  \[\033[1;31m\]\w\[\033[0m\]\$ ' # Everything on single line without space
export PS1='\[\033[0;32m\]\d:\t \[\033[0m\]\[\033[1;31m\]\w\[\033[0m\]$ '
```

Examples of BASH settings

```
# Set custom default editor (Ctrl+X+E and other cases)
export EDITOR='nano'
# Ignore repeated entries in bash history (stored in ~/.bash_history)
HISTCONTROL='ignoreboth'
# Maximal length (number of lines) of bash history (~/.bash_history)
HISTFILESIZE='100000'
# Following two settings save history from multiple terminals
# Normally, only history from last time opened terminal is kept
shopt -s histappend # Append to history, don't overwrite it
# Save and reload the history after each command finishes
export PROMPT_COMMAND="history -a; history -c; history -r;
  ${PROMPT_COMMAND}" # Note its recursive behavior
```

- `shopt` can set various shell options, see e.g. https://www.geeksforgeeks.org/shell-scripting-shopt-command/ (it can be used also in script to set some specific shell behavior)

Aliases and BASH settings I

Alias is short cut — instead of very long command write short alias

```
# Define new alias
alias ll="ls -l"
```

- Can be stored in `~/.bashrc` (or `~/.profile` or `~/.bash_profile`)
- If there are plenty of them, aliases can go to `~/.alias` and `~/.bashrc` then contains `test -s ~/.alias && . ~/.alias || true`

```
# After adding new aliases to ~/.bashrc or ~/.alias or so reload it
source ~/.bashrc # Reload BASH settings to load newly aliases
# Popular ls aliases
alias ls="ls --color=auto" # Make output of ls colored
alias l="ls -laF" # Long list (add details) with hidden files
alias ll='ls -l' # Long list (add details) without hidden files
alias lh='ls -lha' # Long list (add details) with human readable sizes
```

Aliases and BASH settings II

```
alias md='mkdir -p' # Making directories
alias rd='rmdir' # Removing directories
alias o='less' # Text file viewer
alias grep='grep --color=auto' # Enable color in grep
alias df='df -h' # Always human readable output of df (disk free)
# Easier history listing
alias his="history | grep" # Use e.g. 'his ls' to list last 'ls' usage
# Add aliases pointing to software installed outside PATH, ...
# Add ~/.local/bin to PATH (directories with commands and scripts)
export PATH="${PATH}":~/.local/bin:~/bin
```

- `~/.bashrc` can contain anything (technically it's just BASH script), functions, whatever needed
- `~/.bashrc` commonly contain definitions of variables
- Other shells than BASH (KSH, ZSH, ...) have their own configuration files

Directories

- When using `cd`, `cp`, `mv`, ... use `<TAB><TAB>` key suggesting matching names of files and directories and save repeated and unneeded typing
- In command line, **user is always in some directory** — it's crucial to train fluent moving among directories and manipulation with files
  - If lost among directories, run `pwd` to find out current directory and `ls` to see what is around

```
pwd # Print working directory - where we are right now
cd # Change directory (just "cd" or "cd ~" goes to home directory)
cd .. # One directory up; cd ../..; cd ../../another/directory/
cd relative/path/from/current/position # Go to selected directory
cd /absolute/path/from/root # Absolute path starts by "/"
cd - # Go to previous directory
tree # Tree like hierarchy of files and directories
tree -d # List only directories; see tree --help
tree -L 2 # Only up to second level; combine: tree -d -L 3
```

Directories and files

```
du -sh # Disk usage by current directory, -s for sum, -h for nice units
mkdir NewDirectory # Make directory
rmdir DirectoryToRemove # Remove empty directory
ls # List directory content
   # Try parameters -l, -a, -1, -F, -h (with -l or -s), --help
rm -r # Recursive delete - remove also non-empty directories
mv from to # Move files/directories (also for renaming)
mv docs to/sub/directory/ # Move 'docs' to 'to/sub/directory/'
cp from to # Copy, -r (recursive, including subdirectories)
           # -a (keeps all attributes), -v (verbose)
# Copy 'XXX' (recursively with subdirectories and everything) in the
# upper directory into 'sub/directory/'
cp -a ../XXX sub/directory/
# Copy 'doc.txt' from home directory into current directory
cp ~/doc.txt . # Dot stands for current directory
file somefile # Information about questioned file (what it is, ...)
xdg-open somefile # Open file by graphical application as in GUI
```

Tasks on the remote server I

1. Login via SSH to `vyuka.natur.cuni.cz`.
2. Get your path by `pwd`.
3. Go to `/srv/scripts_data` (with `cd`) and explore its content (`ls`).
4. List permissions in `/srv/scripts_data` (slide 68). What do they show? What can you do with the content?
5. How much space does `/srv/scripts_data` consume?
6. Go back to home directory (by `cd`).
7. Create new directory in your home directory (`mkdir`).
8. Copy content of `/srv/scripts_data` into your newly created directory (`cp`).

Tasks on the remote server II

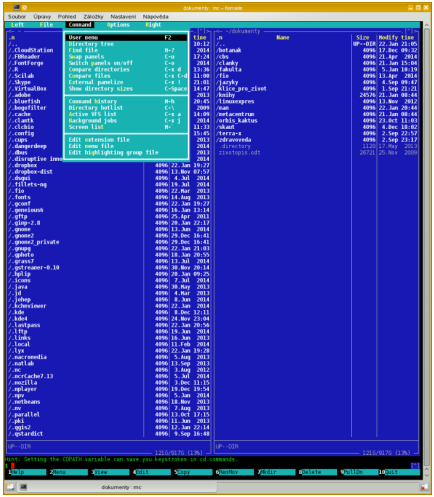
9. Rename that directory with scripts and data using `mv` to any custom name. Who is owner of the files in origin location and in new location? Why?
10. Explore your home directory and its content by command `ls` and `tree` and some files by command `file`. Which hidden files and directories are there? What could it be?
11. Change permissions of the files so that only you can read, write and execute them (`chmod`).
12. Create other directory, see it and then remove (`rmdir`).
13. Can you access directories of another users? Why? If yes, what are your permissions there? Explain it.
14. What are some permissions in `/`? Why?
15. Define some alias (by running `alias` command, not by edit of `~/.bashrc`) and use it.

Tasks on the remote server III

16. Create directory in your home directory and share it with another user so she/he can write there anything (using e.g. `touch somefile` or `mkdir somedirectory`) (work e.g. in pairs). Use everywhere as restricted permissions as possible. Can you figure out solution with or without ACL (slide 71)?
17. Practice moving between `/srv/scripts_data` and your home directory. Use `cd` and `TAB`.
18. Within `/srv/scripts_data` list by single command only `jpg` and `txt` files.
19. Create in your home directory new directory `scripts` and copy there with single command all shell scripts (`*.sh`) files from `/srv/scripts_data`
20. Copy anywhere into your home `/srv/scripts_data` and by single command remove all `jpg` and `sh` files there.

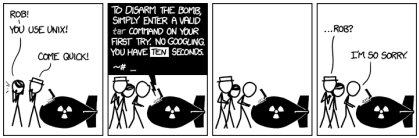
Midnight Commander

- `mc` to launch MC (`F10` to quit), can be used with mouse
- Move (`F6`), copy (`F5`), delete (`F8`), files/directories, `TAB` to switch panes
- Connect to SSH/(S)FTP, ...
- Edit (`F4`) or view (`F3`) text files
- `F2` for quick menu
- `F9` for top menu
- And much more...
- Impossible to live without it :-)
- Task: Which of the previous tasks can you solve with it? Try it.



Compressing and decompressing archives

- `gzip`, `bzip2`, `xz` and `lzma` are able to **pack only one file** — use them together with `tar` to pack multiple files (when used **without** `tar` they move file into archive)
- In Linux, `gzip` (and less `bzip2`) are the most commonly used
- Rar and arj are not used at Linux/UNIX at all
- Zip is probably the most portable between Linux/UNIX and Windows
- `lzma` and `xz` have excellent compression, but can be very slow, use similar algorithm, often confused



https://xkcd.com/1168/

Compressing and decompressing archives

Archive	Compressing command	Decompressing command
*.tar	tar cvf archive.tar file1 file2 file3	tar xvf archive.tar
*.tar.gz / *.tgz	tar czvf archive.tar.gz/.tgz file1 file2	tar xzvf archive.tar.gz/.tgz
*.tar.bz / *.tbz / *.tar.bz2	tar cjvf archive.tar.bz/.tbz/.tar.bz2 file1 file2 file3 file4	tar xjvf archive.tar.bz/.tbz/.tar.bz2
*.tar.xz	tar cvJf archive.tar.xz file1 file2 file3	tar xvJf archive.tar.xz
*.tar.lzma	tar cvf - file1 file2 file3 file4 | lzma > archive.tar.lzma	lzcat archive.tar.lzma | tar xvf -
*.gz	gzip file	gunzip archive.gz
*.bz2	bzip2 file	bunzip2 archive.bz2
*.xz	xz -zv file	xz -d archive.xz
*.lzma	lzma file	unlzma archive.lzma
*.zip	zip -r archive.zip file1 file2 file3 file4	unzip archive.zip

Tasks with archives

1. Compress (and decompress) text file `Oxalis_HybSeq_nrDNA_selection_alignment.fasta` from `scripts_data` with various compressing tools.
2. Compare sizes of original file and compressed outputs.
3. Compress (and decompress) all `foto_oxalis_*.jpg` (*Oxalis* photos) together from `scripts_data` with various compressing tools.
4. Compare sizes of original files and compressed outputs.
5. Which compression tool seems to be the best? In terms of compressing ratio and time needed for compression.
6. Is more effective compression of text files or images? Why?
7. Why is even plain `tar` (without compression, it requires `gzip`, `bzip2`, `xz` or `lzma` to add compression) useful with FAT disks (it applies also for another tools)?
8. Search the Internet to find out how to unpack `arj` and `rar` archives from command line.

Find — looking for files

```
find <where> <what> <what to do> # The most powerful searching tool:
find /... -type d/f -name XXX -print # Most common usage
```

- First `find`'s parameter is location to search — absolute or relative, "`.`" means current directory (the only compulsory parameter)
- `-type` for only directories `d` or only files `f` (without this parameter, files as well as directories are looked for)
- `-name` of the searched files/directories supports wildcards (`*`, `?` and `[...]`), see globing (slide 110)
- `-print` is default action — prints list of results
- `-exec` runs some command with results (some operation, not just listing)
  - All following arguments are argument of the command until "`;`" is encountered
  - `'{}'` is replaced by the current file name being processed
  - Those constructs might require protection by escape ("\") or quotes not to be expanded by shell

Find examples I

Apply some (or something similar) of them to the toy data

```
# Find in /home/$USER/ all JPG files containing string "oxalis"
find /home/$USER/ -name "*oxalis*.jpg" -print
# Find in scripts_data all JPG files and resize them to 1000x1000 px
find scripts_data -name "*.jpg" -exec mogrify -resize 1000x1000 '{}' \;
# Another possibility with xargs (it chains commands - reads input from
# stdin and execute command with given arguments, using all CPU threads)
# Note in the example below -print is not needed as it is default action
find photos/ -name "*.jpg" | xargs mogrify -resize 1000x1000
# How many directories are there in the books directory
find books/ -type d -print | wc -l # wc -l calculates lines
# Change permissions of all files within "files" directory to 640
find files/ -type f -exec chmod 640 '{}' \;
# Find all R scripts in ~/Documents and find in them lines with "DNA"
find ~/Documents -name "*.r" -print | xargs grep -nH DNA # Or
find ~/Documents -name "*.r" -exec grep -nH DNA '{}' \;
```

Find examples II

Apply some (or something similar) of them to the toy data

- `find` is extremely versatile and useful tool — master it (see also `man find` for help)
- `-print` is default action — if it is missing and there is no other actions, results are printed to the screen

```
# Find all executable files within current directory and list them
find . -executable -type f -print
# Delete empty directories within 'doc' directory
find doc/ -type d -empty -execdir rmdir '{}' \;
# Copy all *.sh files from /home to ~/scripts
find /home -name "*.sh" -exec cp '{}' ~/scripts/ \;
# Search for file long_text.txt (exact name) in your home directory
find ~ -name long_text.txt -type f -print
# Find in current directory files from 1 to 100 MB
find . -type f -size +1M -size -100M
# Search in home dir for files named "*.jpg" or "*.txt" and print them
find ~ -name "*.jpg" -o -name "*.txt" # '-print' is default, can be omitted
```

Looking for applications and files

```
updatedb # Must be regularly launched to get "locate" to work
         # It is usually regularly launched by cron task (see further)
locate somename # Searches for files/directories in "locate" database
apropos keyword # Searches for command descriptions containing keyword
which app # Full path to application (shell command)
whereis app # Path to source code, executable and man pages for the command
# Test if executable command exists (good for scripts)
# If "Application" is missing, script ends with error
command -v Application >/dev/null 2>&1 || { echo >&2 "Application is
  required but not installed. Aborting."; exit 1; }
type Application >/dev/null 2>&1 || { echo >&2 "Application is
  required but not installed. Aborting."; exit 1; }
# More reliable than 'which Application || { echo XXX; exit 1; }'
exit 1 # "exit" use to be added (with various numbers) after any error
       # to send term signal 1 - for better handling of various errors.
       # Every termination has exit status number - 0 is normal exit.
       # Exit status 1 and higher number is various error.
```

Searching tasks

1. Use `locate` to find file `long_text.txt` on the server. Is the output absolutely correct? If not, why?
2. Where is executable of `mc`? Why can be such information useful?
3. Which software is related to keywords `permission` and `compress` (use e.g. `apropos`)? Use `man` to explore some of them.

Tasks with `find`

1. Find all `*.vcf.gz` files in whole `/home`. Why do you get errors for some directories?
2. Compress, see and decompress all shell scripts in your home directory.
3. Change permissions of all content of your home directory so that no one else can access it. Consider hidden files, directories and scripts.
4. List all directories in `/etc` up to second level.

BASH globing and wildcards

- BASH itself doesn't recognize regular expressions (from slide 191) — its wildcards have some of functions of regular expressions and can look similarly, but behave differently — Do not confuse!
- `?` — Replaces any single character
- `*` — Replaces any number of any characters (`ls a*` lists all files starting with "a")
- `[]` — Range or a list — `[abcdef]` and `[a-f]` are same
- `[!...]` — Reverse previous case (`!`) — any character except those listed
- `{}` — Expansion (terms inside are separated by commas `,`) — all possible combinations (see next slide for examples)
- `\` — Escapes following character and it doesn't have its special meaning (e.g. `\*` means literally asterisk `*` and not "any number of any characters" as usually)
- For details see `man 7 glob` and `man 7 regex`
- Useful e.g. to list only subset of files or to provide file selection as input for some command

Brace expansion and quotes

```
1 echo a{p,c,d,b}e # ape ace ade abe - all combinations
2 echo {a,b,c}{d,e,f} # ad ae af bd be bf cd ce cf - all combinations
3 ls *.{jpg,jpeg,png} # Expansion to *.jpg *.jpeg *.png, same as:
4 ls *.jpg *.jpeg *.png
5 ls scripts_data/*_R[12].fastq.bz2 # Same as *_R1.fastq.bz2 *_R2.fastq.bz2
```

- Text in single quotes (`'...'`) preserves the literal value of each character within the quotes
- Text in double quotes (`"..."`) preserves the literal value of all characters within the quotes except of dollar (`$`), back tick (`` ` ``) and back slash (`\`) — handling variables
- A double quote may be quoted within double quotes by preceding it with a backslash (`\"` means literally “double quote”)
- Text between back ticks (`` `...` ``) or within `$(...)` will be evaluated and then used as command or argument (see next slide for examples)
  - Syntax with back ticks is deprecated, keep using `$(...)` instead
- This is important when handling file names with non-Latin characters, when working with variables (from slide 113), printing various information within scripts (e.g. slide 121), etc.

Variables in BASH

- Variables contain various information (where to look for the executable programs, name of the computer, various settings, input files, ...)
- Can be local (within a script for some temporal purpose) or global — available for all processes (and users)
- Names commonly written in CAPITALS (just a costume)
- Popular and useful variables
  - `HOME` — location of user's home directory
  - `HOSTNAME` — network name of the computer
  - `LANG` — language settings, encoding, similarly variables `LC_*`
  - `PATH` — paths where to look for applications — all applications have to be in `PATH` or called directly (slide 118)
  - `SHELL` — shell in use (bash or something else)
  - `USER` — user name
  - And many more, commonly specific for particular server

Work with variables

- Names start with `$`, e.g. `$HOME`, but declaration is without `$`, e.g. `MYVAR='XXX'`
- Variable defined within shell session, script, function, etc. will disappear as soon as the session, script, function, etc. is terminated — it must be exported, defined in `~/.bashrc` or so to be preserved

```
printenv # Get all environmental variables and their values
export -p # Get all exported variables and their values
declare -p # Get all declared variables and their values
echo "${VARIABLENAME}" # Get value of particular variable
echo "${PATH}" # Get path where to look for applications
VARIABLE='variablevalue' # Set new variable and its value
                         # Or replace existing variable by new value
export EDITOR='/usr/bin/vim' # Set new default text editor
LISTFILES="$(ls)" # Get output of command 'ls -1' into the variable
echo "${LISTFILES}" # See content of the variable LISTFILES
export GREP_OPTIONS='--color=auto' # Colored grep (see further)
unset VARIABLENAME # Drop variable and its value
```

How quotes influence reading of variable content I

- As soon as variable value (content) should contain anything else than plain Latin characters and numbers, or should combine with another variable, be careful...

```
A=abcdef # Set new variable (no special characters allowed)
echo $A # See variable's content
abcdef # It works
echo '$A' # Single quotes preserve literal value
$A # We see variable's name, not its content
echo "$A" # Double quotes preserve literal value, except $, `, \
abcdef # This also works; not needed here
echo `$A` # Text between back ticks is evaluated as a command and launched
abcdef: command not found # There is no command "abcdef"...
echo $($A) # Same as `...`, this is now recommended way, `...` is legacy
           # Commonly used to fill variable by output of some command, e.g.
           FQTOPROCESS=$(find "${DATADIR}" -type f -name "*.f*q")
echo "Hi, dear $USER" # Compare this and following command...
echo 'Hi, dear $USER' # Single quotes do not evaluate variables
```

How quotes influence reading of variable content II

```
B=abcd$e # The content will be "abcde + $e" or "abcd" (if $e is missing)
echo $B # abcd
C=abcd\$e # \ escapes next character - it is loosing its special meaning
echo $C # abcd$e
C='abcd$e' # '...' keep literal value of the content; as above
# Next command breaks shell - incomplete quotes " - pres then Ctrl+C
D=ab"cde # The variable should contain incomplete quotes ", it fails
echo $D # Nothing - empty
E=ab\"cde # \ escapes next character - it is loosing its special meaning
echo $E # ab"cde
E='ab"cde' # '...' keep literal value of the content; as above
F=abc${USER}de # $USER wouldn't work - {} are needed
echo $F # abcvojtade # To add output of command into the variable
G="abc$(echo ${USER} | tr [:lower:] [:upper:])de" # Modification of var.
echo $G # See what happened with content of $USER
H=abc def # Error: command "def" not found... Why? Fixing:
H='abc def' # "echo $H" outputs "abc def" - typical problem with paths
```

How quotes influence reading of variable content III

- As soon as variable contains only Latin alphanumerical characters, assignment like `MYVAR=whatever` works, but it is not recommended
- If variable should contain another variable, special characters, etc, use double quotes, e.g. `WORKDIR="/home/$USER/data"`
  - It might be called like `echo $MYVAR`, but if there would be non-standard characters anywhere, it'd fail, so prefer `"$MYVAR"` (it *might* work without quotes, but don't risk it) or `"${MYVAR}"` as the most reliable way
- If variable should contain any special character (including space), or you wish to be sure you keep it's literal value, use single quotes, e.g. `MYDATA='~/My Doc/exp 1/data.fsa'` (and then `"${MYDATA}"`)
- Similar for using variables
  - Using `$MYVAR` is fine if it doesn't contain anything special (but are you really sure?)
  - Best practice is to use `"${MYVAR}"` as there is highest security if `MYVAR` would contain special characters, and there is room for various manipulations (next slide)

Manipulating with content of variable

- It is possible to change/remove extension (suffix) of variable, do simple search and replace with its content, etc. (useful in scripts e.g. for input and output files)
- Search and replace syntax is related to `sed` (slide 175)

```
MYVAR='oxalis_assembly_6235.aln.fasta' # Assign new variable
echo "${MYVAR}" # See it - it works: oxalis_assembly_6235.aln.fasta
# Remove the '.fasta' extension
echo "${MYVAR%.fasta}" # oxalis_assembly_6235.aln
# Replace extension '.fasta' with '.fsa'
echo "${MYVAR%.fasta}".fsa # oxalis_assembly_6235.aln.fsa
# Remove any last extension
echo "${MYVAR%.*}" # oxalis_assembly_6235.aln
# Replace 'aln' within variable name with 'aligned'
echo "${MYVAR/aln/aligned}" # oxalis_assembly_6235.aligned.fasta
# More complex manipulations with variable content (try to avoid this)
echo "$(echo "${MYVAR}" | sed "s/assembly/${USER}/")"
# sed allows usage of variables (sed "s/pattern/$SOMEVAR/", see further)
```

The PATH variable

- Lists directories (separated by colon `:`) where the current shell searches for commands
- If some software is installed outside standard locations, the user must specify the full path (or update the `$PATH`)
- In case there are two commands with the same name (e.g. `/usr/bin/somecommand` and `/home/$USER/bin/somecommand`), the order of directories in `$PATH` matters — the first occurrence is used, any possible later ignored
- Computing clusters (like MetaCentrum) use to have special command (e.g. `module`) to load particular software (including particular version) by extending user's `$PATH`

