TD 4 - ISS

Exercice 1.

1.1

L'output de ce code est

```
a
1
b
2
c
3
```

1.2

1.

```
i@pc /home/moi $ ./testFlux > /tmp/fichier1.txt
```

Cette commande va écrire `a b c` sur `fichier1.txt` et écrire les erreurs dans la sortie standard. 2.

```
moi@pc /home/moi $ ./testFlux 2> /tmp/fichier2.txt
```

À l'inverse, dans ce cas-là, les erreurs seront écrites dans `/tmp/fichier2.txt` et la sortie standard sera affichée sur le terminal.

3.

```
moi@pc /home/moi $ ./testFlux > /tmp/fichier3.txt 2> /tmp/fichier4.txt
```

Aucun élément ne sera affiché sur le terminal. Les éléments imprimés sur stdout seront dans `/tmp/fichier3.txt`, et les erreurs seront dans `/tmp/fichier4.txt`.

4.

```
moi@pc /home/moi $ ./testFlux > /tmp/fichier5.txt
```

Création et sauvegarde des éléments imprimés sur la sortie standard dans `fichier5.txt`.

5.

```
moi@pc /home/moi $ ./testFlux > /tmp/fichier5.txt
```

Écrasement du fichier créé dans 4. et écriture des éléments qui sont imprimés sur la sortie standard.

6.

```
moi@pc /home/moi $ ./testFlux > /tmp/fichier6.txt
```

Création et sauvegarde des éléments destinées à être printées sur stdout dans `fichier6.txt`.

7.

```
moi@pc /home/moi $ ./testFlux >> /tmp/fichier6.txt
```

Comme `fichier6.txt` est déjà existant et rempli, cette commande écrit la sortie standard de `testFlux` depuis la fin de `fichier6.txt`.

1.3

Rien, parce que printf envoie au stdout par défaut.

1.4

Dans ce cas, toutes les erreurs seront envoyées au stdout.

1.5

On veut uniquement `abc`, donc il faut rediriger les erreurs `123` vers le trou noir.

```
$ ./testFlux 2> /dev/null
```

1.6

Le `2>` écrase le fichier existant, il faut utiliser `2>>` à la place pour que les erreurs soient écrites à la fin.

1.7

Uniquement les éléments imprimés sur `stdout`.

1.8

Si l'on considère la commande de la question 7, alors `abc`.

1.9

```
$ ./testFlux > /tmp/fichier_test.txt 2>> /tmp/fichier_test.txt
```

Si l'on veut tout dans le même ordre :

```
$ ./testFlux >& /tmp/fichier_text.txt
```

1.10

Comme prévu, seulement les éléments sont sur `stdout`, confirmant que le processus fils hérite les flux du père.

Exercice 2.

2.1

```
#! /bin/bash

# longest_param.sh

if [[ ${#} -eq 0 ]]; then
    echo "Usage ${0} : <param1> <param2> <...>"
    exit
fi

res=""
for x in ${@}; do
    if [[ ${#x} -gt ${#res} ]]; then
        res=${x}
    fi

done

echo ${res}
exit
```

2.2

Cette fonction marche parce que, comme on n'utilise pas de `" "` dans `${line}`, Bash réalise une expansion en utilisant `$IFS`.

```
#! /bin/bash

#longest_word.sh

while read line; do
    ./longest_param.sh ${line}
done

exit
```

2.3 et 2.4

Ça marche pas, il faut utiliser

```
$ ./longest_word.sh < proverbe.txt
```

2.5

```
#! /bin/bash

#longest_word.sh 2.5

if [[ ! -f $1 ]]; then
    echo "Faut passer un fichier comme paramètre"
    exit 1
fi

while IFS= read line; do
    ./longest_param.sh "${line}"
done < ${1}

exit 0
```

On utilise le `IFS=` (vide) comme ça pour le dire à bash de ne pas separer la ligne.

2.6

Parce qu'on à pas de communication entre le fils et le père.

2.7

```
#! /bin/bash

#longest_word.sh 2.7

if [[ ! -f $1 ]]; then
    echo "Faut passer un fichier comme paramètre"
    exit 1
fi
listeMots=""
while IFS= read line; do
    listeMots+=$(./longest_param.sh "${line}")
done < ${1}

echo "$(./longest_param.sh ${listeMots})"

exit 0
```

Summary

File descriptors

A file descriptor is a process-unique identifier (a handle) in the form of a non-negative integer, for files and I/O devices (physical or digital like pipes or network sockets).

The value returned by an `open` syscall is the file descriptor, and is an index into an array of open files kept by the kernel (per process), this is indexed in a system-wide array of all the files opened by all process called file table (recording the mode, read, write, etc and the offset of the file) this table also indexes into an inode table (that has the size, permissions and location on disk).

- File descriptors are stored in unix under the path `/proc/PID/fd`

Redirections

In UNIX there are three types of possible redirections

1. Redirection of I/O

Bash allows to manipulate the output streams of `stdin` (file descriptor 0), `stdout` (fd 1), and `stderr` (fd 2) using various symbols.

The child process inherits the flux of the father.

`stdin`

- `<` Redirects `stdin` so that a command or program reads from a specified file.

`stdout`

- `>` Redirects `stdout` to a file, creating it if it doesn't exist and overwriting its contents if it does.
- `>>` Appends `stdout` to a file, creating it if it doesn't exist.
- `|` Pipes `stdout` to another command as input.

`stderr`

- `2>` Redirects `stderr` to a file, creating it if it doesn't exist and overwriting its contents if it does.
- `2>>` Appends `stderr` to a file, creating it if it doesn't exist.
- `>&1` Redirects `stderr` to `stdout`, effectively merging the two outputs.

`stdout` and **`stderr`**

- `>&` Redirects both `stdout` and `stderr` to another file or file descriptor.
- `|&` Pipes both `stdout` and `stderr` to another command.

2. Command chaining

We can execute multiple commands in a sequential basis, either using an script or using the `;` symbol.

```
./test > fichier.txt; cat fichier.txt
```

3. Using Pipes

One of the most common way of interprocess communication (IPC), we distinguish between `anonymus pipes` and `named pipes` (aka FIFO), in the former the data is buffered by the OS until the other process reads it, in the later, the pipe is managed by the kernel and controls the state, buffer and permissions.