Introduction to Linux

Stefan Becuwe, Franky Backeljauw, Kurt Lust, Carl Mensch,

Michele Pugno, Bert Tijskens, Robin Verschoren

Version Spring 2024

Vlaanderen is supercomputing

VLAAMS SUPERCOMPUTER CENTRUM | *Innovative Computing for A Smarter Flanders*

vscentrum.be

Goals of this course

- The VSC clusters, like most HPC clusters worldwide, use **Linux**-based operating systems.
- Basic concepts:
  - Files and the file system
  - Processes, threads
- Using the **command-line**
  - Starting (and stopping!) programs
  - Files and directories: find, read, create, write, move, copy, delete, …
- **Scripts**: store a series of commands in a file, so we can (re-)use them later.

Linux-like environments

- Microsoft Windows
  - Microsoft Subsystem for Linux (WSL) (≥ Windows 10): https://docs.microsoft.com/en-us/windows/wsl/
  - MobaXterm: https://mobaxterm.mobatek.net
- macOS
  - Terminal app (or iTerm2)
  - For identical commands: install GNU tools using Homebrew (brew.sh) or MacPorts (macports.org)
    e.g. `brew install coreutils findutils gnu-tar gnu-sed grep wget`
    and use (GNU) `gsed` instead of (BSD) `sed`
- or in your browser:
  - www.tutorialspoint.com/unix_terminal_online.php

What is the shell?

- A program
- “Command line interpreter”: waits for input and performs requested tasks.
- Input language is a **scripting language** (variables, iterations, …)
- Provides access to 100s of **commands**.
- Different shell programs exist.
  - On most Linux systems, the default shell is called **Bash** (**B**ourne **A**gain **Sh**ell).
  - On macOS, the default shell is **zsh**. However, **bash** should still be installed by default.

Command line basics

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

  e.g. `$ ls -l /etc/host`↹
- Case sensitive (`myfile` vs `myFile`)
  - macOS is case insensitive by default.
- Spaces separate parts of command (`myfile` vs `my file`)
- Edit command: use arrow keys **`Left`** ← and **`Right`** →
- Command history: use arrow keys **`Up`** ↑ and **`Down`** ↓
- Copy/Paste: **`Ctrl + Shift + c`** and **`Ctrl + Shift + v`**

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:

```
$ command
```

- Arguments: tell a command what to do and how

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

- Options: arguments starting with a dash `-` modify a command's behavior

```
$ command -option
$ command --long-option
```

- In general:

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

Command line arguments

- Interpreted by the command itself → usage depends on the command
  - Order of arguments often doesn't matter.
  - Convention: options first, non-option arguments last.
  - Short options can be combined, i.e. `$ date -R -u` = `$ date -Ru`
  - For some commands, strict rules apply, e.g. `find`
- Meaning of arguments
  - Non-option argument: often a file name

```
$ less myfile.txt
```

  - But not always:

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

Types of commands

- A command can be
  - Any **program** (or script) on the system
  - A **built-in** shell command
  - An **alias**: user-defined shorthand for more complex command.
  - A (user-defined) shell **function**.

Types of commands: examples

- Use the command `type` to learn about other commands, e.g.
  - `$ type date`
  - `$ type cd`
  - `$ type type`
  - `$ type ls`
- Use the command `which` to see where a binary or builtin is located, e.g.
  - `$ which cd`
  - `$ which date`
- Use the command `alias` to see defined shortcuts, e.g.
  - `$ alias` (list all aliases)
  - `$ alias ls` (if `ls` is an alias)

Getting help

- Documentation is available online at https://www.kernel.org/doc/man-pages/

  or from the command line itself

- For shell built-ins: `help`

```
$ help cd
```

- Manual pages for commands:

```
$ man ls
```

- More elaborate: info manuals (also available on the web):

```
$ info ls
$ info info
```

- Ask a command about its use with the `--help` or `-h` option (if available):

Reading man pages

- Scrolling: ↓ / ↑ or **`j`** / **`k`**
- Search for word: **`/ "word" Enter`**
- Find next occurrence of word: **`n`** , previous: **`N`**
- Help for man page viewer: **`h`**
- Quit man page: **`q`**
- Conventions for describing keys

  `^-<key>` = **`Ctrl + <key>`**

  `C-<key>` = **`Ctrl + <key>`**

  `M-<key>` = **`Alt + <key>`**

Searching man pages

- **Example:** searching in `$ man bash`
- `/Commands for Moving` **Enter** Searches for words “Commands for Moving”

Key combination	Action
Ctrl + a (beginning-of-line)	Move to the start of current line.
Ctrl + e (end-of-line)	Move to the end of the line.
Ctrl + f (forward-char)	Move forward a character.
Ctrl + b (backward-char)	Move backward a character.
Alt + f (forward-word)	Move forward to the end of the next word.
Alt + b (backward-word)	Move back to the start of current or previous word.
Ctrl + l (clear-screen)	Clear the screen, leaving the current line at the top of the screen.