```
1 echo $PATH # See the $PATH variable. Sample output is on the next line:
2 /home/$USER/bin:/usr/local/bin:/usr/bin:/bin:/opt/bin:/sbin:/usr/sbin
3 # Adding new directory to $PATH
4 export PATH="${PATH}":/some/new/directory # Ensure to add original $PATH
5 # Do not overwrite $PATH - there would be only the new directory
6 export PATH=/some/new/directory # Wrong! Old $PATH is missing!
```

Reading variables from command line and as output of other commands

This is especially useful in scripting to read input from users or from another commands

```
# Reading variable from user's input from command line
# (some interactive script interacting with the user)
read X # We will read new variable from input (do not use "$" here)
10 # Type any value and press Enter
echo $X # Get value of the variable
10 # It works
echo "$((1 + $X))" # Sum of 1 and variable $X
# Output some command into variable
Y=$(command) # Set as variable output of command
echo $Y # Y will contain output of command
# "command" from previous lines can be e.g.
cat somefile.txt # Read content of somefile.txt; or
find . -name "*.txt" # Save list of matching files
WORKDIR="$(pwd)" # Save current directory into variable
ls -1 | head -n 1 # First file/directory in the current directory
unset X # Destroy this variable
```

BASH expressions

Basic arithmetic operations and more with BASH

```
1  echo $((5 < 7)) # Is 5 smaller than 7? TRUE (1)
2  echo $((3 > 4)) # Is 3 greater than 4? FALSE (0)
3  echo $((16 / 3)) # Division (without decimal part)
4  echo $((16 % 3)) # What remains after aritmetic division
5  x=$((1 + 6)) # Result will be in 'x' - save output into variable $x
6  echo $x # See content of $x
7  x=1 # Set x to 1
8  y=$x+1 # Will this add 1? Why?
9  echo $y # See result
10 y=$(($x + 1)) # Will this work? Why?
11 echo $y # Result
12 echo $(expr length "MetaCentrum and Linux") # Get length of chain
13 # String of 5 characters starting at position 10 of the text
14 echo $(expr substr "MetaCentrum and Linux" 10 5)
15 # Does 1st chain contain 2nd chain (how long)? Get position of first hit
16 echo $(expr index "GNU Linux" "Linux") # If no overlap, return value is 0
```

Notes about variables, expressions and quotes and more

- `expr` works with various operands (see `man expr`) — it used to be utilised also for simple computations, `$((...))` is now preferred
- Exported variables will be lost when logging off
- To make variables permanent, add `export` commands into `~/.profile` or `~/.bash_profile`, or `~/.bashrc` (according to shell and its settings)
- "`~`" means home directory
- `$` marks variables
- `\` escapes following character — it will not have its special meaning (space to separate arguments, ... — next slide)
- If variable is going to contain any special character (`?`, `.`, `*`, ...), the value must be quoted — `"..."` allow escaping of special character or inclusion of another variables, `'...'` keeps absolutely literal value

Chaining commands I

- Key feature of BASH — work with individual commands as with Lego to get new functionality
- `&` — command will be launched in background, terminal is available for next typing: `firefox &` (when launching graphical application, hit **Enter** afterward if there is no active command line prompt)
  - If the application has any output, it goes to the screen, so it can be bit messy...
- `&&` — second command is launched only when first command exits without error (exits with status `0`): `mkdir NewDir && cd NewDir`
- `;` — second command is launched regardless exit status of the first one: `kshfskcbd; hostname`
- `{...}` — commands within curl brackets are launched as one block
  - Closing bracket `}` must be either on new line or preceded by semicolon (`... ; }`)

Chaining commands II

- `||` — second command is launched when first command fails (has non-zero exit status):
  `cd newdir || { mkdir newdir && cd newdir; }`
  - Easy way how to do something when previous command fails, either exit script (`… || exit 1`) or somehow fix it (see above), report problem
    (`… || {echo "It failed!" && exit 1; }`) or so
- `|` — pipe — redirects standard output of one command into standard input of second command: compare `mount` and `mount | column -t`
  - Chains commands, basic redirecting method — common e.g. for various data parsing
  - One of key features for commands and scripts processing scientific data, e.g.
    `bwa mem … | samtools view -bu | samtools sort -o sample.bam`
- Behavior in shells other than bash might be little bit different

Standard input and output and redirects

- Standard input (`stdin`) is standard place where software takes input (keyboard and terminal) and writes results to standard output (`stdout`) — typically monitor (or file)
- Standard error output (`stderr`) is target of error messages — typically also monitor (but can be log file or so)
- `>` redirects output into new place (file, device, another command, ...)

```
cat /etc/group # Print whole file /etc/group
grep users /etc/group > users # Extract from /etc/group lines containing
                              # "users" and write output into new file
cat users # See result
```

- `>>` adds output to the end of the file (`>` rewrites target file)

```
grep root /etc/group >> users # Add new information into existing file
cat users # See result
```

- Task: Practice chaining and redirects from previous and this slide. What is it good for?

Redirects of input/output

- `/dev/null` — "black hole" — can discard anything

```
command 2> /dev/null # Discard only errors (note "2" for errors only)
command > /dev/null # Discard all output (no logging or onscreen output)
```

- `/dev/stdin` — standard input (typically keyboard)
  - In case application reads files, not from standard input:

```
echo "Žluťoučký kůň úpěl ďábelské ó" | iconv -f utf-8 -t cp1250 /dev/stdin
```

- `/dev/stdout` — standard output
  - Typically screen, commonly redirected into file
  - We wish to see errors which would be discarded otherwise:

```
command 2> /dev/stdout # Errors go to screen (typically), not to log
```

- `/dev/stderr` — standard error output
  - Typically screen or log file, right place to send errors to:

```
echo "error" > /dev/stderr
command 2> /dev/stderr # Errors go to standard error log
```

Redirects of standard input and output I

Common way how to save output of some command(s)

```
# If file directory_listing.txt exists, will be overwritten
ls -la > directory_listing.txt
cat directory_listing.txt # See result (same as running "ls -la")
# If file directory_listing.txt exists, new content will be appended
ls -la >> directory_listing.txt # See result 'cat directory_listing.txt'
# We are outputting 'ls -lh' to 'awk' and not to screen, printing only
# selected columns, and parsing with 'column -t' for tabular display
ls -lh # Compare outputs of the 3 commands starting with 'ls -lh'
ls -lh | awk '{ print $NF, " ", $5}' # See further more info about AWK
ls -lh | awk '{ print $NF, " ", $5}' | column -t
```

Tasks with redirects of standard input/output

1. Try everything on this and following slide, be sure to understand the redirects.
2. Try to figure out some other example regarding your practical needs.

Redirects of standard input and output II

```
# Add error output to the end of standard output file
# Note: In the examples below command "commandX" does not exist -
# it produces error "command not found" to be recorded by the log
# and because of redirect, the error is not shown in the terminal.
command > outputfile.log 2>&1 # Example:
{ commandX; ls; } > outputfile.log 2>&1
cat outputfile # See result
# Compare with following. What is the difference?
{ commandX; ls; } > outputfile.log # Where does error and output go?
# Compare two following commands. What is the difference?
{ commandX; ls; } > outputfile.log 2>&1 # Inspect outputfile.log
{ commandX; ls; } >> outputfile.log 2>&1 # Inspect outputfile.log
# Add error output to the error log text file
command >> outputfile.log 2> error.log # Example:
{ commandX; ls; } >> outputfile.txt 2> error.log
cat outputfile.txt # See results
cat error.log # See results
```

Examples of redirects and pipes when working with molecular data I

```
# Extract and sort depth of coverage (how many times was each position
# sequenced) in genomic VCF with multiple individuals
zcat arabidopsis.vcf.gz | grep -o "DP=[0-9]\+" | sed 's/^DP=//' |sort |less
# Convert DNA sequence from FASTQ to FASTA (two of many options)
# Discard FASTQ quality scores and keep only the sequence and its name
bzcat Oxalis_hirta_R1.fastq.bz2 | sed -n '1~4s/^@/>/p;2~4p' > \
  Oxalis_hirta_R1.fasta
bzcat Oxalis_hirta_R2.fastq.bz2 | awk '{if(NR%4==1)
  {printf(">%s\n",substr($0,2));} else if(NR%4==2) print;}' > \
  Oxalis_hirta_R2.fasta
# See both sequences in one view
less Oxalis_hirta_R{1,2}.fasta
# Mapping of trimmed Illumina FASTQ reads to reference FASTA sequence
# and creation of BAM containing the alignment (reference and mapped
# reads, one BAM for each sample)
bwa mem reference.fasta input_R1.fastq input_R2.fastq | samtools view \
  -bu | samtools sort -l 9 -o mapped.bam
```

Examples of redirects and pipes when working with molecular data II

```
# How many FASTQ reads are there in the file
echo "$(("$(wc -l file.fastq)" / 4))"
# Count number of *.FNA or *.fasta files in current directory
find . -maxdepth 1 -name "*.FNA" -o -name "*.fasta" | wc -l
# Same as above, saving output to the file
echo -e "Hey, ${USER}, Number of FASTA files:\t$(find . -maxdepth 1 \
  -name "*.FNA" -o -name "*.fasta" | wc -l)"
# Compressing all *.fq or *.fastq files in parallel using all CPU threads
find . -name "*.f*q" -print | parallel bzip2 -v9 '{}'
# Statistics using BCFtools for all *.vcf.gz files in current directory
echo "Statistics of SNPs in VCF files using bcftools"
for VCFGZ in *.vcf.gz; do
  echo "Processing ${VCFGZ} at $(date)"
  bcftools stats --threads 2 -F reference.fasta "${VCFGZ}" > \
    ${VCFGZ%.vcf.gz}.stats.txt || exit 1
  echo
  done
```

Which system are we using?

```
uname -a # Information about Linux kernel (version, ...)
lsb_release -a # Information about Linux distribution release
cat /etc/os-release # Similar to above command
lscpu # Information about CPU; better than 'cat /proc/cpuinfo'
lsusb # List of devices on USB
lspci # List of PCI devices (graphic card, network card, ...)
lspci | grep -i vga # Get information about graphical card
lshw # Complete list of hardware; e.g. 'lshw -C memory' for RAM
hwinfo # Complete list of hardware; "hwinfo --network" for network devices
neofetch # System information tool
free -h # Available memory (RAM) and swap, -h for nice units
df -h # Free space on disk partitions, -h for nice units
lsmod # List loaded kernel modules
uptime # How long is the system running, number of users, average load
date # Date and time - plenty of options for formatting
mount # Information about mounted file systems
findmnt # Display mounted devices in tree structure
```

Processes — every running program has its own process ID

```
top # Listing of processes, quit using "q"
htop # Nice listing of processes (better version of top), quit using "q"
     # Allows also termination of processes etc.
pstree # See running processes with child processes, recursively
ps # processes related to actual terminal
ps ux # All user's processes
ps aux # All processes
pgrep application # Return PID (process ID) of application
# kill (terminate) process by name or process ID (PID)
# Find which PID has application to terminate
pgrep geany
14639 # Process ID (PID) of respective process
# Kill (stop by SIGTERM) selected application according to above PID
kill -SIGTERM 14639 # SIGTERM is "nice" termination, SIGKILL "brutal"
kill -SIGTERM $(pgrep geany) # Two above commands in one step; note $()
killall -SIGTERM geany # Select by name (more processes with same name)
```

Processes and users

```
# nice - how much resources will task use: from -20 (high priority - not
# "nice" process) to +19 (low priority - very "nice" process), default 0
nice -n 7 hard_task.sh # set priority 7 for newly launched task
renice 15 16302 # Change priority of PID 16302 to 15
sudo renice 15 16302 -u USER # Change priority of USER's process
whoami # What is my user name
id # Information about current user (user ID and group IDs)
who # Who is logged in
w # Who is logged in, more information
users # Plain list of currently logged users
finger # Information about users on current terminals
last # Last logged-in users
passwd # Change password
passwd USER # Change USER's password
groups # List your groups
# If user is added into new group, changes will be effective since
# next login
```

Users and groups

- These commands to manage users and groups do not have to work on all systems — depends on authentication methods used
- Commands modifying users and groups require root authentication

```
# Add new user
useradd new_user_name
# Modify user, see possible modifications
usermod --help
# Delete user
userdel user_name
# Add new group
groupadd new_group_name
# Modify group, see possible modifications
groupmod --help
# Delete group
groupdel group_name
```

Tasks regarding hardware, information and processes

1. Change your password on the remote server and/or virtual machine.
2. Who is and was recently logged in to the remote server?
3. Which information about hardware can you get from your local computer (if you are running Linux) and from course server? Why could such information be useful?
   1. Which disks are mounted?
   2. How many memory (RAM) slots are occupied and what is memory size?
4. Find your user ID on local computer and remote server. Do you think this has any implications when copying files from your local computer to server and back?
5. Terminate from command line some running graphical application. What is difference between `kill` and `killall`?
6. Change priority of some running (graphical) application.
7. Which applications are consuming the most resources (CPU, RAM)?

Network protocols I

- Every network communication protocol (e.g. browsing web or Skype) has its own distinct port ("door to access target computer") — it must be opened (by firewall)
- There are plenty of network services
  - Especially file servers and various data storages support more protocols to access the data — according to operating system and/or use case
  - Sometimes it can be tricky to pick up (and configure) the right protocol
- SSH — secure shell — command-line connection to remote server to work there (port 22), slide 86
  - Plenty of other protocols can run over SSH (SFTP, SCP, SSHFS, rsync, ...)
- Telnet — old deprecated insecure predecessor of SSH, never ever use it (port 23), only low-quality electronics still sometimes use it for administrative access
- FTP — file transfer protocol — outdated, no encryption (port 21), slowly replaced by FTPS (FTP secured) or SFTP/SCP (based on SSH)
  - Sometimes used only for download, e.g. `ftp://ftp.ebi.ac.uk/pub/`

Network protocols II

- FTPS — FTP with added connection encryption for higher security (port 21), common
- SFTP — FTP over SSH — common, secure (port 22), slide 139
- SCP — secure copy — uses SSH, but has restricted possibilities, common, secure (port 22), slide 139
- NFS — network file share/server — very common in UNIX world (commonly used e.g. by CESNET MetaCentrum and data storage — česky), commonly used to permanently connect to network server, share directories, etc. (port 20049), slide 141
  - NFS connection must be set up by administrator
- webDAV — file transfer over web (using WWW server) — not so common, but good (port 80 or 443 — same as WWW), slide 141
  - Accompanied by calDav and cardDav to share calendars and address books over the network
  - Used e.g. by ownCloud/nextCloud (also provided by CESNET — česky) — tool to synchronize and share files etc, similar to Dropbox, OneDrive or Google Drive
- SAMBA — UNIX connection to Microsoft network shares (port 5445), slide 141

Network protocols III

- web — "The Internet" for most of users (port 80 or encrypted 443)
- IMAP (port 143 or 993) and SMTP (port 25 or 465) to connect to e-mail server and send mails
- Messaging protocols like XMPP (Jabber and derived services like Google Talk or Facebook Messenger), IRC, ICQ, Skype, ...
- Databases (if accessible over the Internet), and another special services, use to be available on dedicated ports
- And much more (see `/etc/services`)...
- Port number can be changed in configuration of respective server service
- All firewalls on the way must allow communication on given port — some ports are commonly filtered in restrictive Wi-Fi networks or totalitarian countries

Basic network information and testing

```
hostname # Get name of the computer
ping web.natur.cuni.cz # Ping host. Is it alive? Cancel by Ctrl+C
traceroute www.metacentrum.cz # Get route (path) to the host
mtr seznam.cz # Combines ping and traceroute, quit with "q"
ip a s # Information about all network devices (MAC, IP address, ...)
ip r s # Show routes
# Does SSH work on the host? Verbose (-v), scan (-z), host, port number
# (22 for SSH, can be any)
nc -vz web.natur.cuni.cz 22
man nc # See for more information; "nc" is alias for "netcat"
ss -atn # Information about all network connections
ss -ntplu # Show open TCP/UDP ports
ss -anp # Show active connections
ss --help # More options...
# If using nmap at faculty, firewall disconnects you for 10 minutes!
nmap -r someserver.cz # Scan someserver.cz for opened ports
nmap botany.natur.cuni.cz --script ssh-hostkey # See SSH key
```

Transferring files from/to remote server

- On server side, same files can be accessible (shared) by more methods
- `curlftpfs` allows mount FTP as local directory, but FTP is outdated, insecure and not constructed to that usage...

```
wget https://some.address.cz/internet # Download file(s) from Internet
wget --help # -r for recursive download (whole web), -k to convert links
# curl is predecessor of wget, without parameter "-o" it prints remote
# content to standard output (typically screen)
curl https://some.server.cz/some/files -o localfilename
# Copy files (-r for recursive) over SSH from local computer
# to remote server or vice versa (just flip arguments)
scp -r localfiles remoteuser@remote.server.cz:/remote/path/
scp -r remoteuser@remote.server.cz:/remote/files /local/directory/
# scp behaves like cp, but works over SSH
```

- `rsync` is synchronization tool (commonly used for backups) able to connect to remote server (next slide)

Synchronization with rsync

- `rsync` has huge amount of possibilities (see `man rsync` or `rsync -h`)
- Works locally as well as over network
- It transmits only changes — very efficient
- Suitable for local as well as network backup
- Network address for rsync is written in same way as for `scp`
- `--delete` delete in target location files which are not in source location anymore
- `--progress` show progress percentage for every file
- `--exclude=*.jpg` skip JPG files
- For incremental backups use e.g. `duplicity`

```
1 rsync -arv somedirectory otherplace # All attributes, recursive, verbose
2 rsync -arv localdirectory user@remote.server.cz:/remote/directory/
3 rsync -arv user@remote.server.cz:/remote/data local/directory/
```

Connecting file systems on remote servers

- Target mount point must exist before mounting
- Servers can be accessed by IP address or hostname

```
# Mounting remote server over SSH (sshfs package must be installed)
sshfs USER@vyuka.natur.cuni.cz:/some/dir /local/mount/point
fusermount -u /local/mount/point # Disconnect SSHFS
# Mount Windows server (requires packages samba and cifs)
mount.cifs //windows.server.cz/Some/Directory /mnt/win -o \
  credentials=/path/to/password.file,uid=USER,gid=GROUP
# "password.file" contains login credentials to Windows server:
username=user.name
password=TopSecretPassword1
domain=DOMAINNAME
# Mount NFS share (NFS is common protocol in UNIX world)
mount -t nfs some.server.cz:/shared/directory /local/directory
# Mount webDAV folder (requires package davfs2 to be installed)
mount -t davfs https://owncloud.cesnet.cz/remote.php/webdav/ /local/dir
umount /mount/point # Disconnect CIFS/SAMBA, NFS, webDAV, ...
```

SSH keys I

Secure way how to connect to multiple servers with single key and more

- Secure way how to connect to multiple servers via SSH with single (or no) password using asymmetric encryption
- Command `ssh-keygen` generates pair of keys
  - Private key is typically in `~/.ssh/id_*` file(s) (according to cipher)
  - Public key is typically in `~/.ssh/id_*.pub` file(s) (according to selected cipher) and are copied to target servers
  - Private key is unlocked by password (might be without password) and it then allows to login to any server having the public key
  - Having only public key is not enough to login, private key without password is still not enough, but user must be sure private key is kept securely and not lost or stolen
  - When running it, `ssh-keygen` asks bunch of questions — unless having special needs, keep defaults (hit Enter), using password for key is optional (but recommended)

SSH keys II

Secure way how to connect to multiple servers with single key and more

- File `~/.ssh/known_hosts` contain servers you have ever connected to, and their SSH fingerprints (unique IDs) — if this changes, SSH complains a lot as it could be Man-in-the-middle attack
- File `~/.ssh/authorized_keys` contains public keys allowing logging to the machine
- `~/.ssh/config` can store various settings for particular servers
- SSH keys work in all applications using SSH (SFTP, SSHFS, rsync, Git, many graphical file managers, ...)
- User can have various local passwords on servers, but use single SSH key to connect to all of them, very convenient if connecting to multiple servers

Connect to SSH with key

No need to remember password for every server...

