Vlaanderen
is supercomputing

Introduction to Linux

Ine Arts, Franky Backeljauw, Stefan Becuwe, Kurt Lust, Carl Mensch, Michele Pugno, Bert Tijskens, Robin Verschoren

Version Fall 2024

VLAAMS
SUPERCOMPUTER
CENTRUM

*Innovative Computing
for A Smarter Flanders*

vscentrum.be

Overview

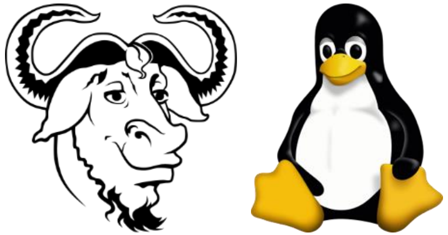
DAY 1 – basics

- The shell
  - exploring the command line
- The filesystem
  - navigating the filesystem
  - manipulating files & directories
  - reading and editing text files
- Useful tools
  - hands-on
- Pipelines & scripting
  - streams & redirection
  - pipelines

DAY 2 – diving deeper

- The environment
  - environment variables
  - aliases & persistent settings
- The shell
  - expansions
- Useful tools
  - regular expressions
- Bash scripting basics
- Extra topics
  - ownership & permissions
  - running & stopping programs

What is GNU/Linux?



- Unix-like computer operating system (OS)
  - free and open-source, worldwide community, active development
- Under the hood: Linux kernel
  - abstraction between hardware and software
  - device drivers, system calls, process and memory management, …
- Typically offers GNU utilities and libraries
  - basic tools to work with files, compile programs, …
  - e.g.: coreutils, binutils, Bash shell, …
- Comes in many flavours, called distributions
  - bundles desktop environments, applications, …

Red Hat Enterprise Linux fedora Rocky Linux openSUSE debian Canonical Ubuntu

Available Linux-like environments

- Microsoft Windows
  - Microsoft Subsystem for Linux (WSL)
  - MobaXterm

- macOS
  - Terminal app (built-in) or iTerm2
  - note: macOS is based on BSD (Unix), thus offering BSD variants of commands
  - use package managers like Homebrew (or MacPorts) to install the GNU utilities
    - e.g. (using Homebrew): `brew install coreutils findutils gnu-tar gnu-sed grep wget`
    - use (GNU) `gsed` instead of (BSD) `sed`

- Use an online terminal emulator
  - e.g.: https://sandbox.bio/tutorials/playground

Vlaanderen
is supercomputing

The shell — Part 1

Exploring the command line

VLAAMS
SUPERCOMPUTER
CENTRUM

Innovative Computing
for A Smarter Flanders

vscentrum.be

What is the shell?

- A program that interprets commands and sends them to the OS

- Sometimes referred to as ”the terminal” or a “**Command Line Interface**” (CLI)
  - waits for input and performs the requested tasks
  - the input language is a **scripting language** (variables, iterations, …)
  - provides access to 100s of commands/programs

- Different shell programs exist
  - on most Linux systems, the default shell is called bash (Bourne Again SHell)
  - note: on macOS, the default shell is zsh, but bash is also available

Command line basics

bash

- $ and text preceding it is called the “prompt”
  - executing a command: type a command after the prompt and press the **Enter** key
  - autocompletion: type part of the command and press the **Tab** key (↹)

```
$ ls -l /etc/host↹
```

- **Linux systems are case and space sensitive**
  - files: `myfile` is not the same as `MyFile`
  - commands: spaces separate parts of commands

- Some keyboard shortcuts when using the Bash shell environment

Left ← and **Right** →	moving around the line	`Ctrl + a`	go to the beginning of the line
Up ↑ and **Down** ↓	browse the command history	`Ctrl + e`	go to the end of the line
`Ctrl + r`	backward history search	`Ctrl + l`	clear the screen

Hands-on

- Enter the following commands and try to interpret the output

```
$ echo Hello, world.
$ date
$ date --utc
$ cal
$ whoami
$ hostname
$ uptime
```

```
$ clear
$ sleep 3
$ time sleep 3
$ who
$ echo $SHELL
$ echo -n Hello, world.
…
```

Anatomy of a command

- Single command: program that does one thing

```
$ command
```

- **Arguments** (parameters): provide the input/output that the command interacts with

```
$ command argument1 argument2 [...]
```

- **Options**: modify a command's behavior (also called *flags*)

`$ command -option`	single dash + one letter (*short* form)
`$ command --long-option`	double dash + one word (*long* form)

- Generally, they compose as follows:

```
$ command [-o]... [--long-option]... [argument]...
```

Arguments & options

- Interpreted by the command itself → usage depends on the command
  - **convention: options first, non-option arguments last**
  - short options can be combined, the order often doesn't matter

    ```
    $ date -R -u = $ date -Ru
    ```

  - for some commands, strict ordering rules apply

    ```
    $ find -maxdepth 2 -type f
    ```

  - non-option arguments often refer to a filename

    ```
    $ less myfile
    ```

  - but not always

    ```
    $ echo "This is an example"
    $ date +"%A %e %B"
    ```

Types of commands

type

- A command can be either:
  - any **program** (or script) on the system
    - use `which` to find out where the program is located/installed
  - a **built-in** shell command
    - get an overview with `man builtin`
  - an **alias** or (user-defined) shorthand for a more complex command
    - use `alias` to see the currently defined aliases
  - a (user-defined) shell **function**

Getting help

man

- Documentation for commands is available as online Linux man pages
  - *There is no shame in using Google or ChatGPT for help, the web is your friend!*

- Or directly from the command line itself
  - ask a command about its use with the `--help` or `-h` options (if available)

```
$ ls --help
```

  - manual pages for commands

```
$ man ls
```

  - More elaborate info manuals

```
$ info ls
```

- Search man pages for keywords

```
$ man -k <keyword>
```

Getting help

man

- Efficiently reading man pages

↓ / ↑ or `j` / `k`	scrolling up or down
`h`	help for the man page viewer
`q`	quit reading the man page

- Searching through man pages

`/ + ”word” + Enter`	search for the given word
`n`	find the *next* occurrence
`N`	find the *previous* occurrence

- Conventions for describing key combinations

`^-<key>` = `Ctrl + <key>`	press Ctrl and the given key together
`C-<key>` = `Ctrl + <key>`	
`M-<key>` = `Alt + <key>`	M stands for “Meta” key (*note: Option on Apple keyboards*)

Vlaanderen
is supercomputing

The filesystem — Part 1

Navigating the filesystem

Manipulating files & directories

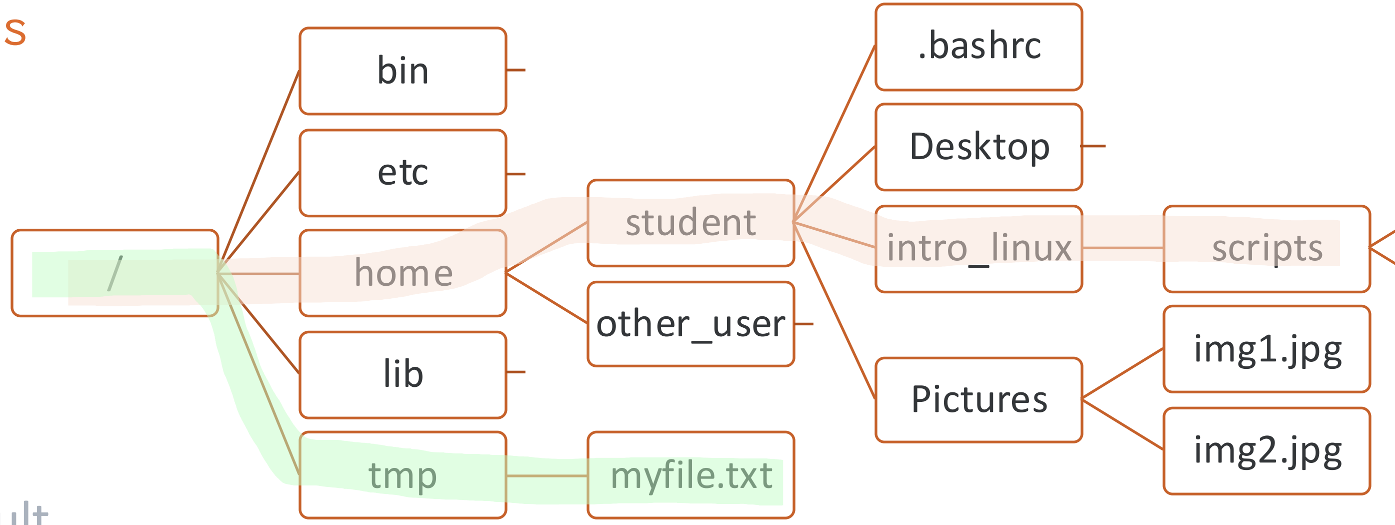
VLAAMS
SUPERCOMPUTER
CENTRUM

Innovative Computing
for A Smarter Flanders

vscentrum.be

The filesystem

- Tree of **directories** and **files**
- **File name** describes the full location (also called *path*) in the file system
  - `/home/student/intro_linux/scripts`
  - `/tmp/myfile.txt`
  - / is called the *root* directory
- Directories are separated by /
- The filesystem is **case sensitive**
  - note: macOS is case insensitive by default



