

---

# Révisions Bash

PRÉSENTÉ PAR LITEAPP, ÉMILE ET  
4DRAGALIA

---

# Le terminal

- Un terminal est une interface avec le système d'exploitation qui permet de configurer et de lancer des programmes
- Par défaut, il est aussi en mesure de lire de la donnée que l'on écrit dans des flux nommées stdout et stderr, et de les afficher à l'écran.

---

# Les processus

- Lorsque l'on lance un programme, l'OS crée ce qu'on appelle un "processus"
- Un processus contient:
  - Une zone de mémoire dédiée (stack, heap, le programme lui même)
  - Une sorte "d'état" (à quelle instruction on se trouve, quel est l'état des registres du programme)
  - Un identifiant unique (PID (process ID))
  - D'autres choses plus complexes...
- Le travail de l'OS est de faire tourner tout ces processus "en parallèle"

---

# Créer un processus

- Pour créer un processus, il suffit d'exécuter la ligne suivante dans un terminal:

```
le_programme argument1 argument2 argument3 ...
```

- Le premier paramètre est le “nom” du programme (un chemin vers ce dernier, si on ne le trouve pas, rien ne se passe)
- On peut mettre autant d'arguments que l'on veut, c'est au programme de comprendre ce qu'ils signifient. Chaque argument est séparé d'un espace.

---

# Les chemins

- Un chemin permet de se déplacer et de désigner les fichiers dans un système de fichiers. Il y a deux types de chemins: les chemins absolus et les chemins relatifs
  - Un chemin absolu s'écrit comme ceci:
    - `/mon/chemin/absolu` pour un chemin partant de la racine de l'ordinateur
    - `~/mon/chemin/absolu` pour un chemin partant de la racine de l'utilisateur (Attention, c'est différent!)
  - Un chemin relative s'écrit comme cela:
    - `./mon/chemin/relatif`
    - `mon/chemin/relatif`
    - Dépend de où on se trouve dans le système de fichier!

---

# Naviguer le système de fichiers

- Pour naviguer le système de fichier, on peut isoler 3 commandes indispensables:
  - ls (list directory contents): Imprime dans le terminal tous les fichiers dans le “current working directory” ou dans le directory spécifié dans un argument.
  - cd (change directory) change le “current working directory”
  - pwd (print working directory): donne le chemin (path) absolu du “current working directory”

---

# Les types de fichiers

- Directories (un répertoire)
  - Des cas particuliers:
    - "." la directory actuelle
    - ".." la directory parent
- Les fichiers (de la donnée brute: du texte, un programme, une image...)
  - Des cas particuliers:
    - Symlinks/Hardlinks
    - Sockets/ Pipes
    - On parle pas de /dev

---

# Les links

```
ln -s source output
```

Symlink (target va juste pointer vers source)

---

# La variable \$PATH

- Il existe une variable globale associée au terminal que l'on appelle PATH qui contient des chemins. Tous les fichiers de ces chemins sont accessibles depuis n'importe où dans le terminal.
  - Exemple: /bin (ou se trouve ls, mv, cp, mkdir, echo, kill...)

---

# Les scripts bash

- Un fichier comme un autre (par convention, ces fichiers finissent en .sh)
- Regroupe une liste de commandes à exécuter
- Peut être exécuté par un utilisateur (comme un programme\*)

---

# Créer un fichier bash

- Étape 1: créer le fichier
  - Par exemple: **touch** my\_bash\_file.sh
- Étape 2: ouvrir le fichier et écrire ce que l'on veut
  - Par exemple: 

```
#!/bin/bash  
echo "Proot"
```
- Étape 3: autoriser l'exécution du fichier
  - Par exemple: **chmod** +x my\_bash\_file.sh (par défaut, on peut pas exécuter n'importe quel fichier sur l'ordi ! (encore heureux))

---

# Créer un fichier bash (2)

- Étape 4: lancer le programme
  - Par exemple: `./my_bash_file.sh`

---

# Les variables en bash

```
#!/bin/bash

# PAS D'ESPACE AU NIVEAU DU SYMBOL '=' !!!
a=10

# Imprime la lettre a
echo a

# Imprime 10
echo $a

# Imprime rien (une ligne vide)
echo $b
```

---

# Le Branching

```
#!/bin/bash

number=$1

if [ -z $number ]; then
    echo "Give me a number!" >&2
    exit 1
fi

if [ $number -eq 42 ]; then
    echo "Based."
elif [ $number -eq 420 -o $number -eq 69 ]; then
    echo "Cringe."
else
    echo "Basic."
fi
```

```
#!/bin/bash

letter=$1

if [ -z $letter ]; then
    echo "Give me a letter!" >&2
    exit 1
fi

if [ ${#letter} -ne 1 ]; then
    echo "Just one letter please!" >&2
    exit 1
fi

case $letter in
    [a-z])
        echo "Small alphabetical character.;;"
    [A-Z])
        echo "Big alphabetical character.;;"
    [0-9])
        echo "Number gang";;
    *)
        echo "Something else";;
esac
```

---

# Quesque \$1?

- \$1, \$2, \$3 ... Les arguments passés au programme (\$0 le path spécifié du programme)
- \$# Le nombre d'arguments du programme
- \$? Le return code du dernier processus qui a fini de s'exécuter (0 si tout va bien)
- \$\$ Le PID du shell
- \$! Le PID du dernier programme mis en background
- \$\* renvoie la liste \$1 \$2 \$3 ...
- ...