```
# Create the key (several options)
ssh-keygen -t rsa -b 4096 # Good security, portable
# ECC gives better security, but not all servers/applications support it
ssh-keygen -t ecdsa -b 521 # Higher security
ssh-keygen -t ed25519 # Same security as ecdsa, higher performance
# Empty (no) passphrase will connect to server without password
# Copy public key to remote server (private key must be kept locally)
ssh-copy-id user@remote.server.cz
# Now, public key is on the server and private key in local computer is
# unlocking the connection
# Unlock the key (no need in some distributions or if there is no
# passphrase) - must be done only once per user session
ssh-add
# Connect as usually
ssh user@remote.server.cz
```

Network tasks I

1. Create SSH key (if you don't have any), copy the public part to `vyuka.natur.cuni.cz` and connect there with SSH.
   1. What are advantages and disadvantages of having the key with or without password?
   2. Is there any reason for having multiple keys? Can you find any examples?
   3. Which services, protocols can use it? Find some examples.
2. Make in your notebook new empty directory and mount there via SSHFS your home directory on `vyuka.natur.cuni.cz` (or on some other server).
   1. Explore it. What is then path to files on the remote server?
   2. Disconnect the server when done.
3. Use `wget` and `curl` to download this presentation to your notebook (link at slide 11).
   - For `curl` be aware of setting of output — what is default behavior?
4. Use `scp` to copy directory with toy data and scripts from your home directory on `vyuka.natur.cuni.cz` into your notebook.

Network tasks II

5. Use `scp` to copy any file from your notebook into your home directory on `vyuka.natur.cuni.cz`.
6. Use `rsync` to update toy data and scripts in your notebook according to version on `vyuka.natur.cuni.cz` in `/srv/scripts_data`.
7. Use `rsync` to update directory with toy data files and scripts in your home directory on `vyuka.natur.cuni.cz` according to version in your notebook.
   - For `rsync` try to use various options, run it several times.
   - Explore help of `rsync`. Can you find there some useful parameters?
8. Get MAC and IP address of your notebook. Why can be such information useful?
9. Ping some web server. Is it alive and well reachable?
10. Why can be output of `traceroute` useful when you have problems with network?

Network tasks III

11. Mount (explore and then unmount) your CESNET ownCloud into your notebook. If unsure, consult help (česky).
    - It is available for all academics in the Czech Republic, just go to `https://owncloud.cesnet.cz` and login with your institutional credentials
    - Alternatively connect to some other ownCloud/nextCloud/generic webDAV server
12. What are advantages and disadvantages of mounting of remote servers (SSHFS, etc.) into local directories?
13. Connect to `vyuka.natur.cuni.cz` (or any other server) using `mc` and copy there and back some files. What are advantages and disadvantages of `mc` when compared with usage of `scp` or `rsync`?
14. Find in graphical interface of your computer how to connect via SSH/SFTP/SCP to transfer files. Connect to `vyuka.natur.cuni.cz` or some other server and transfer there and back some files.

Faculty web server(s)

- Login requires same credentials as to CAS (login name, no ISIC number)
- Faculty information are only in Czech
- It is Linux server running Debian
- Connect with SSH/SFTP/SCP to `web.natur.cuni.cz`
- Mainly used for webhosting, user's address will have form `https://web.natur.cuni.cz/~loginname/`, user can apply for another URL, for space for some projects, ...
- Every department also has dedicated space there, it can be used for various web projects, address can be discussed with IT department
- Personal web can be placed in `public_html` within home folder
- Users can apply for MySQL database or special settings
- Department of Botany (and some other departments) have their own web and file servers

Parallelization with GNU Parallel

- GNU Parallel can distribute task among CPU threads of one computer, or even among different computers in network
- It is not (so) effective for short/small tasks
- Important operands (for more see `man parallel` and next slides)
  - `{}` — input line — whole line read from input source (typically standard input)
  - `{.}` — input line without extension
  - `{/}` — base name of input line — only file name (without path)
  - `{//}` — dirname from input line (filename is removed)
  - `{/.}` — base name of input line without extension
  - `:::` — use arguments from command line instead of stdin (::: is placed after the command and before the argument)
  - `::::` — read from argument files
  - `-j` — number of jobs — if not provided, `parallel` will use all available CPU threads

GNU Parallel examples I

```
# Convert all images from JPG to PNG
find . -name '*.jpg' -print | parallel --bar convert {} {.}.png
# It might be possible to to use "'{}' '{.}.png'" instead
# Resize all images ("\" marks that command continue on next line)
find . -name '*.jpg' -print | parallel convert -resize 500x500 \
  -quality 75 '{}' '{.}-small.jpg' # or
parallel convert -resize 25% '{}' '{.}-small.jpg' ::: *.jpg
convert --help # See help of 'convert' from ImageMagick set
# Find WORD in huge text file (named "longfile" here) - this works
# but it is not possible to get line number (file is red in blocks)
parallel --pipe --block 10M -- grep --color=always WORD < longfile
nl longfile | parallel -k --pipe --block 20M -- grep WORD # Add line num.
# When needed to get phrase or regular expression (use parameter
# "-q" for escaping of shell special characters or extra quotes):
# "--" stops reading parameters for parallel
nl longfile | parallel -qk --pipe --block 20M -- grep "WORD TEXT" # or
nl longfile | parallel -k --pipe --block 20M -- grep '"WORD TEXT"'
```

GNU Parallel examples II

```
# Convert all WAV files into OGG
parallel -X oggenc ::: *.wav # -X parse as many parameters as possible
# Decompress all *.bz2 files in 'archive' directory using 2 CPU threads
parallel -j 2 -X bunzip2 -v ::: archive/*.bz2
# Run in parallel commands from command list file (list of commands)
parallel < command_list.txt # (each command on one line) or
parallel :::: command_list.txt
# Add same text to the end of multiple files
find . -name '*.txt' -print | parallel 'cat block_to_be_added.txt >> {}'
# Replace particular text in multiple files with sed and GNU Parallel
find . -name '*.txt' -print | parallel 'sed -i "s/XXX/YYY/g" {}'
# Launch MrBayes for multiple nexus files and create log file with
# starting and ending date and time
find . -name '*.nexus' -print | parallel 'echo -e "Start: $(date)\n" > \
  {}.log && mb {} | tee -a {}.log && echo "End: $(date)" >> {}.log'
# 'tee' copies output of the program into given log file
```

GNU Parallel examples III and tasks

```
# Create BCFtools statistics for all *.vcf.gz files
find . -name "*.vcf.gz" | parallel "echo '{/}' && bcftools stats -F \
  reference.fasta '{}' > '{.}'.stats.txt"
# Note that 'find' searches also in subdirectories
parallel --help # Basic help for GNU Parallel
man parallel # More information about GNU Parallel parameters
```

Tasks

1. Resize using `parallel` photos `foto_oxalis_*.jpg` to 1000x1000 px.
2. Convert all above JPG files to PNG.
3. Use several options how to run `parallel`.
4. Think about some example solving your practical task where GNU Parallel would be helpful.

Recording output of commands

- Alternative (commonly more convenient) to redirects of outputs to log file (slide 125) is command `tee`
  - Very useful if redirects would be harder to code
- `tee` can record
  - All output of the software (standard as well as error output)
  - Commands (keys) typed by user — it can be later reused to rerun the application — unique feature useful for certain software

```
# tee (-a for append to existing file) records output of any application
command | tee record.txt # tee will record whole output of command
tee record.txt | command # tee will record user input (NO command output)
# If software reads commands from user, we can reuse record next time:
command < record.txt # Empty lines are interpreted as Enter key
                     # Each line is used whenever command waits for new
                     # input (instead of typing, 'record.txt' is used)
```

Launching of tasks at certain time

- `at` can run command at certain time (`atd` daemon must be running)
- Tasks are running in background, outputs are mailed (e.g. to `/var/spool/mail/$USER`)

```
# Check status of atd daemon (it must run), start/stop/enable/disable it
systemctl status/start/stop/enable/disable atd.service
man at # Check for various possibilities of time settings
at HH:MM # Run commands at certain time (hour:minutes)
at> command1 # Add as many commands as you wish (separate by Enter)
at> # When done, press Ctrl+D to cancel giving commands to at
# Instead of manual typing of tasks, run script at certain time
at HH:MM -f somescript.sh # Run somescript.sh at certain time
at -l # List of scheduled tasks (alias is atq)
at -r <number> # Cancel scheduled task (according to number from at -l)
atrm # Alias for previous command
batch # Commands will be executed when system loads drops below 0.8 or
      # other value specified in configuration or startup of atd
```

Automated launching of tasks

- `cron` runs tasks repeatedly (`cron` daemon must be running)
- Scripts for tasks running hourly/daily/weekly/monthly can be copied into respective `/etc/cron.*/` directories
- Can be replaced by SystemD timers files (more complex, but more versatile)

```
# Check status of cron daemon (it must run), start/stop/enable/disable
systemctl status/start/stop/enable/disable cron.service
crontab -l # List user's cron tasks
crontab -e # Edit user's cron tasks (separate columns by spaces):
# Minute, Hour, Day in month, Month, Day in week, Command (absolute path)
# 0-59    0-23  1-31           1-12   0-6 starting with Sunday (WTF?)
  *       *     *              *      *            /usr/bin/command
  10      22    1              *      *    # 1st day in month, 23:11
  0       */3   *              *      *    # Every 3 hours
  0       11    *              *      0    # Every Sunday, 12:00
  30      3     */2            *      *    # Every second day, 4:31
  */15    *     *              *      0,4  # At Sun and Fri every 15 min
```

Text

Various aspects of working with text files

5 Text
- Reading
- Extractions
- AWK
- Manipulations
- Compressed text
- Comparisons
- Editors
- Regular expressions

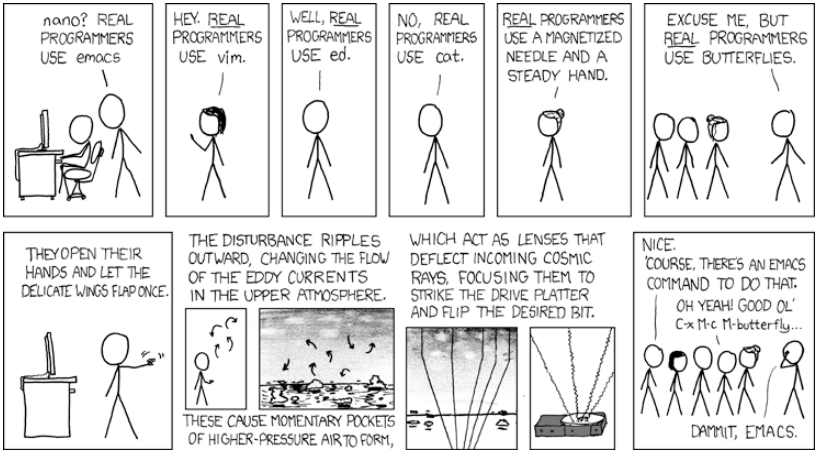
Everything is (text) file

- As UNIX configuration and outputs (logs, ...) are mostly saved as relatively simple text files, manipulations of any type with text files is one of the most common tasks
  - Similar situation is for e.g. molecular data — input/output data use to be text files with simple structure
- One of the most powerful features of BASH
- Some operations are complicated (e.g. complex manipulations with columns, various calculations) it is necessary to use AWK or Perl (probably the most advanced language working with text)
- Text-manipulating tools have very rich implementation of regular expressions (slide 191)
- Most of the operations are done in stream — per line — everything is very fast and memory efficient

Text tasks

Try all the commands from this chapter. It is one of key BASH features.

It **is** important to select **good** text editor...



https://xkcd.com/378/

Importance of good text editor

Can your text editor...?

- Show syntax highlight
- Show line numbers
- Show space between brackets
- Open any encoding and EOL
- Fold source code
- Show line breaks
- Mark lines
- Open multiple files

- Advanced search and replace
- Use regular expressions
- Make projects, add notes
- Use command line
- Check spelling
- Debug source code
- And more...

- Kate
- KWrite
- Vim
- GNU Emacs
- Geany
- Bluefish
- Gedit
- Notepad++
- Sublime
- VS Code
- Nano
- And more...

Converting the text

Prevent bad display and weird errors when launching scripts

```
unix2dos textfile # Convert text file from UNIX to Windows EOL
unix2mac textfile # Convert text file from UNIX to old Mac EOL
dos2unix textfile # Convert text file from Windows to UNIX EOL
mac2unix textfile # Convert text file from old Mac to UNIX EOL
enca -h # See usage - enca converts various encodings (similar to iconv)
enca file.txt # Detects encoding of file.txt
enca -x utf8 file.txt # Convert file.txt into UTF-8
# Converts encoding of input file (ISO-8859-2) to outfile in UTF-8
iconv -f ISO-8859-2 -t UTF-8 infile.txt > outfile.txt
iconv -l # List of available encoding to convert
```

- macOS uses same encoding and EOL as Linux (and rest of UNIX world), so there are no problems with compatibility
- Launching of bash script written on Windows on Linux/macOS will probably fail (because of different EOL)

Read text file

```
cat # Read or join (using redirects) files; 'cat --help' for options
cat long_text.txt # Print content of text file to the screen (stdout)
cat textfile1 >> textfile2 # Append textfile1 to the end of textfile2
nl long_text.txt # Like 'cat -n', prints textfile with line numbers
tac textfile # Like cat, but prints lines in reverse order
more long_text.txt # When textfile is long, prints screen by screen
                   # (space for next screen, q to quit)
# Better version of more - you can scroll up and down by PgUp, PgDown,
# arrows, searching by / (type searched string, hit Enter, n for next,
# twice ESC to quit), q to quit viewing (also used by man)
less long_text.txt
fmt long_text.txt # Basic formatting of text - joining of commented
                  # lines, line breaks to break too long lines, ...
fmt textfile > formatted_file # Save output of fmt into new file
wc long_text.txt # Calculates lines, words and bytes in text file; 'wc -l'
                 # for only lines, '-m' for characters, '-w' for words
mc # It has text viewer (F3) and editor (F4)
```

Get part of text file (by lines) I

```
# head and tail are very convenient to quickly check file structure
head -n N textfile # Print first N lines from textfile
tail -n N textfile # Print last N lines from textfile
head -n-N textfile # Print textfile without last N lines
tail -n+N textfile # Print textfile from Nth line to the end
# Split text file on selected pattern - creates new files xxXY
# Pattern can be regular expression - set it carefully
# {*} says to repeat operation as many times as possible
csplit textfile '/pattern/' '{*}' # pattern itself is inside '/___/'
# Split text file by lines into multiple files
# e.g. suffix will be of length 5 (-a 5), suffix will be numeric (-d),
# list.txt will be split every 10 lines (-l 10) and output names will
# start by 'lists_'
split -a 5 -d -l 10 list.txt lists_
split -t ... # Split by separator, not by newline
# Both csplit and split are very versatile, see their help
```

Get part of text file (by lines) II

```
grep --help # See plenty of options
grep -parameters pattern textfile # Write lines containing pattern
grep user /etc/passwd # Write all lines in passwd file containing "user"
cat /etc/passwd | grep user # Same as above, common, but superfluous style
grep -v user /etc/passwd # Write all lines in passwd NOT containing "user"
grep -c user /etc/passwd # Get number of lines in passwd containing "user"
grep -i USER /etc/passwd # -i for case insensitive
grep -q ... # quiet - no output (only T/F) - good for testing in scripts
grep -ls user /etc/* # -l print files with pattern, -s suppress errors
grep "longer text" textfile # Extract whole phrase (must be quoted)
```

- Grep supports regular expressions, slide 191
- Grep works per-line, multiline patterns are more or less impossible (use AWK or Perl instead) — this is general limitation of basic tools
- Grep (and sed and other tools) in macOS is outdated, missing plenty of functions — use version from Homebrew (slide 80)

Work with columns

- `cut` extracts columns, `paste` joins, `column` reformates
- BASH can not select column according to its name (Perl can do that)

```
cut column/delimiter+field textfile
cut -c 1 /etc/group # Get first character
cut -c 1-5 /etc/group # Get character 1-5
cut -c 4- /etc/group # Get character 4 and more
cut -c 2,5,7 /etc/group # Get characters 2, 5 and 7
cut -d ':' -f 1 /etc/group # Select 1st field separated by ":"
cut -d ':' -f 2-4 /etc/group # Select fields 2-4 separated by ":"
cut -f 1,2 cut_awk_test_file.tsv # get columns 1 and 2 separated by TABs
# Add second file as second column
paste file1 file2 > outputfile
# Output will be two columns (from file1 and file2) separated by TAB
paste -d "|" diff_test_file_1.txt diff_test_file_2.txt # -d for delimiter
# Swapping columns is not very comfortable...
paste <(cut -f 2 cut_awk_test_file.tsv) <(cut -f 1 cut_awk_test_file.tsv)
ls -l | column -t # Reformate input as table (compare with 'ls -l')
```

Examples of usage of head, grep and cut with our data I

```
head long_text.txt # Beginning of the file
# Extract names of FASTA sequences
grep "^>" Oxalis_HybSeq_nrDNA_selection_alignment.fasta
# How many sequences are there?
grep -c "^>" Oxalis_HybSeq_nrDNA_selection_alignment.fasta
# Extract every FASTA sequence containing "AAAAAA" and name of
# respective FASTA sequence
grep -n -B 1 AAAAAA Oxalis_HybSeq_nrDNA_selection_alignment.fasta
# Get column of pairwise identities of sequences exported as TSV from
# Geneious - discard header (1st line, print from 2nd line) and extract
# second column (separated by TAB)
tail -n+2 cut_awk_test_file.tsv | cut -f 2 | less
# Number of occurrences of word "Gregor" in file long_text.txt
grep Gregor long_text.txt | wc -l # Number of matching lines
grep -c Gregor long_text.txt # Number of occurrences
ls -1 *.sh | wc -l # How many BASH script are in current directory
```

Examples of usage of head, grep and cut with our data II

```
# Extract from FASTA only sequences (discard sequence names) into seq.txt
grep -v "^>" Oxalis_HybSeq_nrDNA_selection_alignment.fasta > seq.txt
# Save names of oxalis* JPG files into list_oxalis_photo.txt and see it
ls -1 *oxalis*.jpg > list_oxalis_photo.txt && cat list_oxalis_photo.txt
```

Tasks

1. How many samples of *Oxalis hirta* are in `Oxalis_HybSeq_nrDNA_selection_alignment.fasta`?
2. From `Oxalis_HybSeq_nrDNA_selection_alignment.fasta` extract only sequences of *Oxalis hirta* and save them into `oxalis_hirta.fasta`.
3. From `Oxalis_HybSeq_nrDNA_selection_alignment.fasta` extract only sequences of *Oxalis amblyosepala* and *O. gracilis* and save them into `oxalis_spp.fasta`.

Get a column with awk

- AWK is scripting language mainly for text manipulations
- Can do things other BASH tools can not (easily) do — better manipulation with columns, calculations, work with multi-line patterns, ...
- Has complicated syntax, it is hard to read, it is not similar to other tools — Perl can do more and is more common (learn it instead)...
- Supports regular expressions, slide 191
- Can not select column according to its name (Perl can do that)
- For more information see manuals
  https://www.gnu.org/software/gawk/manual/,
  https://en.wikibooks.org/wiki/An_Awk_Primer and
  https://www.grymoire.com/Unix/Awk.html

```
awk 'regexp { commands parameters }' file # General syntax
```

AWK examples I

```
# Print last column (separated by tab, built-in variable $NF)
awk '{print $NF}' cut_awk_test_file.tsv
# Select 2nd column ($2; separated by tab)
awk '{print $2}' cut_awk_test_file.tsv
# Print columns 3 and 2 (in this order)
awk '{print $3, $2}' cut_awk_test_file.tsv
# Get column 5 and 1 (in this order, separated by ":") from /etc/passwd
# (only lines containing "home") print ", username:" between the columns.
# Note usage of commas and consequences to output.
grep home /etc/passwd | awk -F ':' '{print $5 ", username:", $1}'
# Separate columns by TAB, /^d/ for lines starting with "d" (only dirs)
ls -l | awk '/^d/ { print $8 "\t" $3 }'
# Print on even lines ">", former column 1, new line, former column 2:
# 2 columns into 2 lines (create FASTA from tabular record)
awk '{print ">"$1"\n"$2}' awk_test_file.tab | less -S
# Print field 1, TAB (\t), length of field 2, TAB and field 2
awk '{print $1"\t"length($2)"\t"$2}' awk_test_file.tab
```

AWK examples II

```
# If field (column) 2 contains exactly 100.0%, print whole line ($0)
awk '{if($2=="100.0%"){print $0}}' cut_awk_test_file.tsv
# Field 1 is numeric (less then 5 digits) - add leading zeroes
awk '{printf "%05d\n", $1;}' awk_test_file.tab
# As previous, but add leading zeroes to field 1 and print whole line
awk '{$1=sprintf("%05d", $1); print $0}' awk_test_file.tab
# Field 6 is numeric, select lines where field 6 is higher than 200
awk -F '\t' '$6>200' cut_awk_test_file.tsv # Separated by TABs (\t)
# Print fields 4 and 5 (fields are separated by "_" or TAB)
awk -F '[_\t]' '{print $4, $5}' cut_awk_test_file.tsv
# Precede each line by its line number for all files together, with TAB
# (i.e. print line number (NR) and then whole original line ($0))
awk '{print NR "\t" $0}' diff_test_file_*
# Substitute "a" with "XXX" ONLY for lines which contain "the"
awk '/the/{gsub(/a/, "XXX")}; 1' diff_test_file_1.txt
```

AWK examples III

```
# For every 4th line starting from line 2 of FASTQ file (from line 2
# every 4th line contains the DNA sequence) print its length (bzcat
# prints content of file compressed by bzip2)
bzcat Oxalis_hirta_R1.fastq.bz2 | awk 'NR%4==2{ print length($0) }'
# If sequence is longer that 500 bp (length of field 2), print its name
# (field 1) like this "Seq. name: TAB the sequence name (ID)"
awk '{if(length($2)>500){print "Seq. name:\t" $1}}' awk_test_file.tab
# Extract from awk_test_file.tab sequences over 500 bp, sort theme by name
# and save them as FASTA file
awk '{if(length($2)>500){print $0}}' scripts_data/awk_test_file.tab | \
  sort -n | sed 's/^/>/' | sed 's/[[:blank:]]\+/\n/g' > 500bp.fasta
```

Sorting I

```
sort file # Sorting is influenced by locale setting (e.g. Czech "ch")
LC_ALL=C sort ... # To force use of English locale use
# Take into account only spaces and alphanumerical characters (ignore
# any other)
sort -d textfile # Only spaces and alphanumerical characters
sort -r textfile # Reverse order
sort -f textfile # Ignore character case (not case sensitive)
sort -m textfile1 textfile2 # Merge already sorted text files
sort -u textfile # Print only first of multiple (repeated) entries
# Extract only unique sequences from FASTQ
bzcat Oxalis_hirta_R1.fastq.bz2 | awk 'NR%4==2{print $0}' | sort -u
sort -b textfile # Ignore leading blanks (space on beginning of line)
sort -k 2 -n cut_awk_test_file.tsv # Sort according to 2nd field
# Filters following identical lines - only unique are printed
uniq textfile
sort textfile | uniq # To get unique lines from whole file, sort it first
```

Sorting II

```
uniq -c textfile # Add number of occurrences before each line
uniq -d textfile # Print only repeated lines
uniq -i textfile # Ignore case (not case sensitive)
uniq -s N textfile # Skip first N characters
uniq -u textfile # Print only not-repeated lines
# How many times is each taxon presented in
#  Oxalis_HybSeq_nrDNA_selection_alignment.fasta
grep -o "Oxalis_[a-z]\+" Oxalis_HybSeq_nrDNA_selection_alignment.fasta \
   | sort | uniq -c | sort -r
# Find sizes of directories in /home (ignore errors caused by restricted
# permissions) and sort output according to sizes
du -sh /home/* 2>/dev/null | sort -h | sed 's/\/home\///'
# Sort *.sh scripts according to number of lines
wc -l *.sh | head -n-1 | sort -bn
```

Replacements with tr

- `tr` replaces or deletes characters from standard input and writes result to standard output — use pipes and/or redirects

```
# Replace space by TAB in inputtextfile, save result as outputtextfile
cat inputtextfile | tr " " "\t" > outputtextfile
# Delete "text" from each line and print it to standard output (screen)
cat inputtextfile | tr -d "text"
# Replace every occurrence of A, B, C or D by a new line (\n)
cat inputtextfile | tr "[ABCD]" "\n" > outputtextfile
# Replace capital letters by small ones
tr "[A-Z]" "[a-z]" < diff_test_file_1.txt > outputtextfile.txt
# Alternative (easier reading) of previous command:
cat diff_test_file_1.txt | tr "[:upper:]" "[:lower:]" > outputtextfile
# Replace all new lines (line breaks) by TABs
cat diff_test_file_1 | tr "\n" "\t"  > outputtextfile
# Discard all new lines - output will be one line
tr -d "\n" < textfile > /dev/stdout # stdout is typically screen
tr --help # See another possibilities for pattern to find/replace
```

Replacements with sed

- `sed` supports regular expressions, see slide 191 (same as in `grep` and `vim`), with parameter `-r` can use extended regular expressions (do not confuse — the syntax is slightly different, richer)
- Output is written to standard output — use pipes, redirects or `-i` to modify the file in place (without printing of output)
- macOS has old outdated versions of `grep`, `sed` and other tools (richness of regular expressions is poor) — use versions from Homebrew (slide 80) or search Internet how to modify the patterns...
- Option `-s` separates multiple files (otherwise lines in multiple files are calculated as one stream)
- Option `-n` use to be used when deleting lines or printing only specific lines to suppress other lines (see examples)
- See manuals `https://www.gnu.org/software/sed/manual/` and `https://www.grymoire.com/Unix/sed.html`

Sed examples I

- Various parameters, modificators, operators can be combined...

