VRIJE UNIVERSITEIT BRUSSEL

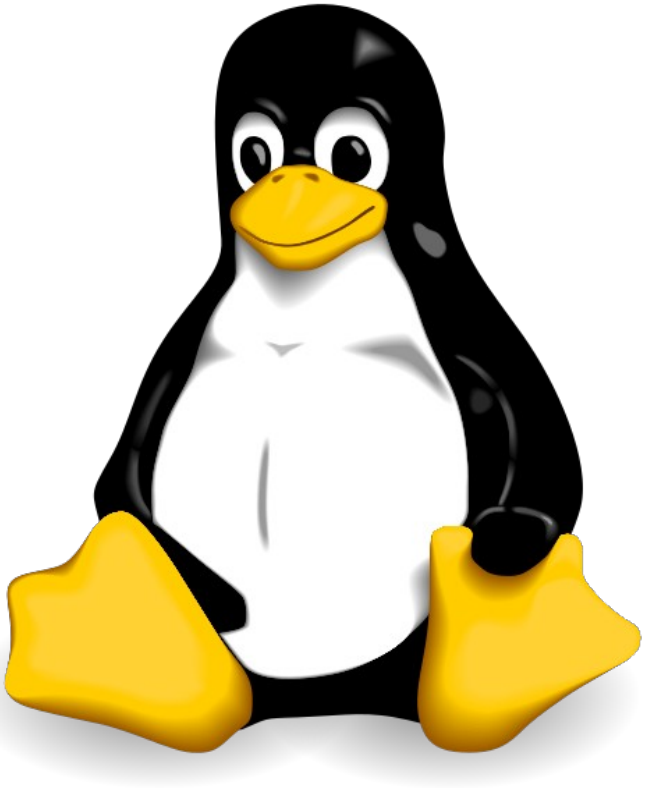
VLAAMS SUPERCOMPUTER CENTRUM

Vlaanderen is supercomputing

Linux Training

VUB – ULB

Stéphane GÉRARD



Fall 2021

Purpose of the course

- NOT a course for system administrators!
- Focus on working with the command-line, emphasis on manipulating and editing files
- Main final objective : being able to write some simple shell scripts

Organization of the course

- Linux walk-through (in common):
  - Why learn Linux?
  - Discovering the Shell
  - Manipulating files
  - Other useful commands
  - Writing simple shell scripts
- Hands-on (individual)

Training material

- Please download a copy of these slides on this webpage :

  https://homepage.iihe.ac.be/~sgerard/

  (slides : Linux_for_beginners.pdf)

PART 1 : Linux walk-through

Why learn Linux?

Linux today

- Linux is everywhere:
  - Most academic computing clusters/supercomputers
  - Many mass storage systems (“Big Data”)
  - Data-centers of major companies like the so-called “GAFAM”
  - Everyday equipment (like the box from your Internet provider)
  - Internet providers infrastructure (webservers,...)
  - Cloud providers
  - Android is based on a branch of the Linux kernel, modified to support hardware found in smart-phones, tablets,…
  - ...

Linux in scientific computing

- Advantages:
  - Portability
  - Modularity
  - Based on open standards
  - Avoid complications and costs due to licenses (especially true on a cluster)
  - Lots of tools available for scientific programming
  - Unix philosophy of tools : only do one thing, but do it well.
  - Easy to interact with development teams
  - True multi-task and multi-user operating system
  - Management of processes
  - Documentation
  - ...

Discovering the Shell

Logging in

- Linux is multi-user. As a practical consequence, you are obliged to identify yourself before using the system: that's the “login” step.
- Logging in to a system typically means to provide a user name (or *login*) and a password.

SSH

- Secure Shell
- To open a session on a distant machine, with a high level of security:
  - Encryption of the communication
  - Identification of the distant machine (to avoid man-in-the-middle attack)

Exercises

- Open your preferred terminal.
- Login to Hydra with ssh:

```
ssh <your_user_name>@hydra.vub.ac.be
```

- Close your connection to Hydra by entering “exit”
- Re-open a new session on hydra
- Is it possible to open several sessions at the same time? Try it!

Shell (1)

- Sometimes called “terminal” or “CLI” (Command Line Interface)
- A **shell** is a command-line interpreter:
  - Executes commands entered by user (*interactive mode*)
  - Executes scripts (*command mode*)
- Different variants: csh, tsh, sh, Bash,…
- Exercise

  To determine which shell you are currently using, type one of the following command:

  ```
  ps $$
  ```

Shell (2)

- Why use command-lines?

  Many things can't be done or are very difficult to do in a graphical environment. Examples:

  - Executing an operation on files matching a given criteria
  - Sending the output of a process in input of another process
  - …
- Working with scripts : task automation, re-usability, modularity,...

Convenient features of Bash

Bash (or *Bourne Shell*) brings a lot of features:

- Process control
- Automatic completion
- Keeps an history of what user has typed
- Copy/paste
- Keyboard shortcuts
- ...

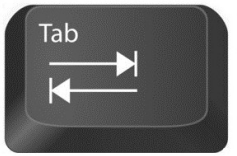
Exercises

- Enter the following informative commands:

```
date
hostname
w
id
whoami
```

  What are these commands telling?

- Auto-completion: type the first 3 letters of the “history” command and type on the TAB key ()



- Enter the “history” command
- Type the letter “l” and type twice on the TAB key
- Try to copy-paste some lines selected in your shell to a text file in your personal computer
- Press on the upwards arrow key (↑) several times → What do you see?

The root user

- You always log in on a system using a given identity (or “login” or “account”).
- On all Unix-like systems, there is a special account: root.
- Root account has ALL the privileges.
- Working with root account is **dangerous**, because:
  - you can execute non-reversible commands that might damage your files/data;
  - you can break your system, making it very difficult to repair if you are not an expert;
  - you can create security holes that can be exploited by “bad guys”.

The prompt

- The prompt is a short message text at the beginning of the command-line.
- On Linux, the prompt is also giving you some basic information, like your user name and the machine to which you are logged in.
- The prompt is ending by a '$' indicating that you are an ordinary user (i.e. not the root user, whose prompt is ending with '#').
- Can be customized by the user

Exercises

- What tells your prompt on Hydra? (Clue: remember the outputs of the “id” and “hostname” commands of the previous exercises)?
- Try to enter the following commands:

```
visudo
useradd toto
cat /var/log/messages
```

What's the common result? Do you have an idea why it fails?

User environment

- Environment is a set of variables that are available to all applications launched in the shell.
- Standard environment variables:
  - PATH = Colon separated list of directories to search for binaries
  - HOME = Current user's home directory
  - USER = Current logged in user's name
  - SHELL = The current shell
  - ...
- To display the actual value of env. variable:

```
echo $<VAR>
```

- Exercises:

  Try the previous command with the variables PATH, HOME, USER and SHELL.

  Verify that the content of the PATH variable is well a colon-separated list of directories.