---

# Les loops

```
#!/bin/bash

while [ $# -ne 0 ]; do
    echo $1
    shift
done
```

Le shift décale tous les \$1, \$2, \$3 etc... vers la gauche, et mets a la poubelle \$1... TRÈS PRATIQUE!!

```
#!/bin/bash

file=$1

if [ -z $file ]; then
    echo "I need a file path!" >&2
    exit 1
fi

while read -r line; do
    echo $line
done <"$file"
```

---

# Interagir avec les fichiers

- 3 opérations
  - > écrire en écrasant le contenu
  - >> ajouter à la fin du fichier
  - < lire le fichier

```
echo "rebonjour" >> fichier.txt  
echo "bonjour" > fichier.txt  
read salutations < fichier.txt
```

# Les flux

- 3 flux « standards » :
  - stdin (0) → lecture
  - stdout (1) → écriture
  - stderr (2) → écriture
- On peut en ajouter d'autres

exec [numéro]{>, >>, <, <>}[fichier]

Ouvre un nouveau flux

Pour lire/écrire sur un flux : **&**[numéro]

echo "bonjour" **>&4**

cat **<&4**

---

# Lecture avancée

```
exec 3<noms.txt
```

```
exec 4<prenoms.txt
```

```
while read nom <&3; do
```

```
    read prenom<&4
```

```
    echo bonjour $prenom $nom
```

```
done
```

- Ouvre les deux fichiers en lecture
- Lis une ligne de chaque fichier en alternant
- Affiche le résultat
  
- On ne peut pas faire ça simplement avec `while read var ; do ... done < fichier`

---

# Les pipes 🙄 (1/3)

mkfifo nomdefichier

echo bonjour > nomdefichier

read valeur < nomdefichier

- Crée un nouveau fichier tuyau
- Ecris dans le tuyau [ouvre – écrit – ferme]
- Lis une ligne [ouvre – lis – ferme]

Les ouvertures (lecture et écriture) sont bloquantes s'il n'y a personne au bout du fil.

---

## Les pipes 🙄 (2/3)

- **Problèmes** quand on ouvre et on ferme un tuyau en lecture quand quelqu'un écrit dessus.
- **Solution** : toujours garder un lecteur et un écrivain avec :  
mkfifo nomdufichier  
exec 4<>nomdufichier
- A partir de maintenant toutes les **ouvertures** sont **immédiates**.

---

# Les pipes 🙄 (3/3)

- « Chez Linux on a des petits programmes qu'on assemble pour faire des trucs cools, pas un gros machin compliqué »
  - *L'art de la guerre*, Sun Tsu
- Pour rediriger le flux 1 (stdout) de proc1 vers le flux 0 (stdin) de proc2 :

```
proc1 | proc2
```



# Quand on a plusieurs processus

C'est la cata

---

# Petite mise en situation

Fichier transaction.sh

  `#!/bin/bash`  
`read argent < comptes_minet.txt # lit`  
`argent=$(expr $argent + "$1") # traite`  
`echo $argent > comptes_minet.txt # écrit`

Fichier comptes\_minet.txt :

100

---

 `$ transaction.sh 50`      `argent = 160`

 `$ transaction.sh 1`      `argent = 100`

---

# Section critique (1/2)

Fichier transaction.sh

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

```
# ...
```

```
read argent < comptes_minet.txt # lit
```

```
argent=$(expr $argent + "$1") # traite
```

```
echo $argent > comptes_minet.txt # écrit
```

```
# ...
```

- Définie pour **chaque** ressource (fichier, appareil, ...)
- AVANT la **première** lecture
- APRÈS la **dernière** écriture
- **Un seul** acteur peut se trouver dans la section critique d'une ressource à la fois

## Section critique (2/2)

Fichier transaction.sh

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

```
./P.sh comptes_minet.txt.lock
```

```
read argent < comptes_minet.txt # lit
```

```
argent=$(expr $argent + "$1") # traite
```

```
echo $argent > comptes_minet.txt # écrit
```

```
./V.sh comptes_minet.txt.lock
```

- Deux programmes magiques :  
P.sh et V.sh
- P.sh ne peut pas terminer tant que le lock existe encore
- **Attention à bien relâcher les locks même quand le programme se termine mal (exit)**

# Les signaux

- SIGINT (CTRL + C)
- SIGTERM [9] (envoyé par kill)
- SIGKILL (meurt)
- SIGSTOP (CTRL + Z)
- SIGCONT (fg ou bg)
- SIGUSR1 / SIGUSR2 (libre)

- Interrompt le programme en cours pour gérer le signal
- kill -[SIGNAL] [pid ou %job]
- trap 'commande' [SIGNAL]

```
liteapp@luna:~$ sleep 30m
^Z
[1]+  Stoppé                  sleep 30m
liteapp@luna:~$ sleep 1h
^Z
[2]+  Stoppé                  sleep 1h
liteapp@luna:~$ kill %1
```