The filesystem

- All data is stored on the file system, a tree of **directories** and **files**.
  ("directory": file containing a list of other files)
- A **file name** describes a location in the file system, e.g.
  - `/home/student/introlinux/scripts`
  - `/tmp/myfile.txt`
  - `/`
- Directories are separated by `/` (Windows uses ****).
- A single `/` is the "root" directory (compare to C:\ on Windows).
- Some commonly used directories (see "Filesystem Hierarchy Standard"):
  - `/home/<username>`: "home" directory, user's personal disk space.
  - `/tmp`: temporary files
  - `/bin`: essential programs

The filesystem

- File name starting with `/` is an **absolute** file name.
- Otherwise: **relative** file name: a path starting from the current **working directory**.
- `$ pwd` prints the current working directory (at login usually your home directory).
- **Example:** relative path from directory `/home/student`:

  `introlinux` = `/home/student/introlinux`
  `introlinux/scripts` = `/home/student/introlinux/scripts`

- Use `..` to refer to a parent directory. Starting from `/home/student`:

  `..` = `/home`
  `../..` = `/`
  `../anotheruser` = `/home/anotheruser`

The filesystem

- Use `$ cd <directory>` to change the current directory, e.g.

```
$ 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.

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

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

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

Example

- Try out the following commands

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

Hands-on

- **Exercise:** Download and extract the following archive files:
  - https://calcua.uantwerpen.be/courses/introduction-to-linux/input.zip
  - https://calcua.uantwerpen.be/courses/introduction-to-linux/scripts.zip
- **Version 1:**
  - Point your web browser to https://calcua.uantwerpen.be/courses/introduction-to-linux
  - Download both files.
  - Extract their contents using the file manager.
  - List the files using the `cd` and `ls` commands.
- **Version 2:** Use only the following commands (and copy/paste):
  - `wget https://[...].zip` to download the files.
  - `unzip` to extract their contents.
  - `cd` and `ls` to list the extracted files.

Wildcards

- Many commands use lists of file names, e.g.

```
$ zip textfiles.zip file1.txt file2.txt ... file100.txt
```

- Wildcards help us generate such lists. Example:

```
$ zip textfiles.zip file*.txt
```

- Bash replaces `file*.txt` by the list of matching files.
- `*` matches everything → `file*.txt` matches any filename which
  - starts with `file`
  - ends with `.txt`

Wildcard types

- `*` 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`

Character classes

- Predefined character classes exist for `[]` wildcard expressions:
- `[[:class:]]` matches any single character of the class
- Example classes:
  - `[:alpha:]` Alphabetic character
  - `[:alnum:]` Alphanumeric character
  - `[:digit:]` Digit 0-9
  - `[:lower:]` Lower-case letter
  - `[:upper:]` Upper-case letter
  - `[:space:]` Whitespace (space, tab, newline, ...)
  - `[:punct:]` Punctuation
- `file[[:lower:]].txt` matches `filea.txt fileb.txt ... filez.txt`
- For a complete list, look up “POSIX character classes”

Hands-on: wildcards

- **Exercise:** which names match the following patterns?
  - `[abcdefghijk]*.pdf`
  - `[[:digit:]]*.doc`
  - `backup.[0-9][0-9][123]`
  - `file[[:lower:]123].txt`

1. Think of an example file name for the pattern.

   `file[[:upper:]].txt` `fileA.txt` matches?

2. Create the file using the command `touch`:

   `$ touch fileA.txt` Creates empty file called `fileA.txt`.

3. Try it out:

   ```
   $ ls file[[:upper:]].txt
   fileA.txt
   ```

   The file appears, success!

Manipulating files and directories

- **Warning:** when deleting/copying/moving files at the command line, there is **no "recycle bin" or undo!**
- `mkdir`: create directories

```
$ mkdir dir1 dir2 dir3
```

- `mkdir -p:` create nested directories

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

- `rmdir`: remove empty directories

```
$ rmdir dir1 dir2 dir3
```

Move, copy, remove

- `$ mv source target`: move (rename) files and directories:
  - If `target` is existing file: overwrite it.
  - If `target` is existing directory: move inside it.
  - `$ mv src1 src2 ... srcn targetdir` move a list of items into existing directory `targetdir`
- `$ cp source target`: copy files and directories:
  - Same rules as `mv`, except:

```
$ cp srcdir target
cp: omitting directory ‘srcdir’
```

  - `$ cp -r srcdir target`: recursive, copy directories + contents
- `$ rm file1 file2 ... filen`: remove (erase) file(-s)
  - `$ rm -r mydir` : recursive, delete directories + contents

With wildcards

- Together with wildcards: very efficient

```
$ mv *.jpeg Photos
```

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

  → typing mistake can be dangerous

- Safety for `cp`, `mv` and `rm`:

  option `-i` or `--interactive` – ask for confirmation before overwriting or deleting.

- Example:

```
$ rm -i file*.txt
rm: remove regular file 'file1.txt'? → y <Enter> to confirm
```

Ownership & permissions

- Every user has a unique **id** / **name** and belongs to one or more **groups**.
- To see your id 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
  - user
  - group
  - others (= all other users who are not a member of the file's group)

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

```
$ 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
```

- `rwxr-xr-x`: three kinds of permissions for “user,” “group” and “others”
  - read: read file’s contents
  - write: modify file’s contents
  - execute: run file as a program
- For directories
  - `x`: enter the directory and access contents
  - `r`: list directory contents
  - `w`: create, delete, rename files (also needs `x`)

Setting permissions

- `$ chmod` can change the permissions for files or directories.
- Set `rwx` permissions using `chmod =`
  - `$ chmod =rw file.txt` give all users `rw-` permissions.
  - `$ chmod u=rw,g=r,o= file.txt` set permissions for user, group and others
- Or use the numbers: 0=none, 1=x, 2=w, 3=wx, 4=r, 5=rx, 6=rw, 7=rwx
  - `$ chmod 640 file.txt`
- Add/remove permissions using `chmod +` or `chmod -`
  - `$ chmod +w file.txt` add `w` permission for all users
  - `$ chmod ug+x,o-r file.txt`
- `-R` Recursive: change permissions on a directory and all its contents:
  - `$ chmod -R go-xr my_private_dir`

Processes and threads

- A **process** is a running instance of a program.
- Several instances of the same program can run at the same time.
- Each process has a unique identifier or **PID**.
- One process can start other processes, **child** processes.
- A process can not access other processes' **memory**.
- Each process consists of one or more **threads**.
  - Threads **share** access to the process' memory.
  - Different threads can run in **parallel** on different CPU cores.
- To run a calculation on **multiple CPU cores**, we can use
  - multiple processes ("distributed memory parallelism")
  - multiple threads in one process ("shared memory")

Looking at processes

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

```
$ ps
  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 <username>` show processes of the given user
  - `$ ps -T` show the threads of each process
- The commands `top` or `htop` show processes together with CPU and memory usage in real time.

