Langage de commande (shell)

Plan du cours en 4 x 3h30

- **OS & UNIX**
- **Prise en main d'un système Unix (term)**
- **Système de fichiers Unix**
- **Un editeur de texte (vi)**
- **Expressions Régulières (regex)**
- **Processus (proc)**
- **Langages de commandes (shell)**
- **Le monde unix (X, net, ssh, ftp, http, mail,...)**

Langage de commande (shell)

- Modes d’exécution
- Les Scripts
- Variables d’environnement
- Bash, Extraction de sous chaînes
- Les tableaux
- Langage de scripts

Modes d’execution

Shell (coquillage) : Enveloppe externe du système UNIX

2 ou 3 versions importantes :

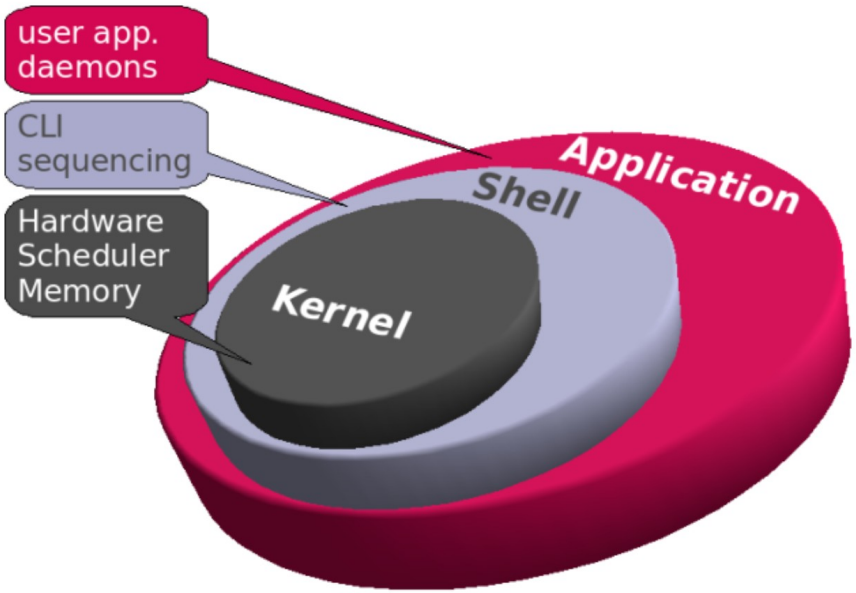
- Shell de base d’ATT, Bourne shell */bin/*sh

  prompt : $
- C-shell */bin/csh disponible principalement sur UNIX BSD 4.3.X (avec des fonctions plus riche*

  prompt : %

Variantes plus récentes (fonction des distributions) :

- */bin/bash, /bin/ksh, /bin/tcsh, /bin/zsh, ...*



Modes d’execution

2 Modes d’executions

Mode Synchrone

- Execution des instructions une par une.
- *Terminées par <CR> ou séparées par ; (sur une même ligne*

Mode Asynchrone :

- Execution des instructions en parralléle (en arrière plan)
- Termine une commande par &

Exemple :

$ mkdir toto

$ ls

≡

$ mkdir toto ; ls ;

```
phil@phildata:~/test$ ls
passwd  test
phil@phildata:~/test$ mkdir toto & ls
[1] 38083
passwd  test
[1]+  Fini                    mkdir toto
phil@phildata:~/test$ mkdir titi ;ls
passwd  test  titi  toto
phil@phildata:~/test$
```

$ mkdir toto & ls

≡

$mkdir &

$ls

Fichiers de commandes executables

Fichier texte executale (droit x, chmod):

```
# !/usr/bin/bash
commande 1 ; commande 2 ;
commande 2 ;
```

Execution d'un script:

$ ./filename

*/usr/*bin/bash indique le type de shell à executer (ksh,...)

Pour faire quoi ?

