Linux

JUNIA-ISEN - M1

2022-2023

Contents

1 Pre-requisites

Pour ce cours nous allons travailler avec des machines virtuelles ou avec une distribution Linux. Pas de WSL !

2 Introduction to Linux

2.1 The operating system

Linux systems can be confusing, there exists a variety of operating system labeled as Linux. You most probably know Ubuntu, Debian or Fedora. They are in fact *distributions*. Many different Linux distributions are available to meet just about any computing requirement you could have. Most distributions are customized for a specific user group, such as business users, multimedia enthusiasts, software developers, or average home users. Every distributions and therefore every Linux system are composed of four main parts:

- The Linux kernel
- The GNU utilities
- A graphical desktop environment
- Application software

Every distributions are different flavors of these four elements.

Each of these parts has a specific job in the Linux system.

The core of the Linux system is the kernel. The kernel controls all the hardware and software on the computer system, allocating hardware when necessary and executing software when required.

Besides having a kernel to control hardware devices, a computer operating system needs utilities to perform standard functions, such as controlling files and programs. The GNU organization (GNU stands for GNU's Not Unix) developed a complete set of Unix utilities, but had no kernel system to run them on.

With the popularity of Microsoft Windows, computer users expected more than the old text interface to work with. This spurred more development in the OSS community, and the Linux graphical desktops emerged. There exists various graphical desktop environment (DE). Few examples are Gnome, KDE, XFCE... Desktop environment runs on top of a display server and a window manager that handle how windows should appear on the screen.

Application softwares can be run on any DE, however some have also been produced to integrate well on a particular graphical environment. They require libraries to be installed, and sometimes those libraries are used by core applications in a specific DE. For instance GNOME uses the GTK library.

2.2 Folders, files and permissions

Unlike some other operating systems, the Linux kernel can support different types of filesystems to read and write data to and from hard drives. Besides having over a dozen filesystems of its own, Linux can read and write to and from filesystems used by other operating systems, such as Microsoft Windows. The kernel must be compiled with support for all types of filesystems that the system will use.

Filesystem	Description
ext	Linux Extended filesystem — the original Linux filesystem
ext2	Second extended filesystem, provided advanced features over ext
ext3	Third extended filesystem, supports journaling
ext4	Fourth extended filesystem, supports advanced journaling
hpfs	OS/2 high-performance filesystem
jfs	IBM's journaling filesystem
iso9660	ISO 9660 filesystem (CD-ROMs)
minix	MINIX filesystem
msdos	Microsoft FAT16
ncp	Netware filesystem
nfs	Network File System
ntfs	Support for Microsoft NT filesystem
proc	Access to system information
ReiserFS	Advanced Linux filesystem for better performance and disk recovery
smb	Samba SMB filesystem for network access
sysv	Older Unix filesystem
ufs	BSD filesystem
umsdos	Unix-like filesystem that resides on top of msdos
vfat	Windows 95 filesystem (FAT32)
XFS	High-performance 64-bit journaling filesystem

Figure 1: The different filesystems supported by Linux distributions

2.2.1 Permissions

User's privileges are defined by a UID (User ID) and a GID (Group ID). Both are set in the `/etc/passwd` file. Groups are defined in the `/etc/group` file.

The UID `0` has a special role: it is always the root account. The UIDs `1` through `99` are traditionally reserved for special system users.

Some Linux distributions begin UIDs for non-privileged users at `100`. Others, such as Red Hat, begin them at `500`, and still others, such Debian, start them at `1000`. The UID `65534` is commonly reserved for nobody, a user with no system privileges, as opposed to an ordinary. Also, it can be convenient to reserve a block of UIDs for local users, such as `1000` through `9999`, and another block for remote users (i.e., users elsewhere on the network), such as `10000` to `65534`. The important thing is to decide on a scheme and adhere to it.

Unlike user 0 (the root user), group 0 does not have any special privilege at the kernel level. Traditionally, group 0 had special privileges on many unix variants — either the right to use su to become root (after typing the root password), or the right to become root without typing a password. Basically, the users in group 0 were the system administrators. When group 0 has special privileges, it is called wheel

On Linux, you need permissions to be able to read, write or execute files or directories.

The file permissions works as follows:

The umask is how you will **restrict**, by default, on newly created files. Think of working the opposite way of the file permissions describe in the image 2.

Here is an example:

You start with the full permission on read, write and execute for **owner**, **group** and **guest** then you substract what you want to restrict. Here we do not want **guest** to be able to use the files, and we restrict write permission for **group**