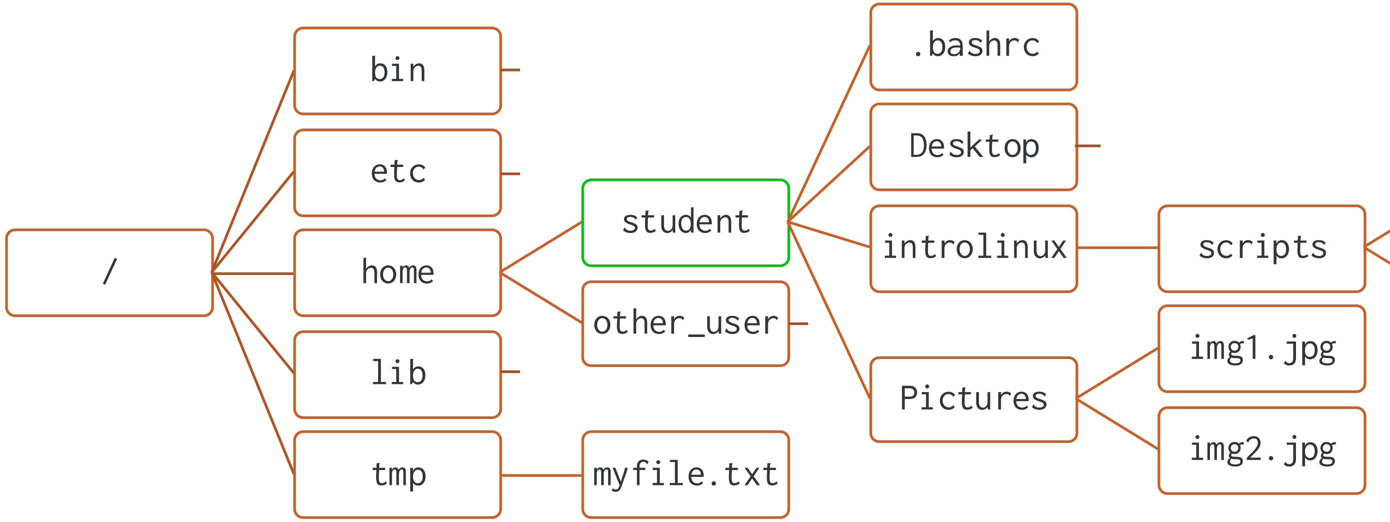
VLAAMS SUPERCOMPUTER CENTRUM

The filesystem

pwd

- **Absolute** file name path starts from root /
- **Relative** file name starts from *current working directory*
- pwd prints the current working directory
  - at login, usually your home directory
- Use .. to refer to a *parent directory*
- E.g., starting from /home/student

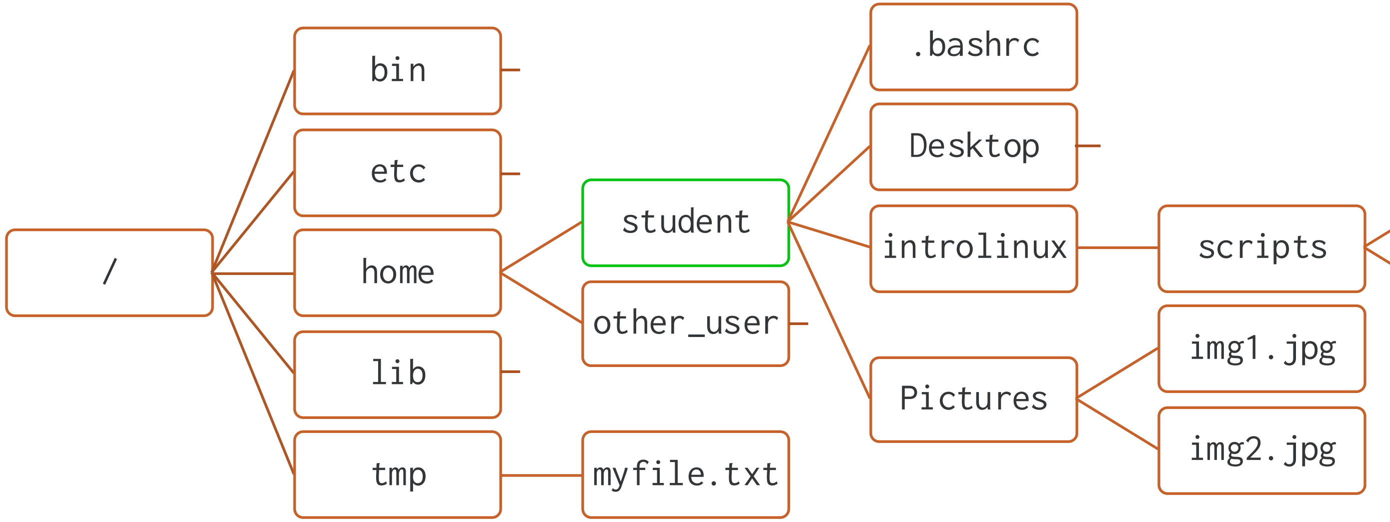
relative path	absolute path
..	/home
../other_user	/home/other_user
../..	/
introlinux	/home/student/introlinux



The filesystem

pwd

- **Absolute** file name path starts from root /
- **Relative** file name starts from *current working directory*
- pwd prints the current working directory
  - at login, usually your home directory
- Use .. to refer to a *parent directory*
- note: on Windows
  - folders are separated by \
  - the filesystem is case insensitive
  - the root indicates a physical partition, e.g. C:\
  - there can be multiple (root) trees



Navigating the filesystem

cd
ls

- Use **cd** `<directory>` to change the current directory

```
$ cd Downloads
$ cd ../Documents
$ cd -              go back to the previous directory
$ cd                go to your home directory
```

- **ls** (without arguments) lists the current directory’s contents

- **~** (“tilde”) is a shorthand for the absolute path to your *home directory*

```
$ cd ~ = $ cd /home/<username>
$ cd ~/Downloads = $ cd /home/<username>/Downloads
```

- A single `.` points to the *current directory*

```
$ cd ./Downloads = $ cd Downloads
```

Hands-on

- Try out the following sequence of commands

```
$ cd
$ ls
$ cd Documents
$ pwd
$ cd ..
$ cd ./Documents
$ pwd
```

```
$ cd /bin
$ ls
$ pwd
$ cd ~
$ pwd
$ cd -
$ pwd
```

Manipulating files and directories

mkdir
rmdir

- **Warning: no “recycle bin” or undo!**
  - be *very* careful when deleting/copying/moving files at the command line!

- mkdir creates directories

```
$ mkdir dir1 dir2 dir3
```

  - create *nested* directories

```
$ mkdir -p topdir/subdir/subsubdir
```

- rmdir removes *empty* directories

```
$ rmdir dir1 dir2 dir3
```

Move, copy and remove

mv
cp
rm

- `mv source target` moves (renames) files and directories
  - if `target` = existing file → overwrite
  - if `target` = existing directory → move inside it

```
$ mv source1 source2 ... target
```

move list of items into existing `target` directory

- `cp source target` copies files and directories
  - same rules as `mv`, except:

```
$ cp srcdir target
cp: -r not specified; omitting directory 'ttt1’
```

  - **recursively** copy directories and their content:

```
$ cp -r srcdir target
```

- `rm file1 file2 ...` removes (deletes) files — *remember: no “recycle bin” or undo!*

```
$ rm -r mydir
```

recursively deletes directories *with* their contents

Using wildcards

- Wildcards help generate lists of filenames, e.g.:

```
$ mv file*.txt target
```

  - Bash replaces `file*.txt` by the list of matching files.

- `*` matches everything → `file*.txt` matches any filename which
  - starts with `file` and ends with `.txt`

- But remember: **no “recycle bin" or undo!**

  → typing mistake can be dangerous!

- *Safety first* for `cp`, `mv` and `rm`

  using **`-i`** or **`--interactive`** asks for **confirmation** before overwriting or deleting

Wildcard expressions

- `*` any sequence of (0 or more) characters

  `file*.txt` → `file.txt file_copy.txt file1.txt ...`

- `?` any single character

  `file?.txt` → `file1.txt file2.txt ... files.txt`

- `[set of characters]` any single character from the given set

  `[fF]ile.txt` → `file.txt File.txt`

- `[!set of characters]` any single character *not* from the given set

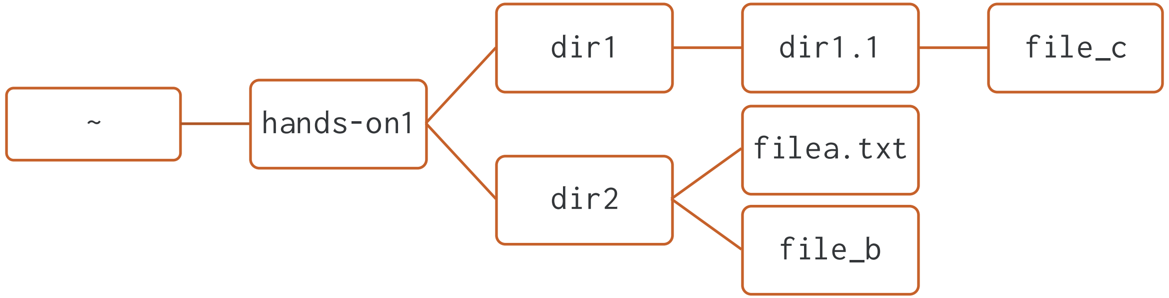
  `file[!123].txt` → `file4.txt file5.txt ... files.txt`

- `[[:class:]]` use a predefined character class

VLAAMS
SUPERCOMPUTER
CENTRUM

Hands-on

- Create new directories and files in your home directory, according to the given diagram
  - use `touch file.txt` to create empty file
  - check your result with `tree ~/hands-on1`
  - *challenge yourself:* do this exercise from your home ~ without using cd



- Let's move things around
  - copy the files in `dir1.1` to its parent directory
  - rename `dir1` to `dir0`
  - copy `dir2` (including its contents) to `dir2_backup`
  - delete the files in `dir2` using wildcards
  - restore the backup directory