- Automatisation de tâches (backup, synchronisation de fichiers entre serveurs, maintenance de base de donénes, ...)
- Déclenchement programmer dans le temps (**cron :** detection de timing)
- Déclenchement un évenement fichier (**incron :** detecte modification de fichiers)

Les variables

Variables d’environnement

Une variable = une donnée stockable le temps du process

VARIABLE=valeur une chaîne de texte en majuscule par convention (sans espace)

En fonction des shell

VAR=valeur

export VAR=valeur

setenv VAR valeur

Accès à la variable :

$VAR ou ‘’ Texte ${VAR}’’

```
phil@phildata:~/test$ VAR=test
phil@phildata:~/test$ echo $VAR
test
phil@phildata:~/test$ echo "Dans un texte ${VAR} à trous "
Dans un texte test à trous
phil@phildata:~/test$
```

```
phil@phildata:~/test$ VAR = Test
VAR : commande introuvable
phil@phildata:~/test$
```

```
phil@phildata:~/test$ set var ma valeur
phil@phildata:~/test$ echo $1 $2 $3
var ma valeur
phil@phildata:~/test$
```

```
phil@phildata:~/test$ read v1 v2 v3
voici un texte
phil@phildata:~/test$ echo $v2
un
phil@phildata:~/test$
```

Les variables - $ env


```
phil@phildata:~/test$ env
SHELL=/bin/bash
SESSION_MANAGER=local/phildata:@/tmp/.ICE-unix/2330,unix/phildata:/tmp/.ICE-unix/2330
QT_ACCESSIBILITY=1
COLORTERM=truecolor
XDG_CONFIG_DIRS=/etc/xdg/xdg-ubuntu:/etc/xdg
SSH_AGENT_LAUNCHER=gnome-keyring
XDG_MENU_PREFIX=gnome-
GNOME_DESKTOP_SESSION_ID=this-is-deprecated
SASS_LIBSASS_PATH=/usr/local/lib/libsass
MANDATORY_PATH=/usr/share/gconf/ubuntu.mandatory.path
GNOME_SHELL_SESSION_MODE=ubuntu
SSH_AUTH_SOCK=/run/user/1000/keyring/ssh
TERMINATOR_UUID=urn:uuid:6aaf36dd-81d0-42bc-a54c-1acaff6dfd11
XMODIFIERS=@im=ibus
DESKTOP_SESSION=ubuntu
GTK_MODULES=gail:atk-bridge
PWD=/home/phil/test
LOGNAME=phil
XDG_SESSION_DESKTOP=ubuntu
XDG_SESSION_TYPE=x11
GPG_AGENT_INFO=/run/user/1000/gnupg/S.gpg-agent:0:1
SYSTEMD_EXEC_PID=2353
NODE_ENV=development
XAUTHORITY=/run/user/1000/gdm/Xauthority
GJS_DEBUG_TOPICS=JS ERROR;JS LOG
WINDOWPATH=2
HOME=/home/phil
USERNAME=phil
IM_CONFIG_PHASE=1
LANG=fr_FR.UTF-8
```

Les variables

Quelques variables utiles

Environnement

HOME : nom du repertoire de l'utilisateur
PATH : répertoires dans lsquels le shell va chercher les commandes
MAIL : boîte mail de l'utilisateur
TERM : modèle du terminal utilisateur
PS1 : prompt principal
PS2 : prompt secondaire
IFS : liste des caractères séparateur
SHELL : nom de votre shell
USER : login du user
DISPLAY : écran du X (machine.somehost.somewhere:0.0)

... $ env <CR> pour voir la liste

Variable locale (au shell)

- Perte du scope du script/shell père au script/shell fils.

$ VAR=toto

Variables d'environnement

- Une variable pour tous les shells descendants (appelés)

$ export VAR='toto'

Variable pré-definies

- Dans un fichier sur ~ (.profile, .bashrc, ... fonction distribution)
- C'est variable s'initialise au login du user1

Les variables

Usage des variables dans un script

$0	Le nom de la commande (i.e. : du script)
$1, $2, etc.	Le premier, deuxième, etc, argument passés au script.
$*	La liste de tous les arguments passés au script.
$#	Le nombre d'arguments passés au script.
$?	Le code de retour de la dernière commande lancée.
$!	Le numéro de process de la dernière commande lancée en tâche de fond.
$$	Le numéro de process du shell lui-même.