Manipulating files

Notations

Designation	Notation
slash	/
back-slash	\
angle brackets	< >
a value still to be defined	<value>
caret	^
tilde	~
hash	#
pipe	|
dot	.
colon	:
semi-colon	;
square brackets	[ ]
curly braces	{ }
star	*

Exercises

- Make sure you can find the previous characters on your keyboard by trying to type them in the shell.

To keep in mind when working with command-lines

- Linux is case-sensitive. Example: “MyFile” is not the same as “myfile”.
- In the shell, there is no trash-bin: once a file is deleted, you can't revert. So, be very careful when you want to delete a file!

Demonstration

Case sensitivity

- Type the following commands:

```
touch MyFile
touch myfile
ls
```

=> We've got two distinct files!

How files are organized

- The Linux file-system is unique and everything is attached to the *root* directory that is noted “/”.
- A directory is just a special kind of file: it can be viewed as a file pointing to a list of files.
- The files are organized in a tree.

Typical directory structure

Main directories

- /: is called the “root directory”
- /bin: contains binaries of basic commands
- /sbin: contains binaries of advanced commands (reserved to administrators)
- /etc: contains the configuration files
- /home: that's where users have their personal files
- /var: data generated by the system and its applications (mainly logfiles, databases,...)
- /tmp: contains temporary files, all users have write access

Main file management commands

- **`ls`**: print the list of files in a directory
- **`pwd`**: print the current directory
- **`cd`**: change the current directory
- **`cp`**: copy files
- **`mkdir`**: directory creation
- **`rmdir`**: directory removal
- **`mv`**: move files
- **`rm`**: delete files
- **`chmod`**: change the permissions of a file

Usual syntax of commands

`Command [option...] [argument...]`

Options are generally preceded by a minus character “-”.

Example :

`ls -l /bin`

`ls`: name of the command

`-l`: option

`/bin`: argument

Most commands accept a list of arguments separated by spaces when there is no ambiguity.

Current directory

- Command : **pwd**

  Prints the absolute path of the *current directory* (or *working directory*)

- Each user has a personal directory, called the *home directory* (or *homedir* for short).

- The current directory can be changed with the **cd** command:

  ```
  cd <destination_directory>
  ```

  Example:

  ```
  cd /tmp
  ```

  → change the current directory to /tmp

Exercises

- Display the path of your home directory with the following command:

```
echo $HOME
```

- Try the following sequence of commands:

```
pwd
cd /tmp
pwd
cd
pwd
```

From the previous sequence, what's the effect of command “cd” without argument?

Listing files (1)

- Command: **`ls`**
- List files in a directory (default is the current directory)
- Default behavior:
  - alpha-numeric order
  - files beginning with a dot are not printed (*hidden* files)

Listing files (2)

- Useful **ls** commands:
  - ls -a
    - → list all the files (including hidden files)
  - ls -l
    - → long listing : display all file information (*metadata*)
  - ls -al
    - → just a combination of the two previous commands
  - ls -lh
    - → long listing with sizes in human readable format
  - ls -t
    - → chronological order (most recent first)
  - ls -ltr
    - → long listing, display in reverse chronological order
  - ls -S
    - → list the biggest files first

Exercises

- Try the previous commands on the directory /tmp.
- Are there hidden files in your home directory? Which ones? What command did you use?
- What's the biggest file in the /sbin directory? What command did you use?
- Convince yourself that “ls” command can take a space-separated list of arguments by entering the following:

```
ls -al /vsc /home
```

Listing files (3)

- Example:

[stgerard@nic112 ~]$ ls -al

total 201926

drwx------	16	stgerard	dntk	68	Oct 16 15:01	.
drwxr-xr-x	489	root	root	492	Dec 1 02:00	..
drwxr-xr-x	2	stgerard	dntk	94	Mar 17 2014	accounting
-rwx------	1	stgerard	mech	196	Feb 22 2013	ask_dns.pl
-rw-------	1	stgerard	mech	19672	Oct 13 15:22	.bash_history
-rw-r--r--	1	stgerard	mech	42	Feb 24 2011	.bashrc
drwx------	2	stgerard	fd060	3	Sep 23 2009	.config
-rw-------	1	stgerard	mech	7746	Feb 21 2011	dns_records
drwxr-xr-x	7	stgerard	dntk	8	May 12 2014	examples

Permissions

Ownership

Size (in octet)

Last modification date

Listing files (4)

- Explanation of “ls -al” output:
  - 1st column: one letter giving the file type, followed by the permissions (9 characters); possible file types:
    - -: ordinary file
    - d: directory
    - l: symbolic link
    - s: socket
    - p: named pipe
    - b: block device
    - c: special file

    (Permissions will be explained later.)
  - 2nd column: contains a number whose meaning depends on the file type. For an ordinary file, it is the number of hardlinks pointing to the file. For directories, it is the number of subdirectories (+2 to take into account the “.” and “..” directories).
  - 3rd and 4th columns: owner and group owner of the file
  - 5th column: file size in bytes
  - 6th column: last modification date
  - 7th column: short (relative) name of the file; if the file is a symbolic link, the target is indicated

Absolute and relative path

- **Absolute path**: the full path, beginning from the root directory.

  Example:

  ```
  /home/sgerard/test1/testfile.txt
  ```

- **Relative path**: only gives the name of a file relative to the current directory.

  Example:

  If your are in the directory `/home/sgerard/test1`:

  ```
  testfile.txt
  ```

Special directory names

- Can be viewed as generic shortcuts:
  - The directory named “.” is pointing to the current directory.
  - The directory “..” is pointing to the parent directory of the current directory.
  - The directory “~” is pointing to the home directory of the current user.
- Example:

  Back to example of previous slide:

  ```
  testfile.txt
  ./testfile.txt
  $PWD/testfile.txt
  ```

  are naming the same file.

Exercises

- Make sure that your current directory is your home directory. (Command?)
- What's the difference between the following commands:

```
ls -al
ls -al ~
ls -al .
```

- Write down the absolute path of the file '.bashrc' located in your home directory? Check that this path is correct with the “ls” command.

Remarks for Windows users (1)

- In the MS world, file names are ending with “.” followed by a few letters (usually 3). This is called the “extension”, and it indicates to the operating system with which application the file must be opened.

  In Linux, you are not obliged to add an extension to file names (although it's a good habit!)

- Avoid spaces in file names, because space is interpreted as a list separator in some commands.

  If a file name contains space(s), use surrounding quotes to get rid of ambiguities.

Remarks for Windows users (2)

- About the usage of “/” and “\” that is different in the two operating systems:
  - In Linux, the “/” is the directory separator, and the “\” is the escape character. (“Escaping” means avoiding interpretation of something.)
  - In Windows, the forward-slash “/” is the command argument delimiter, while the backslash “\” is a directory separator.