Managing processes

- Terminate processes
- Stop and resume processes
- Run processes in background
- 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**.

Managing 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

Terminate a process

- 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.

Threads

- The example program `omp_pi` can run with multiple threads:

```
$ OMP_NUM_THREADS=4 ./omp_pi
```

- `$ ps -T` displays each process' threads:

```
$ ps -T
PID SPID TTY TIME CMD
17058 17058 pts/3 00:00:32 omp_pi
17058 17059 pts/3 00:00:32 omp_pi
17058 17060 pts/3 00:00:32 omp_pi
17058 17061 pts/3 00:00:33 omp_pi
...
```

- `$ top -H` displays CPU usage for each thread in real time.
- When running `top`, hit **f** to display other info (e.g. CPU number).

Streams, redirection, pipelines

- Output of commands is shown in the terminal; some commands read input from the keyboard.

  This is managed using **file descriptors**:

  - Normal output is written to **standard output** (stdout), fd 1.
  - Warnings and errors are written to **standard error** (stderr), fd 2.
  - Commands can read from **standard input** (stdin), fd 0.

- By default, “stdout” and “stderr” file descriptors are attached to terminal,

  “stdin” is read from the keyboard.

- We can redirect output and input:
  - write output to a file
  - send output from one command to input of another command
  - read stdin from a file

Output redirection

- The operator **`i>`** redirects file descriptor **`i`** to a file.

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

  - File `ls-output.txt` is created, contains the command's output.
  - stderr still shown in terminal
  - Inspect the file with `$ less ls-output.txt`

- Redirect stderr:

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

- Multiple redirections for one command:

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

- `>` without fd number redirects stdout (stderr still shown in terminal):

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

Output redirection

- `/dev/null` is a special “file” that discards everything written to it.

  E.g. to hide a program’s output: `$ ./omp_pi > /dev/null`

- Note: **>** creates a new file
  - Existing file with same name is replaced (!)
  - If command produces no output: empty file.
- **1>>** and **2>>** append stdout or stderr to the end of a file, without erasing previous content.

```
$ date >> diary.txt
$ echo "Dear diary, today ..." >> diary.txt
```

- **i>&j**: attach file descriptor **i** to the same file as descriptor **j**

```
$ ls *.txt *.jpg > ls-all1.txt 2>&1      write stdout and stderr to the same file
$ ls *.txt *.jpg 2>&1 > ls-all2.txt      only stdout written to file
```

Input redirection

- Standard input (fd 0) is read from the keyboard. Example: try `$ bc`.
- The input redirection operator `< filename` opens a file, from which the program now reads standard input:

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

- Most commands also accept a file name as an argument. e.g., these commands have the same result:

```
$ less homework.txt
$ less < homework.txt
```

- Redirecting input and output:

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

Pipelines

- We can chain 2 or more commands with the `|` (“pipe”) operator:

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

  stdout from `command1` is directly sent to stdin of `command2`, etc.

- Commands run in parallel, each command processes input as it becomes available.
- Example: scrolling through the list of all processes with `$ less`.

  ```
  $ ps aux | less
  ```

- Create complex commands from simple building blocks.
- Note: to pipe stderr from a command, redirect it to stdout:

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

Frequently used commands

Typical commands for pipelines:

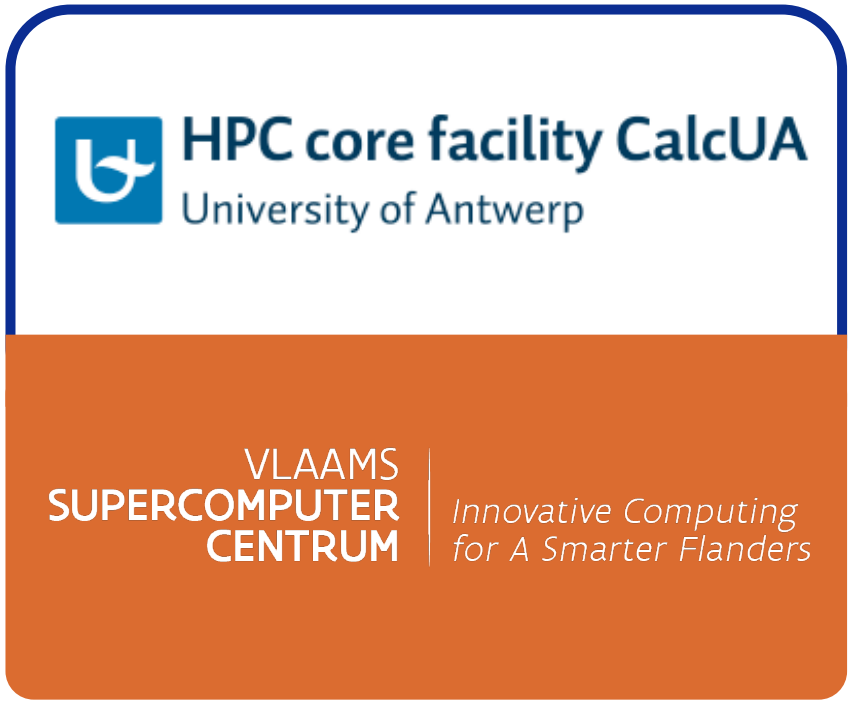
- `wc` print the number of lines, words and bytes of input
- `grep` filter lines which match a given search pattern
- `head` / `tail` print first/last lines of input
- `sort` sort input alphabetically
- `uniq` report or leave out repeated lines
- `sed` transform input (pattern replacement and more)
- `awk` text pattern scanning and processing
- Find more commands at https://www.gnu.org/software/coreutils/manual/

Hands-on: pipelines

- Build pipelines with `ps`, `head` / `tail`, `grep` and `wc` to find out
  - What is the name of the first process (PID 1)?
  - How many processes are **not** owned by user root?