$@ ≡ ‘’$1’’,’’$2’’,... ‘’${n}’’ utile dans les scripts

```
#/bin/bash
echo $0
echo $1 $2 $3;
echo "Nombre de caractères de la 3éme variable: ${#3} "
echo "dernier process en tache de fond ${!} et process de ce shell ${$}"
echo "liste des arguments ${*}"
```

```
phil@phildata:~/test$ ./test.sh variable1 variableautre2 variableencore3
./test.sh
variable1 variableautre2 variableencore3
Nombre de caractères de la 3éme variable: 15
dernier process en tache de fond 38083 et process de ce shell 44684
liste des arguments variable1 variableautre2 variableencore3
phil@phildata:~/test$
```

Bash extraction de chaînes

Modification de chaines de caractères

```
phil@phildata:~/test$ VAR="Une chaine de test"
phil@phildata:~/test$ echo {#VAR}
{#VAR}
phil@phildata:~/test$ echo ${#VAR}
18
phil@phildata:~/test$ echo "Resultat d'une opération: $((${#VAR}*2+1))"
Resultat d'une opération: 37
phil@phildata:~/test$ echo "Remplacement de chaine par autre: ${VAR/chaine/bout/}"
Remplacement de chaine par autre: Une bout/ de test
phil@phildata:~/test$ echo "extraction start pos 2 et longth 5 :${VAR:2:5}"
extraction start pos 2 et longth 5 :e cha
phil@phildata:~/test$ echo "supprime à gauche: ${VAR#Une ch}"
supprime à gauche: aine de test
phil@phildata:~/test$ echo "supprime à gauche: ${VAR#*i}"
supprime à gauche: ne de test
phil@phildata:~/test$ echo "supprime à gauche: ${VAR#*[de]}"
supprime à gauche:  chaine de test
phil@phildata:~/test$ echo "garde jusqu'à : ${VAR%c*}"
garde jusqu'à : Une
```

```
phil@phildata:~/test$ VAR="toto@titi.tata.fr"
phil@phildata:~/test$ echo ${VAR%%.*}
toto@titi
```

Variable tableaux (ARRAY)

Variable avec indexe = (v1 v2 v3)

Séparateur : variable env $IFS (espace, ...)

```
phil@phildata:~/test$ VAR=(Bonjour "Phil C")
phil@phildata:~/test$ echo "${VAR[0]} de la part de ${VAR[1]}"
Bonjour de la part de Phil C
phil@phildata:~/test$
```

```
phil@phildata:~/test$ VAR=(Bonjour "Phil C")
phil@phildata:~/test$ echo "${VAR[0]} de la part de ${VAR[1]}"
Bonjour de la part de Phil C
phil@phildata:~/test$ echo $VAR
Bonjour
phil@phildata:~/test$ echo ${VAR[*]} # tous les items
Bonjour Phil C
phil@phildata:~/test$ echo ${!VAR[*]} # tous les indices
0 1
phil@phildata:~/test$ echo ${#VAR[*]} #Nombre d'éléments
2
phil@phildata:~/test$ echo ${#VAR[0]}
7
phil@phildata:~/test$
```

Langage : simple, double, back quoting

Les Quote servent à isoler un ensemble de mots (l'espace est souvent un séparateur)

Simple quote affiche un contenu

Double quote permet la substitution

```
phil@phildata:~/test$ NB=10
phil@phildata:~/test$ echo 'nombre = $NB'
nombre = $NB
phil@phildata:~/test$ echo "nombre = $NB"
nombre = 10
```

Back quote (altgr 7 : `) permet de recupérer le résultat d'une commande

En bash, identique à $(...)

```
phil@phildata:~/test$ ls
listf.txt  passwd  test  test.sh  titi  toto
phil@phildata:~/test$ sudo chown `id -u`  test.sh
phil@phildata:~/test$ NBFIC=`ls | wc -l`;echo $NBFIC
6
phil@phildata:~/test$
```

Langage : set / shift

Les Quote servent à isoler un ensemble de mots (l'espace est souvent un séparateur)