Exercises

- To illustrate the previous remark about using spaces in file names, let's do the following experiment:
  - Create a file with a space in its name:

    ```
    touch 'my file'
    ```

    (What would have happened without the quotes?)
  - Check the results of the following commands:

    ```
    ls -l my file
    ls -l 'my file'
    ```
  - To clean-up, enter the following:

    ```
    rm 'my file'
    ```

File size (1)

- The fundamental unit is the *bit*, which is the smallest unit of information in a binary system (values : 0, 1).
- By default, size is expressed in *bytes* (or *octets*).
- One byte is 8 bits. The unit symbol is: B.
- In text files, a character is coded on one byte. So, the size of a text file in bytes is equal to the number of characters it contains.
- For multiples, don't forget that in computer sciences, powers of 1024 are used instead of powers of 1000, because 1000 is approximately 2^10.
- Human readable format: size expressed using multiples, making it easier to read.

File size (2)

- Unit symbols used in the shell:

Symbol	Value
K → Kilo	1024 bytes
M → Mega	1024 * 1024 bytes
G → Giga	1024 * 1024 * 1024 bytes
T → Tera	1024 * 1024 * 1024 * 1024 bytes
P → Peta	1024 * 1024 * 1024 * 1024 * 1024 bytes

Exercises

- What's the size in octets of the biggest file in /sbin?
- Get the size in human readable format (clue: option for human-readable format is '-h'). Command?

File globbing (1)

- Substitution characters interpreted by the shell to generate file names:
  - * : match 0 or more characters
  - ? : match 1 character
  - [abc] : match one of the characters listed
  - [!abc] : match any character not listed
  - [a-z] : match one character in the range
  - {pear,peach} : match one word in the list

File globbing (2)

- Some examples:
  - s* : files whose name begins with “s”
  - s? : files whose name begins with “s” followed by one and only one character
  - *.c : files whose name ends with “.c”
  - *[0-9] : files whose name ends with a digit
- File globbing must not be confused with regular expressions (or *regex*), which is a more powerful technique aimed at pattern matching. If you want to learn more about regex:

  https://www.digitalocean.com/community/tutorials/an-introduction-to-regular-expressions

Exercises

- Using the “ls” command with the appropriate globbing expressions in argument, find the following files in /sbin:
  - files whose name begins with 's';
  - files whose name is two characters long;
  - files whose name ends with a dot followed by three characters;
  - files whose name contains the string 'fs'.

File copy

- Command: **`cp`**
- Examples:

```
cp myfile1 myfile2
```

→ create a copy of the file “myfile1” called “myfile2”

```
cp /home/sgerard/test1/myfile.txt .
```

→ create a copy of “myfile.txt” in the current directory

- To copy a directory and its content, you need recursivity option “-R”. Example:

```
cp -R test1 test2
```

→ create a copy “test2” of directory “test1”

Moving a file

- Command: **`mv`**
- Examples:

```
mv file1 file2
```

→ is the same as renaming “file1” to “file2”

```
mv /home/sgerard/myfile.txt .
```

→ move the file “myfile.txt” to the current directory

```
mv dir1 dir2
```

→ renaming directories works the same as for files

Deleting a file

- Command: **`rm`**
- Examples:

```
rm file1
rm /home/sgerard/file1
```

- To delete a directory, the recursive option “-r” must be added:

```
rm -r /home/sgerard/dir1
rm -rf
```

  → with “-f”, non confirmation is asked

- BE CAREFUL USING `rm` COMMAND: IT IS NOT REVERSIBLE!
- A solution: create your own “garbage bin”, a kind of “temp” directory where you can move your old files into.

Exercises

- Make a backup copy of the file .bashrc in your homedir:

```
cp .bashrc .bashrc.backup
```

- Rename the backup:

```
mv .bashrc.backup bashrc_backup_29042020
```

- Delete the backup:

```
rm bashrc_backup_29042020
```

Creating and deleting a directory

- Command: **`rmdir`**

  `rmdir dir1`

  → delete directory “dir1” if this directory is empty

- Command: **`mkdir`**

  `mkdir dir1`

  → create directory “dir1” in the current directory

  `mkdir -p dir1/subdir1/subsubdir1`

  → create directory “subsubdir1” and its parent directory

Exercises

- In your homedir, create the following directories:

```
temporary_directory/exercise1
temporary_directory/exercise2
```

  You can do it in only one step by issuing:

```
mkdir -p temporary_directory/{exercise1,exercise2}
```

- Create a file in the last directory:

```
touch temporary_directory/exercise2/answers.txt
```

- Try to delete directory:

```
rm temporary_directory
rmdir temporary_directory
```

  Why does it fail? Now, try this way:

```
rm -r temporary_directory
```

File permissions (1)

- Remember: Linux is natively a multi-user system, so you need permissions to protect your files against other users.
- Permissions are defined for 3 entities:
  - owner (u);
  - group of the owner (g);
  - the rest of the world (o).
- Three atomic permissions: read (r), write (w), execute (x)
- Giving one bit position for each permission, you get a number. Examples:

rwx	111	7 in decimal
rw-	110	6 in decimal
r-x	101	5 in decimal
r--	100	4 in decimal

File permissions (2)

- What the right to execute (x) means:
  - For a file, makes it executable (binary, script)
  - For a directory, gives the right to enter the directory and access its content
- What the right to modify (w) means for a directory:
  - To have the right to modify the content of a directory, you need to have the write permission on it.
  - In particular, if you don't have the write permission, you can't delete, create or rename a file in this directory.

File permissions (3)

- Some examples of permissions sets:

  775 → rwx rwx r-x

  → on a directory, give the owner and his group the right to read, modify and browse the directory, other users can only read and browse

  664 → rw- rw- r--

  → on a file, give the owner and his group the right to read and modify, other users can only read

  400 → r-- --- ---

  → the owner has the right to read, other users have no access

Changing permissions (1)

- Command: **`chmod`**
- Examples:
  - `chmod 775 file1`

    → give all the rights to owner and his group, other users are allowed to read and execute
  - `chmod 600 file1`

    → give owner the right to read and write, other users have no access

Changing permissions (2)

- Another way to use it, with the following notations:
  - People concerned:
    - u : owner
    - g : group
    - o : other
    - a : all
  - Action:
    - + : add the right
    - - : remove the right
  - Rights that you want to change:
    - r (read)
    - w (write)
    - x (execute)
- Putting all together, in some examples:

```
chmod u+x file1
```

→ add the right to execute to owner

```
chmod go-w file1
```

→ remove the right to modify to owner group and other users

Exercises

- Try to print out the content of /var/log/messages:

  ```
  cat /var/log/messages
  ```

  Why does it fail? Check the permissions of this file:

  ```
  ls -al /var/log/messages
  ```

- Do you have the right to see the content of the following directories:

  ```
  /var
  /var/log
  ```

- Who can see the content of your home directory?
- Let's say you would like to share a document with only 2 of your colleagues. Is it possible?

Transferring files with scp

- Transfering a file from your laptop to hydra.vub.ac.be with scp:

  scp <file_to_copy> <your_login>@hydra.vub.ac.be:~/

  Note that what follows the ":" is the destination on the remote machine. In the command above, the destination is "~/", i.e. your home directory on hydra.