- Using the file `chemistry.txt` in the `input` folder, and the commands `wc`, `grep`, `sort`, `tail` and `uniq`, answer the following:
  - How many courses are there?
  - Which courses are taught by Wouter Herrebout in the first semester?
  - Which are, in alphabetical order, the last 5 course codes starting with **1001WET**?
    - alphabetically sorted by course code, or
    - alphabetically sorted by course title
  - Which course is listed twice?

Inside the shell



Shell: expansion

- "Each time you type a command line and press the **Enter** key, bash performs several processes upon the text before it carries out your command. The process that makes this happen is called expansion."

1. Brace expansion (e.g. `{my,your}file` -> `myfile yourfile`)
2. "~" expansion (e.g. `cd ~` → `cd /home/username`)
3. Variable expansion (e.g. `$HOME` → `/home/username`)
4. Arithmetic expansion (e.g. `$((2 + 2))` → 4)
5. Command substitution (e.g. `$(pwd)` → `/path/to/this/dir`)
6. Word splitting (e.g. `grep 1e semester` vs `grep '1e semester'`)
7. Filename expansion (e.g. `ls file.*` → `ls file.txt file.jpg`)
8. Quote removal

Shell: brace expansion

- `{}` brace expansion
  - List:

```
$ echo Front-{A,B,C}-Back
```

  - Sequences:

```
$ echo {Z..A}
$ mkdir {07..09}-0{1..9} {07..09}-{10..12}
```

  - Nested:

```
$ echo a{A{1,2},B{3,4}}b
```

Shell: “~” expansion

- Tilde (~) expansion:
  - `$ echo ~`

    your own home directory
  - `$ echo ~user2`

    user2’s home directory

Shell: filename expansion

- Filename expansion: wildcards are expanded into matching file names
  - `$ echo *`

    `*` is expanded (non-hidden files in current directory) before `echo` is executed.
  - `$ echo ~/.[a-z]*`

Shell: arithmetic expansion

- Arithmetic expansion: `$((expression))` → result of expression

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

  - arithmetic expression (*only integers in bash!*)
  - operators:

    `+` addition
    `-` subtraction
    `*` multiplication
    `/` integer division
    `%` remainder
    `**` exponentiation

  - single parentheses may be used to group multiple subexpressions:

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

Shell: variable 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: command substitution

- Command substitution: expand `$(command)` to output of command

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

equivalent – but old-fashioned:

```
$ echo I see `ls -A | wc -l` files and subdirs
```

Shell: escapes & quotes

- `$ echo The total is $100.00 # ?!`
- Use “escape” character **** for literal use of special characters (**$**, ****, **`**, **{**, **}**, **(**, **)**, *******, **␣**)

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

- Text inside double quotes **""**: special characters lose their meaning, EXCEPT **$**, **** and **`**

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

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

Shell: escapes & quotes

- No expansion at all inside single quotes, compare:

```
$ 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'
```

- Word splitting: words separated by space become separate arguments

```
$ ls my directory
$ ls 'my directory'
```

- Quote removal: after all expansions, but before executing the command, quotes are removed.

```
$ echo "hello world"
```

  - unless you escape or quote the quotes...

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

Hands on: find

- The command `find` can search files on your file system based on various criteria

  (see reference, or `$ man find ...`).

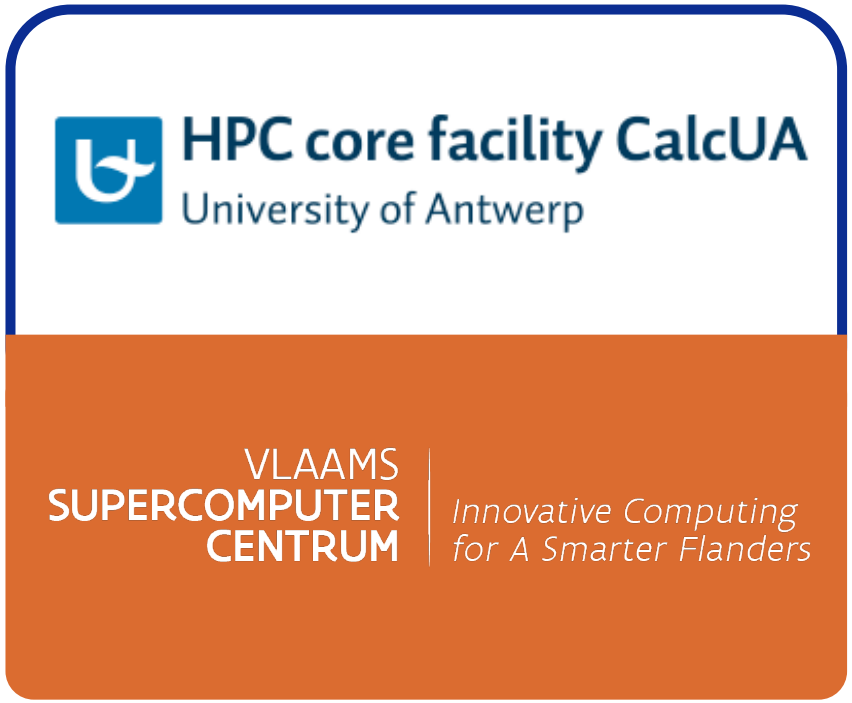
- Build a pipeline with `find` to count the total number of files and directories in your home directory (and its subdirectories).