Set un parametre de position

```
#!/bin/bash
# Fichier set.sh
set a=`ls test*`
echo a =$a
echo "Liste argu du script " $*
```

```
phil@phildata:~/test$ ls
liste.sh   passwd         select.sh  test     titi
listf.txt  selectctrl.sh  set.sh     test.sh  toto
phil@phildata:~/test$ ./set.sh
a =
Liste argu du script  a=test test.sh
```

Shift decale vers la gauche

```
#!/bin/bash
# Fichier shit.sh
while [ $# -gt 0 ]
do
        echo $*
        shift
done
```

```
phil@phildata:~/test$ ./shift.sh Un Deux Trois
Un Deux Trois
Deux Trois
Trois
phil@phildata:~/test$
```

Langage : Odre des commandes

Différence de fonctionnement du shell et d'un langage de programmation

Ordre de substitution :

- Substitutions des variables
- Substitutions des commandes
- Séparation des arguments
- Génération des noms de fichiers

Exemple:

```
cmd='ls -l'; list="ficl fic2"
$cmd $list part* `echo fic3`
```

- Subs. variables `ls -l ficl fic2 part* 'echo fic3'`
- Subs. commandes `ls -l ficl fic2 part* fic3`
- Séparation des paramètres
- Génération, si `part*=parta partb`
  ⇒ `ls -l ficl fic2 parta partb fic3`

Langage : Retour de commande

Toute commande doit retourner un etat

Par convention sous unix :

- 0 tout est ok
- Entre 1 et 255 indique le type de problème

Fonction exit <code etat>

Rappel :

<cmd1> || <cmd2>

<cmd1> && <cmd2>

Mais aussi les structure de contôles comme if, while ...

Langage : cmd expr test

Expression arithmétique

Le shell ne sait pas nativement calculer des expressions, la commande expr le fait

Attention à la syntaxe

```
phil@phildata:~/test$ expr 12+8
12+8
phil@phildata:~/test$ expr 12 + 8
20
phil@phildata:~/test$ echo `expr 12 + 8`
20
phil@phildata:~/test$
```

Test Logique

Retour code 0 = vrai
code autre = false

Utile lors de mise au point de code
Attention à la syntaxe

```
phil@phildata:~/test$ test 1 == 1 && echo "yes" || echo "No"
yes
phil@phildata:~/test$ test 1 == 2 && echo "yes" || echo "No"
No
phil@phildata:~/test$ test 1 -eq 2 && echo "yes" || echo "No"
No
phil@phildata:~/test$ [1 == 1] && echo "yes" || echo "No"
[1 : commande introuvable
No
phil@phildata:~/test$ [ 1 == 1 ] && echo "yes" || echo "No"
yes
```

Langage : Les opérateurs logiques

Opérateurs -a -o

-a (vrai ET)

```
touch foo       # donc foo existe
rm bar          # donc bar n'existe pas

# [ -f foo ]    = vrai si le fichier foo existe
# [ ! -f bar ]  = vrai si bar n'existe pas

# à n'exécuter que si foo existe ET que bar n'existe pas.
[ -f foo -a ! -f bar ] &&
    mv foo bar
```

-o (vrai OU)

```
if [[ "$fichier" == "fichier_interdit" -o ! -f "$fichier" ]]
    then echo "Je ne veux pas lire $fichier ou bien il n'existe pas."
fi
```

Opérateurs comparaison

```
#!/bin/sh
if test 2 -lt 3
then echo "C'est normal."
fi

if test 2 -gt 3
then echo "C'est absurde."
fi

petit=2
grand=3

if test $petit -ne 3
then echo "C'est normal."
fi

if test 2 -eq $grand
then echo "C'est absurde."
fi
```

```
C'est normal.
C'est normal.
```

- `-eq` (*equal*) : « égal à » (signe « = ») ;
- `-ne` (*not equal*) : « différent de » (signe « ≠ ») ;
- `-gt` (*greater than*) : « strictement supérieur à » (signe « > ») ;
- `-lt` (*lesser than*) : « strictement inférieur à » (signe « < ») ;
- `-ge` (*greater or equal*) : « supérieur ou égal à » (signe « ≥ ») ;
- `-le` (*lesser or equal*) : « inférieur ou égal à » (signe « ≤ ») ;

