Practical work 1 - Seashell on the seashore

Duration	Groups	Evaluation
4.5 hours	Solo	PDF report on Edunao

Context and instructions

- You're a ~~system administrator~~ renowned Captain, Linux is your ship, the command line your sails, and you like picking up seashells on the sea shore.
- Read instructions carefully.
- For each exercise, test commands and read associated documentation. There is **purposefully** not a lot of information on how commands should be used. **Please refrain from pasting everything into your favorite LLM and take the time to understand how to use commands and read the associated documentation.** Your future-self will appreciate it.
- Today's manoeuvres will require creating directories. You are free to create them **somewhere** on your file system. Suggestion: in your `$HOME`.

To hand in your work:

- Create a **pdf** document (no docx);
- It MUST contain **your name**;
- Upload it on Edunao.

Tips & tricks about the Shell

To open a terminal: `CTRL` + `ALT` + `T`. Else, use the search bar and type `term`.

Most shells provide some basic features to help you write commands faster.

- Press `TAB` multiple tabs for **autocompletion**.
- `CTRL` + `L` to **clear the screen**.
- `UP ARROW` and `DOWN ARROW` to **navigate the command history**.
- `CTRL` + `R` to **search** commands in the history.
- `CTRL` + `SHIFT` + `C` to copy, `CTRL` + `SHIFT` + `V` to paste.

To get help / more information about a command, use:

```
man mycommand
mycommand --help
```

Starting

By default, we are going to use MyDocker. This means **files will be deleted** soon after the class. Please **back up any file / script** you wish to keep. This can include your command history (see following exercices).

To connect, open a terminal on your system and use the command line provided by MyDocker. You will need to edit it to change the user with "captain":

```
ssh captain@[...]
```

Finally, use the custom password provided in the MyDocker interface.

Alternatively, you **could** use your own Linux system, a virtual machine, or a local Docker container running the following configuration: https://gitlab-research.centralesupelec.fr/samuel.pelissier/r5cybgos-mydocker (this will use the default captain:captain user:password).

Exercise 1: navigation

The first thing a captain should know is to navigate around its ship.

Relevant commands:

```
cd dir # go into dir
cd .. # go back to the parent directory
cd ../.. # go back 2 levels
cd ~ # go to the current user $HOME directory
cd / # go to the root
ls
pwd
mkdir
```

If you do not know what a command does, check its documentation (sailors like old manuscripts, and so should you).

a. What is the first option documented in the manual of the `ls` command?

b. When you connect / open a terminal, what is the current directory? Which command did you use to get this information?

Navigate where we will work for today (suggestion: somewhere in your `$HOME` directory). Create a directory named `tpos1` and navigate into it.

c. Type `cd`. In which directory are you now? Type `cd -`. What happened?

d. How can you list all the files, including hidden files, in a the directory?

If required, go back into the `tpos1` directory. Here, we want to create a subdirectory, with another directory inside it, such as: `tpos1` → `subdir` → `anotherdir`.

e. What happens when you try `mkdir subdir/anotherdir`? What are the two ways of creating a directory in another directory? If needed, check the manual page for the `mkdir` command.

Exercise 2: a ship made of trees

Relevant commands:

```
tree
find
```

So far, we've listed files using `ls`, maybe with some of its options, for instance `ls -lah`. However, it focuses only on one directory, not subdirectories. A cool program named `tree` can be used to view the whole hierarchy.

If we try to use the command, we get the following error:

```
user@DebianNetInstall:~$ tree
bash: tree: command not found
```

> **a.** What are two possible reasons as of why this happens?

Fix the most obvious reason using `apt`. You may need to use `apt update` before searching for / installing the package. If you encounter a permission denied error, it means your current user does not have high enough privileges to run the commands. In that case, use `sudo apt update`.

Now, let's run it: `tree ~`. To view the entirety of all files and directories on your system, you can also run `tree /`.

> **b.** Compare the number of files and directories when running `tree /` with elevated privileges (i.e., as root) and without. Why is there a difference?

Let's investigate why this happens. To **find** files and directories with specific features, we can use the... `find` command!

Using the manual for the `find` command, find the options required to a) select only directories, and b) only executable directories (i.e., those our captain user can access). To navigate inside the manual, you can use `h` to display the summary of helpful commands, and `/` to search for specific keywords, e.g., `/executable`. To navigate between occurences, use `n` (next) and `N` (Nprevious).

> **c.** What command did you use? Cite one example of a directory without executable rights for our current user.

> **d.** Run both `ls` and `tree` on one of the directories we've just found. What are the error messages and differences in behaviors? What possible issue does this create?

Hint: you can also look at the error code of the previous command using `echo $?`.

Exercise 3: moving freight

Ships are used to convey goods, and you're no stranger to moving shipments around.