```
sed 'operator/FindToReplace/Replace/modificator' textfile > newtextfile
# Search and replace ("s") all occurrences ("g") of "find" by "replace"
sed 's/find/replace/g' textfile
# Replace third occurrence of pattern on every line
sed 's/pattern/Replace/3' # 's/.../.../3' replace only third occurrence
sed '1,7s/...' # To work only on particular line, place single number or
sed '5s/...'   # range (e.g. 1,7) right before "s" ("$" for last line)
sed '1~2n;s/F/R/g' # Work on every second line, starting by line 1
sed -n '2~10p' # Print every 10th line, starting with line 2
seq 1 100 | sed -n '2~10p' # Example of above pattern (see "seq 1 100")
# Replace first TAB (\t) on each line by new line (\n)
sed 's/\t/\n/' textfile
# Convert sequences in tabular format into FASTA (place ">" to the
# beginning of the line, replace TAB "\t" by newline "\n")
sed 's/^/>/' awk_test_file.tab | sed 's/\t/\n/' > seq.fasta
```

Sed examples II

```
# Convert FASTQ sequences into FASTA (on every 4th line, starting with
# line 1 replace "@" by ">", print every 4th line, starting by line 2)
bzcat Oxalis_hirta_R1.fastq.bz2 | sed -n '1~4s/^@/>/p;2~4p' > seq.fasta
sed -i 's/find/replace/g' directory/* # Process all files in directory
# Convert all capital letters into lower
sed 's/[A-Z]/\L&/g' inputtextfile > outputtextfile # And vice versa:
sed 's/[a-z]/\U&/g' inputtextfile > outputtextfile
# Groups to remember work in same way in sed, grep as well as vim:
# \(ToRemember\) # Remember expression in brackets and then use
# \Number # Use remembered expression (numbered from one: \1, \2, \3, ...)
# Take output of ls -l and replace value of $USER by "$USER-RULEZZZ"
ls -l | sed "s/\($USER\)/\1-RULEZZZ/g" # Note " to use the variable
# Replace size column (2nd numeric) by "size:TAB<file size>b"
# Second sed replaces any white spaces by single TAB
ls -l | sed 's/\([0-9]\+\)/size:\t\1b/2' | sed 's/[[:blank:]]\+/\t/g'
sed '/^$/d' long_text.txt | less # Delete blank (empty) lines
head long_text.txt | sed '6d' # Delete 6th line
```

Sed examples III

```
sed 's/ *$//' # Delete extra spaces on the end of lines
sed '/GUI/d' # Delete all whole lines containing "GUI"
sed 's/GUI//g' # Delete all occurrences of "GUI" (not whole lines)
sed '4 i\Linux is great.' diff_test_file_1.txt # Insert to 4th line
sed '3 a\Linux is great.' diff_test_file_1.txt # Insert after 3rd line
# Insert text to the beginning of the 3rd line (compare with previous)
# "^" is beginning of line, $ end ('$/...' last line)
sed '3s/^/INSERT/' diff_test_file_1.txt
# From ls -l keep number of links (1st numeric column after permissions)
# and then flip user and group and print it as "group:user"
ls -l | sed 's/ \([[:digit:]]\+\) \([[:alnum:]]\+\) \([[:alnum:]]\+\) /
  \1 \3:\2 /g' # Note separating spaces (previous line ends with space)
ls -l # Compare to the previous command, explain behavior of sed pattern
# Escaping - replace dot by comma (dot means any single character)
sed 's/\./,/g' diff_test_file_1.txt # \ escapes following character
sed 's/./,/g' diff_test_file_1.txt # Compare with the previous example
```

Sed examples IV

```
# Replace any of characters within [...] by some pattern
sed 's/[abcd]/X/g' diff_test_file_1.txt # Compare with reverse case:
sed 's/[^abcd]/X/g' diff_test_file_1.txt # "[^...]" means anything else
sed -i ... file.txt # In-place editing - file is edited, no output to
                    # standard output (no need for redirects and pipes)
ls -l | sed 's/[0-9]\{4,\}/BIG!/' # Replace 4 or more digits by "BIG!"
sed -E ... ; sed -r ... # Use extended regular expressions (see further)
# Remove blank space (spaces or tabs) on beginning of each line
sed 's/^[[:blank:]]\+//' textfile
# Remove suffix from names of *.sh files - compare variants
ls -1 *.sh | sed 's/\.sh$//' # Note '$' to ensure end of line
ls -1 *.sh
find . -name "*.sh" | sed 's/^\.\///;s/\.sh//' # Note chaining of patterns
find . -name "*.sh"  # It searches also in subdirectories
```

- Sed does not perform well on multi-line patterns — better is to use AWK or Perl

Joining

- Generally, most of tools work per-line, `paste` appends columns (see also slide 164)
- `join` compares every matching lines (by default 1^{st} field) and creates all combinations — ensure to have sorted input files with unique text
  - E.g. if 1^{st} file contains `A B` and `A C` and 2^{nd} file `A D` and `A E`, the result will be `A B D`, `A B E`, `A C D` and `A C E`

```
# Add file to the end of another text file
cat file1 >> file2 # file2 will contain both files, file1 is unchanged
# Compare two sorted text files and write shared lines
# (duplicitous lines are shown just once)
join textfile1 textfile2 > outputfile
# If used on wrong files, it can create huge file
seq -f "1 %g" 100 > aaa && less aaa
seq -f "1 %g" 100 > bbb && less bbb
join aaa bbb | wc -l
join --help # See more options...
```

Variants of basic commands for processing of compressed text files I

- Many tools are able to directly handle files (i.e. single text files) compressed by `gzip`
- For files compressed by `gzip` use `zcat` (instead of `cat`), `zdiff` (instead of `diff`), `zegrep` (instead of `egrep`), `zfgrep` (instead of `fgrep`), `zless` (instead of `less`), `zmore` (instead of `more`), ...
- For files compressed by `bzip2` use `bzcat`, `bzdiff`, `bzegrep`, `bzfgrep`, `bzless`, `bzmore`, ...
- For files compressed by `lzma` or `lzma2` (`xz`, `lzma`) use `lzcat`, `lzdiff`, `lzegrep`, `lzfgrep`, `lzgrep`, `lzless`, `lzmore`, ...
- Sometimes these variants are used automatically when user works with compressed file
- `mc` can also do the job
- These tools sometimes don't have all options of the "standard" applications

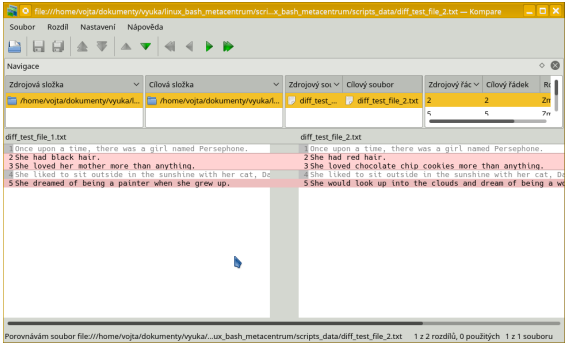
Variants of basic commands for processing of compressed text files II

```
# Extract lines with grep
zgrep FORMAT arabidopsis.vcf.gz
# 'less -S' displays long lines without indent (use R/L arrows to move),
# zless doesn't have such option - use as follows:
zcat arabidopsis.vcf.gz | less -S # zless doesn't have the -S option...
# Display file compressed by bzip2
bzless Oxalis_hirta_R2.fastq.bz2
# Get end of the file
bzcat Oxalis_hirta_R2.fastq.bz2 | tail
# Get number of lines
bzcat Oxalis_hirta_R2.fastq.bz2 | wc -l
```

- Not all commands to work with text have variant to work with compressed files...
- Variants to work with compressed files sometimes don't have all options...
- Sometimes it's the best to pipe `zcat` / `bzcat` and another tool

Comparisons

- Graphically compare two files by e.g. Kompare, DiffMerge, Meld, ...
- Most common is usage of GNU diffutils (next slide), see manual `https://www.gnu.org/software/diffutils/manual/`
- See also guide with examples, česky příklady na diff a porovnání dvou textových souborů



comm and diff

```
cat diff_test_file_1.txt diff_test_file_2.txt # See and use examples
# Compare two sorted columns: 1st column - lines only in textfile1; 2nd
# column - lines only in textfile2; 3rd column - lines in both files
comm textfile1 textfile2
# Don't show 2nd column (similarly -1, -3)
comm -2 diff_test_file_1.txt diff_test_file_2.txt
# Show differences between text files
diff diff_test_file_1.txt diff_test_file_2.txt
  # First number shows line(s) in 1st file, then if add/delete/change
  # and last number shows line(s) in the second file, <> show direction
diff -e diff_test_file_1.txt diff_test_file_2.txt # More simple output
diff -c diff_test_file_1.txt diff_test_file_2.txt # Show context
diff -u diff_test_file_1.txt diff_test_file_2.txt # Better, most common
diff -y diff_test_file_1.txt diff_test_file_2.txt # In two columns
colordiff # Same usage and parameters as previous, colored output
colordiff -u diff_test_file_1.txt diff_test_file_2.txt # Most common
```

diff and patch

- Users of `vim` (slide 188) can display the diff in `vim`, or use `vimdiff`
- In vimdiff swap between panes by `Ctrl+W+W`
  - Can open more files, works like standard `vim` (from slide 188), individual files can be saved

```
# Display diff in vim
diff -u diff_test_file_1.txt diff_test_file_2.txt | view -
# vimdiff can show more colors, launches vim (exit by <ESC>:q! Enter)
vimdiff diff_test_file_1.txt diff_test_file_2.txt
```

- Saves difference between two files — it can be later used as template to modify original file
- Single patch file can contain changes from multiple files
- Patch file from `diff -u` is common way how to send changes, improvements, ...

```
# Create the diff file
diff -u diff_test_file_1.txt diff_test_file_2.txt > difference.diff
# Apply the patch - xxx_file_1.txt is patched according to xxx_file_2.txt
patch < difference.diff # or
patch diff_test_file_1.txt difference.diff
```

Command line text editors

- `nano`, `pico` and `mc` are very simple, just for very basic text editing in command line or until you learn `vim` (graphical version is **gVim**) or `emacs` (graphical versions are also available, just search for **Emacs** in your distribution software manager)
- You can work most of the time in graphical editors (slide 159)
- Emacs and Vim are extremely rich, but having completely different approach — when you get use to one, you can't use the another

```
nano textfile # Enhanced clone of pico, basic simple text editor
pico textfile # Basic simple text editor
mc # Use its internal editor, just very basic (press F4 on the file)
emacs textfile # Extremely feature rich (including file browser and
               # many tools), exit by Ctrl+X and Ctrl+C
vim textfile # Probably the most common, as rich as Emacs (see further)
vimtutor # Launch tutorial to learn Vim (in various languages)
```

- Task: Run `vimtutor` and follow instructions there.

The editors and their usage

- In `nano` and `pico` see bottom line for commands
  - `Ctrl+O` to write the file, `Ctrl+X` to quit the editor, `Ctrl+G` for help (`^` stands for `Ctrl` key)
- In `mc` highlight the file to edit and press `F4`
  - `F2` to save, `F10` to quit, `F1` to help, `F9` for top menu (navigate with arrows, cancel with double `Esc`), it is possible to use mouse
- **Emacs** use huge number of commands with `Ctrl` key (basics on next slide)
  - See `https://www.gnu.org/software/emacs/tour/`, `https://tuhdo.github.io/emacs-tutor.html` and `https://systemcrafters.net/emacs-essentials/absolute-beginners-guide-to-emacs/`
- The most common (but very complex and specific) is **Vim**
  - See `https://vim-adventures.com/` to play a game and learn Vim
- **Emacs** and **Vim** have huge number of possibilities and support for plugins and scripts, but completely different usage style — one person can really learn only one...
- If regularly working in command line, master any command line editor

Emacs basics

- Feature-rich ecosystem — not only text editor, also file manager, debugger, integrated development environment, extra plugins are available, ...
- Keyboard commands are noted as e.g. `C-x`, where "`C`" stands for (mostly) `Ctrl` key, `M-x`, where "`M`" stands mostly for `Alt` key (`Meta`), sometimes for `Win` key

- `C-h C-h` help (twice `Ctrl + H`)
- `C-x C-c` quit (`X-g` for particular buffer, etc.)
- `C-x C-f` open file
- `C-x C-s` save file
- `C-x C-w` save file as
- `C-_` undo
- `C-s` search forwards
- `C-r` search backwards
- `C-left` move one word left
- `C-right` move one word right
- `C-up` move one paragraph up
- `C-down` move one paragraph down
- `M-%` search and replace pattern

Vim

Vim has three different working modes

1. **"Normal"** — nothing is displayed in bottom left corner, every key has some meaning (next slide) — very powerful manipulations with text
   - `i` or `Insert` key to enter *insert mode*, `:` to enter *command mode*
2. **Insert** — in bottom left corner "`-- INSERT --`" is displayed, the most familiar mode, normal typing etc., exit to normal mode by `ESC` key
3. **Command** — in bottom left corner `:` is displayed, awaits commands, exit to normal mode by `Backspace` key (delete "`:`")

- See documentation `https://vim.fandom.com/wiki/Vim_Tips_Wiki`, `https://www.vim.org/docs.php`; česky `http://www.nti.tul.cz/~satrapa/docs/vim/`
- For interactive learning try `https://vim-adventures.com/` (Task: Play it for a while.:-) or command `vimtutor`

Normal and command Vim modes

- **Normal mode** (press `ESC`)
  - `dd` cut current line
  - `r` replace single character below cursor (type then character to be placed instead of the original one)
  - `v` for selection of text (and then e.g. delete by `d` or copy by `y`)
  - `y` copy selection
  - `x` cut selection (always to clipboard, it can be pasted by `p`)
  - `p` paste
  - *number* to get to line of particular line number
  - `u` to undo last change(s)
  - `/ToSearch` search "ToSearch" (once press `Enter`, `n` for next occurrence, quit with `Esc`)
- **Command mode** (press `ESC :`)
  - `w` write file
  - `q` quit
  - `q!` quit and discard changes
  - `%s/...` to search and replace as in `sed`
  - `syntax on/off` turn syntax highlight on/off
  - `set nu` show line numbers

Regular expressions are useful...

- Find text according to a pattern
- Manipulate the text — flip, reformat, replace, ...
- Syntax is variable among programming languages and applications
- There are commonly more solutions for one task
- Well supported in `grep`, `sed`, `awk`, `vim`, `emacs`, ...
- Probably the most advanced is Perl
- Implementations can differ among tools

https://xkcd.com/208/

Regular expressions I

- Implementation in `vim`, `sed`, `grep`, `awk` and `perl` and among various UNIX systems is almost same, but not identical — can be confusing...
- **grep**, **sed** and **vim** require escaping of `+`, `?`, `{`, `}`, `(` and `)` by backslash `\` (e.g. `\+`, see also next slides)
  - These characters have special meaning (see further) — consider if you search for e.g. literal `+` or not (and then you can use it's special meaning)
- **egrep** (extended version, launched as `grep -E ...` or `egrep ...`), **sed** with extended reg exp (`sed -r`) and **perl** do not require escaping (simply just e.g. `+`, not `\+`)
- Mastering regular expressions require practicing – solve practical problems and see their power
- Read https://en.wikibooks.org/wiki/Regular_Expressions, https://www.grymoire.com/Unix/Regular.html, https://www.regular-expressions.info/

Regular expressions II

- Česky `http://www.nti.tul.cz/~satrapa/docs/regvyr/`, `https://www.root.cz/serialy/regularni-vyrazy/` a `https://www.regularnivyrazy.info/`
- Manuals for Grep, Vim, Sed, Awk, Perl (newer Perl 6 Raku), ...

- See sed examples, slide 175; and next slides
- macOS has by default very outdated version of `sed` and another tools — it does not have all advanced features — users need to install e.g. `gnu-sed` formulae from Homebrew (slide 80), similarly for Grep, AWK, ...
- Do not confuse with shell globbing (slide 110) — regular expressions are used withing particular application (GNU Sed, GNU Grep, Perl, ...), while shell globbing is in-build BASH feature
  - Globbing as well as regular expressions match/expand particular text string (in case of globbing typically file names)

Regular expressions III

- Regular expressions mostly must be quoted (`'...'`) **not** to be interpreted by shell, they work mostly with **text** files (their versatility allows to use them to work with e.g. molecular data)

- Word processors (LibreOffice, ...), graphical text editors, etc. usually also support regular expression, more or less following syntax below, but sometimes bit simplified
- `.` — any single character
- `*` — any number of characters/occurrences of pattern (including 0)
- `+` — one or more occurrences of the preceding reg exp
- `?` — zero or one occurrences of the preceding reg exp
- `[...]` — any character in the brackets (can be list like `[abcd]` or range like `[a-kxz4-8_-]`)
- `[^...]` — reverse case — all characters except newline and those listed in brackets

Regular expressions IV

- `^` — first character of reg exp — beginning of the line
- `$` — last character of reg exp — end of the line
- `\{n,m\}` — range (number) of occurrences of single character (from *n* to *m*)
- `\{n\}` — exactly *n* occurrences
- `\{n,\}` — at least *n* occurrences
- `\` — escape following special character (e.g. `\.` to literally search for dot and not “any single character”)
- `|` — either the preceding or following reg exp can be matched (alternation), in `grep` etc. escape it and use as `\|`

Regular expressions V

- `\(...\)` — remembered group reg exp (numbered, starting with 1) — can be called by \n, where *n* is number of the group (starting with 1, see examples further)
- `\<`, `\>` — word boundaries
- `[[:alnum:]]` — alphanumerical characters (includes white space), same like `[a-zA-Z0-9]`
- `[[:alpha:]]` — alphabetic characters, like `[a-zA-Z]`
- `[[:blank:]]` — space and TAB
- `[[:cntrl:]]` — control characters
- `[[:digit:]]` — numeric characters, like `[0-9]`
- `[[:graph:]]` — printable and visible (non-space) characters

Regular expressions VI

- `[[:lower:]]` — lowercase characters, like `[a-z]`
- `[[:print:]]` — printable characters (includes white space)
- `[[:punct:]]` — punctuation characters
- `[[:space:]]` — white space characters
- `[[:upper:]]` — uppercase characters, like `[A-Z]`
- `[[:xdigit:]]` — hexadecimal digits
- `^$` — blank line
- `^.*$` — entire line whatever it is
- ` +` — one or more spaces (there is space before plus)
- `&` — content of pattern that was matched

Grep and sed examples I

- Be sure to understand all syntax on this and following slide...