Hands-on

- Which names match the following patterns?

```
[abcdefghijk]*.pdf
backup.[0-9][0-9][123]
[Ff]ile?.*
file_[[:digit:]].txt
```

```
file_1.txt
cv.pdf
backup-001
backup.182
introLinux.pdf
A.pdf
File_C.docx
thesis.pdf
Filea.txt
backup.634
```

Reading and editing text files

- **Reading** (displaying) text files
  - `cat` → display the entire content of a text file
  - `more` → display the content of a file one screen at a time
  - `less` → allows forward and backward navigation and searching (*less is more*)
  - `head -n <x>` or `tail -n <x>` → print the first/last *x* lines of a file

- Create or edit text files using **editors** that run inside the terminal
  - `nano` → simple and straightforward text editor (user-friendly, easy to use interface)
  - `vi` → stands for visual interface, takes some practice (use ”modes” for insert or commands)
  - touch → create an empty file (or update the timestamp of the file if it already exists)

Useful tools — Part 1

Hands-on & examples

Hands-on

- Scenario: a colleague sends you a link to a dataset (here: zip-file) and you want to know how many inputs there are in the file `squeue.txt`
  - *note: step by step instructions and commands are given*
    - *it is up to you to look up the correct usage*
  - Download the file https://calcua.uantwerpen.be/courses/introlinux/input.zip — use `wget`
  - Extract (or unzip) the files — use `unzip`
    - *can you look at the content of the zip file without unzipping it?*
  - Locate the file named `squeue.txt` — use `tree` and `find`
    - *which tool was better suited?*
  - Count the number of lines in this file — use `wc`

Hands-on

- Scenario: you download some scripts and you quickly want to know the value of a parameter
  - *note: step by step instructions and commands are given*
    - *it is up to you to look up the correct usage*
  - Download the files: https://calcua.uantwerpen.be/courses/introlinux/pi_montecarlo.tar.gz
  - Extract the files — use `tar`
    - *pay close attention to the options*
  - You encounter two scripts with a similar name: `script01_new.py` and `script01_latest.py`
    Show the difference between the two files, but ignore white spaces — use `diff`
  - Show the line where parameter `n_points` is assigned — use `grep`

Download & extract files

wget
unzip
tar

- Download files with **wget**

```
$ wget https://[...].zip
```

- ZIP file format

```
$ unzip file.zip

$ zip -r file.zip
```

- TAR / TAR.GZ

```
$ tar -zxf file.tar.gz target_dir

$ tar -zcf file.tar.gz source_dir
```

  - TAR stands for Tape Archive — also called “*tarball*”
  - more common in Unix/Linux environments
  - preserves file permissions, ownership, and timestamps, making it more suitable for backups and archives

What I type:

```
$ tar czf data.tar.gz data
$ tar xzf data.tar.gz
```

What I say in my head:

"create ze file"

"extrakt ze file"

Comparing files and directories

diff

- Detect differences between text files

`$ diff -i file1 file2`	ignore case
`$ diff -w file1 file2`	ignore all white space
`$ diff -y file1 file2`	output in two columns
`$ diff -r dir1 dir2`	recursively compare directories

Hands-on

- Scenario: you see that a colleague opens file `student_scores.csv` with *comma separated values* in Excel to sort the data by Score
  - Not on your watch — you use Miller like a pro!
  - *note: parts of the commands are given, complete them using the documentation*
- Start by reading Miller in 10 minutes
- Install the Miller command **mlr**
  - on Linux: `apt install miller` (Ubuntu) or `yum install miller`
  - on macOS with Homebrew: `brew install miller`
- Pretty-print the .csv file — `mlr --icsv ???`
- Sort (and pretty-print) the .csv by the values of field Score — `mlr --icsv ???`

Processing text-formatted structured data

mlr

- Why our sysadmin loves **Miller** (*obligatory slide!*)
  - easily query, shape and/or reformat CSV, TSV, JSON, … data files
  - pretty-print data files, convert between file formats
  - using compact verbs instead of a programming language

- Some examples

`$ mlr --icsv --ojson cat example.csv`	convert `example.csv` to JSON format
`$ mlr --c2j cat example.csv`	use a keystroke-saver flag
`$ mlr --csv tail -n 4 example.csv`	print header and last 4 lines
`$ mlr --c2p cut -f user,jobid example.csv`	pretty-print only fields `user` and `jobid`
`$ mlr --t2x -N filter '$1 == "Fedora”’ then cut -f 2,3 then sort -n 2 distrostab.txt`	filter on Fedora, show version and date

Pipelines & scripts — Part 1

Streams & redirection

Pipelines

Streams, redirection, pipelines

- Output and input (**I/O**) of commands is managed using *streams* and *file descriptors*
  - **streams** provide an interface with powerful formatted input and output functions (high-level)
  - under the hood, streams use **file descriptors** (fd) to keep track of the I/O-resources (low-level)

stream	readable name	fd	purpose
stdout	standard output	**1**	for normal output
stderr	standard error	**2**	for printing warnings and errors
stdin	standard input	**0**	from which commands receive input

  - *by default, “stdin” is read from the keyboard, while “stdout” and “stderr” are sent to the terminal*

- We can **redirect** the output and input streams, to
  - write output to a file
  - send output from one command to input of another
  - read stdin from a file

Output redirection

- To redirect an output stream, use operator **`i>`** with its associated file descriptor (fd) **`i`**

- Redirect standard output (stdout)

```
$ ls 1> ls-output.txt
$ ls > ls-output.txt
```

without fd: redirects stdout

  - the file `ls-output.txt` is created and contains the command’s output
  - note: stderr is still shown in terminal

- Redirect standard error (stderr)

```
$ ls wrong-filename 2> ls-error.txt
```

- Redirect both stdout and stderr

```
$ ls *.txt *.jpg 1> ls-output.txt 2> ls-errors.txt
```

to *different* files

```
$ ls *.txt *.jpg > ls-output-and-errors.txt 2>&1
```

to the *same* file

Output redirection

>>

- Hiding a program's output

```
$ ls > /dev/null
```

  - /dev/null is a special "file" that discards everything written to it

- note: redirecting (>) creates a new file
  - if a file exists with the same name, it will be overwritten!
  - if the command produces no output, the file will be empty

- Append stdout and/or stderr to the end of a file, *without erasing previous content*

```
$ date >> diary.txt
$ echo "Dear diary, today ..." >> diary.txt
$ ls notfound 2>> ls-errors.txt
$ ls *.txt *.jpg >> ls-output-and-errors.txt 2>&1
```

Input redirection

- Standard input (stdin) is by default read from the keyboard — example: try with **bc**

- The input redirection operator `< filename` opens a file, and the program processes it as input

```
$ echo "2 * 17" > homework.txt
$ bc < homework.txt
34
```

  - useful for automating commands that normally require user input
  - or for reading from specific sources (devices) directly

- Note: for commands that accept a file name argument. these commands have the same effect

```
$ less homework.txt

$ less < homework.txt
```

- Redirecting both standard input and standard output

```
$ bc < homework.txt > answers.txt
```

Pipelines

- Combine several commands by chaining them using the “pipe” operator **|**

```
$ command1 | command2 | command3 [| ...]
```

  - a *pipeline* creates a flow of data between commands
  - stdout from `command1` is directly sent to stdin of `command2` (etc)
  - the commands run in parallel, each command processes input as it becomes available

- Example: scrolling through the list of all processes with **`ps`** and `less`

```
$ ps aux | less
```

- Create complex commands from simple building blocks

```
$ who | cut -d' ' -f1 | sort | uniq > users
```

- note: to pipe stderr from a command, redirect it to stdout

```
$ command1 2>&1 | command2
```

Hands-on

- Given the file `chemistry.txt`, how many courses are thought by Wouter Herrebout in the first semester?
  - *note: use pipelines whenever possible!*
  - Investigate the file — use `cat`
  - Print only the lines belonging to the first semester — use `grep`
  - Of those lines, select the lines containing Wouter Herrebout — use `grep`
  - Count the resulting number of lines — use `wc`

Hands-on

- Which are, in alphabetical order, the last 5 course codes starting with 1001WET? Write them to a new file.
  - Alphabetically sorted by course code
  - Sort the lines in alphabetical order — use `sort`
    - *pay close attention to the options*
  - Of those lines, select the last 5 — use a pipe and `tail`
  - Write the output to a new file
  - Edit your pipeline to instead sort alphabetically by course nam

Hands-on

- Which course is listed twice in the file chemistry.txt?
  - Print each unique line of the file, with the number of times it occurred — use `uniq`
    - *carefully read the last line of DESCRIPTION in the man page*
  - Print the line with the highest count — use pipelines, `sort` and `tail`

Hands-on

- Take a look at the file `squeue.txt`
  - *note: this file shows a list of jobs that were submitted to the cluster*
  - Build pipelines to
    - find how many jobs are running
    - check how many jobs are running per user
    - show how many jobs per user are running, sorted in descending order
    - sort the jobs on partition zen2 by their state and job id
    - count the number of jobs per number of nodes
    - count the number of jobs per state
    - give, per user, the number of running jobs as well as the number of nodes in use
  - *question: is there a command to do this without pipelines?*

Overview of frequently used commands

- Typical commands for pipelines

`cat`	concatenate files (useful to print out file content)
`grep`	filter lines which match a given search pattern
`head` / `tail`	print first/last lines of input
`sort`	sort input alphabetically
`uniq`	report or leave out repeated lines
`wc`	print the number of lines, words and bytes of input
`sed`	transform input (pattern replacement and more)