- To transfer a directory and its content, add the option "-r":

  scp -r <dir_to_copy> <your_login>@hydra.vub.ac.be:~/

Finding help on commands

- Command **`man`** followed by the command name:
  - Help is displayed in page format
  - Use arrows or Page-up/Page-down to navigate in pages content
  - Exit man pages by typing “q” key
- To search for a word in the man pages, just type “/” followed by the word, and then:
  - Search for the next occurrence with “n”
  - Search for the previous occurrence with “N”
- Command name followed by option **`-h`** or **`--help`**
- Command **`info`** followed by the command name: usage is similar to **`man`**.
- Online help:

  https://www.gnu.org/software/coreutils/manual/html_node/index.html

Other useful commands

Commands to handle text files

- **`cat`**: concatenate files and print on the standard output
- **`more`**, **`less`**: view file content page by page
- **`head`**/**`tail`**: view first/last part of a file
- **`wc`**: print newline, word, and byte counts of a file
- **`sort`**: sort lines of text files
- **`diff`**: compare files line by line
- **`grep`**: print lines matching a pattern
- **`cut`**: remove sections from each line of files
- **`touch`**: change file timestamps
- **`echo`**: display a line of text

Handling text files
`cat`

- To concatenate and print files on the standard output (or STDOUT, i.e. the screen by default)
- Examples:

```
cat file1
```

→ print the content of “file1” to STDOUT

```
cat file1 file2
```

→ print the concatenation of “file1” and “file2” to STDOUT

- Exercise:

  Try the following command:

```
cat /proc/cpuinfo
```

Handling text files
`more` and `less`

- To print the content of a file in page mode
- Typical usage :

```
more <filename>
less <filename>
```

- Exit the command by typing “q”
- Similar commands, but “less” allows backward and forward movements with arrows and PageUp and PageDn keys.
- Exercises:

  Try these two commands on /proc/cpuinfo

Handling text files
`head` and `tail`

- To view only a part (beginning or end) of a file:
  - head: to view the beginning of a file
  - tail: to view the end of a file
- Examples:

```
tail -n 20 filename
```

→ show the last 20 lines of filename

```
head -n 20 filename
```

→ show the first 20 lines of filename

- Exercises:

  Try the two previous commands on /proc/cpuinfo

Handling text files
wc

- To print some statistics about a textfile: word, newline, and byte counts.
- Examples:

```
wc -l myfile
```

→ print the number of lines in “myfile”

```
wc -w myfile
```

→ print the number of words in “myfile”

Handling text files
`sort`

- To sort the lines of a text file
- Examples:

```
sort file1
```

→ print the sorted content of “file1”

```
sort -o file2 file1
```

→ sorted content of “file1” is sent to “file2”

```
ls | sort -r
```

→ the same as above, but in reverse order

- Remark: sort doesn't modify the file content!
- Exercises: in the hands-on part

Handling text files
diff (1)

- To compare files line by line
- Example 1:

```
$ cat file1
  peach
  lemon
  orange
$ cat file2
  peach
  lemon
  pear
  orange
$ diff file1 file2
  2a3
  > pear
```

Read the output like this:
“If you add line 3 of file2 after line 2 of file1, you get file2.”

Handling text files
diff (2)

- Example 2:

```
$ cat file1
peach
lemon
orange
$ cat file2
peach
lemon
pear
strawberry
orange
$ diff file1 file2
2a3,4
> pear
> strawberry
```

Read the output like this:
"If you add line 3 to 4 of file2 after line 2 of file1, you get file2."

Handling text files
`grep`

- To print only lines matching a pattern
- Only prints the lines that match the criterion given in option (*horizontal slicing*)
- Examples:

```
grep 'apple' file1
```

→ print the lines from file1 that contain 'apple'

```
grep -v 'apple' file1
```

→ print the lines from file1 that don't contain 'apple'

- Exercises: in the hands-on part

Handling text files
cut
(1/3)

- To remove sections from each line of file
- Only prints the part of each line that is described by options (*vertical slicing*)
- Two basic ways of slicing:
  - Based on character positions → option “-c”
  - With structured lines (line = list of values separated by a character): based on fields positions → option “-f” with “-d”
- Defining slices:

N	N'th byte, character or field, counted from 1
N-	from N'th byte, character or field, to end of line
N-M	from N'th to M'th (included) byte, character or field
-M	from first to M'th (included) byte, character or field

Handling text files
cut
(2/3)

- Example 1:

```
cat fruits.txt
  apple
  pear
  apricot
cut -c 1-3 fruits.txt
  app
  pea
  apr
```

Handling text files
cut
(3/3)

- Example 2:

```
cat beatles.txt
  1:Paul:bass
  2:Ringo:drums
  3:George:guitar
cut -f2 -d: beatles.txt
  Paul
  Ringo
  George
```

Handling text files
`touch`

- To change the timestamps (access and modification times) of a file to the current time. If the file doesn't exist, it is created empty, unless options “-c” or “-h” are used.
- Example:

```
touch test.txt
```

→ access and modification times of 'test.txt' are set to the current time; if the file 'test.txt' doesn't exist, it is created empty

- Exercises: in the hands-on part

Handling text files `string arguments`

- When giving a string as argument to a command, it should always be surrounded by simple quotes ('), or double quotes (“”) if it contains ambiguous characters (like spaces for example):
  - Strings surrounded by simple quotes: characters inside the string are not interpreted by the shell.
  - Strings surrounded by double quotes: some special character sequences are interpreted because they have a special meaning. For example:

    \\ backslash

    \b backspace

    \n new line

    \t horizontal tab

Handling text files
echo

- To display a line of text on the standard output
- It is frequently used with stdout redirection to modify the content of a text file.
- Examples:

```
echo 'This is a sentence written on one line.'
echo 'This is a sentence \n written on one line.'
echo -e 'This is a sentence \n written on two lines.'
```

- Exercises: try the previous examples

Some other useful commands

- **wget**: to dowload files from the Internet
- **date**: print or set the system date and time
- **df**: report file system disk space usage
- **du**: estimate disk space usage
- **tar**: manipulate archives
- **file**: determine file type
- **which**: show the full path of a command
- **find**: search for files

Some other useful commands `wget`

- To download files from Internet
- Example:

```
wget https://archive.org/download/ulysses04300gut/ulyss10.txt
```

→ download the file “ulyss10.txt” from the URL in argument

- Exercises:
  - Try out the previous command.
  - The downloaded file contains the text of a famous novel. Based on the file size, compute how many such books we could store on a 1 TB hard drive?