Relevant commands:

```
mv
cp
ln
```

First, let's create some files corresponding to your freight.

```
mkdir dock
mkdir deck
echo "rum" > dock/barrel
```

We are now ready to load a barrel of our finest liquor from the harbour's dock to the deck of the ship. Don't hesitate to check the content of each folder using `ls`.

In practice, the harbour contains multiples docks, corresponding to multiple folders: e.g., `dock`, `dock2`. Instead of using the `mkdir` command, **copy** the content of the `dock` folder into `dock2`.

> **a.** What option is required to copy a directory?

We will now **move** the barrel from one of the docks to the deck.

> **b.** What command should you use to move the barrel? Is it still in the `dock` directory?

Finally, and please suspend your disbelief for this one, we would like to create a symbolic link from the barrel in `dock2` to one on the deck.

> **c.** What commands are required? Using `ls`, how is the link materialised?

> **d.** What happens if you move or delete the original file a symlink points to?

Exercise 4: our first script

Using the interactive command line is fine, but sailors would rather automate tasks and have everything neatly organized on the ship. It is now time to create our own command. Our goal is to use a script reading a file containing the position of a secret treasure. This file should be readable only by its owner (you, the Captain), not other sailors on the ship ~~users on the system~~.

We will **not** use a graphical text editor (e.g., VSCode) for this part. Indeed, some ships ~~servers~~ may not come with a graphical environment and it is important to be able to edit files with default tools. In this exercise, we will use `nano`.

Relevant commands:

```
chmod
cat
echo
ls
nano
```

First, create a file named `script.sh` containing the following insanely complex code. Note: when using `nano`, the `^` in shortcuts displayed at the bottom of the screen corresponds to `CTRL`.

```
#!/bin/bash

echo "full sail!"
```

Set the executable permission then run the script using `./script.sh`. It should print "*full sail!*".

> **a.** What are the two (2) ways of setting the executable permission on your script?

> **b.** The first line corresponds to the shebang (`#!/bin/bash`). Why is it needed?

A captain is nothing without their log; we would like to print the command history. As said in introduction, you can access previous commands via arrows. Additionally, the `history` command displays all the previous commands, which can be saved in a file named `~/.bash_history`.

To test the various ways of displaying the history:

- Use the `history -a` command (which sets up the `~/.bash_history` file if it does not exist).
- Disconnect (using `exit` or `CTRL` + `D`) and reconnect.

Finally, display the history and update `./script.sh` so instead of printing "*full sail!*", it prints the full command history.

> **c.** What is a security risk associated with having access to the command history?

Now, create a file named `secret.txt` containing the secret location of your treasure on its first line.

> **d.** Based on the *relevant commands* provided earlier, you have two ways of writing the secret string to the file. Which are they? Following up question (c), which one should be preferred for security reasons?

Modify `./script.sh` so it prints the content of `secret.txt`.

> **e.** Currently, who can read the content of `secret.txt`? What command should you run to protect this file (i.e., to make sure only the owner can read/write it)?

Exercise 5: who's the real captain?

A complex ship such as yours runs numerous processes, started by various crewmates. We will explore how Linux' philosophy of **everything is a file** and its role in maintaining process protection can be illustrated.

Relevant commands:

```
sudo
sleep
ps
kill
top
ps
```

On your ship, some crewmates like to take a little nap sometimes, as a treat. Create a script named `sleep_pid.sh` containing the following code:

```bash
#!/bin/bash
# Start sleep for 60 seconds in the background
sleep 60 &
# Capture the PID of the background process
PID=$!
# Output the PID
echo "Sleep process started with PID: $PID"
```

... and of course, as a tyrannical captain, you like to wake them up by stopping the corresponding process.

We will run this script and retrieve the PID of the sleep process. Linux exposes process information through a virtual filesystem at `/proc`. Hence, you can check the corresponding folder with: `ls /proc/$PID/` (replace `$PID` by the integer returned by `sleep_pid.sh`).

For the following question, run `sleep_pid.sh` with and without `sudo` and compare results.

> **a.** What are the permissions differences between `/proc/$PID/exe`? Who's the owner of the process started using `sudo`?

> **b.** If you try to stop the process (i.e., using `kill`), do you have the permission to do so?

Exercise 6: a new crewmate

Now that we saved the treasure location in a file, we want to make sure no other sailor can access it. So far, you **should** be the only sailor on your ship ~~only user on the system~~, which is a bit sad.

Hence, we will recruit a new crewmate, *Bob*.

Relevant commands:

```
groups # list the groups of the current user
whoami # print the current user
useradd
passwd
usermod
su
groupadd
chgrp
```

In this exercise, we will use both `sudo` and `su`. Important difference: `sudo` allows you to execute a single command with elevated privileges, whereas `su` switches your entire session to another user.

As Bob probably does not have an account your ship, the first step is to create it (this step may require `sudo`). Do not forget to set their password as well.

Additionally, both Bob and you should belong to the group *crewmate*.