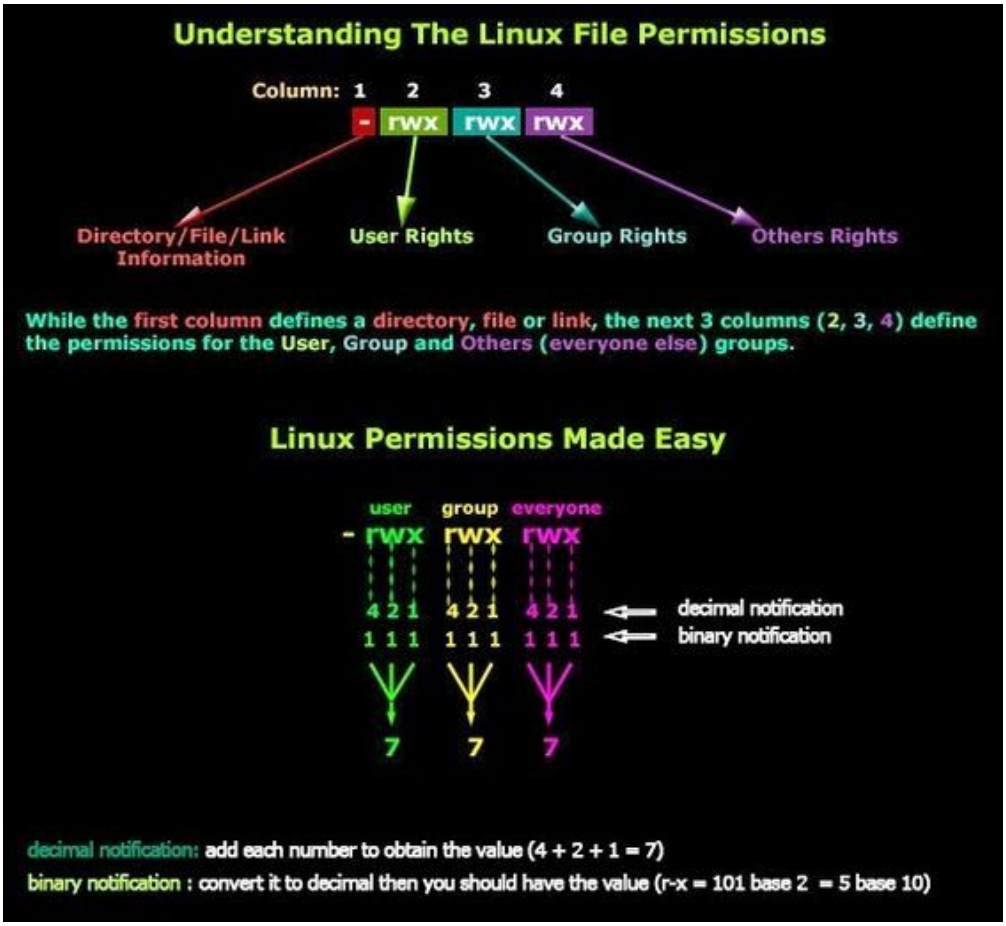

Figure 2: Linux file permission system

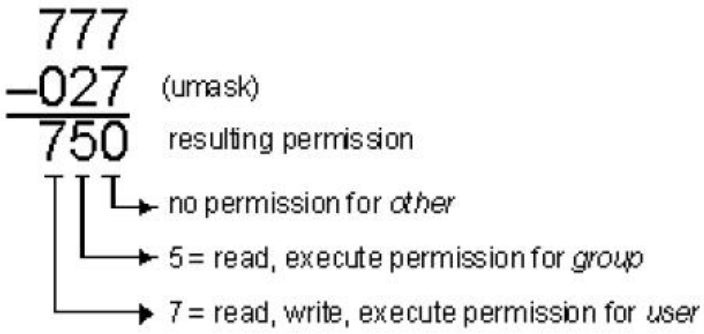

Figure 3: `umask` property to remove the permissions to a device or a file

2.3 Kernel and shell

2.3.1 Linux kernel

The kernel is primarily responsible for four main functions:

- System memory management
- Software program management
- Hardware management
- Filesystem management

System memory management Not only does the kernel manage the physical memory available on the server, but it can also create and manage virtual memory, or memory that does not actually exist. It does this by using space on the hard disk, called the swap space. The kernel swaps the contents of virtual memory locations back and forth from the swap space to the actual physical memory. This allows the system to think there is more memory available than what physically exists

The memory locations are grouped into blocks called pages. The kernel locates each page of memory either in the physical memory or the swap space. The kernel then maintains a table of the memory pages that indicates which pages are in physical memory and which pages are swapped out to disk. The kernel keeps track of which memory pages are in use and automatically copies memory pages that have not been accessed for a period of time to the swap space area (called swapping out), even if there's other memory available. When a program wants to access a memory page that has been swapped out, the kernel must make room for it in physical memory by swapping out a different memory page and swapping in the required page from the swap space. Obviously, this process takes time and can slow down a running process. The process of swapping out memory pages for running applications continues for as long as the Linux system is running.

Software program management The Linux operating system calls a running program a process. A process can run in the foreground, displaying output on a display, or it can run in the background, behind the scenes. The kernel controls how the Linux system manages all the processes running on the system. The kernel creates the first process, called the init process, to start all other processes on the system. When the kernel starts, it loads the init process into virtual memory. As the kernel starts each additional process, it gives it a unique area in virtual memory to store the data and code that the process uses.

The Linux operating system uses an *init* system that utilizes run levels. A run level can be used to direct the init process to run only certain types of processes. There are five init run levels in the Linux operating system. At run level 1, only the basic system processes are started, along with one console terminal process. This is called single-user mode. Single-user mode is most often used for emergency filesystem maintenance when something is broken. The standard init run level is 3. At this run level, most application software, such as network