Some other useful commands

`date`

- To print or set the system date and time
- Examples:

  `date`

  → print the current date and time

  `date --date='2 days ago'`

  → print the date of the day before yesterday

  `date +%s`

  → print the date in Unix format

  (= number of seconds since the epoch, i.e. 1970-01-01 00:00:00 UTC)
- Exercises: try out the three previous examples
- More examples:

  https://www.gnu.org/software/coreutils/manual/html_node/Examples-of-date.html

Some other useful commands

`df`

- To report file system disk space usage
- Examples:

  `df`

  → display all file systems and their disk usage

  `df -h`

  → same as above, but use “human readable” format
- Exercises: try out the two previous commands

Some other useful commands

du

- To estimate file space usage
- Examples:

```
du -h /home
```

→ summarizes disk usage of directory /home recursively

→ sizes are printed in human readable format

```
du -h --max-depth=1 /home
```

→ summarizes disk usage for level 1 subdirectories

→ sizes are printed in human readable format

- Exercise: get the size of your homedir in human format

Some other useful commands

`tar`

- To manage archives (= many files gathered into a single tape or file)
- Examples:

```
tar cvzf backup.tgz /home
```

→ archives the content of /home into a file backup.tgz

→ compression is used (hence the “z” in options)

```
tar xvzf backup.tgz
```

→ extracts and uncompresses files from backup.tgz

Exercises

- Create a directory 'test_tar' with two files, 'file1' and 'file2' in it.
- Create a compressed archive of the previous directory:

```
tar cvzf test_tar.tgz test_tar
```

- Check the existence of the archive.
- Delete the directory 'test_tar'
- Extract the archive:

```
tar xvzf test_tar.tgz
```

- You should have recovered the directory the directory 'test_tar'.

Some other useful commands `which`

- Displays the absolute path of commands
- To find the absolute path, the shell with do a search in each directory mentioned in the PATH environment variable.
- Examples:

  `which df`

  → shows the full path of command 'df'

  `which showq`

  → shows the full path of command 'showq'

- Exercises:

  Try the previous examples and verify that the paths of these commands are mentioned in the PATH variable.

Some other useful commands

`file`

- To determine the type of a file
- Examples:

```
file /usr/bin/du
```

→ /usr/bin/du: ELF 64-bit LSB executable, x86-64

```
file TQC_UJF_10.pdf
```

→ TQC_UJF_10.pdf: PDF document, version 1.4

Some other useful commands

`find`

- To search for files in a directory hierarchy
- Syntax:

```
find [option...] [path...] [expression]
```

- Examples:

```
find . -name "*.c"
```

→ find in current directory all files ending with “.c”

```
find . -type d -name "*s"
```

→ find in current directory all directories ending with “s”

```
find ./dir1 -type f -ok rm {} \;
```

→ find all files in current directory and delete them asking a confirmation

- Exercises: in the hands-on part

Writing simple shell scripts

What is a shell script

- A shell script is a text file containing commands that will be executed in a sequential way by the shell interpreter.
- The goal of shell scripting is to automate the execution of a series of tasks, reducing human intervention at a minimum.
- Ideally, a shell script begins with a special line called “shebang” that gives the absolute path of the interpreter. The standard shebang is:

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

Shell scripting limitations

- Shell scripting is perfect to launch some shell commands with arguments and options (*wrapper scripts*).
- However, shell scripting suffers from severe limitations:
  - Natively limited to integer calculations
  - Not optimized for computation
  - Poor syntax, sometimes not very handy
  - ...
- For all these reasons, you should not use shell scripts for scientific computation. Instead, use an advanced scripting language like Python or Julia.

Creating a shell script

- Only tool needed : a text editor (see next slides)
- Once the content of the script is edited, make it executable with the following command:

```
chmod u+x <your_script>
```

- To launch your script:

```
./<your_script>
```

Text editing

- Command-line text editors:
  - Most of the time, they are installed by default and/or they have few requirements
  - Examples: vi, vim, nano, emacs…
  - Some editors may require some practice before you can feel at ease with them (that's the case with vi, vim and emacs).
- Graphical text editors:
  - Graphical interface → easier to use
  - Examples : gedit, Kate, nedit,...
  - Require X11 forwarding if working on a distant machine

Text editing : nano (1)

- Launching nano:

  ```
  nano <name_of_the_file_to_edit>
  ```

  You can also just type “nano” without specifying a filename. You will then be asked to give a filename when exiting the program.
- Text editing: just like in any other text editor
- Commands are called using shortcuts. At any time, the main relevant commands are reminded at the bottom of the nano window.
- Shortcut: CTRL key (noted with a caret, i.e. “^”), followed by a lowercase letter.

Text editing : nano (2)

- If you need help about shortcuts/commands: CTRL + g
- Copy/paste:
  - text selection with the mouse;
  - copy and paste with the “Edit” menu of the window.

Text editing : micro

- Offers similar functionalities + syntax colouring
- Alt+G: list of keyboard shortcuts
- Ctrl+G: Help
- Ctrl+Q: Quit
- Ctrl+S: Save
- Ctrl+Z: Undo
- Ctrl+Y: Redo

Exercises

- Let's write our first shell script called “script.sh”:
  - We will launch nano to edit the content:

```
nano script.sh
```

  - The content:

```
#!/bin/bash
echo ‘Hello World!’
```

  - Exit nano by typing 'CTRL X' and press 'Y' to save
  - Make the script executable:

```
chmod u+x script.sh
```

  - Run the script :

```
./script.sh
```

Variables

- Assigning a value to a variable:

  Syntax:

  ```
  <variable_name>=<value>
  ```

  Examples:

  ```
  myvar1='example of a string value'
  myvar2=”another example”
  myvar3=123
  ```

- Using the value of a variable:

  You need to append the '$' sign at the beginning of the variable name.

  Examples:

  ```
  echo $myvar1
  echo “The number is $myvar3”
  ```

The “if” command (1/2)

- Syntax:

```
if [ <condition> ]
then
    <commands_if_condition is true>
else
    <commands_if_condition_is_false>
fi
```

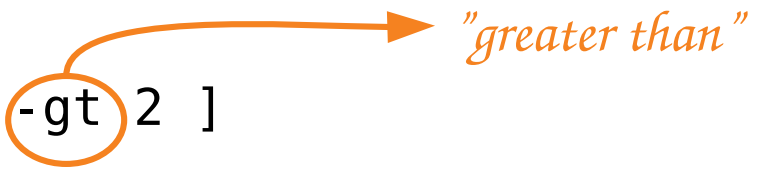
- The condition is an expression that evaluates to true or false (*boolean*).

The “if” command (2/2)

- Example:

  The following code :

```
myvar=3
if [ $myvar -gt 2 ]
then
    echo “variable greater than 2”
fi
```



  will result in printing the sentence “variable greater than 2”.

- Exercise :

  Try the previous example in your shell. Redo the exercise with myvar=2 to check what happens if the condition is not fulfilled.

Nested “if” commands

- Syntax:

```
if [ <condition> ]
then
    <commands_if_condition is true>
elif [ <condition> ]
then
    <commands_elif_condition_is_true>
else
    <commands_elif_condition_is_false>
fi
```

- The “elif” command can be understood as “else if”. There can be as many “elif” as needed.