```
# Extract sequences with at least 5 A bases in line
grep "A\{5,\}" Oxalis_HybSeq_nrDNA_selection_alignment.fasta
# Extract DNA sequence string ATCG or ATGC
grep "ATCG\|ATGC" Oxalis_HybSeq_nrDNA_selection_alignment.fasta
# Get quality of Illumina reads mapping to reference in genomic VCF
zcat arabidopsis.vcf.gz | grep -o "MQ=[[:digit:]]\+" | sed 's/^MQ=//'
# How many times there is a direct speech (text between "...")
grep -o '"[[:upper:]][a-zA-Z0-9,\.\?\! ]\+"' long_text.txt | wc -l
# Add after dot on the end of the line by extra line break
sed 's/\.$/.\n/' long_text.txt | less
# Add HTML paragraph tags (<p> and </p>)
sed -e 's/^/<p>/' -e 's/$/<\/p>/' long_text.txt | less
# Make first word of every paragraph bold in HTML (<strong>...</strong>)
sed -e 's/^/<strong>/' -e 's/^[[:graph:]]\+/&<\/strong>/' long_text.txt
```

Grep and sed examples II

```
# How many times is each word in the text
grep -o "\<[[:alpha:]]\+\>" long_text.txt | sort | uniq -ic | less
# List all Internet web links
grep -o 'https\?://[a-zA-Z0-9\.()/:\-]\+' long_text.txt
# Convert selected letters to upper case
sed 's/[acegikmoqsuwy]/\U&/g' diff_test_file_1.txt
# From file listing (compare with 'ls -l') remove permissions and number
# of links on the beginning, flip user and group ownership and add labels
# Note usage of numbered groups
# Note that unmatched part o line is intact
ls -l | sed 's/^[[:graph:]]\+[[:blank:]]\+[0-9]\+[[:blank:]]\+
  \([[:alnum:]]\+\)[[:blank:]]\+\([[:alnum:]]\+\)/GRP: \2\tUSR: \1/g'
# Create list of samples (e.g. as input in script for some application)
SAMPLESLIST=$(find . -name "*.jpg" | sed 's/^\.\///' | sed 's/^/-I /' |
  tr "\n" " ")
echo $SAMPLESLIST # What would be difference from quoted "$SAMPLESLIST"?
application $SAMPLESLIST -method X -out Y ... # Rationale of such listing
```

Regular expressions tasks

1. Remove "`S`" codes, replace underscore by dot and space (`. `), and capitalize initial "`O`" in FASTA names in `oxalis_assembly_6235.aln.fasta`, e.g. from `>o_annae_S499` to `>O. annae`.
2. Extract from `arabidopsis.vcf.gz` values of `DP` (only numbers), sort them and print on single line, separated by commas.
3. Determine, which sequence(s) of `Oxalis_HybSeq_nrDNA_selection_alignment.fasta` has block of missing data (`N`) or spaces (`-`) longer than 10 bp.
4. From file `cut_awk_test_file.tsv` remove with `sed` column `Description` (`"Assembly of # reads: ..."`).
5. Think about any task (manipulation with your data, ...) you are (sometimes) dealing with, which could be simplified/solved by using regular expressions. Try to solve it. Discuss it with others.

Scripting

Basics of writing scripts in BASH

6 Scripting
- Basic skeleton
- BASH variables
- Functions
- Reading variables
- Branching the code
- Loops

Keeping correct syntax might be sometimes tricky. Use some checker helping you to fix your code, e.g. ShellCheck (on-line or locally `shellcheck myscript.sh`).

Basic script

- Every script begins with `#!/bin/bash` (or alternative for another shells, Perl, ...)
- Add any commands you like...
- Every script should end with `exit` (but it is not necessary)
- After writing the script, add execution permission (`chmod +x noninteractive.sh`), launch with `./noninteractive.sh`
- The most simple script:

```
#!/bin/bash
# Simple non-interactive script - no communication with user,
# only list of commands - prints user name, date, $PATH and free space
echo "Hi, ${USER}, today is $(date) and your PATH is ${PATH}."
echo "Free space on your disk:"
df -h
echo
exit
```

Special BASH internal variables available in the script (selection)

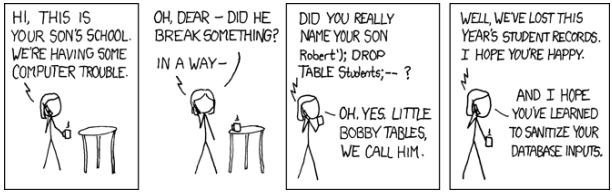
- These variables can be used within script e.g. to parse arguments provided by the user
- `$1` , ... (number from `1` up to number of parameters) — individual positional parameters (see further examples)
- `$0` — path of the starting script
- `$#` — number of command-line arguments
- `$*` — all of the positional parameters, seen as a single word, must be quoted (i.e. `"$*"`)
- `$@` — same as `$*` , but each parameter is a quoted string — the parameters are passed intact, without interpretation or expansion, each parameter in the argument list is seen as a separate word, should be quoted (i.e. something like `"$@"`)
- `$$` — process ID (PID) of the script itself
- `$?` — Exit status of previous command, function, or the script itself
- See more variables...

Functions in BASH

Pieces of code, which can be used repeatedly

```
# Declare new function within script
function MyNewFunction1 {
  echo "Hello, ${USER} from $(groups) on ${HOSTNAME}!"
  }
# Use it in a script as any other command
...
MyNewFunction1
...
# Use with variables - provide parameters for the function
# See following script examples for another input - same as in scripts
function MyNewFunction2 {
  echo "The sum is $(("$1" + "$2"))."
  }
# Use it in the script as any other command
MyNewFunction2 5 8 # For example
...
```

It is important to check user input...



https://xkcd.com/327/

- By accident or purpose (attack), user can enter unexpected value
  - In the "best" case, the script "just" crashes
  - Script can behave unexpectedly, returning very weird results, damage data
  - Internal functions/commands can return error messages, which are hard to understand
  - Attacker can e.g. modify web content (XSS, ...), obtain private data, root privileges, ...
  - Applies also to scientific data — wrong input use to have unexpected outcomes...
- Programmer should always check if user input is correct, filter it

Script reading two variables

```
#!/bin/bash
# Arguments are read from command line as parameters of the script
# Order has to be kept (well, not in this case, but generally yes)
echo "Sum of two numbers $1 and $2 is $(("$1" + "$2"))." # $((calc...))
# "$#" is available every time and contains number of parameters
# (variables) given to the script
echo "Number of parameters is: $#"
# "$*" is available every time and contains all supplied parameters
echo "Those parameters were supplied: $*"
#  "$0" is available every time and contains script path
echo "Path to the scrip is: \"$0\""
exit
```

When done, do:

```
chmod +x interactive1.sh
./interactive1.sh 8 9 # Or select any other two numbers
```

- There is no checking of input values, nothing advanced, ...

Variables will be interactively provided by the user

```
#!/bin/bash
# Arguments are read from user input (script asks for them)
echo "Please, input first value to sum and press 'Enter'"
read -r V1
echo "Please, input second value to sum and press 'Enter'"
read -r V2
echo "Sum of two numbers ${V1} and ${V2} is $(("V1" + "V2"))."
# Note above that "$" is unnecessary on arithmetic variables
echo
exit
```

When done, do:

```
chmod +x interactive2.sh
./interactive2.sh # Values will be provided when script asks
```

- There is no checking of input values, nothing advanced, ...
- See next slide to read the variable in `while` cycles to ensure it is correctly entered

Ensuring user interactively provides correct input (while)

- Detailed explanations of all features used here are in various following slides... See scripts `interactive2{whiles,functions}.sh`

```
... # Following code replace lines 3 and 4 from previous script
NUMBER='^[0-9]+$'
echo "Please, provide a number as input value:"
while : # Start of while cycles - run until correct input is provided
  do # Star of the body of the cycles
  read -r INPUT # Here the input from keyboard is received
  if [[ ${INPUT} =~ ${NUMBER} ]]; then # Test if INPUT is a number
    echo "OK, input value is $INPUT."
    break # We have correct value, we can break the cycles and continue
    else # What to do if the user did not provided correct value
      echo "Error! You provided wrong value!" # Tell the user
      echo "Try again (the number):" # Ask user for new input value
    fi # End of the conditional evaluation
  done # End of the while cycles
... # The code continues... Such check is needed for every variable...
```

Ensuring user interactively provides correct input (function) I

```
# Regular expression to check if the provided input is a~number
NUMBER='^[0-9]+$' # From beginning (^) to end ($) only numbers
# Function to read and check user input (this goes to beginning)
function checkinput {
  while :
    do # Star of the body of the cycles
      read -r INPUT # Here the input from keyboard is received
      if [[ ${INPUT} =~ ${NUMBER} ]]; then # Test if $INPUT is a number
        echo "OK, input value is ${INPUT}."
        break # We have correct value, we can break and continue
        else # What to do if the user did not provide correct value
          echo "Error! You provided wrong value!" # Tell the user
          echo "Try again (the number):" # Ask user for new input value
        fi # End of the conditional evaluation
    done # End of the while cycles
  } # Read variable is in INPUT
```

Ensuring user interactively provides correct input (function) II

```
# Replace line 4 of interactive2.sh by (similarly for line 6)
...
checkinput # Use function declared on previous slide
V1="${INPUT}"
...
checkinput # Recycle the function to read next variable
V2="${INPUT}"
...
```

- User can provide as arguments...
  - Input/output data file names
  - Parameters for running whatever analysis — can be passed to some scientific software within script or so
  - Which branch of the code to run
  - ...

Provide named parameters

```
#!/bin/bash
# Script has only one parameter ($1) provided
# "case" is evaluating provided parameter and behaving accordingly
case "$1" in
  d|disk) # "|" means alternatives - more possible inputs
    echo "Your disk usage is:"
    df -h;;
  u|uptime)
    echo "Your computer is running:"
    uptime;;
  # This should be every time last possibility - any other input
  # User is then notified he entered nonsense and gets some help
  *) # Any other input
    echo "Wrong option! "
    echo "Usage: 'd' or 'disk' for available disk space or 'u' or"
    echo "  'uptime' for computer uptime"
# Ends on next slide...
```

Notes to previous script

```
# ...end from previous slide
    exit 1;; # In this case, exit with error code 1
  esac
exit
```

- First make `interactive3.sh` executable and launch it via e.g. `./interactive3.sh d` or `./interactive3.sh uptime` or so
- Function `case` has basic checking of input available — as last parameter use `*)` — any other input except those defined above will produce some warning message, error or so
- In same way can be added more parameters (by multiple use of `case` or by wrapping `case` by `while` loop), in the latter variant order of parameters does not have to be kept and all parameters are compulsory
- `case` can evaluate simple regular expressions, e.g. `[Uu]ptime`, `d*`, ...
- This is the most simple usage, more complex possibilities are ahead

Provide parameters, verify them and behave accordingly I

```
#!/bin/bash
# From the beginning (^) to the end ($) at least one (+) number ([0-9])
NUMBER='^[0-9]+$'
function usagehelp { # Function to print help - we will use it four times
  echo "Usage: number1 plus/minus/product/quotient number2"
  echo "Use plus for sum, minus for difference, product"
  echo "  for multiplication or quotient for quotient."
  exit 1 # End up with an error
  }
if [[ "$#" -ne "3" ]]; then # Do we have 3 parameters provided?
  echo "Error! Requiring 3 parameters! Received $# ($*)."
  usagehelp # The function to print help
  fi # "=~" means testing if $1 fits to regular expression in $NUMBER
if [[ ! $1 =~ ${NUMBER} ]]; then # Is parameter 1 number?
  echo "Parameter 1 is not an integer!"
# Continues on next slide...
```

Provide parameters, verify them and behave accordingly II

```
# Remaining part from previous slide...
  usagehelp # The function to print help
  fi
if [[ ! $3 =~ ${NUMBER} ]]; then # Is parameter 3 number?
  echo "Parameter 3 is not an integer!"
  usagehelp # The function to print help
  fi
case "$2" in
  plus) echo "$(($1 + $3))";;
  minus) echo "$(($1 - $3))";;
  product) echo "$(($1 * $3))";;
  quotient) echo "$(($1 / $3))";;
  *) echo "Wrong option!"
    usagehelp # The function to print help
    ;;
  esac
exit
```

Provide parameters, verify them and behave accordingly III

```
# Make in executable and run it...
chmod +x interactive4.sh
./interactive4.sh 7 plus 5 # For example...
```

- Note that we can evaluate input parameters in any order
- Compare syntax styling of `case` on previous slide and in script file `interactive4.sh`
  - Each option can be on single line, or on multiple lines
  - For each `case` option there can be any number of commands
  - There can be any number of `case` options — convenient way how to evaluate multiple options in single step
  - Se following slides for most common usage of `case`
- `case` can be used anywhere in the code, not only to evaluate input parameters (as possible alternative to `if-else` branching)

Multiple switches in classical UNIX form (no positional) I

- Following code use to be near beginning of the script to evaluate input
- See `interactive5.sh` for complete example
- `getopts` reads short (one-letter) parameters, they can have input value (marked by `:`)

```
#!/bin/bash
# All provided values are evaluated in while cycles...
while getopts "hvi:o:a:" INITARGS; do # Switches are -h -v -i -o -a
  case "${INITARGS}" in # $INITARGS contains the parameters to evaluate
    h|v) # Accept parameters "-h" or "-v" for help
      echo "Usage options..."
      exit # Terminate after providing help
      ;; # End of this option
    i) # Parameter "-i" accepts some value (e.g. "-i inputfile.txt")
      ... # Do some checking etc...
      INPUTFILE="${OPTARG}" # $OPTARG always contains value of parameter
      ;; # End of this option
# ...continues on following slide...
```

Multiple switches in classical UNIX form (no positional) II

```
# ...starts on previous slide...
    o) # Parameter "-o" accepts some value (e.g. "-o outputfile.txt")
      ... # Do some checking etc...
      OUTPUTFILE="${OPTARG}" # $OPTARG always contains value of parameter
      ;; # End of this option
    a) # Parameter "-a" accepts some value (e.g. "-a X" for number)
      # Check if provided value makes sense (integer between 10 and 300)
      if [[ "${OPTARG}" =~ ^[0-9]+$ ]] && [[ "${OPTARG}" -ge 10 ]] &&
        [[ "${OPTARG}" -le 300 ]]; then # The condition is long...
        VALUE=${OPTARG} # $OPTARG always contains value of parameter
        echo "Value is OK: ${VALUE}"
        else
         echo "Error! For parameter \"-a\" you did not provide an"
         echo "  integer ranging from 10 to 300!"
         exit 1
        fi
# ...continues on following slide...
```

Multiple switches in classical UNIX form (no positional) III

```
# ...continuing from previous slide...
      ;; # End of this option (see previous slide)
    ?)
      echo "Invalid option(s)!"
      echo "See \"$0 -h\" for usage options."
      exit 1
      ;; # End of this option
    esac
  done # ...the end.
# Check if all required values are provided
if [[ -z "${INPUTFILE}" ]] || [[ -z "${OUTPUTFILE}" ]]; then
  echo "Error! Name of input and/or output file was not provided!"
  echo "See \"$0 -h\" for help usage..."
  exit 1
  fi
# ...ends on following slide...
```

Multiple switches in classical UNIX form (no positional) IV

```
# ...continuing from previous slide...
if [[ -z "${VALUE}" ]]; then
  echo "Warning! Value for \"-a\" was not provided! Using default (10)."
  VALUE=10
  fi
# Do the job...
for (( I=1; I<="${VALUE}"; I++ )); do # Repeat task number of times
  echo -ne "Cycle ${I}...\r" # Write number of cycle and return cursor to
       # the beginning of the line to overwrite the number in next step
  sleep 1s # Wait 1 second - just for fun ;-)
  # Do the task - append input to the output - note usage of variables
  cat "${INPUTFILE}" >> "${OUTPUTFILE}" #Do the job: append input to output
  done
echo -ne "\n" # Reset cursor to new line
echo "Done!"
exit
```

Multiple switches in classical UNIX form (no positional) V

- Script `interactive5.sh` contains complete example
- This is the classical way how to use UNIX switches used in most of commands
- `while` loop encapsulating `case` ensures we evaluate all provided parameters regardless their number or order
  - Be prepared that user can use the arguments in any combination and order (e.g. calling `-h` together with any other switch) — avoid in code doing several things at once

```
# Make it executable
chmod +x interactive5.sh
# Start with displaying help
./interactive5.sh -h # Or ./interactive5.sh -v
# Try it as common command line tool
./interactive5.sh -i input.txt -o output.txt -a 50
# Order of parameters doesn't matter
./interactive5.sh -o out.txt -a 50 -i input.txt
```

Simple providing of input file I

```
#!/bin/bash
# We expect exactly one parameter
if [[ "$#" -ne "1" ]]; then
  echo "Error! Exactly one parameter is required!"
  exit 1
  fi
# Verify that file exists and is readable
if [[ ! -r "$1" ]]; then
  echo "Error! The file provided does not exist or is not readable!"
  exit 1
  fi
# Do the operation with input file..
echo "Size of the file is $(du -sh "$1" | cut -f 1)."
echo "The file has $(wc -l "$1" | cut -d ' ' -f 1) lines."
echo "Making backup of the file $1..."
# Ends on next slide...
```

Simple providing of input file II

```
1 # ...the end from previous slide
2 # Copy file to backup (*.bak). If it succeeds, report it, if it fails
3 # exit with error (still handling single variable $1)
4 { cp "$1" "$1".bak && echo "Backup saved as $1".bak; } || \
5   { echo "Error! Making backup of $1 failed!"; exit 1; }
6 echo "Done!"
7 exit
```

- Common way for simple scripts — check input and do something with single input file
- Note lines 4 and 5 above — common way to report success as well as handle failure

```
1 # Make it executable
2 chmod +x interactive6.sh
3 # Use the script with some text file...
4 ./interactive6.sh long_text.txt
```

If branching (examples are elsewhere)

```
# Basic variant - commands are done only if condition is met
if condition; then
    commands
  fi
if condition; then # Two branches - when condition is met and when not
    commands1 # condition is TRUE
  else
    commands2 # condition is FALSE - all other cases
  fi
# Join together two (or more) if branches
if condition1; then
    commands1
  elif condition2; then
      commands2
    else
      commands3
    fi # Branching can be long, hard to read and susceptible to errors
```

Evaluation of conditions I

- Basic method to branch code — do something according to certain condition
- Very versatile, usually there are more options how to write desired conditioning
- Avoid long chaining of conditions using `elif` statement (previous slide) — susceptible to mistakes, hard to debug
- “ `[ … ]` ” (always keep space inside around it) is function to evaluate expressions
  - `if [ "${VAR}" -eq 25 ]` or alternatively `test "${VAR}" -eq 25` instead
  - `if [ "${VAR}" == "value" ]; …`
    - Escaping variables and values by double quotes (`"…"`) is recommended (to be sure), but not strictly required all the time
  - `if [ ! -f regularfile ]; …` — `!` reverts condition
  - Single-bracket conditions — file, string, or arithmetic conditions
  - Double-bracket syntax — enhanced (now preferred)
    - Allow usage of regular expressions and globing patterns

Evaluation of conditions II

- Word splitting is prevented — `${STRINGVAR}` can contain spaces etc.
- Expanding file names — `if [[ -a *.sh ]]` (variant with only one bracket doesn't work when there are multiple sh files)
- More detailed test, e.g. `if [[ $num -eq 3 && "${STRINGVAR}" == XXX ]]`…

- Common tests
  - `-eq` — Equal to (like `==`)
  - `-lt` — Less than
  - `-gt` — Greater than
  - `-ge` — Greater than or equal to
  - `-le` — Less than or equal to
  - `-f "${FILE}"` — True if `$FILE` exists and is a regular file (not link or so)
  - `-r "${FILE}"` — True if `$FILE` exists and is readable

Evaluation of conditions III

- `-w "${FILE}"` — True if `$FILE` exists and is writable
- `-x "${FILE}"` — True if `$FILE` exists and is executable
- `-d "${FILE}"` — True if `$FILE` exists and is a directory
- `-s "${FILE}"` — True if `$FILE` exists and has a size greater than zero
- `-n "${STR}"` — True if string `$STR` is not a null (empty) string
- `-z "${STR}"` — True if string `$STR` is a null string
- `"${STR1}" == "${STR2}"` — True if both strings are equal
- `"${STR}"` — True if string `$STR` is assigned a value and is not null
- `"${STR1}" != "${STR2}"` — True if both strings are unequal
- `-a` — Performs the `AND` function (`[ ... -a ... ]` or `[ ... ] && [ ... ]`)
- `-o` — Performs the `OR` function (`[ ... -o ... ]` or `[ ... ] || [ ... ]`)

Evaluation of conditions IV

- `if` statement can test whatever returning `TRUE` / `FALSE`, commonly something like `if grep -q XXX; then …` (see also further)
- Do not confuse globing patterns and regular expressions when using `[[ … ]]`
  - **Shell globing:** `if [[ "${STRINGVAR}" == ?[sS]tring* ]]; then` – `?` represents single character `[]` any character inside and `*` zero or more characters
  - **Regular expressions:** `if [[ "${STRINGVAR}" =~ .[sS]tring.* ]]…` – `.` represents single character (`?` would be zero or one occurrence of preceding expression), `[]` any character inside and `.*` zero or more occurrences of any single characters
  - Same expression is interpreted in different ways

Selecting version of RAxML according the CPU type I

- RAxML can take advantage of modern CPUs to highly speed up calculations, with or without parallelization — every version has separated binary, user must select, see README
- After compilation, the script can select e.g. on remote server appropriate version
- RAxML binaries must be in `$PATH`
- See `raxml_if.sh` for whole script

```
if grep -iq avx2 /proc/cpuinfo; then # Does the CPU support AVX2?
  RAXML='raxmlHPC-AVX2' # Select appropriate binary
  elif grep -iq avx /proc/cpuinfo; then # Does the CPU support AVX?
    RAXML='raxmlHPC-AVX' # Select appropriate binary
    elif grep -iq sse3 /proc/cpuinfo; then # Does the CPU support SSE3?
      RAXML='raxmlHPC-SSE3' # Select appropriate binary
      else # The very last option
        RAXML='raxmlHPC' # Slowest oldest CPU...
  fi # End of branching
${RAXML} -s ${INPUT} # All the parameters as usually...
```

Selecting version of RAxML according the CPU type II

Multiple branching in one step

- Same task as on previous slide, but instead of if-then branching it is using `case`
- See `raxml_case.sh` for whole script