Langage : Les opérateurs logiques

Opérateurs sur les fichiers

-e (exists) : test si fichier existe

-f (file) : test si existe et si c'est un fichier de base

-d(directory) : vérifie si le repertoire existe

-L(link) : vérifie si le fichier est un lien symbolique

-s (size) : verifie si le fichier n'est pas vide

-r -w -x (readable writable executable):verifie qu'on peut écrire

-nt -ot (newer than older than) : vérifie si un fichier est plus récent / vieux qu'un autre

```
#!/bin/sh

if test -e ~/.emacs
  then echo "~/.emacs existe."
  else echo "~/.emacs n'existe pas."
fi

if test -d ~/.emacs
  then echo "~/.emacs est un répertoire."
  else echo "~/.emacs n'est pas un répertoire."
fi

if test -f ~/.emacs
  then echo "~/.emacs est un fichier."
  else echo "~/.emacs n'est pas un fichier."
fi

if test ~/.vimrc -nt ~/.emacs
  then "~/.vimrc est plus récent que ~/.emacs."
fi
```

Langage : Les opérateurs logiques

Booléen

! inverse le code de retour

```
phil@phildata:~/test$ [ -f foo ] && echo "Le fichier foo existe."
phil@phildata:~/test$ touch foo
phil@phildata:~/test$ [ -f foo ] && echo "Le fichier foo existe."
Le fichier foo existe.
phil@phildata:~/test$ rm foo
phil@phildata:~/test$ [ -f foo ] || echo "Le fichier foo n'existe pas."
Le fichier foo n'existe pas.
phil@phildata:~/test$ [ ! -f foo ] && echo "Le fichier foo n'existe pas."
Le fichier foo n'existe pas.
phil@phildata:~/test$ [ ! -f foo ] || echo "Le fichier foo existe."
phil@phildata:~/test$
```

En shell :

if <condition>
then
<commande>
[elif <condition>
Then
else
<commande>]
fi

```
# Premier cas
if [ -f foo ]
   then echo "Le fichier foo existe."
else continue
fi

# Deuxième cas
if [ -f foo ]
   then continue
else echo "Le fichier foo n'existe pas."
fi

# Troisième cas
if [ ! -f foo ]
   then echo "Le fichier foo n'existe pas."
else continue
fi

# Quatrième cas
if [ ! -f foo ]
    then continue
else echo "Le fichier foo existe."
fi
```

Langage : Structure case

```
case chaîne in
  motif) commandes ;;
  motif) commandes ;;
esac
```


Structure case

Case permet un code plus lisible que elif

Attention à la syntaxe ;; (arreter l'execution)

```
if test $PWD = $HOME
    then echo "Vous êtes dans votre répertoire d'accueil."
elif test $PWD = $HOME/bin
    then echo "Vous êtes dans votre répertoire d'exécutables."
elif test $PWD = $HOME/bin/solaris
    then echo "Vous êtes dans votre répertoire d'exécutables pour Solaris."
elif test $PWD = $HOME/prive
    then echo "Vous êtes dans votre répertoire privé."
else echo 'Vous êtes dans un répertoire quelconque, qui n'est ni $HOME,'
    echo 'ni $HOME/bin, ni $HOME/bin/solaris. '"Vous êtes dans $PWD."
fi
```

```
case "$PWD" in
"$HOME") echo "Vous êtes dans votre répertoire d'accueil.";;
"$HOME/bin") echo "Vous êtes dans votre répertoire d'exécutables.";;
"$HOME/bin/solaris") echo "Vous êtes dans votre répertoire d'exécutables pour Solaris.";;
"$HOME/prive") echo "Vous êtes dans votre répertoire privé.";;
*) echo 'Vous êtes dans un répertoire quelconque, qui n'est ni $HOME,'
    echo 'ni $HOME/bin, ni $HOME/bin/solaris. '"Vous êtes dans $PWD.";;
esac
```

Langage : Structure case

```
case chaîne in
  motif) commandes ;;
  motif) commandes ;;
esac
```