Writing conditions (1/3)

- Conditions on strings:

Condition	Meaning
`$string1 = $string2`	Returns true if the 2 strings are identical
`$string1 != $string2`	Returns true if the 2 strings are different
`-z $string1`	Returns true if the string is empty
`-n $string1`	Returns true if the string is not empty

Writing conditions (2/3)

- Conditions on numbers:

Condition	Meaning
`$num1 -eq $num2`	Returns true if the 2 numbers are equal
`$num1 -ne $num2`	Returns true if the 2 numbers are not equal
`$num1 -lt $num2`	Returns true if num1 is strictly little than num2
`$num1 -le $num2`	Returns true if num1 is little than or equal to num2
`$num1 -gt $num2`	Returns true if num1 is strictly greater than num2
`$num1 -ge $num2`	Returns true if num1 is greater than or equal to num2

Writing conditions (3/3)

- Conditions on files:

Condition	Meaning
`-e $filename`	Returns true if the file exists
`-d $filename`	Returns true if the file is a directory
`-f $filename`	Return true if the file is not a directory

Remember that Linux considers (almost) everything as a file! So, a directory is viewed as particular type of file.

Complex conditions

- Logical operators:
  - || → logical “or”
  - && → logical “and”
- Syntax:

```
[<condition1>] || [<condition2>]
[<condition1>] && [<condition2>]
```

While loop

- Principle:

  While a given condition is true, a block of commands is repeated.

- Syntax:

```
while [ <condition> ]
do
   <block_of_commands>
done
```

For loop

- Principle:

  A block of commands is repeated for each value found in a list.

- Syntax:

```
for <variable> in <list>
do
    <block_of_commands>
done
```

Process management

Process and program

- *Process*: a running instance of a program.
- *Program*: an executable file (sometimes referred to as *binary*) that has been compiled from source code into machine code.
- *Daemon*: a process that runs continuously in the background, rather than under the direct control of a user.
- Each process is identified by a unique id, the *PID*.

Viewing processes

- Main commands to get a quick overview of the processes:

  `ps aux`

  → a snapshot of all processes

  → 1st column shows the user

  → 2nd column shows the PID of each process

  → last column shows the command

  `top`

  → a dynamic real-time view of the processes

  → displays information about resources (CPU & Mem) consumption

  → exit by typing 'q'

  `pstree`

  → processes are displayed in tree

  → interesting to see the relations between processes

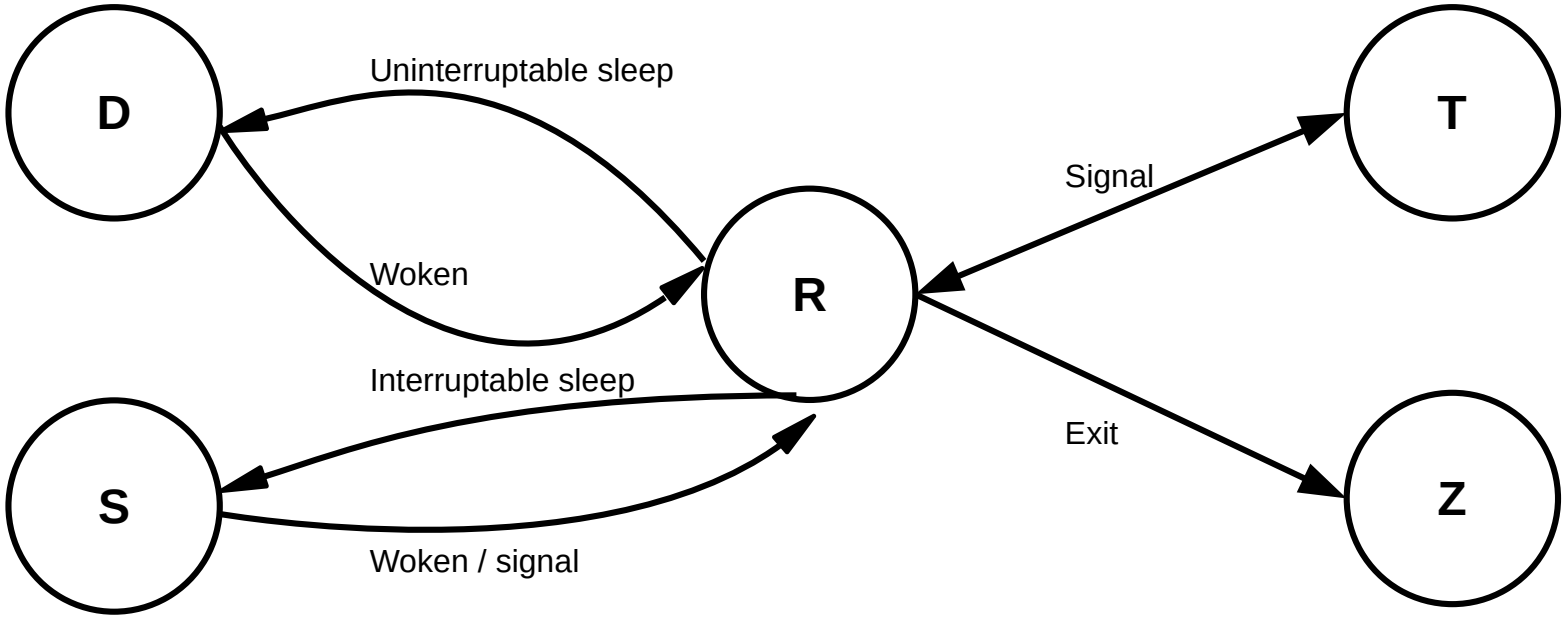
Exercises

- Try the 3 previous commands.
- Focusing on the non-root processes, can you see if there are other users busy on your login node?
- Identify the heaviest processes running on your login node, in terms of CPU time and/or memory.

Process states

- Since there are generally more processes than CPUs/cores, not all processes can be running at the same time.
- A process can evolve through different states:
  - Runnable
  - Sleeping
  - Uninterruptable sleep state
  - Defunct or Zombie

Process life cycle



Process in background

- When dealing with long process
- Just add an ampersand (&) at the end of the command before typing ENTER

  Example:

```
[stgerard@nic50 ~]$ sleep 100 &
[1] 23019
```

  The number between brackets is the job id, not to be confused with the corresponding process id (PID) that is the number right after.

- To view all the processes in background:

```
jobs
```

Killing a process

- Graceful way:

  You want to stop the process the normal way (like you would do by doing CTRL F4 or CTRL C), saving its state. Command:

  ```
  kill -SIGTERM <pid>
  ```

- Forcefully:

  When your process became unresponsive… Command:

  ```
  kill -SIGKILL <pid>
  ```

Process state codes

- From documentation of “ps”:

  R running or runnable (on run queue)

  D uninterruptible sleep (usually IO)

  S interruptible sleep (waiting for an event to complete)

  Z defunct/zombie, terminated but not reaped by its parent

  T stopped by a job control signal

  […]

Exercises

- This exercise illustrates how you can stop a process and let it continue in background:

  `sleep 100`

  → will pause during 100 seconds

  `CTRL Z`

  → put the sleep process in “stopped” state

  `ps -o pid,state,command`

  → to check the status of the sleep process

  `bg`

  → the process is put back running in background

Running long tasks

- If you terminate a shell session, you kill all the processes that you've launched during this session.
- There are solutions to keep a process running after shell exit:
  - disown
  - nohup
  - ...
- However, the good practice is to execute long tasks via job submission.