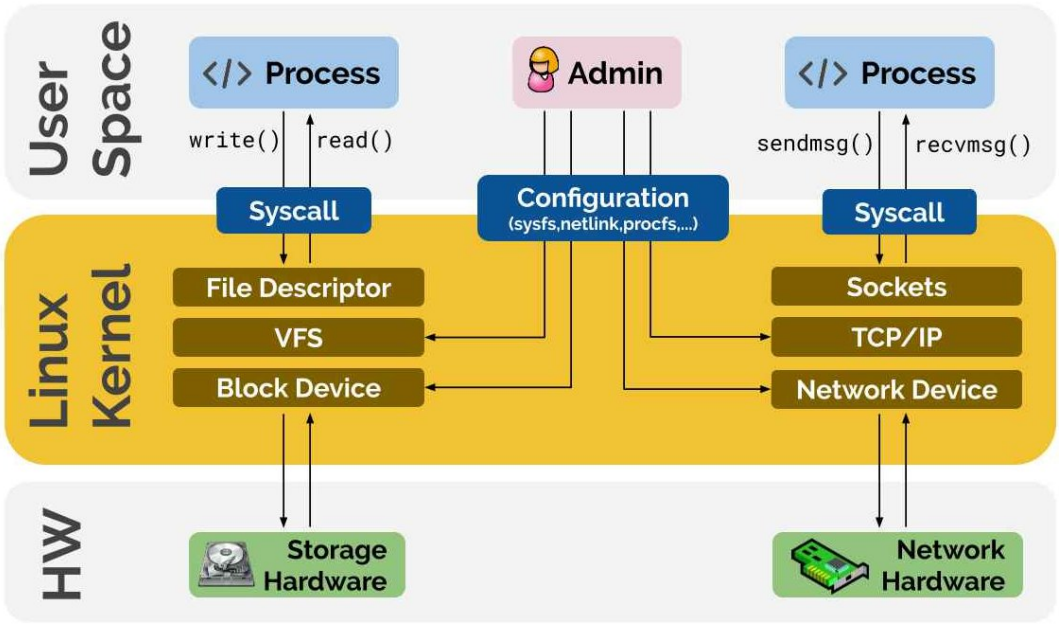

Figure 4: The core of the Linux system is the kernel. The kernel controls all the hardware and software on the computer system, allocating hardware when necessary and executing software when required.

support software, is started. Another popular run level in Linux is run level 5. This is the run level where the system starts the graphical X Window software and allows you to log in using a graphical desktop window.

Hardware management Any device that the Linux system must communicate with needs driver code inserted inside the kernel code. The driver code allows the kernel to pass data back and forth to the device, acting as a middle man between applications and the hardware. Two methods are used for inserting device driver code in the Linux kernel:

- Drivers compiled in the kernel
- Driver modules added to the kernel

Programmers developed the concept of kernel modules to allow you to insert driver code into a running kernel without having to recompile the kernel. Also, a kernel module could be removed from the kernel when the device was finished being used.

Filesystem management See above (2.2) for details about filesystem management.

2.3.2 The shell

The GNU/Linux shell is a special interactive utility. It provides a way for users to start programs, manage files on the filesystem, and manage processes running on the Linux system. The core of the shell is the command prompt. The command prompt is the interactive part of the shell. It allows you to enter text commands, and then it interprets the commands and executes them in the kernel.

There are several popular shells or command interpreters for the Unix / Linux systems.

`sh`: Bourne Shell. The oldest and installed on all Unix-based OS, but poor in functionality compared to other shells.

`bash`: Bourne Again Shell. An improved shell, available by default on Linux and Mac OS X

`ksh`: Korn Shell. A powerful shell quite present on the proprietary Unix, but also available in free version, compatible with bash (Korn's shell includes Bourne's shell)

`csh`: C Shell. A shell using a syntax close to the C language

`tcsh`: Tenex C Shell. An improved C Shell

`zsh`: Z Shell. Pretty new shell with the best ideas of bash, ksh and tcsh

2.4 The terminal and the first commands

Many graphical terminal emulator packages are available for the graphical environment. Each package provides its own unique set of features and options.

To access the graphical terminal emulator, you can usually press simultaneously `CTRL` + `ALT` + `T`. You can navigate to the application drawer and find `Console` or `Terminal`.

Info. Do not underestimate the use of the help command `man`.

2.4.1 Install packages

First we will try to install packages.

Debian and Ubuntu : `apt-get`

Fedora : `dnf`

CentOS and RedHat : `yum`

Arch and Manjaro : `pacman`

Question 2.1. Try to install a package on your system. And try to find the executable in the folder structure.

2.4.2 Process management

`command` : Launch a command

`command &` : Launch a command on background

`[ctrl z]` followed by `bg` : Put a command on background

`ps` : View the list of user processes

`kill` : Stop a process

`top` : View the list of all processes

Question 2.2. Run a command, put it into the background and try to put it back into the foreground.

Question 2.3. Put a command into the background and review the list of all processes to find it there.

2.4.3 Paths

Absolute path: `/dir/sub_dir_1/sub_dir_2/file`

Relative path from working directory `./dir`: `sub_dir_1/sub_dir_2/file` (notice there is no `/` at the beginning)

2.4.4 Wilcards

`/` system's root directory

`.` working directory

`..` directory one level up

`~` or `$HOME` login directory

`~otheruser` otheruser's home directory, if otheruser permission access is granted

`*` all files

2.4.5 Directory management

`pwd` : View the name of the working directory

`cd` : Change working directory

`ls` : View the content of a directory

`mkdir` : Create a directory

`rmdir` : Delete a directory

2.4.6 File management

`touch` : Create an empty file and change file date/time

`mv` : Rename or move a file

`cp` : Copy a file into another file

`rm` : Delete a file

Question 2.4. Play a little bit with the commands above to be familiar with them.

2.4.7 File content

`cat` : Display the content of a file or merge a list of files and display the result

`more` : Display the content of a file page per page

`less` : Display the content of a file page per page

`head` : Display the beginning of a file

`tail` : Display the end of a file

`wc` : Count the number of words, lines or characters

`diff` : Find differences between two files

Question 2.5. Create a file in a newly created folder and enter a random text. Use the commands above to understand the differences.