Structure case

- Permet une meilleure lisibilté

```
case $var in
  [0-9]*) echo "$var est un nombre.";;
  [a-zA-Z]*) echo "$var est un mot.";;
  *) echo "$var n'est ni un nombre ni un mot.";;
esac
```

```
#!/bin/sh
# Fichier "canards"

echo "Quel est ton nom ?"
read nom

case $nom in
"Riri"|"Fifi"|"Loulou") echo "Ton oncle s'appelle Donald, $nom.";;
*) echo "Tu n'es pas un neveu de Donald.";;
esac
```

Langage : Structure while

```
while condition
    do commandes
done
```

shell

```
echo "Tapez le mot de passe :"
read password

while test "$password" != mot de passe correct
do echo "Mot de passe incorrect."
    echo "Tapez le mot de passe"
    read password
done

echo "Mot de passe correct."
echo "Bienvenue sur les ordinateurs secrets du Pentagone."
```

```
#!/bin/sh
# Fichier "50fichiers"

numero=1

while test $numero != 51
    do
    # la commande "touch" permet de créer un fichier vide :
    touch fichier"$numero"

    # cette commande ajoute 1 à la variable "numero" :
    numero=$(($numero + 1))
done
```

Langage : Structure until

```
until condition
    do commandes
done
```


Comment choisir until ou while (choisir la condition la plus simple)

```
echo "Tapez le mot de passe :"
read password

until test $password = mot de passe correct
do echo "Mot de passe incorrect."
    echo "Tapez le mot de passe"
    read password
done

echo "Mot de passe correct."
echo "Bienvenue sur les ordinateurs secrets de la NASA."
```

Langage : Structure for

```
for var in liste de chaînes
do commandes
done
```


On utilise for quand on ne veut pas strictement répéter des commandes mais pour pouvoir modifier du contenu en fonction des éléments.

Liste les fichiers du repertorie courant.

```
#!/bin/sh
# Fichier "liste"

for element in *
   do echo "$element"
done
```

```
phil@phildata:~/test$ vim liste.sh
phil@phildata:~/test$ chmod u+x liste.sh
phil@phildata:~/test$ ./liste.sh
liste.sh
listf.txt
passwd
test
test.sh
titi
toto
phil@phildata:~/test$
```

```
#!/bin/sh
# Fichier "50fichiers-for"

# la commande "seq a b" compte de a à b
for numero in `seq 1 50`
    do touch fichier"$numero"
done
```

Langage : Une variable select

Pour permettre de choisir dans une liste et éviter les erreurs

```
#!/bin/bash
# select.sh

echo "Êtes-vous favorable au remplacement du vert par le rouge ?"
echo "(Répondez « pour » ou « contre »)"
read opinion
# M'envoyer le résultat par mail
echo "$opinion" | mail phil
echo "Êtes-vous favorable au remplacement du vert par le rouge ?"
select opinion in Pour Contre
    do break
done
# Affiche le resultat
echo "$opinion"
```

```
phil@phildata:~/test$ chmod u+x select.sh
phil@phildata:~/test$ ./select.sh
Êtes-vous favorable au remplacement du vert par le rouge ?
(Répondez « pour » ou « contre »)
fuuuu
./select.sh: ligne 8: mail : commande introuvable
Êtes-vous favorable au remplacement du vert par le rouge ?
1) Pour
2) Contre
#? 1
Pour
phil@phildata:~/test$
```

```
#!/bin/bash
# Fichier "selectctrl"

echo "Êtes-vous favorable au remplacement du vert par le rouge ?"
select opinion in Pour Contre
    do
        case $opinion in
            # Laisser passer ceux qui répondent correctement à la
 question
            "Pour"|"Contre") break;;

            # Au cas où des zozos tapent sur autre chose que 1 ou
 2
            *) continue;;
        esac
done
echo "$opinion"
```

Imbrique les boucles les unes dans les autres

```
phil@phildata:~/test$ vim selectctrl.sh
phil@phildata:~/test$ chmod u+x selectctrl.sh
phil@phildata:~/test$ ./selectctrl.sh
Êtes-vous favorable au remplacement du vert par le rouge ?
1) Pour
2) Contre
#? k
#? 9
#? 2
Contre
phil@phildata:~/test$
```