```
# Determine which CPU is available and which binary use then
CPUFLAGS=$(grep -i flags /proc/cpuinfo | uniq)
case "${CPUFLAGS}" in
  *avx2* | *AVX2*) # Does the CPU support AVX2?
    RAXML='raxmlHPC-AVX2';; # Select appropriate binary
  *avx* | *AVX*) # Does the CPU support AVX?
    RAXML='raxmlHPC-AVX';; # Select appropriate binary
  *sse3* | *SSE3*) # Does the CPU support SSE3?
    RAXML='raxmlHPC-SSE3';; # Select appropriate binary
  *) # The very last option
    RAXML='raxmlHPC';; # Slowest oldest CPU...
  esac # End of branching
${RAXML} -s ${INPUT} # All the parameters as usually...
```

For loops I

- `for` loops are available in practically every programming language
- BASH allows plenty of variants how to declare repetitions
- In `for` loop we know in advance number of repeats — numerical sequence, list of files, ...
- Common way how to do same operation with multiple files

```
# Ways how to declare number of repetitions
# Variable $I contains in every repeat number from 1 to 10 - number of
# respective repeat
for I in $(seq 1 10); do echo "${I}"; done # "seq" is outdated
for I in 1 2 3 4 5 6 7 8 9 10; do echo "${I}"; done
for I in {1..10}; do echo "${I}"; done
for (( I=1; I<=10; I++ )); do echo "${I}"; done
# One line for cycle for resizing of multiple images
# Variable $JPGF contains in every repeat one-by-one name of input file
# (each JPG) processed in the respective turn (creates tuhmbnails th-...)
for JPGF in *.jpg; do convert ${JPGF} -resize 100x100 th-${JPGF}; done
```

For loops II

- Basic way to process more files

```
# More commands in a block
for JPGF in *.jpg; do
  echo "Processing JPG file ${JPGF}"
  file "${JPGF}" # Get information about currently processed file
  # Create tuhmbnail. If it fails, skip following commands
  convert ${JPGF} -resize 100x100 thumbs-${JPGF} || continue
  echo "File thumbs-${JPGF} created"
  done
# Passing through each loop can be influenced using conditions and
# subsequent skipping of rest of the loop
for ...; do # Start cycles as you need
  commands1 # commands1 will be executed in any case
  if (condition); then # Set some condition to skip commands2
    continue; fi # Go to next iteration of the loop and skip commands2
  commands2
  done
```

While and until loops I

```
# while loop is evaluating condition and if it is equal to 0 (TRUE)
# the loop body is launched, repeatedly while the condition is met
while condition; do
    commands
  done
# Like while loop, but until condition is NOT equal to zero (not met)
until condition; do
  commands
  done
# Compare differences between while (more common) and until loops
I=0 # Assign initial value
# Repeat while I is lower or equal 10; in every step increment I by 1
while [ $I -le 10 ]; do echo "Value: $I"; I=$(($I + 1)); done
I=20 # Assign initial value
# Repeat until I is lower then 10; in every step decrement I by 1
until [ $I -lt 10 ]; do echo "Value: $I"; I=$(($I - 1)); done
```

While and until loops II

```
# 'continue' skips to another loop turn, 'break' terminates running
# of whole loop (no further turns) and skips to following commands
while ...; do # Start loops as you need
  commands...
  if [condition]; then # If something happens
    break # End up the loops and continue by following commands
    fi
# While loops are popular to process every line of input file
# The text file use to contain e.g. list of files to process (if for
# whatever reason 'for' loop construction is impractical)
while read TEXTLINE; do # Run loops on every line of text file
  commands... # TEXTLINE contains in each turn one line of the file
  done < text_file_to_process.txt
# Infinite loops - common when waiting for some condition to proceed
while :; do echo "Press CTRL+C to exit..."; done
for (( ; ; )) ; do echo "Press CTRL+C to exit..."; done
```

What is worth of scripting

- **Any task you need to repeat from time to time requiring more commands** — almost anything :-) Programmers are lazy. Do not manually do work worth of trained apes.
  - Take advantage of usage of loops, conditions, variables, ...
- Processing of multiple files
  - Various manipulations (conversions, ...)
  - Parsing (e.g. extracting information from multiple CSV or logs)
- Repeated running of some analysis (requiring multiple steps)
  - Keep settings and all preparing and post-processing steps
- Anything you do often and it requires non-trivial set of commands
- And much more...

Start scripting, coding

Now you should have enough knowledge to start writing scripts to do such tasks. Start with something simple, solving some your practical need (e.g. for MetaCentrum, from slide 262). You'll gather experience and improve.

Scripting tasks

1. Write short script (or at least concept how it should work) to print file name of JPG file and its dimensions in px. Use e.g. `file` or ImageMagick `identify` to get the dimensions.
2. Write short script (or at least concept how it should work) taking as input arguments file name and taxon name, verify their validity, extract respective FASTA sequence and print it to standard output.
   - Work on file `Oxalis_HybSeq_nrDNA_selection_alignment.fasta`.
3. Think about several tasks where you would use BASH scripts to automatize repeated work and/or processing of multiple files.
   - Think about usage of various loops (and/or GNU Parallel), if-then branching, various conditions, etc.
   - Prepare concept(s) of such script(s).

Keep in mind...

Do not do simple repeated work worth of trained monkey manually. It is silly. Every such task is worth of (one-line) script to work instead of you.

Software

7 Software
- Packages
- Compilation
- Java
- Windows applications
- Scientific applications
- Games

Package management I

Installation of software

- **Package** — an application or its part (documentation, plug-ins, translations, ...)
- Packages are available in **repositories** (directories) on the internet
  - System has list of applications available
  - Updates and bug fixes are installed for all applications using one interface (GUI or command line) — very reliable
  - Packages are digitally signed — security
  - User can set custom repositories to get more package resources
  - Repositories can be added whenever needed — check documentation for your distribution (at least basic “how-to”)
- The most different task among Linux distributions
- Packages have dependencies — required shared libraries and so on — use package manager and try to avoid downloading packages from the internet outside repositories
- Read manual for your distribution!

Package management II

Installation of software

- Package is basically an archive and system has configured directories where to unpack it — binaries are commonly in `/usr/bin/`, shared libraries in `/usr/lib` and `/usr/lib64`, data in `/var`, ...
- User should not care where parts of packages go to — system is taking the care — user can only damage it
- Shared libraries are installed automatically whenever required
- As all files are placed in standard defined directories, it is very simple to use them also for another applications
- Applications not available in repositories, neither as distributional package should be installed into `~/bin` for current user or `/usr/local` for all users (binaries then go into `/usr/local/bin` and so on)

Package management III

Installation of software

- Common distributions use to provide convenient graphical tool to manage software
  - Ubuntu Software Center
  - Synaptic — feature rich, graphical, advances, for any DEB distribution (Debian, Ubuntu, Mint, ...)
  - Aptitude — feature rich, command-line, advanced, for any DEB distribution (more advanced version of Apt)
  - DPKG — low-level, any DEB-based distribution
  - YaST Software for openSUSE (feature rich, graphical as well as command-line)
  - Zypper — feature rich, command-line, advanced, for openSUSE
  - DNF — feature rich, command-line, advanced, for Fedora and another RPM based distributions (replacing older Yum)
  - RPM — low level, any RPM-based distribution
  - GNOME software — in most of distributions using GNOME
  - And many more...

Package management IV

Installation of software

- All do same task — as soon as user master one, others are equally simple
- Distributions use to provide convenient update applet notifying about awaiting updates
- There use to be web services to look for packages, also from other sources — openSUSE, Debian, Ubuntu (+ Launchpad and PPAs), Fedora, ...
- The task is always same, the exact work-flow and commands more or less differ among distributions...
- Tools like Android Google Play, Apple Store or Windows Store are inspired from Linux (but terribly simplified)...

Task

Install some new software into your Linux. Use graphical as well as command-line tools. See following slides for instructions.

Package management in command line in openSUSE and SLE (basic commands) I

- Root password is required: use `sudo …` or `su -`
- Package name `*.rpm`
- `zypper in package` — install *package*
- `zypper rm package` — remove *package*
- `zypper ref` — refresh repositories
- `zypper up` — update
- `zypper dup` – upgrade to newer release of whole distribution (see documentation)
- `zypper ps -s` – check which running applications (including SystemD services) should be restarted after update of packages

Package management in command line in openSUSE and SLE (basic commands) II

- `zypper se` *`term`* — search *term*
- `zypper pa --orphaned --unneeded` — list packages, which can be safely removed
- `yast sw_single` — interactive manager
- `zypper lr` — list repositories
- `zypper ar` *`repository`* — add *repository* (URL of remote `*.repo` file)
- `zypper rr` *`repository`* — remove *repository* (name according to `zypper lr`)
- `zypper mr` *`repository`* — modify *repository* (see `man zypper` first, or use `yast sw_single`)

Package management in command line in openSUSE and SLE (basic commands) III

- `zypper pa` *package* — get information about particular *package* or another query (e.g. list of dependencies, see `man zypper`)
- `opi` *package* — search (and possibly install) *package* in OBS in all repositories contributed by users (take care, there are no quality checks)
- `rpmconfigcheck` — check which configuration files in `/etc` have new version after update of some software — compare respective configuration files with new `*.rpmnew` files
- `rpm*` commands for other tasks
- `man zypper`, `man rpm` — usage help

Package management in command line in Debian/Ubuntu and derivatives (basic commands) I

- Root password is required: use `sudo ...` or `su -`
- Package name `*.deb`
- `apt install package` — install *package*
- `apt remove package` — remove *package*
- `apt update` — refresh repositories
- `apt upgrade` – upgrade packages
- `apt dist-upgrade` – upgrade to newer release of whole distribution
- `apt search term` — search *term*
- `apt autoremove` — clear unneeded packages

Package management in command line in Debian/Ubuntu and derivatives (basic commands) II

- `apt autoclean` — cleanup of files after uninstalled packages
- `aptitude` — interactive manager
- `cat /etc/apt/sources.list` — list repositories
- `nano /etc/apt/sources.list` — add/remove/edit repositories
- `dpkg-*`, `apt-*` commands for other tasks
- `aptitude` is used in similar way as `apt` (e.g. `aptitude install` *`package`*, ...) or as interactive tool (just run `aptitude`)
- Similar tasks can be done with `apt`, `apt-get`, or `aptitude`
- `man aptitude`, `man apt-XXX`, `man dpkg-XXX`, `man apt` — usage help

Package management in command line in RedHat, Fedora, CENTOS and derivatives (basic commands) I

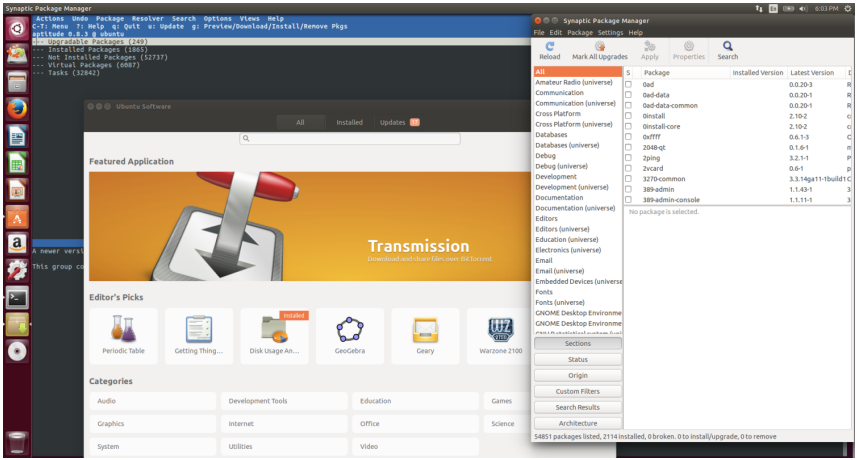
- Root password is required: use `sudo ...` or `su -`
- Package name `*.rpm`
- `dnf install package` — install *package*
- `dnf remove package` — remove *package*
- `dnf check-update` — refresh repositories, check for updates
- `dnf upgrade` — upgrade packages
- `dnf search term` — search *term*
- `dnf autoremove` — clear packages
- `dnf info package` — get information about *package*

Package management in command line in RedHat, Fedora, CENTOS and derivatives (basic commands) II

- `dnf repolist` — list repositories
- `dnf config-manager` — manage repositories and another settings
- `rpm -Uvh package.rpm` — install locally downloaded `package.rpm`
- `rpm -e package` — remove *package*
- `rpm*` commands for other tasks
- In older releases of Fedora, RedHat and CENTOS, `yum` is used instead of `dnf` in nearly identical way
- `man dnf` (or `man yum`), `man rpm` — usage help

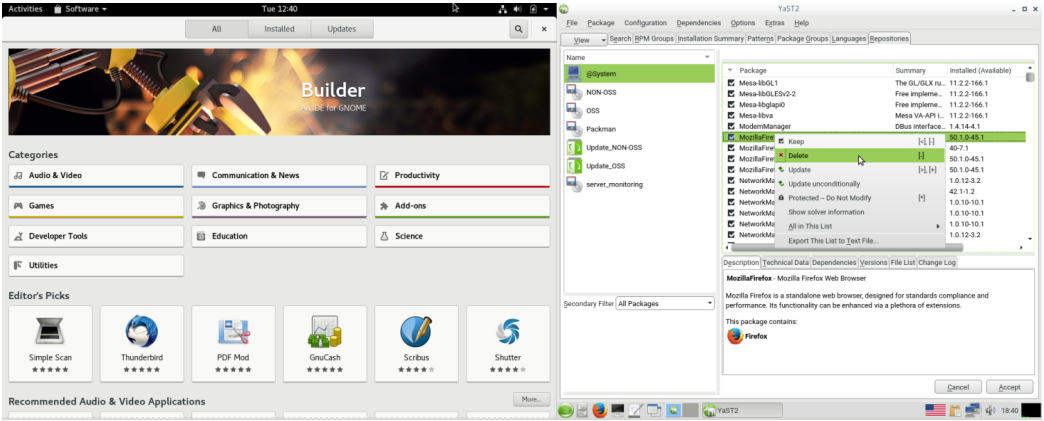
Graphical package managers I

Ubuntu Software, and Synaptic and text-based Aptitude for all DEB-based distributions



Graphical package managers II

GNOME Software in Fedora and YaST in openSUSE



- Practically every common general distribution has some graphical tool... Explore it...

Manuals for package management

- openSUSE and SUSE Linux Enterprise: https://doc.opensuse.org/documentation/leap/startup/html/book-startup/part-reference-software.html and https://en.opensuse.org/Portal:Zypper
- Red Hat, Fedora, CENTOS, Scientific Linux and others: http://yum.baseurl.org/ and newer https://docs.fedoraproject.org/en-US/quick-docs/dnf/
- Debian (and derivatives): https://www.debian.org/doc/manuals/debian-reference/ch02.en.html and https://wiki.debian.org/PackageManagement
- Ubuntu (and derivatives): https://help.ubuntu.com/stable/ubuntu-help/addremove.html and https://help.ubuntu.com/community/AptGet/Howto
- And another distributions...

Basics of compilation

- Some software is distributed only as source code (e.g. downloaded from GitHub) written in languages like C or C++ — user has to *compile* it to get working binary executable
- Compilation creates binary specific for particular operating system and hardware platform — can be tuned for optimal performance
- Interpreted languages (Bash, Perl, Python etc.) don't have to be compiled (but it is possible) — they need their interpreter to run, relative easily portable among hardware platforms and OS (similar to Java and its Virtual Machine)
- Applications requiring compilation usually have good instructions
- **If you don't have to do it, don't do it.** Solving problems can be complicated — contact someone skilled or author of the application, consult ChatGPT...

```
# General schema within application directory with the source code:
./configure # Many possible parameters, settings for compilation
            # Not required every time
make # Basic building command, sometimes only this is required
make install # Placing everything into correct placed, sometimes required
```

Install tools needed for compilation

- You need to install compilation tools for your distribution and programming languages you are going to use
- Commonly, extra dependencies are required to compile the application
  - Packages for compilation use to end with `-dev` or `-devel` (e.g. if the software requires package `zlib` to run, install also its developmental version `zlib-dev(el)` to be able to compile it)
  - All requirements should be listed in README and/or INSTALL documents of particular package — user must install them manually...

```
# openSUSE and SUSE Linux Enterprise
zypper in -t pattern devel_basis devel_C_C++
# Debian, Ubuntu and derivatives like Linux Mint and others
apt install build-essential # Or "aptitude install build-essential"
# Red Hat, CENTOS, Fedora and derivatives (2 options - dnf or yum)
dnf groupinstall "Development Tools" "C Development Tools and Libraries"
yum groupinstall "Development Tools" "C Development Tools and Libraries"
```

Compilation of RAxML

- Available from `https://github.com/stamatak/standard-RAxML`
- Before compilation check README
- This example does not require to run `make install`, it does not have extra dependencies for compilation, it requires specifying of particular source file by `make -f` (there are multiple `GCC` files, no one main)

```
mkdir raxml # Create working directory
cd raxml/ # Go there
# Get source code from GitHub (svn downloads only changed files)
svn co https://github.com/stamatak/standard-RAxML/tags/v8.2.12
cd v8.2.12/ # Go to newly created directory
ls # List files
rm -rf Windows* # No need of Windows version - delete it
# Compile standard version (other versions are available for better CPU)
make -f Makefile.gcc
rm *.o # Remove unneeded files (temporal for compilation)
./raxmlHPC -h # Launch it - see RAxML help
```

Compilation of SAMtools

- See `https://www.htslib.org/download/`
- Ensure packages `zlib` and `zlib-dev(el)` are installed — required for running and compilation, see INSTALL and README

```
wget https://github.com/samtools/samtools/releases/download/1.10/
  samtools-1.10.tar.bz2 # Download SAMtools
tar xjvf samtools-1.10.tar.bz2 # Unpack the archive
cd samtools-1.10/ # Go to the unpacked directory
./configure --help # See various configuring options
./configure # Configure settings for compilation (default settings)
./configure --without-curses # Compile without ncurses support
make # Compile the software - check if there is error
     # Ensure developmental files for zlib (and ncurses) are available
sudo make install # Copy products into final location - default /usr/local
sudo make prefix=/where/to/install install # Install into custom location
make prefix=/home/$USER/bin install # Binary into /home/$USER/bin/bin
make clean # Cleanup - final files are already in the destination
```

Launching Java applications

- Java is probably the most portable language working on any operating system — the only condition is to install Java virtual machine (JVM)
- Linux usually use OpenJDK — search for packages named `*openjdk*`
- Let's download e.g. FigTree from
  `https://github.com/rambaut/figtree/releases`