Nesting commands

- Notation: $(...)
- Useful when the argument of a command is the result of another command
- Examples:

```
ls -al $(which bash)
touch testfile_$(date +%F)
```

- Exercises: try the previous examples

Input, output and error

- In Unix-like operating systems, each process is automatically assigned three data streams: one input stream, called *standard input* (or *stdin*), and two output streams, called *standard output* (or *stdout*) and *standard error* (or *stderr*).
- By default:
  - *stdin* is the keyboard
  - *stdout* and *stderr* are the screen

I/O redirection (1)

- **stdin**, **stdout** and **stderr** can be redirected to a file:

  > : redirect standard output (overwrites)

  < : redirect standard input

  >> : redirect standards output (adds)

  2> : redirect standard error

  2>&1 : redirect standard output and error

I/O redirection (2)

- Examples:

```
ls -al > dircontent.txt
```

→ print the output of “ls -al” in file “dircontent.txt”

```
date >> dircontent.txt
```

→ add the date at the end of “dircontent.txt”

```
./mycommand.sh 2> error.txt
```

→ send error messages of the script in file “error.txt”

```
./mycommand.sh > output.txt 2> error.txt
```

→ send output in “output.txt” and error messages in “error.txt”

```
./mycommand.sh > output_and_error.txt 2>&1
```

→ send output and errors in the same file “output_and_error.txt”

Pipe

- Linux is multi-task. One practical consequence: it is possible to send directly the output (stdout) of a process to feed the input (stdin) of another process.
- To interconnect processes, a pipe (or tube) is used. It is represented by a “|”.
- Examples:

```
ls | sort
ls | sort -r
find /u/ | sort | more
```

Exercises

- Try the following commands:

```
ls -al > curdircontent.txt
find /sbin/ | sort > sbincontent.txt
```

  Try to figure out what these commands are doing, and check that the content of the resulting files is in line with your expectations.

- Try the following command:

```
ls -al /user/brussel/*/$(whoami) /dontexist/$(whoami)
```

  The previous command should generate an error message. Why?

  You can separate the normal output from the error messages by doing this:

```
ls -al /user/brussel/*/$(whoami) /dontexist/$(whoami) >
output.txt 2> error.txt
```

  Check the content of the files “output.txt” and “error.txt”.

Running applications

- Just type the application name and validate. Tip: don't forget to use TAB key to benefit from auto-completion!
- In some cases, you might get a “command not found” message, meaning that the binary was not found in PATH. Solutions:
  - Either add the directory of the binary to the PATH variable;
  - Or type the full path of the binary;
  - Or get into the directory of the binary and type :

```
./name_of_binary
```

Environment

- Ordinary variables are local in the sense that they are only visible in the context where they are defined. On the contrary, an environment variable is one that can be inherited to all the processes that are created after it has been defined.
- To print all your environment variables and their actual values, use “env” command.
- An environment variable can be a list of values (separated by “:” in Bash shell).
- To print the value of a variable, its name must be preceded by a “$”:

  `echo $<MYVAR>`

  Example: `echo $MYVAR`
- To change the value of an environment variable:

  `export <MYVAR>=<VALUE>`

  or, in case of a list:

  `export <MYVAR>=$<MYVAR>:<VALUE>`
- Exercise: see the hands-on part

Useful references

- www.tldp.org
  - The Linux Documentation Project
- www.linfo.org
  - High quality information about Linux and Free Software
- www.ibm.com/developerworks/linux/
  - A very good starting point to learn basics of Linux
- www.gnu.org
  - One of the most complete documentation on the GNU commands
- https://software-carpentry.org/lessons/index.html
  - Basics skills needed in research computing

PART 2: Hands-on sessions

Hands-on sessions

- Session 1: Mastering commands
- Session 2: Writing shell scripts
- Lab 1: Exploring a big text file

Recommendations for the hands-on sessions

- Please take note of your answers.
- If you don't find the answer to an exercise, don't hesitate to ask me or my colleagues for help.
- Answers to the exercises are available on this page:

  https://homepage.iihe.ac.be/~sgerard/

Hands-on
Session 1

man is your friend!

- Using the man command, find the meaning of these commands:

```
ls -F
date --reference=/etc/passwd
```

- Using the man command, find the correct command to display the current date in “dd/mm/yyyy” format.
- Using the man command, find the correct command to display the current date in Unix time (i.e. the number of seconds that have elapsed since 1970-01-01 00:00:00 UTC).

Hands-on
Session 1

Doing research in file content with **grep**

- We will do the exercises on a logfile that you can download here:

  http://homepage.iihe.ac.be/~sgerard/20171002.tgz

  (Clue: use 'wget' with option '--no-check-certificate')
- The logfile has been compressed within a tar archive. Extract the file from the archive.
- How many lines contain the word LOG_ERROR?
- Find all events (lines) corresponding to 'Exit_status=0'? How many such events are there?
- Find all events (lines) corresponding to an 'Exit_status' not equal to 0. How many such events are there?

Hands-on
Session 1

Sorting files with **sort**

- Download this tarball and extract its content:

  http://homepage.iihe.ac.be/~sgerard/cpusecnodes.tgz

- Have a look at its contain to understand how it is structured.
- Sort the file on the first column redirecting the output to a file 'cpusecnodes.sorted'. Clue 1: the sort being based on a numeric value, look at the man pages of 'sort' to find the correct option. Clue 2: use I/O redirection with '>'.

  Warning: The file to sort being quite big, please be patient !

Hands-on
Session 1

Sorting numbers with **sort**

- Create a file file_to_sort.txt with this content:

  02

  01

  10

  2

  12

- Sort the file using “sort”. Are the number in the correct numeric order?
- Now, sort the file using “sort -n”.
- Using man, find the correct option to remove duplicates, and test it.

Hands-on
Session 1

Monitoring long processes with **time** and **top**

- Open a second shell session
- In the first shell, launch this command:

```
time sort cpusecnodes > cpusecnodes.test
```

  While this command is running, have a look at the top statistics on the second shell. Is the sort process using a lot of resources? Is it running on a single core?
- When the previous sorting is complete, look at the statistics. The real time is the time elapsed from your point of view.

Hands-on
Session 1

Sorting and filtering command results thanks to pipes

- Sort the output of command “ps aux” by user name
- Print the output of command “ps aux” without the processes belonging to root

Hands-on
Session 1

Comparing files with **diff**

- Download these 2 files:

  https://homepage.iihe.ac.be/~sgerard/nodelists/20161001

  https://homepage.iihe.ac.be/~sgerard/nodelists/20171001

  These are lists of a machines that were reported to be in production in computing cluster at two different dates (01/10/2016 and 01/10/2017).
- Using diff command, find out which machines were removed from or added to the cluster.

Hands-on
Session 1

File searching with **find**

- Find files in /bin with size bigger than 100K.
- Find files in /etc whose name is ending in ".conf".
- What's this command doing:

```
find ~/ -name "*.txt" -mtime -60 -exec cat {} \;
```

Don't forget: man is your friend!