- Show the result using `echo`, i.e. print a message like

  `/home/student contains <x> files and directories.`

  - How to count regular files and directories separately?
  - How to find a file with a specific name?

The environment



Environment variables

- Recall: the shell has variables
  - set value for variable `myvar`:

    `$ myvar=some_value` no spaces around '=', no spaces in `some_value` unless using quotes
  - get `myvar`'s value ("variable expansion"):

    `$ echo $myvar`
  - "Plain" variables: only exist in the shell itself

    `$ set` display all variables
  - **Environment** variables are special: passed on to processes started from the shell.

    `$ export myvar` make `myvar` an environment variable

    `$ printenv` display environment variables
  - Environment variables are another way to influence the behaviour of programs (e.g. **`OMP_NUM_THREADS`**).

Environment variables

- `EDITOR` The name of the program to be used for text editing.
- `SHELL` The name of your shell program.
- `HOME` The pathname of your home directory.
- `LANG` Defines the character set and collation order of your language.
- `PWD` The current working directory.
- `OLDPWD` The previous working directory.
- `PATH` A colon-separated list of directories that are searched when you enter the name of an executable program.
- `PS1` Prompt String 1. This defines the contents of your shell prompt.
- `USER` Your username.
- `TMPDIR` Directory for temporary files

Environment

- Example: access environment variables from a Python script:

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

Environment

- Persistent settings for your environment:
  - Applied once at login:
    - `/etc/profile` (system wide, for all users)
    - `~/.bash_profile`
    - `~/.bash_login`
    - `~/.profile`
  - Applied every time you start a shell:
    - `~/.bashrc`

      see also bash manual page under “INVOCATION”

      You can also define your own aliases and functions here.

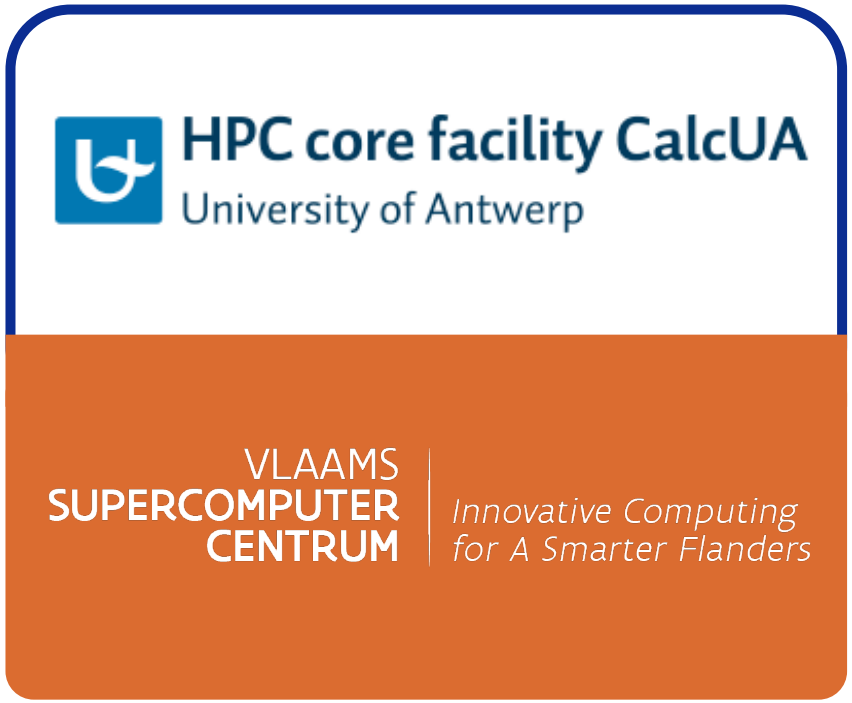
alias

- Substitute a string for the first word of 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
```

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

Writing shell scripts



Shell scripts

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

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

```
$ bash myscript.sh
```

- Commands are executed one after the other, just as if you entered them manually
- Commands are separated by new lines, or by semicolon ‘;’

Editing text

- Editors available on (almost) any Linux system, run inside terminal, syntax highlighting included
  - **nano** – simple editor
    - open ("read") = **Ctrl + r**
    - save ("write out") = **Ctrl + o**
    - exit = **Ctrl + x**
  - **vi** – the default Unix editor. Takes some practice.
    - vim: "vi improved", run vimtutor for a quick tutorial or look for a "cheat sheet"
  - **emacs** – more advanced editor
- Others
  - Visual Studio Code – versatile, portable (https://code.visualstudio.com/)
  - gedit – GNOME text editor
  - notepad++ – available for Windows (https://notepad-plus-plus.org)
  - TextEdit – comes with macOS (use "plain text" format for scripts)
  - …

Editing text

- **Remark:** line endings are encoded differently under Windows and Unix.
  This might introduce some problems for text files (especially job scripts).
  If you created your script file in a Windows environment, we advise to convert your “Windows style” (“carriage return + line feed”) file into a “Unix style” (“newline”) file in the following way:

```
$ dos2unix filename
```

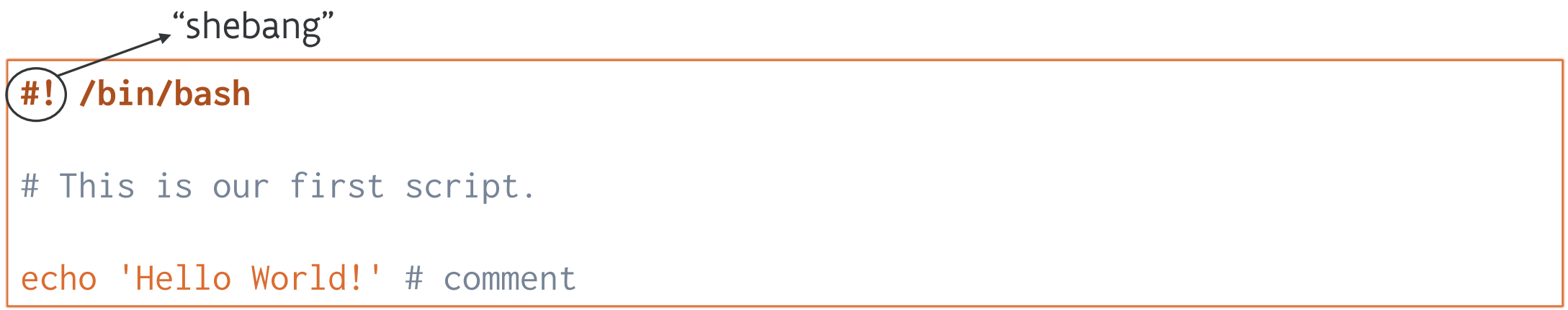
- Beware: `filename` will be overwritten!
  - Can be avoided by using the `-n` option

```
$ dos2unix -n inputfilename outputfilename
```

- A suitable text editor can do this as well

Shell scripts

➢ `$ cat scripts/script01.sh`



`$ 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
  ```

- This works for 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).

- Spaces (between parts) are optional

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

Shell scripts

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

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

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

- Remarks:
  - User variables can not start with a digit: **$1**, **$2**, … are special variables (command line arguments, see later)
  - Setting a variable: without **$**, e.g., `myname=`**`some_value`**
    - Remember: no spaces around `=`
  - Variable expansion: with **$**, e.g., `echo` **`$`**`myname`

Checking commands

- 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
```

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
```

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`

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

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

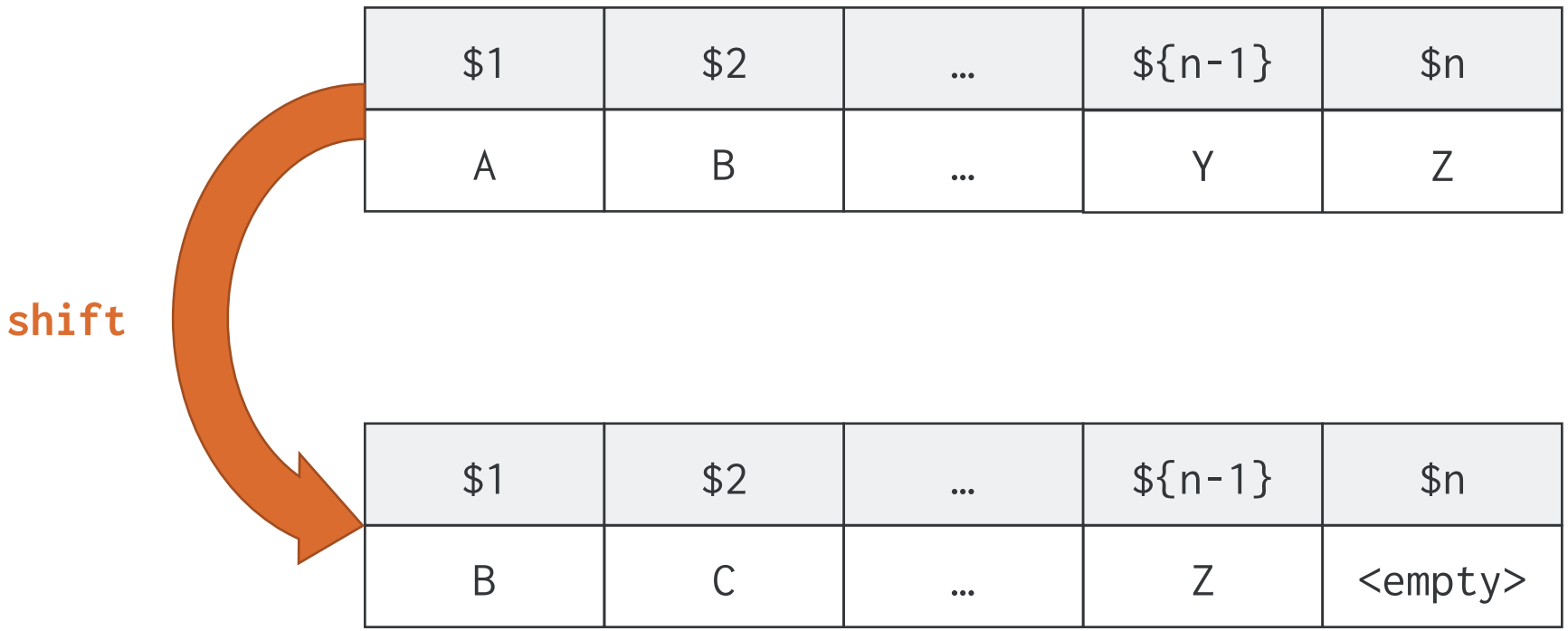
```
#!/bin/bash                                                    # script08.sh

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.



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
```

case

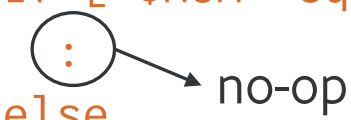
```
case word in
  pattern1) commands1 ;;
  pattern2) commands2 ;;
  ...
  patternn) commands_n ;;
esac
```

```
#!/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
```



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`

Some extra info

- 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:

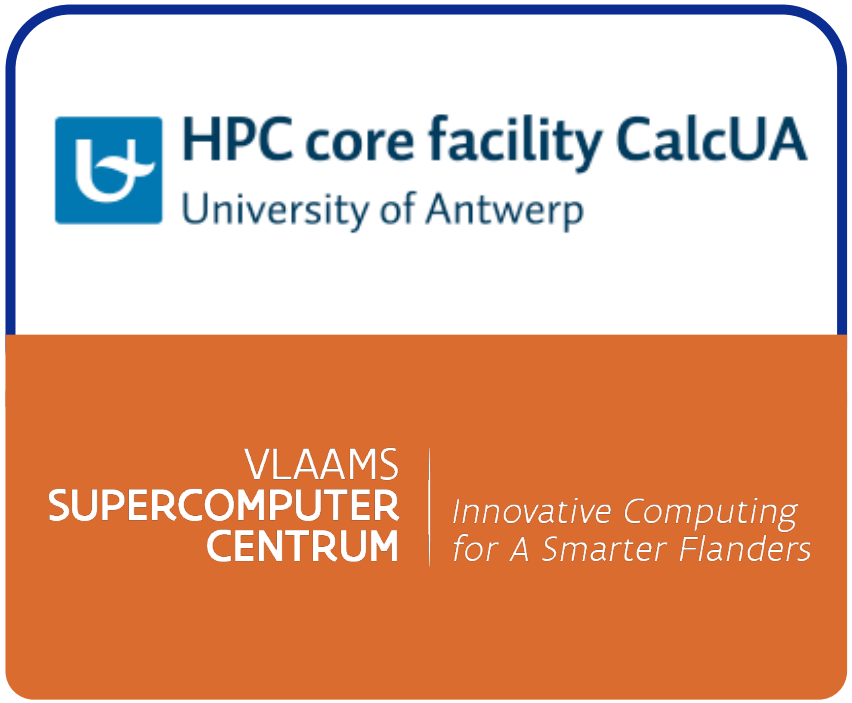
  www.tldp.org/LDP/Bash-Beginners-Guide/html/sect_02_03.html
- More info on bash options such as `-x`:

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

Hands on: scripts

- Modify `script10.sh` in such a way that the longest word in all files is shown, instead of showing the longest word per file.

Regular expressions



Regular expressions

- 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

- Often called “regex”
- Symbolic notation used to match text patterns.
- Similar to wildcards (*, [], ?), but more powerful
- Note: many programs and programming languages support regular expressions:
  - `grep`, `sed`, ...
  - Text editors, e.g. `emacs`
  - Python, Perl, Matlab...
  - ...

  But notation and supported patterns are often slightly different...

Regular expressions

- Create some test files to play with regular expressions

```
$ cd
$ ls /bin > dirlist-bin.txt
$ ls /usr/bin > dirlist-usr-bin.txt
$ ls /sbin > dirlist-sbin.txt
$ ls /usr/sbin > dirlist-usr-sbin.txt


$ touch .zip 1.zip 1zip 22.zip 2zip
```

Metacharacters

- Regex can contain literal characters and digits,

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

- but also “metacharacters” for repetitions, grouping, alternatives, …
- Two notations for metacharacters:
  - basic regular expressions (BRE):

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

  - extended regular expressions (ERE – grep -E):

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

  - To turn a metacharacter into a literal character: add ‘\’

```
$ grep '\$100\.0' accounts.txt
$ grep "\\\$100\\.0" accounts.txt # careful with "" !!!
$ grep '\\\$100\\.0' accounts.txt # careful with '' !!!
```

Metacharacters

- `.` Match any character

  ```
  $ grep -h '.zip' dirlist*.txt
  ```

  vs. `ls *zip; ls *.zip; ls | grep .zip`

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

  ```
  $ grep -h '^zip' dirlist*.txt
  $ grep -h 'zip$' dirlist*.txt
  $ grep -h '^zip$' dirlist*.txt
  ```

Character classes

- `[]` character class

`[bg]zip`	matches `bzip` and `gzip`
`[b-g]zip`	matches `bzip`, `czip`, `dzip`, …, `gzip`
`[^bg]zip`	matches any zip not preceded by `b` or `g`
`[^b-g]zip`	matches any zip not preceded by `b`, …, `g`
`^[A-Z]`	matches any word beginning with an upper case letter
`^[-AZ]`	matches any word beginning with `-`, `A` or `Z`

Repetitions

- `?` Match preceding element zero or one time
- `*` Match preceding element zero or more times
- `+` Match preceding element one or more times
- `{}` (or `\{\}`) 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

Sub-expressions, alternatives

- `()` sub-expression

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

- `|` alternatives

  `bzip|gzip` matches `bzip` and `gzip`

  `(b|g)zip` matches `bzip` and `gzip`, using grouping

  `(b|g|un)zip` matches `bzip`, `gzip` and `unzip`

- Examples:

  `a|b` matches lines containing either `a` or `b`

  `a.*b` matches lines containing both `a` and `b` (in that order)

- If using BRE: write `\( ... \| ... \)`

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

Hands on: regex

- Use the file **`/usr/share/dict/words`**:
  - How many five letter words do you find?
  - 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`**?
- Give a regular expression which recognizes phone numbers of the following form:

  ```
  (03) 265 38 60
  ```

  with or without brackets and spaces. You can find some examples of correct and incorrect numbers in **`phonenr.txt`**