- Find more commands in the GNU core utilities manual

Sneak preview — Shell scripts

bash

- **shell script** = text file containing a series of commands

- Example script “`myscript.sh`”

```
my_analysis input.data > my_results/science.txt
tar -cvzf my_results.tar.gz my_results
rm input.data
```

- Run (execute) the script

```
$ bash myscript.sh
```

- note:
  - commands are separated by newlines or by semicolons ‘`;`’ (as in the terminal)
  - commands are executed one after the other, just as if you entered them manually

Vlaanderen
is supercomputing

The environment

Environment variables

Aliases & persistent settings

Environment variables

export

- We can use variables in the shell

`$ myvar=some_value`	set the value for variable myvar
`$ echo $myvar`	get the current value of myvar — called “*variable expansion*”
`$ set`	display all variables

  - no spaces around ‘=’
  - no spaces in `some_value` unless using quotes
  - there are “plain” variables — they only exist in the running shell itself

- **Environment variables** are special

`$ export myvar`	make `myvar` an environment variable
`$ printenv`	display environment variables

  - they are passed on to processes started from the shell
  - they can influence the behaviour of programs (e.g. OMP_NUM_THREADS)

Environment variables

- Some standard environment variables

PATH	a colon-separated list of directories that are searched when you enter the name of an executable program
HOME	the path name of your home directory (~)
USER	your user name
SHELL	the name of your shell program
PWD	the current working directory
TMPDIR	directory for temporary files (usually /tmp)

- Example: access an environment variable from within a Python script

```
$ python3 -c 'import os
> print("hi there,", os.getenv("USER"), "!")'
```

Aliases

alias

- Substitute a string for a simple command
- `$ alias <name>=<value>` means that `$ <name>` will be replaced by `$ <value>`
- Handy to set default options and simplify your commands

`$ alias ls="ls -F --color=auto"`	append filetype indicator, colorize output
`$ alias lart="ls -Falrt --color=auto"`	show hidden files, recently modified first

- Removing (deleting) aliases

`$ unalias <name>`	removes the alias for <name> (in the current shell)
`$ unalias -a`	removes all aliases (in the current shell)

Environment startup

- User-defined aliases, variables and functions are reset when restarting the shell

- Store the settings so they are persistent for your environment
  - applied every time you start a (interactive) shell:

    `~/.bashrc` you can define your own **aliases** and **functions** here

  - applied once at login:

    `/etc/profile` system wide, for all users

    `~/.bash_profile`

    `~/.bash_login`

    `~/.profile`

Vlaanderen
is supercomputing

The shell — Part 2

Expansions

Shell expansions

- When you type a command line and press **Enter**
  - the shell performs several processes on the text before it carries out your command
  - the process that makes this happen is called **expansion**

Variable expansion

- **$**variable_name → variable’s current value
  optional **{}**: **${**variable_name**}**

```
$ echo $USER
$ set                     display all variables
$ echo $SUER              what if variable doesn’t exist?
$ echo ${USER}_home
$ echo $USER_home         doesn’t work without {}!
$ myvar='Hello, world!'   set a variable
$ echo $myvar
```

Shell expansions

Arithmetic expansion

- `$((expression))` → result of expression

```
$ echo $((10 + 5 + 3))
```

  - arithmetic expression — note: only integers in bash!
  - operators: `+`, `-`, `*`, `/` , `%` (remainder), `**` (exponentiation)
  - single parentheses may be used to group multiple subexpressions:

```
$ echo $(( (5**2) * (3*4) ))
```

Command substitution

- `$(command)` → output of command

```
$ echo We are now $(date)
$ echo I see $(ls -A | wc -l) files and subdirs
```

Shell expansions

Escaping special characters & using quotes

```
$ echo The total is $100.00 # ?!
```

- Use “**escape**” **character** \ for literal use of special characters ($, \, `, {, }, (,), *, ␣)

```
$ echo The total is \$100.00
```

- Inside single quotes '' special characters lose their meaning → no expansion at all

```
$ echo text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER
$ echo 'text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER'
$ echo "text ~/*.txt {a,b} $(echo foo) $((2+2)) $USER"
```

- Inside double quotes "" special characters lose their meaning except $, \, `

```
$ echo "$USER $((2+2)) $(cal)"
$ echo "The total is \$100.00"
```

Shell expansions

Other

- Word splitting: words separated by space become separate arguments

```
$ touch "two words.txt"
$ ls -l two words.txt
$ ls -l "two words.txt"
$ ls -l two\ words.txt
$ ls -l two↹
```

- Quote removal: after all expansions, quotes are removed unless you escape or quote the quotes

```
$ echo "hello world"
$ echo \"hello\" '"world"'
```

Useful tools — Part 2

Regular expressions

Regular expressions

- Often called “*regex*”
- Symbolic notation used to **match text patterns**
- Similar to wildcards (*, [], ?), but more powerful

- Many programs and programming languages support regular expressions:
  - `grep`, `sed`, …
  - Text editors, e.g. `emacs`
  - Python, Perl, Matlab…

  Though slight differences can exist in notation and supported patterns

Regular expressions

grep

- Example: counting animals in the Bible.

```
$ grep -Eo ' (dragon|serpent|lion|eagle)s? ' bible.txt | sort | uniq -c
```

```
     10      dragon
      4      dragons
     10      eagle
      3      eagles
     43      lion
     13      lions
     14      serpent
      4      serpents
```

Regular expressions

- Literal characters and digits.

```
$ grep lion bible.txt
```

- “**Metacharacters**”: repetitions, grouping, alternatives, ...

- Two notations for metacharacters:
  - **basic regular expressions** (BRE):

    ```
    ^ $ . [ ] * \( \) \{ \} \? \+ \|
    ```

  - **extended regular expressions** (ERE):

    ```
    ^ $ . [ ] * ( ) { } ? + |
    ```

- *Slides use ERE for readability.*

Regular expressions

Metacharacters

grep -E

- `.` Match any character.

```
$ grep -h '.word' /usr/share/dict/words
```

  - Remark difference with wildcards:

```
$ touch .zip 1.zip 1zip 22.zip 2zip
$ ls *zip
$ ls *.zip
$ ls | grep .zip
```

- `^ $` anchors: beginning (^) or end ($) of line.

```
$ grep -h '^word' /usr/share/dict/words
$ grep -h 'word$' /usr/share/dict/words
$ grep -h '^word$' /usr/share/dict/words
```

Regular expressions

Character classes

- `[]` character class

`[lw]ord`	matches `lord` and `word`
`[l-w]ord`	matches `lord`, `mord`, `nord`, …, `word`
`[^lw]ord`	matches any `ord` not preceded by `l` or `w`
`[^l-w]ord`	matches any `ord` not preceded by `l`, …, `w`
`^[A-Z]`	matches any word beginning with an upper case letter
`^[-AZ]`	matches any word beginning with `-`, `A` or `Z`

Regular expressions

Repetitions

- `?` Match preceding element zero or one time
- `*` Match preceding element zero or more times
- `+` Match preceding element one or more times
- `{}` Match preceding element a specific number of times:

`{n}`	exactly *n* times
`{n,m}`	at least *n* times, at most *m* times
`{n,}`	at least *n* times
`{,m}`	at most *m* times

- Examples:

`A*`	matches <empty string>, A, AA, ...
`.*`	matches any sequence of characters
`\$[1-9][0-9]{2,}`	match any amount of `$100` or more

Regular expressions

Sub-expressions, alternatives

- `( )` sub-expression

`(bla)+`	matches 1 or more repetitions of `bla`

With `\`*n* you can refer to the *n*-th subexpression

- `|` alternatives

`word|lord`	matches `word` and `lord`
`(w|l)ord`	matches `word` and `lord`, using grouping
`(w|l|sw)ord`	matches `word`, `lord` and `sword`

Regular expressions

Basic vs. extended regular expressions

- Extended regular expressions: `grep -E` or `egrep`
- Examples:

```
$ egrep 'Et|Ut' /usr/share/dict/words
$ grep 'Et\|Ut' /usr/share/dict/words
```

find `Et` or `Ut` in **`/usr/share/dict/words`**

```
$ grep -Eh '^(bz|gz|zip)' dirlist*.txt
$ grep -h '^\(bz\|gz\|zip\)' dirlist*.txt
```

begins with bz or gz or `zip`

```
$ grep -Eh '^bz|gz|zip' dirlist*.txt
$ grep -h '^bz\|gz\|zip' dirlist*.txt
```

begins with bz or contains gz or contains `zip`

Regex: overview

.	Match any	(BRE)
^ $	anchor beginning or end of line	
[ ]	character classes	

repetitions: repeat preceding element:

?	0 or 1 times	\?
*	0 or more times	
+	1 or more times	\+
{x}	x times	\{x\}
{x,y}	more than x, less than y times	\{x,y\}

( )	subexpression	\(\)
|	alternative	\\\|
\n	n-th subexpression	

Hands-on

- Use `grep -E` on the file `/usr/share/dict/words`:
  - Which words start with `chemi`?
  - which words contain both `her` and `bout`? (answer using 1 regular expression)
  - which words start with a capital letter and contain two consecutive letters `a`?
  - how many five letter words do you find? (use a pipeline)

Find and replace with regex

sed

- sed Stream editor.
  - Editing on a stream of text (standard input or set of files) using regular expressions
  - Typical usage: search and replace

    ```
    sed 's/regexp/replacement/'
    ```

    - By default: only first occurrence on each line;
      to replace all occurrences: add ‘g’ at the end
    - By default: case sensitive
  - Powerful but somewhat complex
  - *For larger tasks, you might choose awk, Perl, Python, ...*

Hands-on

- Find and replace all instances of “`chemie`” by “`scheikunde`” in the file `chemistry.txt` and write the output to a new file.
  - make sure the replacement is case insensitive
  - do the replacement directly in the file