2.4.8 Quick help on a command

There are too many commands to experiment with. As a matter of fact, this course will not cover all of them. As a quick help you can use this command `curl cheat.sh/command`, where you replace *command*, by the command you want. For instance `curl cheat.sh/cat` will display this:

```
→ curl cheat.sh/cat
 cheat.sheets:cat
# POSIX way in which to cat(1); see cat(1posix).
cat -u [FILE_1 [FILE_2] ...]

# Output a file, expanding any escape sequences (default). Using this short
# one-liner let's you view the boot log how it was show at boot-time.
cat /var/log/boot.log

# This is an ever-popular useless use of cat.
cat /etc/passwd | grep '^root'
# The sane way:
grep '^root' /etc/passwd

# If in bash(1), this is often (but not always) a useless use of cat(1).
Buffer=`cat /etc/passwd`
# The sane way:
Buffer=`< /etc/passwd`

 cheat:cat
# To display the contents of a file:
cat <file>

# To display file contents with line numbers
cat -n <file>

# To display file contents with line numbers (blank lines excluded)
cat -b <file>

 tldr:cat
# cat
# Print and concatenate files.
# More information: <https://www.gnu.org/software/coreutils/cat>.

# Print the contents of a file to the standard output:
cat path/to/file

# Concatenate several files into an output file:
cat path/to/file1 path/to/file2 > path/to/output_file

# Append several files into an output file:
cat path/to/file1 path/to/file2 >> path/to/output_file

# Number all output lines:
cat -n path/to/file

# Display non-printable and whitespace characters (with `M-` prefix if non-ASCII):
cat -v -t -e path/to/file
```

Figure 5: Quick help using the *curl* command for `cat`.

3 First scripts with command lines

Shell scripting is the set of commands to be executed by the shell. It is said to be the combination of long and repeatable command sequences into one script so that it can be executed as and when required. The main idea behind creating a shell script is to lessen the load of the end-user.

A script file is just a text file with a shebang at the very beginning. A shebang is started with `#!` following by the binary of the shell you want to use. Here we will use `bash`, then the complete shebang is `#!/bin/bash`

Once your script is finished, you need to make it executable. To do so, in the terminal, you can type `chmod +x myScript.sh`, if the name of your script is `myScript.sh`. Then to execute your script you have to type `./myScript.sh` directly in your terminal.

3.1 Script parameters

`$#` : This parameter represents the total number of arguments passed to the script.

`$0` : This parameter represents the script name.

`$n` : This parameter represents the positional arguments corresponding to a script when a script is invoked such `$1`, `$2`...

`$*` : This parameter describes the positional parameters to be distinct by space. For example, if there are two arguments passed to the script, this parameter will describe them as `$1`, `$2`.

`$$` : This parameter represents the process ID of a shell in which the execution is taking place.

`$!` : This parameter represents the process number of the background that was executed last.

`$@` : This parameter is similar to the parameter `$*`.

`$?` : This parameter represents exit status of the last command that was executed. Here 0 represents success and 1 represents failure.

`$_` : This parameter represents the command which is being executed previously.

`$-` : This parameter will print the current options flags where the set command can be used to modify the options flags.

3.2 Create scripts with one command

Question 3.1. Following the commands you used in the introduction, try to create a script that move a file into a different folder.

Question 3.2. As a second task, create a script that takes as a parameter the PID of a process to monitor.

3.3 Create scripts with multiple commands

Question 3.3. Complete the first script by adding information about the current path of the file (after moving it, use the command `dirname`). And finally display the content of file you moved

Question 3.4. Try to think of a script that takes as a parameter the name of a text file and a sentence to find in the text file. (Hint: use `grep` to filter text.)

3.4 Outputs and display

Before or after each command, it is possible to print messages in the terminal. These messages are useful to see what the script is executing or they can help you debug your script. The command is `echo 'text'`

Question 3.5. Complete the scripts above to print information about the process (for instance *Moving file* or *Displaying file content...*)

Question 3.6. Create a script that calculates the operation on two numbers given as parameters. Display the value using `echo`

While it is possible to display information as you execute a script. Sometimes it is necessary to redirect the output to a file. It is done by using `./script.sh > file.txt` to create or override the content of a file. If you want to append to an already existing file, you can use `>>` instead of a single `>`.

Question 3.7. Output the result of the calculation from the script above in a new file. Do the same multiple times and append the new results instead.

3.5 Text manipulation

The goal of this subsection is to get familiar with text manipulation. It is useful to automate string manipulation in a text file, or to apply filters (for instance, selecting the extension in a filename). The sed editor is called a stream editor, as opposed to a normal interactive text editor. In an interactive text editor, such as vim, you interactively use keyboard commands to insert, delete, or replace text in the data. A stream editor edits a stream of data based on a set of rules you supply ahead of time, before the editor processes the data.

Here's the format for using the sed command: `sed options script file`

An example of the `sed` command can be the following:

```
1       echo "This is a test" | sed 's/test/big test/'
```

The `s` is for `substitute`. `/` are delimiters. The first word is the string to change (`test`) and the second is the target (`big test`).

Question 3.8. Try the command and understand how it is working. Review the man page and try different option (special sequences, global, confirm...).

4 Advanced scripting

4.1 Variables

The Linux shells use specific environment variables by default to define the system environment. They are available for every softwares in your system. Some variables are used by the system itself. For instance, the `PATH` variable declares a list of paths where software binaries can be found. A software requiring an external application will look into the list of paths defined in `PATH` to search for the specific application. If the binary is not found, the software will return an error. An environment variable is defined by the dollar sign following by the name of the variable. To display the content of the variable you can do `echo $VARIABLE`. Try the command with your `PATH`. The command to create new environment variable is `export VARIABLE`. The dollar sign is required only to call the variable.