```
# Go to directory where you downloaded it
cd directory/with/downloaded/figtree
# Decompress downloaded archive
tar zxvf FigTree_v1.4.4.tgz
# Go to created directory
cd FigTree_v1.4.4/
ls * # List files, also in subdirectories
# Launch it (command java launches *.jar files)
java -jar lib/figtree.jar
# Limit its memory usage to 128 MB
java -Xmx128m -jar lib/figtree.jar
```

Windows applications on Linux I

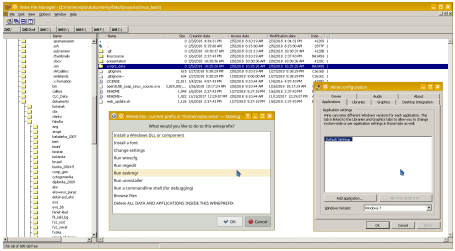
- Applications written for one operating system do not work on the other systems...
  - They must be written in portable language like Java or script like Perl, Python or BASH
  - Otherwise we need an emulator — not everything works
- Windows has since v. 10 possibility to run Linux applications, other option (for more Windows versions) is Cygwin (application must be specially compiled to work on Cygwin)
- To run Windows applications on Linux use Wine
  - Search for packages named `wine` and install it
  - Sometimes, extra functionality is in extra packages — check `wine-*`
- To run DOS application on Linux use dosbox (package `dosbox`)
- As soon as Wine is installed, just click to Windows `*.exe` file...
- Windows applications are installed into `~/.wine/` where Windows directory structure is created, launchers use to be placed to standard application menu into **Wine** section

Windows applications on Linux II

- Use `winecfg` to change settings (e.g. version of Windows — can be different for each application, custom DLL library, ...)
- `winefile` starts Windows file browser, `notepad` Notepad, `winemine` Mines
- To install some extra parts required by some applications use `winetricks`
  - Usage use to differ according to distribution and GUI
  - Browsing and selecting items to install can be bit messy...
  - It can be hard to check application requirements — if it fails, check if it is listed at `https://appdb.winehq.org/` and/or run it from command line like `wine application.exe` and inspect errors in output
- Before installing Windows application under Wine, check if there is some native Linux application to fit your needs...
  - Plenty of applications are available for more operating systems
  - Linux distributions use to have external contributor's sites to provide more packages
  - For many Windows-only applications there are fully comparable alternatives

Windows applications on Linux III

- Some applications do not work under Wine (from various reasons), some complex packages are supported commercially (I have no experience with it)
- Wine is well compatible with rest of the Linux hosting system, but it is also considerable to install Windows in e.g. VirtualBox (or another virtualization platform), if needed



`winefile`, `winetricks` and `winecfg`

Where to get scientific applications for Linux I

- There is no universal simple solution
- As first step look if particular package is available for your distribution
  - Debian Science `https://wiki.debian.org/DebianScience` — same packages are available also for Ubuntu, Mint and all the DEB derivatives
  - openSUSE Science project `https://en.opensuse.org/Portal:Science`
  - Fedora scientific packages `https://fedoraproject.org/wiki/Scientific_Packages_List`
  - Major distributions have web services (e.g. Launchpad for Ubuntu or openSUSE Build Service) where users can create their own personal repositories and package their favorite packages (like myself) — search...
- There are projects creating repositories with custom management system providing (among others) plenty of scientific applications
  - Probably most known is Conda
  - Homebrew on Linux is derived from macOS Homebrew, works also on WSL

Where to get scientific applications for Linux II

- Authors sometimes provide on homepages RPM/DEB/precompiled Java binaries/... — check them first
- If there is no other option, download source code and compile yourselves...
  - Always check documentation for requirements and instructions
  - Solving issues requires experience, can be tricky, ask for help and don't give it up
- Scientific programming languages like R, Python, Perl, Julia, Matlab etc. have their own system to install language-specific packages
  - Distributions sometimes provide packages at least for part of the packages of such languages — typically at least for Python, Perl or R — this is especially convenient for packages having plenty of complex dependencies
- Packages in various repositories are sometimes not up-to-date with newest version released — if so, contact maintainer of distributional package and kindly ask for update (distributional webs use to have easy ways how to do it)

How to game on Linux

- Plenty of games are in repositories of all major Linux distributions — just search
- Many open-source games are multiplatform
- Some distributions have extra repositories for games, e.g. openSUSE
- See `https://www.gamingonlinux.com/` for news etc.
- A lot of Windows-only games can be used with Wine
- Install Steam (package is often available in standard repositories) and there activate Proton (version of Wine to run plenty of Windows games on Linux via Steam)
- Although not all games available for Windows are available for Linux, there are more games than you have time... ;-)

MetaCentrum

8 MetaCentrum
- Information
- Usage
- Tasks
- Graphical connection
- Archive data storage
- More services

CESNET and MetaCentrum I

- CESNET (česky) is organization of Czech universities, Academy of Science and other organizations taking care about Czech backbone Internet, one of world leading institutions of this type
- CESNET provides various services (česky)
  - Massive computations — MetaCentrum (česky)
  - Large data storage (česky)
  - FileSender (česky) to be able to send up to 1.9 TB file
  - Cloud (česky) — computing (HPC) cloud similar to e.g. Amazon Elastic Compute Cloud (EC2), Google Compute Engine or Microsoft Azure
  - ownCloud (česky) to backup and/or sync data across devices (default capacity is 100 GB, user may ask for more) — similar to e.g. Dropbox, Google Drive or Microsoft OneDrive
    - It is possible to connect by webDAV to ownCloud (slide 139) — many applications support it
    - It is possible to share calendars and/or address books via calDav and cardDav among devices and/or people
- Services accessible without registration

CESNET and MetaCentrum II

- ownCloud https://owncloud.cesnet.cz/
- FileSender https://filesender.cesnet.cz/
- Go to web and log in with your institutional password

- Services requiring registration (and approval)
  - To use MetaCentrum fill registration form https://metavo.metacentrum.cz/en/application/form (česky)
  - To use data storage fill registration form https://einfra.cesnet.cz/perun-registrar-fed/?vo=storage
  - After registration for MetaCentrum, user can join MetaCloud via https://perun.metacentrum.cz/fed/registrar/?vo=meta&group=metacloud
  - Login is using EduID (česky)
- Information about data storage https://du.cesnet.cz/en/start (česky) contains detailed usage instructions

CESNET and MetaCentrum III

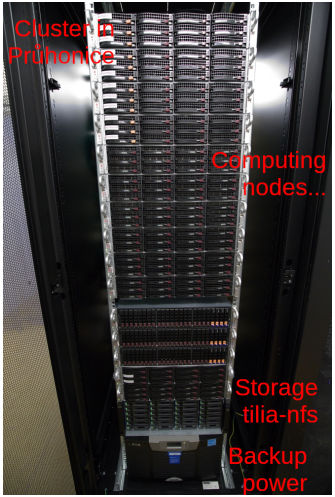
- Information about MetaCentrum https://www.metacentrum.cz/en/ (česky) and wiki https://wiki.metacentrum.cz/wiki/Main_Page (česky) (main information for users containing all needed documentation)
  - The documentation is being moved into https://docs.metacentrum.cz/
- Information about MetaCloud https://docs.cloud.muni.cz/ (old wiki page)
- Also available is Galaxy https://usegalaxy.cz/ (same login as to MetaCentrum) — web based bioinformatics framework (more information at documentation)
- Current state and usage of resources is available at https://metavo.metacentrum.cz/en/ (česky)
- Manage your user account at http://metavo.metacentrum.cz/en/myaccount/ (česky)
- Personal view on actual resources and running tasks is at https://metavo.metacentrum.cz/pbsmon2/person

CESNET and MetaCentrum IV

- List of available applications `https://docs.metacentrum.cz/software/`
- MetaCentrum has 11 frontends where users log and thousands of computers doing the calculations — they are not accessed directly to run task
- Most of computers are running Debian GNU/Linux
  - Basic usage as with any other Linux server — SSH & command line...
  - Tasks are mainly submitted via scripts (see following slides)

Terminology

- Node (computing server) — single computer, worker finally doing your computational job
- Cluster — bunch of nodes in rack (special case storing nodes, use to contain also network switch, backup power, ... e.g. cluster in Průhonice, česky)
- Grid — clusters distributed in various places, connected into single system by fast network (e.g. MetaCentrum, česky)
- Supercomputer — extra powerful computer, nowadays mostly special cluster, e.g. IT4Innovations in Ostrava



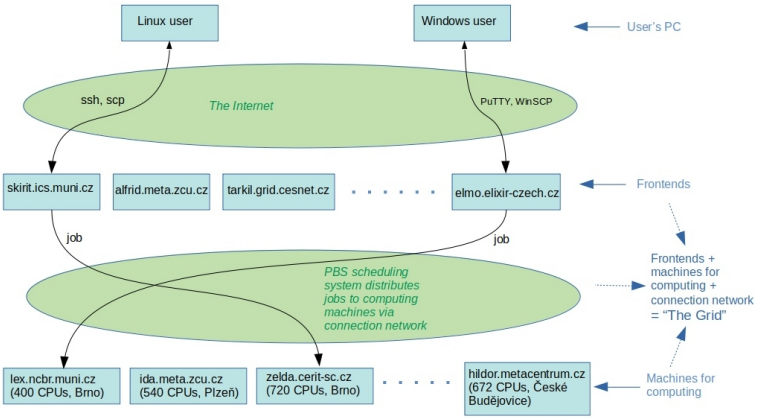
Distributed topology I

- MetaCentrum clusters are distributed in various Czech cities (and institutions)
- Clusters use to have (at least small) storage array (accessible via `/storage/…`)
- There are special archive storages (česky) in Jihlava, Ostrava and Plzeň

Distributed topology II

- Users have different quota on different arrays (depending on home organization etc.)
- There are 11 frontends where user logs via SSH and prepare computing task for submission etc. (see further)
- Distributed topology might be bit confusing — user has different home directory when logging to different frontends — recommended is to use as few frontends and storages as possible
- Some special services (like OnDemand) start on particular storage (here `brno3`)
- Generally, computing task loads data from selected storage and can start on any cluster
- Orientation is plenty of storages might be sometimes tricky — ensure where you are and where your data are supposed to be
  - Preferably use single frontend and storage to avoid confusions and issues with paths

Basic workflow



From old beginners guide

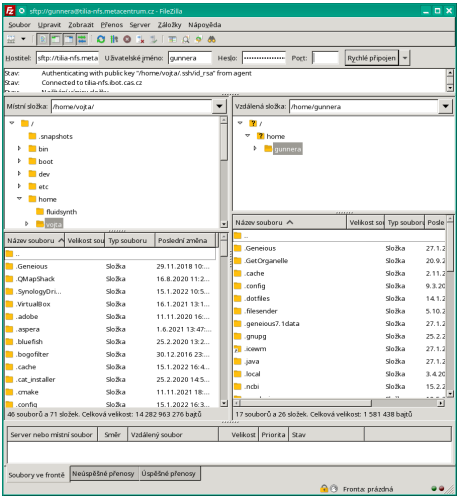
MetaCentrum usage

1. Transfer data into one of storages by e.g. `scp` from command line, WinSCP from Windows or FileZilla from any OS
2. Connect via SSH to selected frontend. Same credentials are used for all frontends, for SSH login as well as file transmissions
3. In home directory on the storage (accessed via frontend) prepare all needed data and non-interactive script to do the calculations
4. Submit job using `qsub`. Tasks are not launched immediately, but submitted into queue and system decides when it will be launched
5. Monitor job progress. Check results. Etc...

```
# Login to selected server (tarkil is located in Prague)
ssh USER@tarkil.metacentrum.cz
# Continue as in any other command line...
qsub ... # Submit the job (see later)
```

File transfers to MetaCentrum

- Graphical applications: e.g. FileZilla or most of Linux file managers
- Protocol is SSH/SSH2/SFTP/SCP, port 22, server is selected storage address — if possible, select any and keep using it
- All servers (storages, frontends, computing nodes) are accessible under domain `*.metacentrum.cz` — e.g. `tarkil.grid.cesnet.cz` is same as `tarkil.metacentrum.cz`
- See slide 139 and following to command-line transfers of files



Basic skeleton of script running tasks I

```
#!/bin/bash
# Modify the script according to your needs!
# Set data directories
WORKDIR="my_data_dir" # Or something else
DATADIR="/storage/XXX/home/${LOGNAME}" # Or other storage
# There is e.g. directory /storage/praha1/home/gunnera/my_data_dir
# containing all the data needed for calculations
# Clean-up of SCRATCH (it is temporal directory created by server) - the
# commands will be launched on the end when the job is done
trap 'clean_scratch' TERM EXIT
trap 'cp -a "${SCRATCHDIR}" "${DATADIR}"/ && clean_scratch' TERM
# Change working directory - script goes to the directory where
# calculations are done
cd "${SCRATCHDIR}"/ || exit 1 # If it fails, exit script
# Prepare the task - copy all needed files from working directory into
# particular computer which will finally do the calculations
# Ends on following slide...
```

Basic skeleton of script running tasks II

```
# ...begins on previous slide; copy data - if it fails, exit
cp -a "${DATADIR}"/"${WORKDIR}"/* "${SCRATCHDIR}"/ || exit 1
# Prepare calculations - load required application modules
# See https://docs.metacentrum.cz/software/
# Every application module is loaded by "module add XXX"
module add parallel/20210922-gcc-10.2.1-iiyjqem # In this (example) case
module add mrbayes/3.2.7a                       # GNU Parallel and MrBayes
# Launch the analysis - calculate MrBayes for multiple files
# Note Parallel will distribute task among 8 CPU threads (-j 8), so that
# qsub must in this case contain select=1:ncpus=8 (see further)
find . -name "*.nexus" -print | parallel -j 8 'mb {} | tee {.}.log'
# Copy results back to $DATADIR directory
cp -a "${SCRATCHDIR}" "${DATADIR}"/ || export CLEAN_SCRATCH='false'
# This is all needed, the script is ready to be launched...
exit
```

- Make `metacentrum.sh` executable and modify it to fit your needs...
- If it was written on Windows, convert EOL (and encoding, slide 160)...

Launching of tasks

- See https://docs.metacentrum.cz/computing/run-basic-job/
- Personal view https://metavo.metacentrum.cz/pbsmon2/person has nice overview of available resources and tasks and allows comfortable construction of submission command

```
# We will run up to 5 days (120 hours), require one physical computer
# with 8 CPU threads, 24 GB of RAM, 10 GB of disk space and we get all
# information mails (for abort, beginning, exit)
qsub -l walltime=120:0:0 -l select=1:ncpus=8:mem=24gb:scratch_local=10gb \
  -m abe metacentrum.sh
# Check how the task is running (above web) and
qstat -u $USER # Information about $USER's jobs (queued and running)
qstat 123456789 # The task ID is available from commands above or mail
qstat -f 123456789 # Print a lot of details
qdel 123456789 # Terminate scheduled or running task
qextend 123456789.meta-pbs.metacentrum.cz 12:0:0 # Prolong walltime (12 h)
```

Key MetaCentrum commands

- MetaCentrum is "just" normal Linux server — work there as on any other Linux system
- Command `module` loads/unloads selected application
- Tasks (BASH scripts) are submitted for computing by `qsub` — the script must copy the data into `$SCRATCHDIR` and do all calculations there
  - It has plenty of options how to specify requirements (see next slide and help)
- Queued and running jobs can be seen by `qstat -u $USER` (`qstat` has much more options) and any job can be terminated by `qdel 123456789` (number from `qstat`)

```
module add <TAB><TAB> # Load some module
module rm XXX # Unload selected module
module list # List of currently loaded modules
qsub ... # Submit task for computing - select any parameters needed
qstat -u $USER # See $USER's running and queued jobs
qdel 123456789 # Termite task (number from qstat)
```

Scheduling details I

- Specify needed time
  - Always `hours:minutes:seconds`, so e.g. for 4 weeks use
    `-l walltime=672:0:0` ($28 \cdot 24$), for two days and 12 hours
    `-l walltime=60:0:0`
  - User may extend it with `qextend` (see `qextend info`), otherwise ask support (write them in advance)
- Ask for as much RAM as you need (e.g. `-l mem=8gb` to request 8 GB)
  - If the task is going to require more, than allowed, system kills it…
  - If user doesn't use all required RAM, the system temporarily lowers priority for future tasks (RAM is limiting resource, do not waste it)
  - It can be hard to estimate (no easy general advice to estimate needs for particular task) — do some experiment and see how your application behaves with your data…
- Disk space is relatively free resource, user can ask more to have some reserve (e.g. `-l scratch_local=10gb` to request 10 GB)

Scheduling details II

- Specify how many physical computer(s) you are going to use (e.g. `-l select=1` for one machine) and number of CPU threads on each machine (e.g.
  `-l select=1:ncpus=8` for 1 machine with 8 cores or
  `-l select=2:ncpus=4` for 2 machines, each with 4 CPU threads)
  - It use to be necessary to specify correct number of threads for the application (e.g. `parallel -j 4`) — the application sees all CPUs on the machine, but can't use them
  - If the application consumes less than required, the system temporarily lowers priority for future tasks, if it try to use more, it will be very slowed down or killed by the server
  - If using more physical machines, ensure correct settings of e.g. MPI (see documentation for respective software you are using)
- If requesting e-mails (e.g. `-m abe` to get mail about abort, beginning and exit of the task) and submitting plenty of tasks by some script, it can result in plenty of mails...

Scheduling details III

- Every user has certain priority increased by acknowledgments (**it's mandatory to acknowledge MetaCentrum when using it**) in publications to MetaCentrum, and lowered by intensive usage of the service (the usage is calculated from past month)
- After submission of the task, check in the queue in which state it is — sometimes it can't start because of impossible combination of requested resources or so
- User can check load of machines
- For more options read
  `https://docs.metacentrum.cz/computing/pbs-resources/`
  - Request special CPU (AMD, graphical, ...), e.g. CPU with AVX2
    `-l select=cpu_flag=avx2`
  - SSD local storage (e.g. `-l scratch_ssd=1gb`)
  - Request particular location, ...
  - `https://metavo.metacentrum.cz/pbsmon2/qsub_pbspro` helps with preparation of `qsub` command

Common problems with launching the tasks

- Script fails because of wrong PATH or missing file — ensure all needed files are transferred and applications receive correct paths
  - Rather do not use absolute paths (starting with `/`) — only relative
- Not all required applications are correctly loaded
  - Check documentation and load all needed applications
  - Names of binaries are sometimes little bit different — contain names of versions, etc.
- Estimation of time needed to run the task
  - No really good solution...
  - Make some trials and try to estimate...
  - There are very different CPUs available (with different speeds) — it is possible to require particular CPU type (but it reduces number of available nodes...)
- Problems with CPU and memory
  - Hard to estimate...
  - Some applications allow setting number of CPU threads — check documentation
  - If using Java application, it often helps to request one more CPU thread for Java itself
  - Limiting memory is problematic, e.g. Java allows `java -Xmx8g -jar ...`, some applications also — check documentation

Get to task's working directory

- Go to `https://metavo.metacentrum.cz/pbsmon2/person` and click to list of your tasks and click to selected task
- Search for information **exec_host** (address of node doing the task) and **SCRATCHDIR** (temporal directory for all data and results)
- Sometimes one needs to monitor task progress or influence it
- It is not possible to directly modify running task, but at least check (and possibly modify) input data and see outputs

```
# From any MetaCentrum frontend login to respective node running the task
ssh exec_host # No need to specify user name; e.g. mandos9
# Go to SCRATCH directory
cd SCRATCHDIR # e.g. /scratch/USER/job_12345678.meta-pbs.metacentrum.cz/
# There are working data of currently running task...
# Check whatever you need...
```

Running R tasks on MetaCentrum

- There are only some R packages, to get more create own package library and use it in scripts (see e.g. `.libPaths()` within `R`)
- Be careful about paths!
- In the `metacentrum.sh` script load R, e.g.
  `module add r/4.1.3-gcc-10.2.1-6xt26dl` and start there R script as usually `R CMD BATCH script.r`

1. Login to selected front node via SSH
2. Create somewhere new directory for R packages `mkdir rpkgs` (or use default `~/R/`)
3. Start R `R` and install **all** R packages needed for the task — install them into the `rpkgs` directory `install.packages(pkgs=…, lib="rpkgs")`
4. In the R script `*.r` load the packages from the `rpkgs` directory
   `library(package=…, lib.loc="/storage/…/rpkgs")`
5. Ensure all needed outputs are saved from the R script

Interactive tasks

```
# Secure we can log off in the meantime
screen # Or tmux
# Again launch qsub according to actual needs
# Note "-I" for interactive session and missing script name
qsub -I -l walltime=1:0:0 -l select=1:ncpus=1:mem=1gb:scratch_local=1gb
# Wait for job to start... User automatically gets to the computing node
cd $SCRATCHDIR # Go to $SCRATCHDIR - work there
# After we get the interactive task, we are on new server
# Do needed work...
# When done, be sure to copy results to the storage
hostname # See where we are - we can connect to that server directly
ssh USER@given.server.cz # User name and password are the same
                         # Server address is output from "hostname"
# When you logout, the task is done (use screen to secure connection)
```

- Work as on normal Linux server...
- With `screen` we can disconnect as usually and let tasks run in background

MetaCentrum task

1. Upload to any MetaCentrum fronted files `metacentrum_oxalis.sh` and `oxalis_assembly_6235.aln.fasta` from `scripts_data`.
2. See `metacentrum_oxalis.sh`, be sure you know what it is doing, that you understand every command there.
3. If needed, edit `metacentrum_oxalis.sh` (e.g. correct paths).
4. Submit the task via `qsub`.
5. Monitor the task with `qstat`.
6. Login with SSH to computing node where the task is running, go to `$SCRATCHDIR` and see its progress. Look at output of `ps ux`. What does it say?
7. Wait for the task to be done and explore output (e.g. open the `treefile` in FigTree).
8. If you encounter any error, try to find its source and fix it.

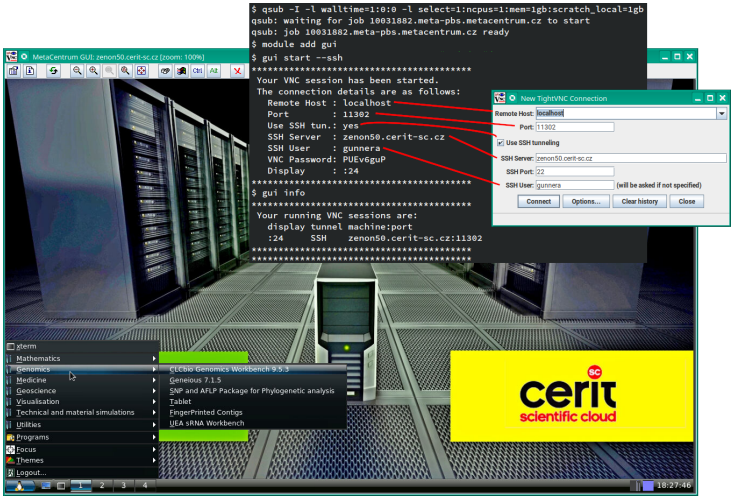
Graphical interactive task

- See https://docs.metacentrum.cz/software/graphical-access/
- Download and install TightVNC (Windows installer or Java Viewer)

```
screen # Secure we can log off in the meantime
# Again launch qsub according to actual needs
# Note "-I" for interactive session and missing script name
qsub -I -l walltime=1:0:0 -l select=1:ncpus=1:mem=1gb:scratch_local=1gb
# Wait for job to start... User automatically gets to the computing node
# After we get the interactive task, we are on new server
module add gui # We need to add GUI module
gui start --ssh # Start GUI (see above link for details)
gui info # Print information about running VNC sessions - use for TightVNC
```

- Work as on normal Linux desktop (ensure to work in `$SCRATCHDIR`)...
- It provides limited amount of resources, not suitable for big tasks

Running VNC


$ qsub -I -l walltime=1:0:0 -l select=1:ncpus=1:mem=1gb:scratch_local=1gb
qsub: waiting for job 10031882.meta-pbs.metacentrum.cz to start
qsub: job 10031882.meta-pbs.metacentrum.cz ready
$ module add gui
$ gui start --ssh
Your VNC session has been started.
The connection details are as follows:
Remote Host : localhost
Port : 11302
Use SSH tun.: yes
SSH Server : zenon50.cerit-sc.cz
SSH User : gunnera
VNC Password: PUEv6guP
Display : :24
$ gui info
Your running VNC sessions are:
display tunnel machine:port
:24 SSH zenon50.cerit-sc.cz:11302
MetaCentrum GUI: zenon50.cerit-sc.cz (zoom: 100%)
New TightVNC Connection
Remote Host: localhost
Port: 11302
Use SSH tunneling
SSH Server: zenon50.cerit-sc.cz
SSH Port: 22
SSH User: gunnera (will be asked if not specified)
Connect Options... Clear history Close
xterm
Mathematics
Genomics
Medicine
Geoscience
Visualisation
Technical and material simulations
Utilities
Programs
Focus
Themes
Logout...
CLCbio Genomics Workbench 9.5.3
Geneious 7.1.5
SNP and AFLP Package for Phylogenetic analysis
Tablet
FingerPrinted Contigs
UEA sRNA Workbench
cerit
scientific cloud
18:27:46

CESNET archive data storage

- Read documentation `https://du.cesnet.cz/` and connection instructions `https://du.cesnet.cz/en/navody/sluzby/start` (česky)
- Generally, it is possible to connect via FTPS, NFS, SAMBA (Windows network drive), SCP/SFTP or SSH — more options how to get to same resource
- See slide 262 for general information and 139 for connecting information
- After logging via SSH, it is possible to work as on any other server
- Users get information about selected storage location and paths after registration (there are several locations, user get space on one of them)
  - There are plenty of storages, be careful where you are connecting to and what is the path — e.g. when connecting directly to `du4.cesnet.cz`, paths are `/tape_tape/...`, from any other MetaCentrum node **`/storage/ostrava2-archive`**`/tape_tape/...`
- All storage locations are accessible from any MetaCentrum front node via directories in `/storage` (e.g. `/storage/ostrava2-archive/tape_tape/VO_...`)

Shared space on CESNET data storage

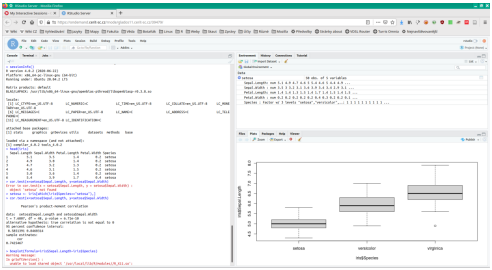
- Users can ask (česky) for creation of shared space
  - Normally, the space is private only for particular user
  - Groups allow more users to share data
  - Data storage admins will instruct users regarding locations, paths, permissions, etc. (it is specific for each case)
- Users must carefully set permissions!
  - Sharing is done by specific UNIX group
  - Users must set group ownership to particular group and permissions e.g. `770` for directories and `660` for files to avoid access of any other users
  - All members of the group must be able to manipulate the data