- Rewrite `MM/DD/YYYY` in `distros/distrostab.txt` as `YYYY-MM-DD`
  - match the pattern `MM/DD/YYYY` by using 3 subexpressions
  - construct the replacement by referring to the subexpressions

Find and replace with regex

sed

- **sed** [options] *<script>* <file>

-n	silent: suppress automatic printing
-i	edit file in place
-E	use extended regex

  - Script: *[line selection] <command>*

n[,n2]	line number n (until n2)
$	last line
/regex/	lines that match regex

    - Command:

s/regex/repl/	replace matches for regex by repl
a	append text after current line
d	delete current line
<command>I	case insensitive
<command>g	'global' -> act on all occurrences on this line

Other useful sed commands

sed

Examples

```
$ sed -n '1,5p' distros.txt
```

print only lines 1 to 5

```
$ sed '/Fedora/a from Redhat' distros.txt
```

```
$ sed '/Fedora/d' distros.txt
```

only non-matches (equivalent of `grep -v`)

```
$ sed -i '1d' distros.txt

$ echo "front front" | sed 's/front/back/'

$ sed "s+/home+/thuis+g"
```

Vlaanderen
is supercomputing

The filesystem — Part 2

Ownership & permissions

VLAAMS
SUPERCOMPUTER
CENTRUM

*Innovative Computing
for A Smarter Flanders*

vscentrum.be

Ownership & permissions

id

- Every user has a unique **id** / **name** and belongs to one or more **groups**

- To see your id, name and groups, run `id`

uid	your user id
gid	primary group id
groups	list of all groups you are a member of

- Every file or directory belongs to a user and a group with different access permissions for
  - **U**ser
  - **G**roup
  - **O**thers = all other users who are not a member of the file's group

Ownership & permissions

- Use `ls -l` to see ownership and permissions:

```
$ ls -l scripts
total 512
-rwxr-xr-x 1 vsc20xxx antwerpenall  76 Feb  8 12:43 script01.sh
...
permissions     user     group     size modif.time   filename
```

- `-rwxrwxrwx` three kinds of permissions for “user,” “group” and “others”

permission	files	directories
read	read file’s contents	list directory contents
write	modify file’s contents	create, remove & rename files *(also needs x)*
e**x**ecute	run file as a program	enter directory & access contents

Setting permissions

chmod

- chmod can change the permissions for files or directories

- Add/remove permissions using chmod + or chmod -

```
$ chmod +w file.txt          add write permission for all users
$ chmod g-w file.txt         remove write permission for group
$ chmod ug+x,o-r file.txt
```

- Or using numbers instead, where 0=none, 1=x, 2=w, 3=wx, 4=r, 5=rx, 6=rw, 7=rwx

```
$ chmod 640 file.txt
```

- -R Recursive: change permissions on a directory and all its contents:

```
$ chmod -R go-xr my_private_dir
```

Change ownership

chown

- chown can change the owner and group of files and directories.

```
$ chown owner file.txt
$ chown owner:group file.txt
$ chown :group file.txt
```

  - -R recursive.

```
$ chown -R owner:group my_dir
```

Running programs

Processes and threads

Processes and threads

- A **process** = running instance of a program.
  - has a unique identifier or **PID.**
  - can start other processes: **child** processes.
  - consists of one or more **threads**.

- Threads **share** access to the process' memory,
  but processes **cannot** access other processes' **memory**.

- Parallelization on **multiple CPU cores**:
  - multiple processes ("*distributed memory parallelism*").
  - multiple threads in one process ("*shared memory*").

Processes and threads

Looking at processes

ps

- The command ps prints information on running processes.
  - `$ ps` show processes **in current shell**

    ```
      PID TTY          TIME CMD
     8627 pts/12   00:00:00 bash
    19621 pts/12   00:00:00 ps
    ```
  - `$ ps x` show **all** processes of current user
  - `$ ps ax` show all processes of **all users**
  - `$ ps u` show username, CPU and memory usage
    (can be combined with previous, e.g. `$ ps axu`)
  - `$ ps -u <user>` show processes of the given user

- The commands top or htop show processes together with CPU and memory usage in real time.

Processes and threads

Managing processes

bg
fg

Foreground processes

- Example: run `xclock` with `$ xclock -update 1`

  The process is started, you have no prompt.

- To **terminate** the foreground process, press `Ctrl + c`

  `xclock` disappears, the prompt returns.

- To **stop** (pause) the foreground process, press `Ctrl + z`

  The process is stopped in the background, the prompt returns.

`$ fg`	process resumes in the **foreground**.
`$ bg`	process continues in the **background**.

Processes and threads

Managing processes

&

Background processes

- To start a process in the background, terminate the command by &

```
$ xclock -update 1 &          bash prints the job number and PID, e.g. [1] 9582
```

- Multiple background jobs: use `$ jobs` to see a list:

```
$ xclock -update 1 &
[1] 9582
$ xclock -update 1 &
[2] 9588
$ jobs
[1]- Running xclock -update 1 &
[2]+ Running xclock -update 1 &
```

- Use the job number to control different processes, e.g.

```
$ fg %2          run job 2 in the foreground
```

Processes and threads

Terminating processes

kill

- *Reminder*: `Ctrl + c` terminates the **foreground** process.
- Use the command `kill <PID>` to terminate any process (owned by you)

`$ kill 12345`	Terminate process with id 12345. The process may belong to another shell.

- `kill %<jobnum>` terminates a **background** process:

`$ kill %2`	Terminate job 2, with time for cleanup.
`$ kill -KILL %2`	Terminate job 2 **immediately**.

- Use `$ kill -STOP` and `$ kill -CONT` to pause/resume processes.

Scripting — Part 2

Bash scripting

Shell scripts

- **shell script** = text file containing a series of commands
- Example script “`myscript.sh`”

```
my_analysis input.data > my_results/science.txt
tar -cvzf my_results.tar.gz my_results
rm input.data
```

```
$ bash myscripts.sh
```

run (execute) the script

- note:
  - commands are separated by newlines or by semicolons ‘`;`’ (as in the terminal)
  - commands are executed one after the other, just as if you entered them manually
- Example scripts in https://calcua.uantwerpen.be/courses/introlinux/scripts.zip

Shell scripts

file
dos2unix

- Note about **line endings**
  - line endings are encoded differently under Windows and Unix/Linux
    - Windows style: carriage return + line feed (CRLF, `\r\n`)
    - Unix/Linux style: newline (**`\n`**)
  - this can introduce problems with bash scripts

- Check which encoding is used:

```
$ file filename
```

- If needed, convert your “Windows style” file into a “Unix/Linux” style:

```
$ dos2unix -n inputfile outputfile
```

- A suitable text editor can do this as well

Shell scripts

- $ cat scripts/script01.sh

“shebang”

```
#! /bin/bash

# This is our first script.

echo 'Hello World!' # comment
```

`$ bash script01.sh` call the interpreter (bash) ourselves

`$ chmod +x script01.sh`

`$ script01.sh` doesn’t work because work dir is not in PATH!

`$ ./script01.sh` **the interpreter from the ‘shebang’ is used**

Shell scripts

- `#!` is called “**shebang**”. It tells the system which interpreter should execute the script.
  - For a bash script:

    `#!/bin/bash`
  - Spaces (between parts) are optional

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

- Any scripting language, not just bash.
  - Example for Python:

    `#!/usr/bin/python3` uses that specific Python executable
  - or preferably:

    `#!/usr/bin/env python3` uses the first `python3` found in `PATH`
  - `PATH` *is modified when using software modules (see HPC@UAntwerp)*

Shell scripts

```
#!/bin/bash                                                      # script02.sh

currenttime=$(date +"%x %r %Z")
myname=$USER

echo "id: $myname, current time: $currenttime"
```

- Remember:
  - Setting a variable: without **$**, no spaces around = e.g., `myname=`**`some_value`**
  - Using a variable (variable expansion): with **$** e.g., `echo` **`$`**`myname`

- User variables can not start with a digit: $1, $2, ... are special variables

  ‘command line arguments’

Command line arguments

```
#!/bin/bash                                                              # script07.sh

echo "Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
...
\$9 = $9 "
```

```
$ ./script07.sh these are four arguments
$ ./script07.sh 'this is a single argument'
```

- More than 9 args? → `${10}`, `${11}`, …
- List of all command line arguments: `$@`
- Last arg? `${!#}` or `${@: -1}` or *(Bash only)* `$BASH_ARGV`

For loop

`for variable in list; do commands; done`

```
#!/bin/bash                                                    # script09.sh
for i in A B C D; do
  echo $i
done
```

- ➢ `list` can be any bash expression resulting in a list, e.g.

  `for file in *.txt; do ... done`   loop over each `txt` file

- ➢ if "`in list`" is omitted, `for` loops over the command line arguments

For loop

```
#!/bin/bash                                              # script09b.sh


for i in $(seq 1 10); do
  echo $i
done


for i in $(seq 11 0.75 20); do
  echo $i
done


for i in {21..30}; do
  echo $i
done
```

Hands-on

- Write a script that adds up all command line arguments
  - loop over all command line arguments
  - add each argument to the total — use arithmetic expansion `$(( ))`
  - test your script with different inputs — make sure your script is executable

- What do you expect to happen when instead of integers you input:
    - text?
    - decimals?
  - *test your expectations!*

Hands-on

- Write a script that loops over each command line argument and that
  - creates a directory `dir_<argument>` in the current location
  - copies a template file `input_<argument>.txt` into this directory
  - replaces “`<param>`” in this file by the value of the argument
- *Challenge yourself!*
  - we want the name of the template file as the first command line argument
  - run previous script without changes, with this new argument – what happens?
  - try to fix what went wrong — look into the `shift` command

Hands-on

- Here is an example of a script which some more logic structures

  if

  while

  ~~case~~

  break / continue

  ~~functions~~

  ...