Question 4.1. Create a new environment variable in your terminal and display the content.

It is convenient to gather all user-created variable in one place. For that in your `$HOME` directory, there exists multiple files to list the variables. In practice two of them are used `.bashrc` and `.bashenv`. They are hidden files (a dot exists in front of the name). `.bashrc` is used in interactive shell. Meaning that everything that is defined in the file is available when you interact with a terminal or a terminal-based software. `.bashenv` is used in non-interactive cases.

A cool feature of environment variables is that they can be used as arrays. To set multiple values for an environment variable, just list them in parentheses, with values separated by spaces: `mytest=(one two three four five)`. To select an item: `echo ${mytest[2]}`. In the previous example, `mytest` is a local variable.

Info. Any variables set but not exported by the parent shell are local variables. Local variables are not inherited by a subshell.

4.2 Command substitutions

One of the most useful features of shell scripts is the ability to extract information from the output of a command and assign it to a variable. After you assign the output to a variable, you can use that value anywhere in your script. This comes in handy when processing data in your scripts. The usual format is `$()`.

Question 4.2. Try the command substitution by assigning the output of the `date` command.

Question 4.3. First, install `fzf`, it is a fuzzy finder that has a terminal prompt to select one output. Create a script that `find` all file in a given directory, and `cat` the content of the file you selected with `fzf`.

In principle you can as well use command substitution with any codes you might have. For instance, a python script that outputs values in the terminal can be used using command substitution.

Question 4.4. Try the command substitution with a python script.

4.3 Conditions and loops

Conditions are useful to control your script behaviour. It can bypass unnecessary parts if a condition is met for instance. In bash script, you can find the traditionnal `if`:

```
  if [[ conditional_statement ]]

then

  command

fi
```

As for loops, you can use the `for`:

```
#!/bin/bash
for i in LIST
do
   echo "Welcome $i times"
done
```

Here `LIST` can be integers `1 2 3 4 5`, a list of files, the output of a command (using command substitution)...

Question 4.5. By combining both condition and loops, create a script that checks if files in a given directory have `.bak` files, and if not, the script creates them.

Question 4.6. Using command substitution, check the permissions of the .bak files from the previous script and if they are not equal to `644`, change them. (Hint: To extract the permissions use the command `stat -c '%a' FILE`)

Question 4.7. Be creative and find a practical script with the knowledge you have until now!

Often, you must iterate through items stored inside a file to process data. For instance, extract the name or values on each line and in a column. Next example, we will try to separate lines and items stored in `/etc/passwd` which list users, their user and group ID...

```
#!/bin/bash
for entry in $(cat /etc/passwd); do
    echo "Values in $entry"
done
```

Question 4.8. Try the previous script think of what is wrong with this script. (Hint: try to `cat` the file in your terminal and see what's the output)

By default, the *internal field separator* (the field tat separates two items) or IFS is space. Therefore when there is a space, the script reacts as if there is a new line. Instead, we want to separate items when there is actually a new line. It's done via defining the variable `IFS=$'\n'`.

```
#!/bin/bash
IFS=$'\n'
for entry in $(cat /etc/passwd); do
    echo "Values in $entry"
done
```

Question 4.9. Add a new nested for loop and try to isolate elements of each line.

4.4 Parameters and options

The goal of this section is to have more advanced knowledge on how to handle parameters and inputs. To iterate through input parameters, numbers are given to the position of the input parameters. `$1` is the first positional parameter, `$2` is the second etc. There is another great way to iterate through command line parameters, especially if you don't know how many parameters are available. You can just operate on the first parameter, shift the parameters over, and then operate on the first parameter again. The command is called `shift`. Here is an example of the shift command :

```
#!/bin/bash
echo
count=1
while [ -n "$1" ]
do
```

```
    echo "Parameter #$count = $1"
    count=$[ $count + 1 ]
    shift
  done
```

Question 4.10. Try the code above and understand how it is working

Often you'll run into situations where you'll want to use both options and parameters for a shell script.

```
#!/bin/bash
echo
while [ -n "$1" ]
do
  case "$1" in
    -a) echo "Found the -a option" ;;
    -b) echo "Found the -b option";;
    -c) echo "Found the -c option" ;;
    --) shift
    break ;;
    *) echo "$1 is not an option";;
  esac
  shift
done
#echo the parameters
count=1
for param in $@
do
  echo "Parameter #$count: $param"
  count=$[ $count + 1 ]
done
```

Question 4.11. Try the code above and understand how it is working. Take good attention to the `case` command as we will use it in later exercices.

Question 4.12. Modify the script above so that options require a parameter (`./script.sh -a param1 -b param2`) instead of separating options and parameters with `--`

File Descriptor	Abbreviation	Description
0	STDIN	Standard input
1	STDOUT	Standard output
2	STDERR	Standard error

Figure 6: File descriptions for STDIN, STDOUT and STDERR as well as their corresponding numbers.

5 Scripts control

So far, you've seen two methods for displaying the output from your scripts:

- Displaying output on the monitor screen
- Redirecting output to a file

Both methods produced an all-or-nothing approach to data output. There are times, however, when it would be nice to display some data on the monitor and other data in a file. For these instances, it comes in handy to know how Linux handles input and output so you can get your script output to the right place.

The Linux system handles every object as a file. This includes the input and output process. Linux identifies each file object using a file descriptor. The file descriptor is a non-negative integer that uniquely identifies open files in a session (see figure 6).