```
# Change group ownership to XXX
chgrp -R XXX /tape_tape/VO_XXX 2>/dev/null
# ("-R" to modify also subdirectories; "2>/dev/null" to discard errors)
# Set correct permissions to directories and files
find /tape_tape/VO_XXX -type d -exec chmod 770 '{}' \; 2>/dev/null
find /tape_tape/VO_XXX -type f -exec chmod 660 '{}' \; 2>/dev/null
```

OnDemand

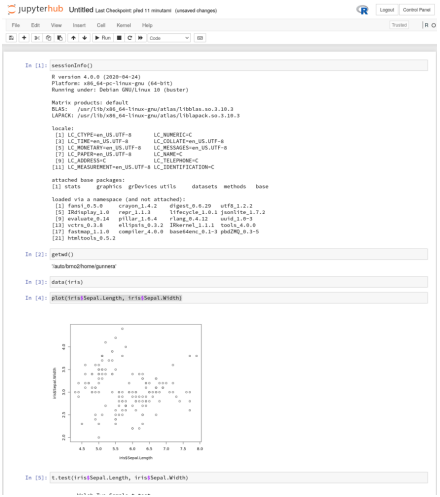
Applications in web browser

- It allows to run selected interactive applications in web browser
- See https://docs.metacentrum.cz/software/ondemand/ and https://ondemand.grid.cesnet.cz/
- Applications start in `/storage/brno3-cerit/home/$USER/` — ensure to have everything needed there



Jupyter Notebook

- Web service allowing to record code as well as its output for languages like BASH, R, Python, ...
- Convenient for recording and sharing code, interactive work, ...
- Use Jupyter Hub for MetaCentrum users
  - Data are typically in `/storage/brno2/home/USER/`
- Available also as part of OnDemand (previous slide), or bit experimental CERIT hub, which allows to select custom storage and also custom docker image (see documentation)



MetaCentrum cloud

- MetaCentrum cloud (česky) allows to run complete Linux distribution (Debian, CentOS, ...), where you can install any software, like on any other computer
- Useful e.g. for various testing or applications with very special requirements
- Go to `https://cloud.metacentrum.cz/` and see documentation
- Login to running virtual machine requires SSH keys to be set up, access is normally via SSH
- Users must have knowledge about Linux administration and follow all security best-practices
- Setting up more than one virtual machine requires good knowledge about Linux networking (user has single public IP address)

Git

9 Git
Git principle
Git basics

https://xkcd.com/1597/

Git and version control I

- Git is version controlling (česky) system (nowadays the most common) — it traces changes among all versions — absolutely crucial for any software development
- Older (nowadays not so common) version controlling systems is Subversion (SVN); there are many more (Bazaar, Mercurial, ...)
- Probably the best textbook for Git is Chacon's Pro Git
  - Dostupná i česky (včetně prvního vydání)
- Changes and their history is stored in repository (local or network, shared or private) — it is possible to view any historical state and differences between any versions
- It is possible to trace who and when did what
- Branching and merging of branches helps with making of big changes
  - When new branch is created, it contains copy of current state
  - User selects in which branch she/he is working, the branches are diverging, user can commit in every branch independently

Git and version control II

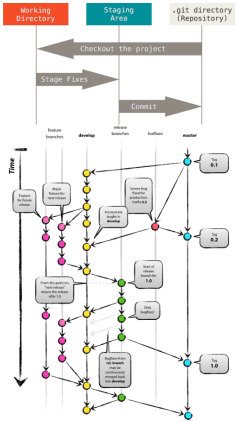
  - Commits in various branches can be compared and branches can be merged again
  - Key feature of Git — branches are useful when user starts to work on any big change in the project (the “main” part is intact while developers can freely work on major change)
- Users work on a project in some directory as usually and from time to time commit local changes to staging (temporal area) and then push them to remote or local repository (directory storing all Git history of particular project, can be anywhere)
- Every commit is checkpoint in the history
  - Can be tagged (named), e.g. particular released version of software, or project stage
  - User can compare any two commits or commit and current state
- Stagging area serves as local “container” of changes to be pushed to central Git repository (or to be modified, discarded)
- Central repository keeps whole history of the project
  - Every user has full copy of the central Git repository — can be large
- Git was developed by Linus to trace development of Linux kernel, now it is probably the most used version control tool, used also by Microsoft to trace development of Windows :-)

GitHub and others

- GitHub is currently probably the most popular platform to host development of open-source projects, see documentation
- Probably second most common on-line service providing Git repository is GitLab
  - User can create an account on GitLab (as on GitHub), or download and install GitLab system on own server (like we did, česky)
- SourceForge used to be more popular in the past, still harbors plenty of interesting projects
- Others are e.g. Bitbucket, Codebase, ...
- For on-line services, many tools are available only for paying customers
- All such services provide Git repositories accessible through standard tools (from command line, see further; or from some special application), through web browser
  - Exact usage might differ from standard Git, check respective help pages first
- Git repository can be easily hosted e.g. on any Linux server...

Git principles

- Three main areas
  1. Working directory
  2. Staging (changes awaiting to be pushed to central repository)
  3. Git repository (remote/local)
- Everyone has whole repository and history (it's "distributed") — very robust
- Flexible branches
  - Very convenient
  - Keeping work structured
  - Separation of tasks
  - Keeping more versions of the project
- Homework: play Oh My Git! to learn more and practice



https://git-scm.com/, https://nvie.com/

Working with Git — start and sending changes

```
# Create a new central repository (e.g. on a server) in empty directory
git init --bare # No need when establishing project e.g. on GitHub
# Create a new repository for new project (in empty directory)
git init # No need when cloning from existing repository
# If you did not start by cloning, add connection to server
git remote add origin <location> # Do only once on the beginning
# <location> can be remote server or local path
git remote add origin . # For repository within working directory
# Or checkout (make a copy) of existing local or remote repository
git clone /path/to/local/repository # Locally mounted repository
git clone username@host:/path/to/remote/repository # Over SSH
git clone https://github.com/V-Z/course-linux-command-line-bash-
  scripting-metacentrum.git # Clone from web, e.g. GitHub
# Add files to trace with Git
# Ignored files (or patterns) can be listed in .gitignore file
git add <files> # Or "git add *"
git status # See changed files, commits in staging, etc.
```

Working with Git — branching and getting changes

```
# Commit changes to prepare them to send to repository
git commit -m "Message..."
# Push changes into the repository (regardless where it is)
git push origin master # See further for selection of branches
# Making new branch and switching to it
git checkout -b NewFeature # Now we are in branch NewFeature
# Switch back to branch master
git checkout master # Generally, "git checkout <branch>"
# Delete the branch (changes there are lost, must be in another branch)
git branch -d NewFeature # Delete local branch
git push origin --delete <branch> # Delete remote branch
# New branch must be also pushed to the remote server
git push origin <branch>
# List branches (current is marked by asterisk on the beginning)
git branch
# Download news from central server (work of colleagues, etc.)
git fetch # Downloaded to staging, not applied yet to local files
```

Working with Git — tags, logs and more

```
# Update local repository to the newest version from central repository
git pull # Fetch and merge remote changes (before commit)
# Merge another branch into the current one
git merge <branch>
# In case of conflict, git shows editor and user must fix it manually
git add <file with conflict> # Needed to re-add conflicting file
# To see changes before merging
git diff <source_branch> <target_branch>
# Tagging e.g. milestones, released versions of software, etc.
git tag <name> <commit id> # <name> can be custom, <commit id> from log:
git log # Newest is on top, see also "git log --help"
git log --graph --oneline --decorate --all # Full long log
# Discard local changes for particular file
git checkout --- <file>
# Discard all local changes
git fetch origin # Overwrite local changes
git reset --hard origin/master # If local repository is broken...
```

Working with Git — summary, settings and more

- Basically repeat in `git add`, `commit` and `push` to get your work to the repository, and `fetch` and `pull` to download news from central repository
- `diff` to see local changes, `status` to see state
- `log` and `branch` show history and branches
- `branch`, `checkout` and `merge` for branching code and merging changes back to master (main) branch

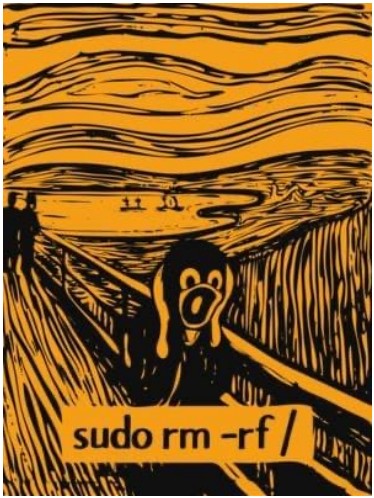
```
git diff # See changes in particular text files
gitk # Graphical interface
git config color.ui true # Set output to be colored
git config format.pretty oneline # Nicer log output
~/.git # Contains user's settings, .git in every repository contains
       # data and settings for particular repository
       # Default behavior of Git can be heavily altered
```

Git tasks

1. Clone over SSH repository `USER@vyuka.natur.cuni.cz:/srv/dadaism` (use your credentials on the testing server) and go to `dadaism` directory.
2. Communicate with others to avoid conflicting edits.
3. Add some text files, edit existing text files. Do not add images or another non-text files, do not add large files.
4. Push your changes back to the repository.
5. Fetch changes done by others.
6. See history of changes, who did what, etc.
7. Use at least `git` commands `clone`, `diff`, `status`, `add`, `commit`, `push`, `fetch`, `pull`, `log` and `gitk`.
8. You can try to play with branches — `branch`, `checkout`, `merge`.

Administration

10 Administration
System services



Managing system services

- Different among distributions — several main methods
- In Linux, most common is SystemD, less common older init scripts and RC scripts
- macOS and various BSD and other systems have different methods
- Used to manage system services like networking, cron, web server, database, ...
- Read documentation of your distribution first!
- Most of actions require root authentication
- For SystemD there are also graphical managers (YaST, Systemd-kcm for KDE, GTK Systemd Manager, web-based Cockpit, ...)

```
# SystemD - huge amount of possibilities
systemctl enable/disable/status/start/stop servicename # TAB helps
# RC scripts
rcservicename status/start/stop # TAB helps to select service
# Init scripts
/etc/init.d/servicename status/start/stop # TAB helps to select service
```

SystemD usage (few examples)

```
# List installed services and their status
systemctl list-units --type service
# Enable/disable/see status/start/stop/restart/... service
systemctl enable/disable/status/start/stop/restart servicename # TAB...
# Show overridden config files after upgrade
systemd-delta
# Analyze boot time (how long does each service take to start)
systemd-analyze blame # Text output
systemd-analyze plot > filename.svg # Same in graphics
# Log for particular service
journalctl -u servicename
# Last logged messages (press Ctrl+C to exit)
journalctl -f
# Log records since last boot
journalctl -b
# Time and date information and management
timedatectl
```

The End

11 The End
Resources
The very end

Resources to learn to work in the terminal I

- openSUSE (general handbook)
  https://doc.opensuse.org/documentation/leap/startup/html/book-startup/part-bash.html
- Ubuntu (general handbook)
  https://help.ubuntu.com/community/UsingTheTerminal
- BASH full reference manual
  https://www.gnu.org/software/bash/manual/ (advanced)
- Debian (general handbook) https://www.debian.org/doc/manuals/debian-reference/ch01.en.html
- Guide to Unix https://en.wikibooks.org/wiki/Guide_to_Unix
- BASH for beginners https://tldp.org/LDP/Bash-Beginners-Guide/html/Bash-Beginners-Guide.html (the site has plenty of good resources)

Resources to learn to work in the terminal II

- Grymoire for UNIX wizards https://www.grymoire.com/Unix/
- Linux tutorial https://ryanstutorials.net/linuxtutorial/
- Getting Started with BASH https://www.hypexr.org/bash_tutorial.php
- Bash Guide https://mywiki.wooledge.org/BashGuide
- TutorialKart https://www.tutorialkart.com/bash-shell-scripting
- Česky
  - Učebnice Linuxu https://www.abclinuxu.cz/ucebnice
  - Příkazová řádka Ubuntu https://wiki.ubuntu.cz/syst%C3%A9m/p%C5%99%C3%ADkazov%C3%A1_%C5%99%C3%A1dka/termin%C3%A1l
  - Příkazová řádka Fedory https://wiki.mojefedora.cz/doku.php?id=navody:prirucka:prompt
  - Johančiny Pohádky z příkazové řádky https://eldar.cz/kangaroo/binarni-sxizofrenie/johanka-pohadky-z-prikazove-radky.html

How to ask for help I

- Never ever ask simple silly lazy questions you can quickly find in manual or web
- People on mailing lists and forums respond voluntarily in their spare free time — do not waste it — be polite, brief and informative
- Imagine you should answer — which information do you need?
- Be as specific and exact as possible
  - Write exactly what you did (“It doesn’t work!” is useless...)
  - Copy/paste your commands and their output, especially error messages — they are keys to solve the problem
  - Try to search web for the error messages (or their parts)
  - Try to provide minimal working example — add at least part of your data (if applicable) so that the problem is reproducible
  - Specify name and version of distribution and of the problematic software
- **OSS is free as freedom of speech — not as free beer!**
  - As soon as you don’t pay for support, you can’t blame anyone for lack of responses

How to ask for help II

- Most of software we use to process our data is provided under “best effort”, without warranty...
- Plenty of scientific software is not written by professional programmers, authors often do not foresee everything what could happen and they could have troubles when fixing reported issues...
- There are plenty of reasons some software doesn’t work — usage/data author didn’t expect, unsupported version of operating system, author’s mistake, user’s mistake, unexpected interaction with another software, ...
- Authors wish their software to be useful — constructive feedback, reporting bugs and wishes is welcomed, but it must be provided in the way useful for the developer

- Try to find the best place to ask your question — specific forum for particular distribution or software use to be the best option
- Learning command line is like learning foreign language — it takes time...
- Reading documentation is not wasting of time!

ChatGPT and others

- Usage of generative AI, large language models (LLM) like ChatGPT can help a lot
- Very good to improve code and fixing non-working code
- Can well explain existing code — good starting point
- Good for adaptation of existing code for user's needs
- Limited usage to generate new code — can misunderstand request, code can be technically wrong, but not well working, can be incomplete or limited in corner-cases
- Without at least basic understanding of subject it is hard to realize if ChatGPT solution is correct — user should be at least basically able to work without AI, otherwise there is high risk of failures
- One should never rely on tool with unclear functionality and problematic usage of of users' data and copyrighted content

Main general fora I

Question must have certain form!

Before asking, ensure your question is in answerable form (previous slides).

- Sloppily asked question can't be answered at all...
- Check documentation, manuals and search the Internet before asking

- Probably the best are fora from StackExchange `https://stackexchange.com/sites`
- General forum for programmers `https://stackoverflow.com/`
- UNIX forum `https://unix.stackexchange.com/`
- Forum for administrators `https://superuser.com/`
- Questions mainly (not only) related to servers `https://serverfault.com/`
- Uncle Google is your friend here (*"how to XXX in BASH/Linux"*)...

Main general fora II

- Bioinformatics and related topics is discussed on Biostars and StackExchange
- Do not hesitate to ask on the forum or contact developers, preferably through some public forum or mailing list, they usually respond quickly and helpfully... — they wish their software to be working and useful
- Plenty of bigger projects have their own web fora or e-mail conferences — search for it to ask on right place
- In Linux/OSS world, e-mail conferences are sometimes more popular, than web forums or various social networks — try them
- If you find a bug, report it according to instructions given by the project
- Plenty of software packages have bug (issue) trackers (e.g. on GitHub) to report any problem with the software and discuss — search them on software homepage
- Programming languages like R or Python have their own discussion fora, commonly specific for particular field

openSUSE

- Homepage https://www.opensuse.org/
- Wiki (knowledge base) https://en.opensuse.org/Main_Page
- Documentation https://doc.opensuse.org/
- News https://news.opensuse.org/, blogs https://planet.opensuse.org/
- Forums https://forums.opensuse.org/
- Mailing lists https://lists.opensuse.org/
- Community https://opensuse-community.org/, short community guide https://opensuse-guide.org/

Debian, Ubuntu, Linux Mint and derivatives

- Debian https://www.debian.org/
  - Documentation, wiki https://wiki.debian.org/
  - Support https://www.debian.org/support
- Ubuntu https://ubuntu.com/
  - Support https://ubuntu.com/support
  - Ask Ubuntu (probably the best forum) https://askubuntu.com/
  - Forum https://ubuntuforums.org/
  - Documentation https://help.ubuntu.com/
  - Kubuntu https://kubuntu.org/
  - Kubuntu forum https://www.kubuntuforums.net/
  - Xubuntu https://xubuntu.org/
  - Lubuntu https://lubuntu.me/
- Linux Mint https://www.linuxmint.com/
  - Documentation https://www.linuxmint.com/documentation.php
  - Forums https://forums.linuxmint.com/

Fedora

- Homepage https://fedoraproject.org/
- Communication and help overview https://fedoraproject.org/wiki/Communicating_and_getting_help
- Wiki https://fedoraproject.org/wiki/Fedora_Project_Wiki
- Official forum https://discussion.fedoraproject.org/
- Documentation https://docs.fedoraproject.org/
- Community forum https://fedoraforum.org/

GNOME, KDE and XFCE

- GNOME https://www.gnome.org/
  - Help for users https://help.gnome.org/users/
  - Wiki https://wiki.gnome.org/
- KDE https://kde.org/
  - Forum https://discuss.kde.org/
  - UserBase wiki https://userbase.kde.org/
  - Application store https://store.kde.org/
  - KDE for education https://apps.kde.org/categories/education/
  - Blogs https://planet.kde.org/
- XFCE https://xfce.org/
  - Documentation https://docs.xfce.org/
  - Wiki https://wiki.xfce.org/
  - Forum https://forum.xfce.org/
  - Blogs https://blog.xfce.org/

LibreOffice

- LibreOffice https://www.libreoffice.org/
  - Document Foundation https://www.documentfoundation.org/
  - Ask LO https://ask.libreoffice.org/
  - Wiki of LO https://help.libreoffice.org/ and DF https://wiki.documentfoundation.org/Main_Page
  - Documentation https://documentation.libreoffice.org/
- Česky
  - Novinky a informace https://www.openoffice.cz/
  - Fórum https://forum.openoffice.cz/
  - Podrobná příručka https://www.root.cz/knihy/libreoffice-writer-prakticky-pruvodce/ (jedna z vůbec nejlepších dostupných knih)

České weby — zdroje informací a fóra I

- ABC Linuxu https://www.abclinuxu.cz/
  - Učebnice GNU/Linuxu https://www.abclinuxu.cz/ucebnice
- Root https://www.root.cz/
- LinuxExpres https://www.linuxexpres.cz/
  - Správa linuxového serveru https://www.linuxexpres.cz/praxe/sprava-linuxoveho-serveru
- LinuxDays (největší konference) https://www.linuxdays.cz/
- Seminář Install fest https://installfest.cz/
- Konference OpenAlt https://openalt.cz/
- OpenOffice/LibreOffice https://www.openoffice.cz/
  - Podrobná příručka https://www.root.cz/knihy/libreoffice-writer-prakticky-pruvodce/
  - Fórum https://forum.openoffice.cz/
- Ubuntu https://www.ubuntu.cz/

České weby — zdroje informací a fóra II

  - Wiki https://wiki.ubuntu.cz/
  - Fórum https://forum.ubuntu.cz/
- Fedora https://mojefedora.cz/
  - Wiki https://wiki.mojefedora.cz/doku.php
  - Fórum https://forum.mojefedora.cz/
  - Příručka https://wiki.mojefedora.cz/doku.php?id=navody:prirucka:obsah
- Linux Mint https://www.linux-mint-czech.cz/
  - Fórum https://forum.linux-mint-czech.cz/
- Johančiny Pohádky z příkazové řádky https://eldar.cz/kangaroo/binarni-sxizofrenie/johanka-pohadky-z-prikazove-radky.html

The end

Our course is over...

...I hope it was helpful for You...

...any feedback is welcomed...

...happy Linux hacking...

...any final questions?

Typesetting using XeLaTeX on openSUSE GNU/Linux January 13, 2024