  - *Try to figure out what it does*

```
#!/bin/bash                                       # script12.sh
while echo -n "enter number: "; read NUM
do
  if [ $NUM -eq $NUM ] 2>/dev/null; then
    :
  else
    echo " $NUM is not a number"
    continue
  fi
  if [[ $(( $NUM % 2 )) -eq 0 ]]; then
    echo " $NUM is an even number"
    continue
  fi
  echo " $NUM is an odd number"
  break
done
```

Course feedback

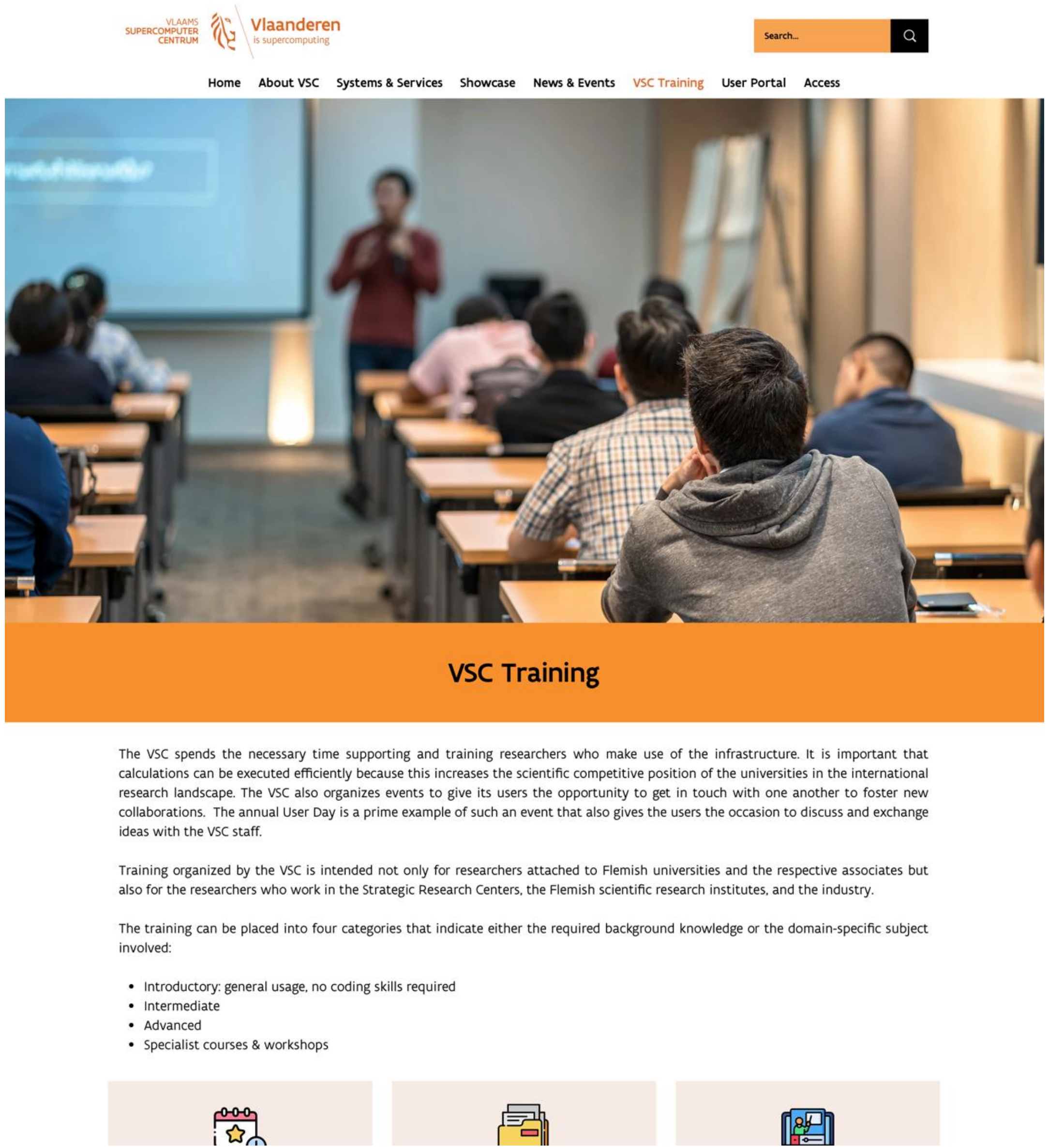
- Please fill in our short questionnaire before Nov 30
- Let us know what you liked and how we can improve our courses
- Thank you for your participation!

Links

- linuxcommand.org/tlcl.php
- free-electrons.com/doc/legacy/command-line/unix_linux_introduction.pdf
- www.ibm.com/developerworks/linux/
- www.howtogeek.com/tag/linux/
- Greg's Wiki Bash Guide: mywiki.wooledge.org/BashGuide
    - Common mistakes: mywiki.wooledge.org/BashPitfalls
- www.tldp.org
    - Advanced bash guide: www.tldp.org/LDP/abs/html/
- Cheat sheets: devhints.io

More training

- hpc.uantwerpen.be
- www.vscentrum.be
- www.vscentrum.be/training



VSC Training

The VSC spends the necessary time supporting and training researchers who make use of the infrastructure. It is important that calculations can be executed efficiently because this increases the scientific competitive position of the universities in the international research landscape. The VSC also organizes events to give its users the opportunity to get in touch with one another to foster new collaborations. The annual User Day is a prime example of such an event that also gives the users the occasion to discuss and exchange ideas with the VSC staff.

Training organized by the VSC is intended not only for researchers attached to Flemish universities and the respective associates but also for the researchers who work in the Strategic Research Centers, the Flemish scientific research institutes, and the industry.

The training can be placed into four categories that indicate either the required background knowledge or the domain-specific subject involved:

- Introductory: general usage, no coding skills required
- Intermediate
- Advanced
- Specialist courses & workshops

VLAAMS
SUPERCOMPUTER
CENTRUM

Supplemental material

More bash scripting structures

if

- Example:

```
if ls file.txt
then echo "That file exists."
else echo "That file doesn't exist."
fi
```

- Generic form

```
if test1; then commands1
elif test2; then commands2
elif ...
else commandsn
fi
```

if

- Most frequently used command with `if` is

```
if test expression
```

or its equivalent form

```
if [ expression ]
```

- bash has an extended replacement

```
if [[ expression ]]
```

which is easier to use, e.g. in combination with variables

if: test expressions

```
#!/bin/bash                                                          # script04.sh

x=5
if [ $x -eq 5 ] ; then
  echo "x equals 5."
else
  echo "x does not equal 5."
fi
```

equivalent to:

```
if test $x -eq 5; then ...

if [[ $x -eq 5 ]]; then ...
```

test expressions: files

`file1 -nt file2`	file1 is newer than file2
`file1 -ot file2`	file1 is older than file2
`-d file`	file exists and is a directory
`-f file`	file exists and is a regular file
`-s file`	file exists and has size > 0
`-L file`	file exists and is a symbolic link
`-r file`	file exists and is readable
`-w file`	file exists and is writable
`-x file`	file exists and is executable

…

- ➢ Search for “bash file test operators” (or `man test`) to see more exotic ones…

test: text strings

`-n string`	the length of the string > 0
`-z string`	the length of the string = 0
`string1 = string2`	strings are equal
`string1 != string2`	strings are not equal
`string1 > string2`	string1 sorts after string2
`string1 < string2`	string1 sorts before string2

test: integers

`int1 -eq int2`	int1 = int2
`int1 -ne int2`	int1 ≠ int2
`int1 -le int2`	int1 ≤ int2
`int1 -lt int2`	int1 < int2
`int1 -ge int2`	int1 ≥ int2
`int1 -gt int2`	int1 > int2

test: combining

- Combining test expressions:

	[ ]	[[ ]]
AND	-a	&&
OR	-o	||
NOT	!	!

- Example:

```
if [[ $((x % 5)) -eq 0 && $((x % 2 )) -eq 0 ]]
then
       echo "$x is a multiple of 10"
fi
```

while

- while test; do commands; done

```
#!/bin/bash                                                            # script06.sh

count=1
while [ $count -le 5 ]; do
  echo $count
  count=$((count + 1))
done
echo "value of count: $count"

echo "Finished."
```

while

```
#!/bin/bash                                                     # script06b.sh

while read jobid partition jobname user state rest; do
  echo $jobid $state
done < squeue.txt
```

- Alternatively (one-liner at prompt):

```
      $ cat squeue.txt | while read line; do ... done
```

- Combining `while` and `read`: easy (quick & dirty) way to process lines of output

  (no worries about how many spaces separate fields).

- `squeue.txt` can be found in the `input` folder

for

`for variable in words; do commands; done`

```
#!/bin/bash                                                     # script09.sh
for i in A B C D; do
  echo $i
done
```

- `words` can be any bash expression resulting in a list, e.g.