5.1 Output and input redirections

The `STDIN` file descriptor references the standard input to the shell. For a terminal interface, the standard input is the keyboard. However, you can also use the STDIN redirect symbol to force the cat command to accept input from another file other than STDIN. It uses the < symbol.

```
1       cat < file
```

The `STDOUT` file descriptor references the standard output for the shell. On a terminal interface, the standard output is the terminal monitor. All output from the shell (including programs and scripts you run in the shell) is directed to the standard output, which is the monitor. We've already used it, as the symbols are > or >>.

```
1       ls -l > file
```

Lastly, the STDERR file descriptor references the standard error output for the shell. This is the location where the shell sends error messages generated by the shell or programs and scripts running in the shell. By default, the STDERR file descriptor points to the same place as the STDOUT file descriptor (even though they are assigned different file descriptor

values). This means that, by default, all error messages go to the monitor display. To redirect errors only to a file, the command has to be followed by the file descriptor number 2. For instance:

```
ls -al badfile 2> file
```

Errors are redirected to `file`.

Question 5.1. Try the `ls -al` command with files that exist and do not exist and redirect both data and error to files. Using one of your previous script, try to implement error and data redirections to files.

To permanently redirect data and errors using `exec 1>data_out` and `exec 2>error_out`, which will redirect data to `data_out` and errors to `error_out`.

Question 5.2. Modify one of your previous scripts to implement these kinds of redirections.

These redirections are useful to create data and error logs

If you have lots of data that you're redirecting in your script, it can get tedious having to redirect every echo statement. Instead, you can tell the shell to redirect a specific file descriptor for the duration of the script by using the exec command by adding `exec 1> data_out` at the very top of your script to redirect data, and use `2` instead of `1` for errors.

Until now, we have redirect outputs. In the same fashion, it is possible to redirect inputs. The first way is to wait for the user to input from the keyboard. It is done using the `read` command.

```
#!/bin/bash
read -p "Enter your name: " first last
echo
echo Hello $first $last, welcome to my program.
```

The second way is to permanently redirect inputs from a file as we did for outputs.

The exec command allows you to redirect STDIN from a file on the Linux system: `exec 0< input_file`

Question 5.3. Using the knowledge on redirections, try to redirect outputs and inputs : first using the terminal and your keyboard and second, permanently with files. Use any scripts you want to realize that exercice.

Question 5.4. As a second script, using a while loop, create a prompt that asks for a number to be input. Create a stream of output by displaying in the STDOUT if an integer is even and in the STDERR if the integer is odd. Use file descriptors to redirect the streams.

Question 5.5. Let's do a practical script to populate a database using MySQL. Review the following code and create a .csv file to work with this script

```
#!/bin/bash
outfile='members.sql'
IFS=','
while read lname fname address city state zip
do
  cat >> $outfile << EOF
  INSERT INTO members (lname,fname,address,city,state,zip) VALUES
  ('$lname', '$fname', '$address', '$city', '$state', '$zip');
EOF
done < ${1}
```

You can tweak the script to increase your productivity with working with SQL tables!

5.2 Automation

To automate scripts, it's usually done via the system scheduler. In most modern Linux distributions, it's done via systemd.

Here is an example of how to do this:

- Create a script file with the commands that you want to automate. For example, let's say that the script is called "myscript.sh" and it is located in the /home/USER/scripts directory (Replace user by your actual user).
- Create a systemd service file that will run the script. This file should be named something like "myscript.service" and it should be located in the /etc/systemd/system directory.
- Open the service file in a text editor and add the following information:

```
[Unit]
Description=My Script Service

[Service]
Type=oneshot
ExecStart=/bin/bash /home/user/scripts/myscript.sh

[Install]
```

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

This service file tells systemd to run the script as a one-shot service, meaning that it will run once and then exit. The ExecStart line specifies the path to the script that you want to run.

Reload the systemd configuration with the command "systemctl daemon-reload"

You can then schedule the execution of the script using the "systemd timer" feature, for that you need to create a timer unit file, for example: "myscript.timer" and put it in the same directory as the service file, with the following content:

```
[Unit]
Description=Run MyScript every 5 minutes

[Timer]
# Timer settings
OnBootSec=5min
OnUnitActiveSec=5min
Unit=myscript.service

[Install]
WantedBy=timers.target
```

This will make your script to run every 5 minutes after the system starts or the previous execution of the script finishes.

Enable and start the timer using the commands "systemctl enable myscript.timer" and "systemctl start myscript.timer"

Now, your script will run automatically every 5 minutes, and you can check the status of the script, timer and service with the commands "systemctl status myscript.service", "systemctl status myscript.timer", and you can stop and start them with the commands "systemctl stop myscript.service" and "systemctl start myscript.service" respectively.

Question 5.6. Try to automate the execution of a script every minute that backups the content of a folder to a different place.

6 Functions

As you start writing more complex shell scripts, you'll find yourself reusing parts of code that perform specific tasks. Sometimes, it's something simple, such as displaying a text message and retrieving an answer from the script users. Other times, it's a complicated calculation that's used multiple times in your script as part of a larger process. In each of these situations, it can get tiresome writing the same blocks of code over and over in your script. It would be nice to just write the block of code once and be able to refer to that block of code anywhere in your script without having to rewrite it : It's where functions come handy!