Hands-on
Session 1

Vertical slicing with **cut**

- Create a file “fruits.txt” with the following content:

```
abricot
banana
pear
```

  Find the command to print out the first 3 letters of each line.

- Create a file “beatles.txt” with the following content:

```
1:Paul:bass
2:Ringo:drums
3:George:guitar
```

  Find the command to print out the second column.

Hands-on
Session 2

A shell script taking two arguments

- Write this code into a file “name.sh”:

```
#!/bin/bash
# example of using arguments to a script
echo "My first name is $1"
echo "My last name is $2"
echo "Total number of arguments is $#"
```

- Make the file executable:

```
chmod u+x name.sh
```

- Run the script with its two arguments:

```
./name.sh <first_name> <last_name>
```

Hands-on
Session 2

A shell script with a “for” loop

- Write this code into a file “loop.sh”:

```
#!/bin/bash
for i in $(seq 1 10)
do
        echo  “number $i”
done
```

- Make the file executable:

```
chmod u+x loop.sh
```

- Run the script:

```
./loop.sh
```

Hands-on
Session 2

Listing files with a “for” loop

- Write this code into a file “loop_files.sh”:

```
#!/bin/bash
for fic in $(ls)
do
    echo “$fic”
done
```

- Make the file executable:

```
chmod u+x loop_files.sh
```

- Run the script:

```
./loop_files.sh
```

Hands-on
Session 2

Generating Fibonacci numbers with a “for” loop

- Write this code into a file “fibo.sh”:

```
#!/bin/bash
arg=$1
echo "Here are the $arg first Fibonacci numbers :"
a=0
b=1
echo $a
echo $b
for i in $(seq 1 $arg)
do
    c=$[$a + $b]
    echo $c
    a=$b
    b=$c
done
```

- Check the script:

```
chmod u+x fibo.sh
./fibo.sh 10
```

Hands-on
Session 2

Discovering the “while” loop

- While the condition of the “while” command is true, you keep on looping. To illustrate this, we will create a infinite loop.
- Write this code into a file “infinite_while_loop.sh”:

```
#!/bin/bash
echo "Press CTRL+C to stop the loop"
while true
do
    echo "You are caught in an infinite loop"
    sleep 1
done
```

- Check the script:

```
chmod u+x infinite_while_loop.sh
./infinite_while_loop.sh
```

Hands-on
Session 2

Interactive script with “read” and “while”

- The “read” command waits for the user to type in something until the ENTER key has been pressed. What the user has typed is kept in the variable following the “read” keyword.
- Write this code into a file “interactive_loop.sh”:

```
#!/bin/bash
line='start'
while [ $line != 'stop' ]
do
    read line
done
```

- Run the script:

```
chmod u+x interactive_loop.sh
./interactive_loop.sh
```

How to stop this script?

Hands-on
Session 2

Generating lists with the “while” loop

- We will use the “while” loop to print the list of the square integers little than a given integer.
- Write this code into a file “list_squares.sh”:

```
#!/bin/bash
limit=$1
sq=1
int=1
while [ $sq -lt $limit ]
do
    echo $sq
    int=$[$int + 1]
    sq=$[$int * $int]
done
```

- Check the script:

```
chmod u+x list_squares.sh
./list_squares.sh 125
```

Hands-on
Session 2

A script that prints to a file a report of all the processes

- Write this code to a file “report.sh”:

```
#!/bin/bash
pstree -apnu > “report_$(date +%s)”
```

- Make the file executable:

```
chmod u+x report.sh
```

- Run the script :

```
./report.sh
```

- Check the result

Hands-on
Session 2

Understanding environment variables

- Enter the following command:

```
export MYOWNVAR=”afancyvalue”
```

- Write this code to a file “print_var.sh” :

```
#!/bin/bash
echo Value of the variable: $MYOWNVAR
```

- Make the file executable:

```
chmod u+x print_var.sh
```

- Run the script:

```
./print_var.sh
```

- Check the result

Hands-on
Session 2

Discovering the “if” command

- Read the section **The “if” command** on this page:

  https://www.digitalocean.com/community/tutorials/how-to-write-a-simple-shell-script-on-a-vps-part-3

- Try the following code in a shell script:

```
#!/bin/bash
if [ 20 -lt 30 ]
then
    echo “20 is indeed less than 30”
fi
```

Hands-on
Session 2

Using comparison operators in “if” commands

- In this exercise, we will write a script that takes an integer as argument and then checks if its value is strictly greater than 10.
- Write the following code in a shell script “check_integer.sh”:

```
#!/bin/bash
if [ $1 -gt 10 ]
then
    echo “your value is greater than 10”
else
    echo “your value is less than or equal to 10”
fi
```

- Make the script executable and check it:

```
chmod u+x ./check_integer.sh
./check_integer.sh 1
./check_integer.sh 10
./check_integer.sh 11
```

Hands-on
Session 2

Nesting “if” using “elif” commands

- In this exercise, we will write a script that takes an integer as argument and then checks if its value is strictly greater or equal or strictly less than 10.
- Write the following code in a shell script “check_integer_improved.sh”:

```
#!/bin/bash
if [ $1 -gt 10 ]
then
    echo “your value is stricly greater than 10”
elif [ $1 -eq 10 ]
then
    echo “your value is equal to 10”
else
    echo “your value is strictly less than 10”
fi
```

- Make the script executable and check it:

```
chmod u+x ./check_integer_improved.sh
./check_integer_improved.sh 1
./check_integer_improved.sh 10
./check_integer_improved.sh 11
```

Hands-on
Session 2

Using string checking operators in “if” commands

- Write this code to a file “check_arg.sh”:

```
#!/bin/bash
if [ -z $1 ]
then
    echo “Error : Argument missing !”
    exit 1
fi
echo “Argument : $1”
```

- On this page, find the meaning of the '-z' in the condition of the 'if':
  https://www.digitalocean.com/community/tutorials/how-to-write-a-simple-shell-script-on-a-vps-part-3
- Make the file executable and test it:

```
./chmod u+x check_arg.sh
./check_arg.sh myargument
./check_arg.sh
```

- What's the role of the line 'exit 1'? (Hint : check the code without it)

Hands-on
Session 2

Using functions

- Write this code to a file “function_add.sh”:

```
#!/bin/bash
add() {
    sum=$[$1 + $2]
    echo $sum
}
add 1 3
```

- Make the file executable and test it:

```
./chmod u+x function_add.sh
./function_add.sh
```

Lab 1

Exploring a big text file (1/2)

- Getting the text file:

```
wget http://www.gutenberg.org/files/4300/4300-0.txt
```

- Using **ls** command, get:
  - the size in bytes and in human readable format
  - the permissions (who has the right to read?)
- Getting some statistics on the file content:
  - using **wc**, get the number of words and lines

Lab 1

Exploring a big text file (2/2)

- Parsing the file with basic commands:
  - Navigate through the file with **less**, and do some basic word search (suggested keyword: “Molly”, “Bloom”,...)
  - Same exercise but with **nano**
  - Check case sensitivity of the **grep** command with keyword “molly” and “Molly”. Find in the man pages how to make grep case-insensitive.