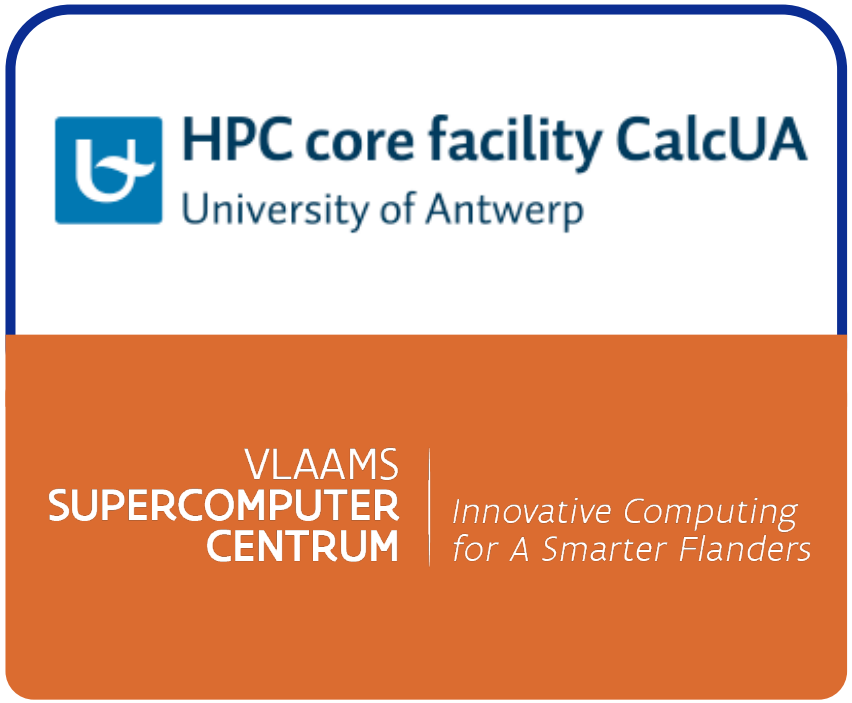
Hands on: squeue

- Use the file **`squeue.txt`**:
  - How many jobs are “Running”?
  - How many jobs per user are “Running”?
  - How many jobs per user are running, sorted in descending order?
  - Give the number of jobs per number of nodes, sorted.
  - Give, per user, the number of Running jobs and the number of nodes in use.

Hands on: courses

- Use the file **`chemistry.txt`**:
  - Create one directory per course code (first column).
  - Create in each directory a file where the file name is the family name of the first teacher (take into account spaces).
  - Put in each file the name of the course.

Other useful programs



diff

- Detect differences between files

`-i`	ignore case
`-w`	ignore all white space
`-y`	output in two columns
`-r`	recursively compare directories

```
$ diff -i file1 file2
$ diff -y file1 file2
$ diff -r dir1 dir2
```

```
$ diff distros.txt distrostab.txt
$ diff -w distros.txt distrostab.txt
```

sed

- Stream editor
  - Editing on a stream of text (standard input or set of files) using basic 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, …

sed

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

`-n`	suppress automatic printing of pattern space
`-i`	edit file in place

- 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`
`p`	print current line
`a`	append text after current line
`d`	delete current line
`<command>I`	case insensitive
`<command>g`	'global' -> act on all occurrences on this line

sed

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

```
$ sed -n '/SUSE/p' distros.txt
```
only matches (equivalent of `grep`)

```
$ sed '/SUSE/p' distros.txt
```
everything + matches

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

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

```
$ sed '/Fedora/d' distros.txt
```
only non-matches (equivalent of `grep -v`)

```
$ sed 's/chemie/scheikunde/I' chemistry.txt
```

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

sed

- Rewrite MM/DD/YYYY in `distrostab.txt` as YYYY-MM-DD.

1. Regular expression for MM/DD/YYYY:

   ```
   [0-9]{2}/[0-9]{2}/[0-9]{4}$
   ```

2. Insert subexpressions:

   ```
   ([0-9]{2})/([0-9]{2})/([0-9]{4})$
   ```

3. Construct replacement: `\n` refers to the *n*-th subexpression, so we want:

   ```
   \3-\1-\2
   ```

sed

4. Result (do not forget the metacharacters), to be entered on one line, without extra spaces

```
$ sed 's/\([0-9]\{2\}\)\/\([0-9]\{2\}\)\/\([0-9]\{4\}\)$/\3-\1-\2/' \
    distrostab.txt
```

or, using another delimiter + instead of /,

```
$ sed 's+\([0-9]\{2\}\)/\([0-9]\{2\}\)/\([0-9]\{4\}\)$+\3-\1-\2+' \
    distrostab.txt
```

or, using ERE instead of BRE,

```
$ sed -E 's+([0-9]{2})/([0-9]{2})/([0-9]{4})+\3-\1-\2+' distrostab.txt
```

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`)

`mlr` (Miller)

- Textual data processing
  - Query, shape or reformat CSV, TSV, JSON, … data files
  - Compact verbs instead of programming language
  - Typical usage: process data files, pretty-print data, format conversion

    ```
    mlr [flags] {verb} [verb-dependent options ...] {zero or more file names}
    ```

  - https://miller.readthedocs.io/
  - https://miller.readthedocs.io/en/latest/10min

mlr (Miller)

```
$ mlr --csv cat example.csv
```
print contents of `example.csv`

```
$ mlr --icsv --ojson cat example.csv
```
convert `example.csv` to JSON format

```
$ mlr --c2j cat example.csv
```
convert `example.csv` to JSON (keystroke-saver flag)

```
$ mlr --icsv --opprint cat example.csv
```
pretty-print `example.csv`

```
$ mlr --csv tail -n 4 example.csv
```
print header and last 4 lines of `example.csv`

```
$ mlr --c2p cut -f user,jobid example.csv
```
pretty-print only fields `user` and `jobid`

```
$ mlr --tsv -N --oxtab filter '$1 == "Fedora"' then cut -f 2,3 then sort -n 2
distrostab.txt
```
filter on Fedora, show only version and date

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-contained

```
#! /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
```

Feedback welcome:

- Please fill out this (short, anonymous) survey:

  https://tiny.cc/calcua-linux-intro-survey

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

Training & Events

The VSC spends the necessary time on 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 give 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

We have created a range of Online Training videos, accessible via our YouTube channel.