Langage : Les functions

Permet de découper un programme en unité « logique

Le problème des programmes sans fonction :

- Peu lisible
- Répétition de codes
- Faire evoluer le code =>
  - Difficile de trouver ce qu'on veut modifier
  - Peu fiable (risque de changer des valeurs avec des effets de bord

Etapes de création de fonction :

- Definir ce qu'elle doit faire
- Définir les entrèes les sorties

```
#!/bin/bash
# function.sh

# Function nomfctA

nomfctA() {
 echo "Info passées en parametre" $*
}

nomfctA para1 para2;
nomfctA;
```

```
phil@phildata:~/test$ ./function.sh
Info passées en parametre para1 para2
Info passées en parametre
phil@phildata:~/test$
```

Langage : Les functions

Une fonction peut être vue comme un script :

- Si un script s'appel S et une fonction porte le même nom (lorsque'on invoque S c'est la fonction qui prime
- Avec les fonctions, on peut surcharger des commandes de base en ajoutant dans PATH le nom d'un script ayant une function identique à une commande

Attention de ne pas mélanger les entrées de script et de fonction :

```
#!/bin/bash
# testfunction

echo "Script = $1"
function bon () { echo "Fct = $1"; }
bon toto
```

```
phil@phildata:~/test$ ./testfunction.sh titi
Script = titi
Fct = toto
phil@phildata:~/test$
```

Langage : Les functions, return

Une fonction ne peut retourner qu'une valeur entre 0 et 255

- Comment retourner une chaîne de caractères ?

```
#!/bin/bash

fctA() {
        echo "valeurA";
        return 0;
}

retval=$(fctA)
status=$?

echo "the function fctA return $retval and code $status"
```

```
phil@phildata:~/test$ ./fctreturnecho.sh
the function fctA return valeurA and code 0
phil@phildata:~/test$
```

Langage : Conclusion sur le shell

Langage complet pour automatiser des commandes

- Trouver les commandes adaptées :

  $ man -k {keyword} quand on cherche qq chose

  $ man nomcommande (comprendre le formalisme [option] … parametres

  $ cmd1 && cmd 2 ; cmd1 || cmd2 ; cmd1 | cmd2 ; cmd1 < fichier ; cmd1 | xargs cmd2 ;

  Si besoin d’ une commande specifique créer un script avec des fonctions et une récuperation de parametre $1 $n , $* $#

- Attention :
  - Les espaces dans les instructions interprétés comme des commandes
  - Les arguments sont séparés par des espaces
  - Une commande par ligne, sinon il faut un ;
  - Un ensemble de mot doit être entre ‘’ (avec substitution) ou ‘(sans substitution) ou `(back quote) interpreté

Le monde Unix

Plan du cours en 4 x 3h30

- **OS & UNIX**
- **Prise en main d'un système Unix (term)**
- **Système de fichiers Unix**
- **Un editeur de texte (vi)**
- **Expressions Régulières (regex)**
- **Processus (proc)**
- **Langages de commandes (shell)**
- **Le monde unix (X, net, ssh, ftp, http, mail,...)**