  `for file in *.txt; do ... done`   loop over each `txt` file

- if “`in words`” is omitted, `for` loops over the command line arguments

for

```
#!/bin/bash                                                    # script09b.sh


for i in $(seq 1 10); do
  echo $i
done


for i in $(seq 11 0.75 20); do
  echo $i
done


for i in {21..30}; do
  echo $i
done
```

for

```
#!/bin/bash                                                      # script10.sh

for i; do
  if [[ -r $i ]]; then
    max_word=
    max_len=0
    for j in $(strings -n 2 $i); do
      len=${#j}
      if [[ $len -gt $max_len ]]; then
        max_len=$len
        max_word=$j
      fi
    done
    echo "$i: '$max_word' ($max_len characters)"
  fi
done
```

read

- Create variables and read their values from standard input

```
#!/bin/bash                                                        # script05.sh

echo -n "Please enter an integer -> "
read int

echo -n "Enter one or more values > "
read var1 var2 var3 var4 var5

echo "int = ${int}, var1 = ${var1}, ..."
```

- Remarks:
  - -n prevents echo from printing a new line
  - extended version: see `script05a.sh`

Command line arguments

```
#!/bin/bash                                                         # script07.sh

echo "Number of arguments: $#
\$0 = $0
\$1 = $1
\$2 = $2
...
\$9 = $9 "
```

```
$ ./script07.sh these are four arguments

$ ./script07.sh 'this is a single argument'
```

- More than 9 args? Use `shift` (see next slide) or `${10}`, `${11}`, ...
- Last arg? `${!#}` or `$BASH_ARGV` (Bash only) or `${@: -1}`
  - Space in `${@: -1}` is required to avoid confusion with `:-` expansion

Command line arguments

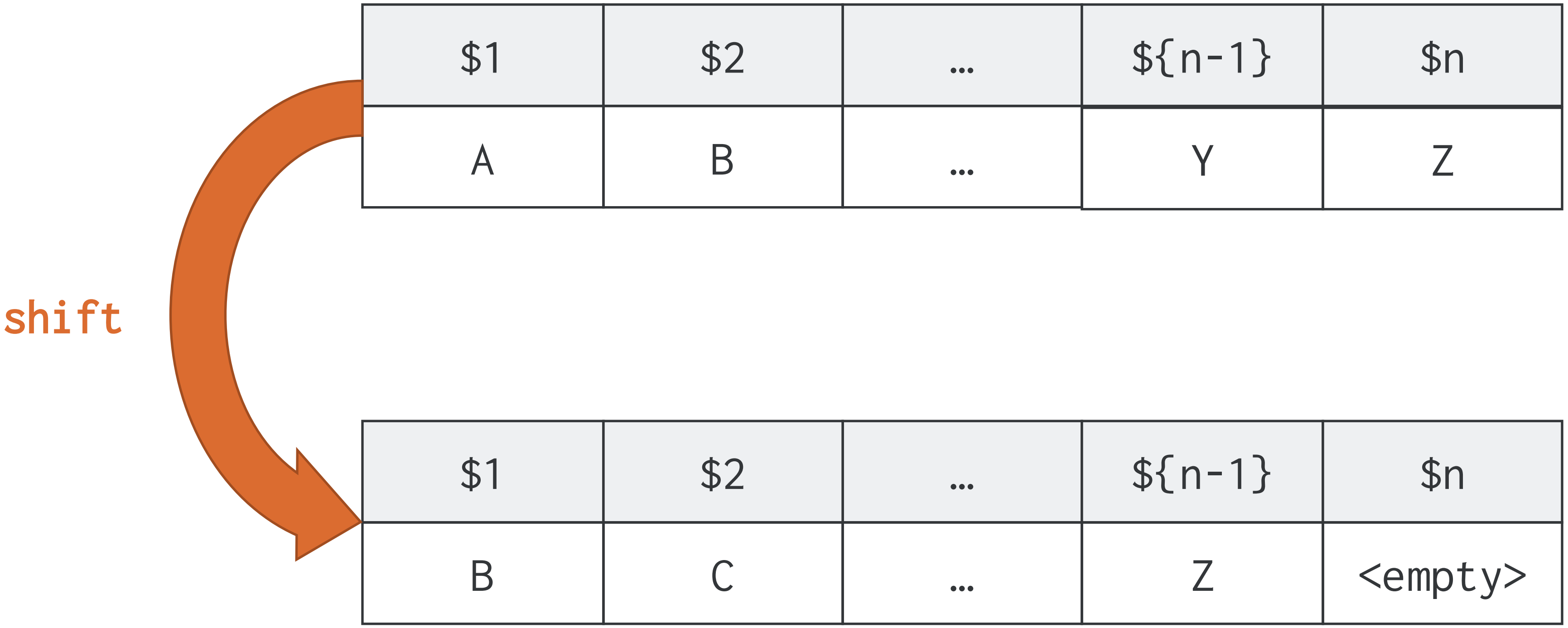
```
#!/bin/bash                                                      # script08.sh

echo "first argument in list: $1"
echo "last argument in list: ${@: -1}"

count=1
while [[ $# -gt 0 ]]; do
  echo "Nr of arguments left = $#"
  echo "Argument $count = $1"
  count=$((count + 1))
  shift
done
```

Command line arguments

- Each time `shift` is executed, the value of $# is reduced by one, the value of $2 is moved to $1, the value of $3 is moved to $2, etc.



$1	$2	…	${n-1}	$n
A	B	…	Y	Z

$1	$2	…	${n-1}	$n
B	C	…	Z	<empty>

case

```
case word in
  pattern1) commands1 ;;
  pattern2) commands2 ;;
  ...
  patternn) commands_n ;;
esac
```

```
#!/bin/bash                                                     # script11.sh

read -p "enter word > "
case $REPLY in
  [[:alpha:]])       echo "single alphabetic character." ;;
  [ABC][0-9])        echo "A, B, or C followed by digit." ;;
  ???)               echo "is three characters long." ;;
  *.txt)             echo "is a word ending in '.txt'" ;;
  *)                 echo "is something else." ;;
esac
```

break and continue

```
#!/bin/bash                                                   # script12.sh
while echo -n "enter number: "; read NUM
do
  if [ $NUM -eq $NUM ] 2>/dev/null; then
    :
  else
    echo " $NUM is not a number"
    continue
  fi
  if [[ $(( $NUM % 2 )) -eq 0 ]]; then
    echo " $NUM is an even number"
    continue
  fi
  echo " $NUM is an odd number"
  break
done
```

no-op

Functions

```
#!/bin/bash                                                      # script03.sh
function func {              # shell function
  echo "use func for $1"
  return
}

echo "step 1"
func "step 2"
echo "step 3"
```

- Useful for sequence of commands that is often repeated
- Functions can also take arguments
- Example using functions defined in another file: `script03a.sh` and `script03b.sh`

Debugging

- How to detect and handle errors in a script?
- A finished command has an **exit status**. Convention:
  - success → exit status 0
  - error → exit status non-zero (status values differ for each command)
- The special variable “?” holds the last process’ exit status:

```
$ ls existing_file
existing_file
$ echo $?
0
$ ls missing
ls: cannot access missing: No such file or directory
$ echo $?
2
$ echo $?
0
```

Debugging

- Putting `set -x` at the beginning of your script will print out all steps as they are executed.

  It's a way to follow what's going on if your script behaves unexpectedly.

- Likewise, `set -e -u` will stop the script if any command fails or when an empty variable is used.

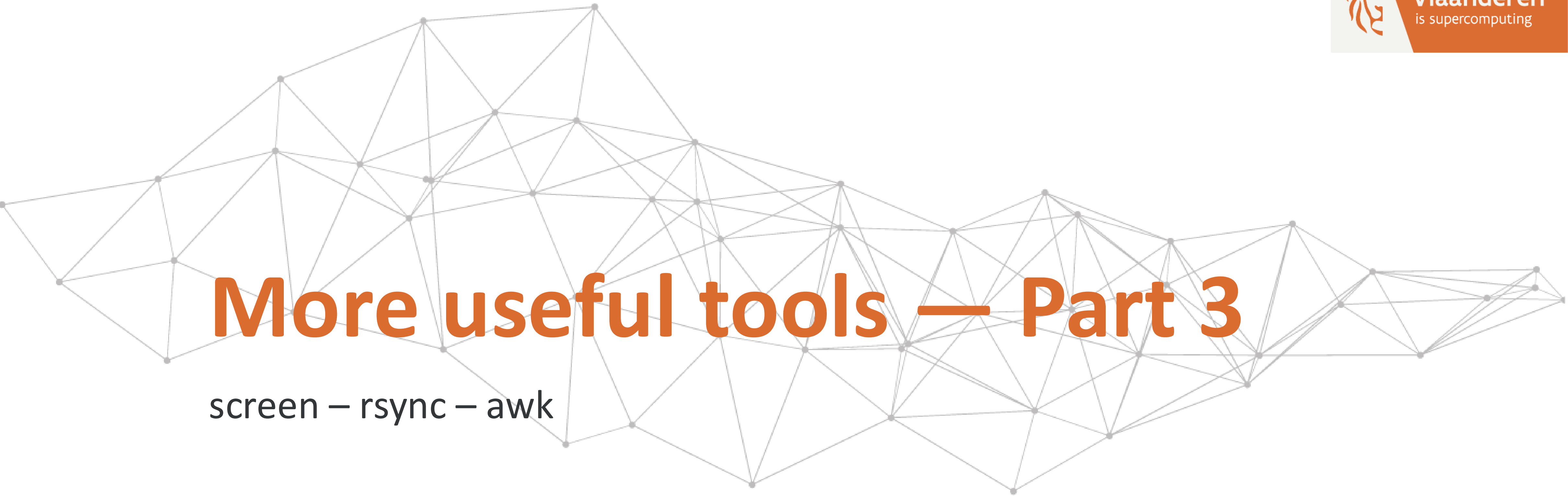
- More info on debugging:

  www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_02_03.html

- More info on bash options such as `-x`:

  www.tldp.org/LDP/abs/html/abs-guide.html#OPTIONS

Vlaanderen
is supercomputing



More useful tools — Part 3

screen – rsync – awk

Screen

- Use multiple shell windows from a single SSH session.
- Keep a shell active even through network disruptions.
- Disconnect and re-connect to shell sessions from multiple locations.
- Run a long running process without maintaining an active shell.
- Similar applications:
  - `tmux`
  - `byobu`
  - …