Once the account is created, you can now switch to Bob's account by using the following command:

```
su - bob # su stands for Substitute User (or Switch User if you will)
```

In this terminal session, you now are connected as Bob and can test commands as this user. You can test that by using `whoami` and `groups`.

> **a.** What happens when you try to read the `secret.txt` file when connected as Bob?

Times has passed and Bob has proved to be a loyal and skillful sailor. You decide to allow him to read the `secret.txt` file containing th treasure location while keeping access for yourself.

> **b.** What commands should you use to do so?

> **c.** If you used a command allowing all crewmates to read the secret file, what could you have done instead to keep it readable only for you two?

Exercise 7: pipes and redirections

Being a captain does not mean you should ignore know how your engine works. We'll do some plumbing and demonstrate how **pipes** (`|`) and redirections (`>` or `>>`) behave.

Relevant commands:

```
grep
cat
sort
wc
```

Create a first script called `parots.sh` that outputs the names of parrot species:

```
#!/bin/bash

echo "Amazon Parrot"
echo "Caique Parrot"
echo "Cockatiel"
echo "Cockatoo"
echo "Conure Parrot"
echo "Electus parrot"
echo "Lovebird"
echo "Macaw"
echo "Parakeet"
echo "Parrolet"
```

Our goal is to filter this outputted list using `grep` to only select our favorites (those containing "parrot").

> **a.** How can you filter species containing "Parrot"? How can you filter species containing "parrot"? (Note the lowercase **p**.) How can you filter species containing both lowercase and uppercase versions of "parrot"? Should "*Parrolet*" be listed as well?

Now that we know how to select a sublist of parrot species containing *parrot*, we would like to order them in reverse alphabetical order.

> **b.** What **one-line command** did you use to both select the sublist and order it?

We would like to save the result of the previous command into a file named `favorite_parrots.txt`.

> **c.** What command should be used to do so? If you run the command again, is the content of the file identical, or did you append new text at the end? Mention which one you chose and the alternative.

Finally, we would like to know the *number* of our favorite parrot species.

> **d.** Suggest two ways of counting the parrot species: a) a one-liner piping and combining the various commands, and b) an alternative using `favorite_parrots.txt` directly. Intuitively, what are the pros & cons of each approach?

Exercise 8: from the figurehead to the stern

Remember when we created a cool script to print the position of our secret treasure? Turns out, it is not practical to type `~/Documents/tpos1/script.sh` everytime we want to know where to go. As the Captain, you want to be able to call this scripts wherever you are on the ship. Similarly to `ls`, we won't need to type `/usr/bin/ls`.

Relevant commands:

```
# Refresher: slides 15 to 19 of lecture 3
printenv
which
```

> **a.** When typing a command, where is the shell looking for executables/scripts? What are the current possible values?

First, let's rename `./script.sh` to `./printmap` in its current directory so its name is more evocative. You can try to execute it; it should still work as usual.

Second, we want to make it available as a global command. To do so, we can either a) move it in one of the paths discovered previously, or, and this is what we'll do, b) add it to the `/opt/` directory (for `opt`ional packages, not required for the system to run).

> **b.** Is `/opt/` one of the places where the shell looks for executables? What is required to make `printmap` available as a global command?

Proceed accordingly. You should now be able to run `printmap` from anywhere in the system (note that, similarly to other system commands such as `ls`, there is no `./` prefixing `printmap`).

> **c.** Disconnect and reconnect (using `exit` or `CTRL` + `D`). Is the command still available? Why? If not, update your setup to make it available everytime you connect.

> **Bonus.** While you're in `~/.bashrc`, cite one alias already defined. If you were to define an alias in this file, would it be available on other machines you connect to?

Exercise 9 (bonus): a custom backup solution

It is common knowledge among sailors that navigation logs are important. Hence, a good Captain regularly saves their data.

Relevant commands:

```
zip
find
jq # might require an installation
date
```

Create a script that:

- Lists files in the current directory.
- Loops through them and selects only files with the `.log` extension (prepare a test environment accordingly).
- Creates an archive (e.g., a zip file) containing these files.
- Moves the archive to a `backup` folder.

> **Improvement 1.** Instead of using the current directory, use a parameter such as `./backup.sh "/path/to/dir"` backs up `/path/to/dir`.

> **Improvement 2.** The created archive should use the current date and time as filename.

> **Improvement 3.** Add your script as a cronjob that runs every day at 1am.

> **Improvement 4.** Instead of backuping `.log` files only, use a JSON configuration file containing a list of extensions that should be backed up. Pass only the filename as a parameter such as: `backup.sh "/path/to/config.json" "/path/to/dir"`.

> **Improvement 5.** Create another script that automatically deletes any file in the `backup` directory that is older than 1 month. IMPORTANT: this is a destructive operation. Make sure that it does not delete more than expected.

> **Improvement 6.** Add color to your script outputs.