6.1 Create functions

There are two formats you can use to create functions in bash shell scripts. The first format uses the keyword function, along with the function name you assign to the block of code:

```
function function_name {
  commands
}
```

The second format for defining a function in a bash shell script closely follows how functions are defined in other programming languages. As an exemple, a simple function printing `hello, world` is:

```
hello_world() {
   echo 'hello, world'
}
```

To call functions, simply use its name, for instance `hello_world`

You can as well define and use variables as usual in those functions. Try the following script:

```
var1='A'
var2='B'

my_function() {
  local var1='C'
  var2='D'
  echo "Inside function: var1: $var1, var2: $var2"
}

echo "Before executing function: var1: $var1, var2: $var2"

my_function

echo "After executing function: var1: $var1, var2: $var2"
```

Question 6.1. Execute this simple program and explain what `local` is doing (put this in a shell script and execute the script).

To return values or strings, you can use the following method:

```
my_function() {
  local func_result="some result"
  echo "$func_result"
}

func_result="$(my_function)"
echo $func_result
```

Here the local variable `func_result` is created, and you can print to STDOUT the result using `echo`. Then you can pass the result of the function to a global variable using `$(my_function)`. In the same way, it is possible to return the variable using the command `return` directly, instead of using `echo`.

Lastly, to have functions asking for an input, you can use the following:

```
greeting () {
  echo "Hello $1"
}

greeting "Joe"
```

Which is very similar to the positional argument used in scripts.

6.2 Use of functions

The default way to check the exit status of a function is to use the `$?` variable after the function has executed. This variable holds the exit status returned by the last command within the function.

Question 6.2. Write a script with two functions, one which outputs a correct value and one which outputs an error. Check the exit status of each functions.

Functions provide a way to organize and compartmentalize code, making it easier to understand and debug. By breaking code down into smaller, self-contained units, it becomes easier to isolate and fix problems.

As in other programming languages, libraries of functions can be written in a separate file. In the main script, you library can be called using `source /path/to/the/library`

Question 6.3. Create a library of functions and a script. The script will ask you to select from a list a folder to backup. The list will be built from a `find` function and displayed using `fzf`. Think of creating a robust script that organizes well the backups (for instance, don't put backups from different folders into a single one!). Use file descriptors and output the results and errors into log files.

6.2.1 Select from a menu

One interesting use case of bash scripts and functions is to create menu and select from a menu to perform actions.

The function of the menu could look like this :

```
function menu {
clear
echo
echo -e "\t\t\tSys Admin Menu\n"
echo -e "\t1. Display disk space"
echo -e "\t2. Display logged on users"
echo -e "\t3. Display memory usage"
echo -e "\t0. Exit program\n\n"
echo -en "\t\tEnter option: "
read -n 1 option
}
```

Question 6.4. Try the function above to understand how it works. Then, create a script to monitor your system using functions from the menu above.

Info. To select from the menu and perform the selected action, you could do a `while [ 1 ]` loop with a `case` selection that you have already seen before! The selection `0` will `break` to exit the loop. Disk space is done via `df -k`, logged on users `who`, and memory usage `cat /proc/meminfo`

6.3 Declaring global functions

One of the main disadvantages of defining shell functions directly on the command line is that they are not persistent and will disappear when the shell is closed. This can pose a problem for more complex functions that are intended to be used multiple times.

It is possible to create functions directly in the `.bashrc` file in the home directory, however it is important to be cautious when editing this file as it may already contain some defined items. The recommended approach is to add your own functions at the bottom of the existing file.

Here's an example of doing that:

```
# Source global definitions
if [ -r /etc/bashrc ]; then
. /etc/bashrc
fi

function addem {
echo $[ $1 + $2 ]
}
```

After modifying the `.bashrc`, in an existing shell, use the `source ~/.bashrc` command to update your shell. An alternative is to start a new shell.

Question 6.5. Create a function in your `.bashrc` that updates your `PATH` variable. To update the `PATH` you need to append a folder at the end of the `PATH` variable: `export PATH=$PATH:/path/to/add`.

Remark 6.1. BE CAREFUL and copy the content of the `PATH` variable before modifying it! (`echo $PATH`)

7 Practical scripts

From now on, you are almost all set to be free to create any script you want to be more productive with your system. Scripts are useful to automate, and gain producitivity during your day. Either as an user, as an admin or a developper. In this section, the goal is to have example of practical scripts.

7.1 Archives and zip files

An operating system always need backups. Especially to save important files when you want to update hardware and need to reinstall the whole operating system, you want to reuse part of the configuration you have. It is pretty easy to copy the important files, but to be more productive and to not forget, you can automate this process.

Question 7.1. Create a script that backup files (take a random folder located at `/home/USER/.config` or take any files you find important) and create a copy into destination folders based on `months` and `days`.

Question 7.2. Complete the script above by creating zip files of the content. The name of the zip file should indicate the hour and minute the backup has been made (for instance : `archive_1053` if it has been created at 10:53 AM)

Question 7.3. Automate this process to create regular backups (every hour, or every day at 8:00 AM...)

The folders structure should resemble to this :

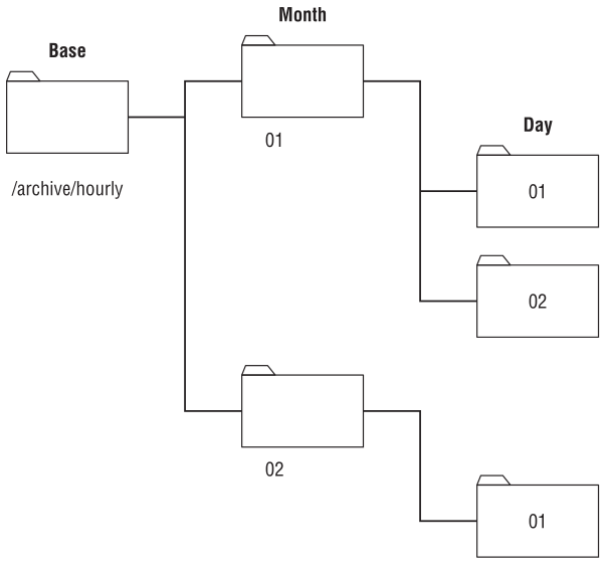

Figure 7: Folder structure for the archive and zip files script

7.2 Finding files

As the time goes on, you accumulate a lot of files and sometimes, that can be difficult to find the proper file. A computer can be extremely efficient in finding files, especially if you know what filetype you want or what is its name.