Screen

➢ `$ screen`	start a new "screen”
➢ `$ screen -S screen1`	start a "screen" named screen1
➢ `$ screen -ls`	overview of (in)active screens
➢ `$ screen -r`	reattach after detach or connection drop
➢ `$ screen -x`	attach to a non-detached screen session (multi display mode)

➢ Key combinations:

○ `Ctrl + a, d`	detach
○ `Ctrl + a, c`	open a new window
○ `Ctrl + a, n`	goto next window
○ `Ctrl + a, p`	goto previous window

➢ Do not forget the host on which you launched the screen command ;-)

rsync

- Efficient transfer and synchronization of files and directories over network
- Like scp or rcp, but more options
- Typical usage: copy from source to destination, useful for backups/large transfers

```
$ rsync [options] <source> <destination>
```

- source or destination may be remote (but not both)
- Some notable options (combine with an alias to avoid retyping):

-a	archive mode; keeps links, permissions, … (implies -r)
-r	recurse into directories
-v	verbose mode
-z	compress data during transfer
-H	preserve hard links
--progress	show transfer progress

rsync: files

`$ rsync file user@server`	copies `file` locally (!)
`$ rsync file user@server:`	copies `file` to `~` on server (mind the `:`)
`$ rsync file user@server:file2`	copies `file` to `~` on server, renamed `file2`
`$ rsync file user@server:test/`	copies `file` to `~/test` on server (mind the `/`)
	remote dir `~/test/` created if non-existant
`$ rsync file user@server:/home/user/`	copies `file` to `/home/user/` on server
`$ rsync user@server:file ~`	copies remote `file` to local `~`
`$ rsync user@server:dir/file ~`	copies remote `~/dir/file` to local `~`

rsync: directories

`$ rsync user@server:dir ~`	**skips directory**, so does nothing
`$ rsync user@server:dir/ ~`	**skips directory**, so also does nothing
`$ rsync -r user@server:dir ~`	copy remote `dir` to local home dir (creates `~/dir`)
`$ rsync -r user@server:dir/ ~`	copies **content** of remote `dir` to local home dir `~`
`$ rsync -r dir user@server:`	copy local `dir` to remote home dir (creates remote `~/dir` if non-existant)
`$ rsync -r dir user@server:dir2`	copies local `dir` to remote `dir2` (result: `user@server:dir2/dir`)
`$ rsync -r dir/ user@server:dir2`	copies **content** of local `dir` to remote `dir2` (result: `user@server:dir2/`)
`$ rsync -r user@server:dir dir2`	copies remote `dir` to local `dir2` (result: `dir2/dir`)
`$ rsync -r user@server:dir/ dir2`	copies **content** of remote `dir` to local `dir2` (result: `dir2/dir`)

awk

- Textual data processing
  - Processing of a stream of text (standard input or set of files)
  - Typical usage: list patterns and desired actions for that pattern

    ```
    awk 'pattern1 { action1 } pattern2 { action2 } …' files
    ```

    - By default, each line of a file is a “record”
      - Several “fields” per record, separated by whitespace
    - awk loops over all records, for each record:
      - evaluates each pattern
      - if pattern is true (non-zero result): execute associated action
  - Powerful, but can become as complicated as you want
  - https://www.gnu.org/software/gawk/manual/

awk: patterns

- Pattern elements

`BEGIN`	beginning of file
`END`	end of file
`1`	`always`
`0`	never
`<`*`empty`*`>`	always

  - https://www.gnu.org/software/gawk/manual/html_node/Pattern-Overview.html

- Expressions

`<value1>` `==` *`<value2>`*	comparison (similar for `!=`,`<`,`>`,`<=`,`>=`)
`<`*`value`*`> ~ /<`*`regex`*`>/`	`value` matches with `regex` (similarly `!~` for absence of match)

  - Logical expressions like Bash tests: AND (`&&`), OR (`||`) and NOT (`!`)
  - https://www.gnu.org/software/gawk/manual/html_node/Expressions.html

awk: actions

- Grouped between braces `{}`
- Some building blocks:
  - `$`*`n`* value of *n*-th field in current record
  - `print` prints to stdout
  - `printf` prints to stdout with extra formatting options
  - `next` stops further processing of current record and continues with next record
  - `+`, `-`, `*`, `/`, `%`, `**`, `++`, `--` arithmetic and increment operations
  - *`<var>`* `= …, <var> += …, <var> -= …` variable assignment
- Separate action statements are separated by semicolon (`;`) or line-break
- https://www.gnu.org/software/gawk/manual/html_node/Statements.html
- https://www.gnu.org/software/gawk/manual/html_node/Action-Overview.html

awk: examples

- ➢ Print every record/line of the file (both are equivalent):

```
$ awk '1 {print}' squeue.txt

$ awk ' {print}' squeue.txt
```

- ➢ Print only jobs where 5th field (STATE) in each record equals “Running”:

```
$ awk '$5 == "R" {print}' squeue.txt
```

- ➢ Print 4th field (USER):

```
$ awk '$5 == "R" {print $4}' squeue.txt
```

awk: variables

- Similar to Bash variables
  - Built-in
    - `FS` field separator (whitespace by default)
    - `OFS` output field separator
    - `RS` record separator (whitespace by default)
    - `ORS` output record separator
    - `NR` number of records processed (total number of records in END block)
    - `NF` number of fields in a record
    - Can be overwritten (in any action)
  - User-defined
    - Assigned in action (`<var> = <value>`)
    - Scalar: numeric (`1234`, `6.02e+23`), string (`"abc"`), regex (`/<regex>/`), …
    - “Associative” arrays: `fib[8] = 21`, `g["earth"] = 9.81`

awk: examples

- Print lines 5-19:

```
$ awk '5 <= NR && NR < 20 {print}' squeue.txt
```

- Keep number of running jobs for user id076 :

```
$ awk '$5 == "R" && $4 == "id076" {nrj +=1}
      END {print nrj}' squeue.txt
```

- Keep number of running jobs per user, but print only for id076:

```
$ awk '$3 == "R" {nrj[$4]+=1}
      END {print nrj["id076"] }' squeue.txt
```

- Keep number of running jobs per user, and print for all users:

```
$ awk '$3 == "R"  {nrj[$4]+=1}
      END { for (u in nrj) print u, nrj[u] }' squeue.txt
```

awk: built-in functions and utilities

- GNU awk has several built-in functions, ranging from `sin`, `cos`, `tan` to internationalization:
  - https://www.gnu.org/software/gawk/manual/html_node/Built_002din.html
- GNU awk also contains several other built-in POSIX utility clones:
  - `cut`
  - `egrep`
  - `id`
  - `sort`
  - `split`
  - `tee`
  - `uniq`
  - `wc`
  - These are not identical clones of the POSIX utilities, but similar in use.
  - https://www.gnu.org/software/gawk/manual/html_node/Clones.html

awk: built-in utilities examples

- Calls to these utilities must be surrounded by double quotes.

- Example: get the number of running jobs per user, in sorted order

```
$ awk '$3 == "R" {nrj[$4]+=1}
       BEGIN {print "UID" "#jobs"}          print a header
       END { for (uid in nrj) print uid, nrj[uid] | "sort" }' squeue.txt
```

- Example: get the number of running jobs for users, but print only for `id`*x3y*–form usernames

```
$ awk '$3 == "R" {nrj[$4]+=1}
       BEGIN {print "UID" "#jobs"}
       END { for (uid in nrj) print uid, nrj[uid] | "egrep ^id.3." }' squeue.txt
```

  - Note: in this case `egrep` only filters the data from the `for` loop, so we still get the header.
  - This program counts all running jobs, although it only displays the ones we want.
    What should you modify to only count the number of running jobs for `id.3.` user IDs? How?

awk: output

- You can modify the output separators by setting the corresponding variables.
  - E.g. if your data contains whitespaces, separate fields with commas or colons or vice versa.
- Example: get the running jobs and sum of number of nodes in use (7th field) per user, separate output by colons

```
$ awk '$5 == "R" {nrj[$4]+=1; nrn[$7]+=$7}
       BEGIN {OFS=":"; print "UID", "run", "nodes"}
       END { for (uid in nrj) print uid, nrj[uid], nrn[uid] | "sort " }'
       squeue.txt
```

awk: scripts

- Write long or frequently re-used awk programs in files and use them with awk -f.

- Example: get a comma-separated list of currently running jobs, per user

```
$5 == "R"                                         # get_running_jobs.awk
{
        if ( length(jobs[$4]) == 0 ){      # if jobs still empty
                jobs[$4] = $1              # fill it with value of 1st field
        }else{
                jobs[$4] = (jobs[$4] "," $1) # join strings
        }
}
BEGIN {OFS=":"; print "UID", "running jobs"}
END { for (uid in jobs) print uid, jobs[uid] }
```

```
$ awk -f get_running.awk squeue.txt
```

awk: scripts

- Even better: make the program self-contaned

```
#! /usr/bin/env -S awk -f                         # get_running_jobs_exec.awk

$5 == "R"
{
        if ( length(jobs[$4]) == 0 ){       # if jobs still empty
                jobs[$4] = $1               # fill it with value of 1st field
        } else {
                jobs[$4] = (jobs[$4] "," $1) # join strings
        }
}
BEGIN {OFS=":"; print "UID", "running jobs"}
END { for (uid in jobs) print uid, jobs[uid] }
```

```
$ chmod +x get_running_exec.awk
$ ./get_running_exec.awk squeue.txt
```