Question 7.4. Create a command-line script that ask for a particular filetype to be found on your computer. The results is passed to `fzf` so you can select and filter which file you want to use. Based on that, pipe the selection to open the file using the default file opener for this extension (Hint: `xdg-open`). Alternatively, you can use `case` to create specific cases if you know filetypes you want to find.

7.3 Scraping from URL

When it comes to shell script programming, the internet is not typically the first thing that comes to mind. The command line environment can seem disconnected from the dynamic, graphical world of the internet. However, there are a variety of tools that can be utilized in shell scripts to access data on the web and other network devices. These utilities can bridge the gap between the command line and internet, allowing you to access web content in your shell scripts. The Lynx program, which is almost as old as the internet itself, was developed in 1992 by students at the University of Kansas. It is a text-based browser that allows you to browse websites directly from a terminal session. Here's the basic format of the lynx command: `lynx options URL` where URL is the HTTP or HTTPS destination you want to connect to, and options are one or more options that modify the behavior of Lynx as it interacts with the remote website.

When you incorporate Lynx in a shell script, the main purpose is typically to extract a specific piece of information from a webpage. This process is known as "screen scraping", where the goal is to locate and capture data from a specific location on a webpage and use it in your shell script. It is a technique used to programmatically retrieve data from a graphical screen in order to automate certain tasks. In order to *download* a webpage into `STDOUT`, the option to use is `-dump`.

Question 7.5. Try to dump the weather page of Lille from weather.com : https://weather.com/fr-FR/weather/today/l/db749df24acdde958fc5a2c673b6ba1017b235853163a3c928af67f08127401e

The next thing is capture the temperature from today's forecast. We will look for `Matin` for instance. We can filter information using `sed`. Create a `matin.sed` file and put

```
/Matin/{
n
p
}
```

This sed script will look for Matin, skip a line and take the prompt. After you dumped the page in your STDOUT, pipe the result into the sed command using sed -n -f matin.sed. Verify that you have the temperature of today's morning.

Question 7.6. Try to scrap the temperature of the afternoon. (Hint: To remove space in a variable you can echo the variable and pipe it into tr -d '[:space:]')

Question 7.7. Create a script that scrap information about today's forecast in Paris and Marseille and display the results in the STDOUT in a intelligible way.

Now, you now how to screen scrap information about most of websites on the internet!

7.4 Using graphical softwares with your scripts

The terminal and the command lines are not where we most spend our time on. Instead, we mostly use graphical interface thanks for their usefulness and easiness of use. A graphical interface can present result in a much more beautiful way. It can present as well results of a script. In this section, you will interact with rofi and your bash scripts. Rofi is a lightweight, highly customizable, and easy-to-use application launcher and window switcher for Linux and Unix-like operating systems. It is a free, open-source software project that is written in the C programming language. Rofi can be used as an application launcher, a window switcher, or both. It allows users to quickly launch applications, switch between open windows, and perform other tasks done by shell scripts

In order to pass information from a script to rofi, a simple script can be as follows :

```
1       ./my_script.sh | rofi -dmenu
```

Question 7.8. Using rofi and bash scripts, create a script that prompt, first, the type of extension file you want to find on your computer (for instance *.txt,.pdf,.doc*...). Then when you select the file, open it with the default application.

For the next script, we will use the conversion software pandoc that can convert files between different extensions. For instance, it is useful for translating a *.md* (Markdown) into a *.pdf* using latex. It can be used as well to convert a *.pdf* into a *.docx*... It is almost limitless!

Question 7.9. Create a script using rofi that list all the files you want to convert. The script will prompt which extension you will convert the file to.

8 Project

8.1 Search into PDF files

The goal of this project is to search information and text in PDF files. As we digitalize more and more documents, we can end up with hundreds or even thousands of documents. In all these documents, important information or text need to be retrieved. Instead of searching manually, we can ask our computer to do it instead. As an example, a researcher wants to look for a scientific paper that has been written by a specific person. By using a script, the researcher can input the name of the author and the computer will search in the database all the papers containing the name of the author.

The task for this project is the following:

- First, the database will be based on a configuration file (placed in the folder `/home/USER/.config/pdf_search/`). This configuration file will contain all the folders to look into.
- Create a script that is interfaced with `rofi` (section 7). The `rofi` prompt will ask for the keyword to search in the PDF database.
- The results will be dumped into a text file and a new `rofi` prompt will present the results.
- By selecting a PDF in this new `rofi` prompt, this will open the PDF.
- As the results are in a text file. That would be interesting to be have a history of previous search. You could then create a menu asking to search for new keywords or present previous results.
- The script will monitor errors as well. It will create a error log file to monitor all potential errors (search not going through, error in the config file, cannot read a pdf...).
- Bonus: if `rofi` is not installed, prepare a command-line version of the script.
- Hint: To search into PDFs, you can use `ripgrep-all` or `pdfgrep`.

Write the script using what you have learned during the module: Use functions, use redirections and command substitutions...