Monde Unix : une industrie

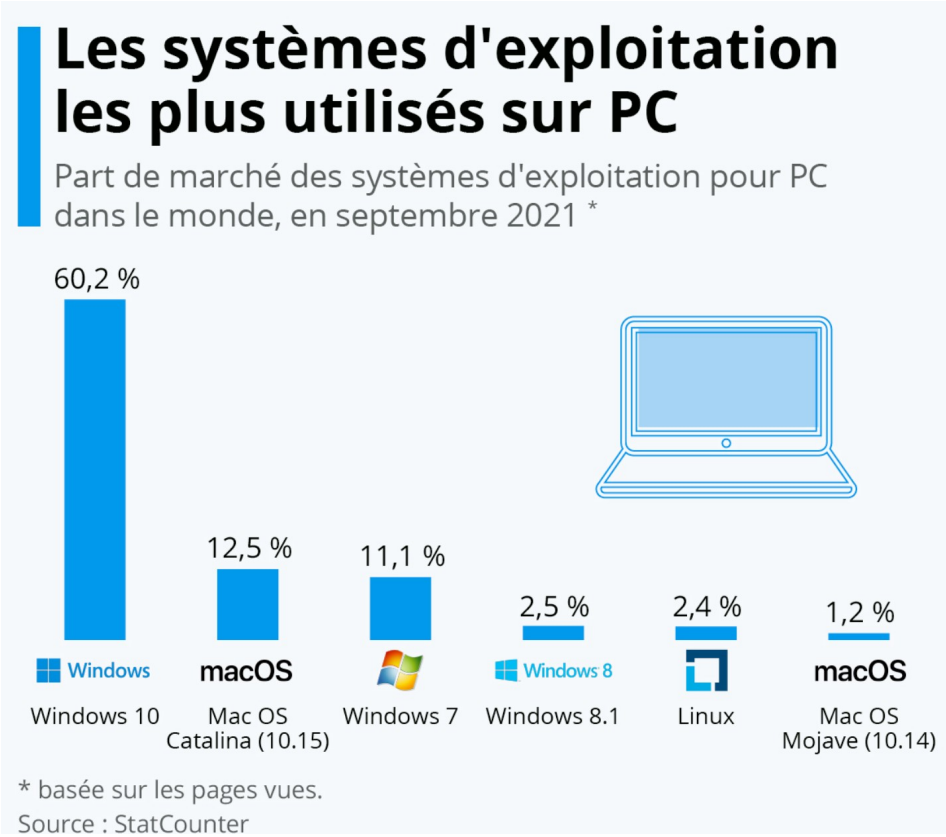
Décentralisé et Communiquant à l’image d’internet

- Propriétaire / libre:
  - Diversité des outils (spécificités par machines, reflexe man)
  - Une communauté d’entraide technique (qui veut apprendre peut)
  - Pouvoir expérimenter pour tester (sur tout type de machine)
  - Scalable en maitrisant les coûts (libre)

- Productif:
  - Courbe d’apprentissage mais bénéfice pour la sécurité, la robustesse (KISS et selection naturelle) pas forcément simple mais efficace sur la durée.
  - S ‘adapte à son niveau (débutant au profil avancé on apprend toujours)
  - Une méthode de dev qui a fait ses preuves largement utilisé aujourd’hui dans les dev en mode integration continu

Monde Unix : Principalement serveur/embarqué

shell



Parts de marché des systèmes d'exploitation dans le monde en 2022

Vous trouverez ci-dessous les systèmes d'utilisation disposant du plus grand nombre d'utilisateurs dans le monde

Filtres

1. Windows Server 11.24%
2. UNIX 2.23%
3. Ubuntu 2.00%
4. CentOS 1.33%
5. Debian 1.28%
6. Red Hat 0.11%
7. FreeBSD 0.10%

Monde Unix : Des outils portables

Les applicatifs sont portables sur tout OS

- Aujourd'hui, tout peut se compiler sur les OS avec des machines classiques (X86 ARM, ….).
- On trouve tous les langages de programmations (SDK) toutes les bases de données
- Des commandes similaires à tous les OS (ping, traceroute, ...)
- La virtualisation permet de deployer des environnements dédiés à une application

La question :

- **Quelle architecture pour quel coût en fonction des compétences d'une équipe**
- **Quelle évolutivité d'une architecture**
- **Le choix de l'OS dépend de l'écosystème (MacOS pour les designer, windows pour les electroniciens, Linux pour le web, windows pour l'informatique de gestion qui évolue vers le web…)**

La bonne réponse : celle qui répond à un besoin et qui peut évoluer avec le moins de bloquage.

Monde Unix : les commandes du quotidien

Quelques cmd :

- $ alias ff=’find / -name * -print’ ; unalias ff ;
- $ history ; !numero ; pour retrouver d’ancienne commande et executé par son numero
- $ neofetch ; info sur sa distribution
- $ wget url ; pour telecharger un fichier html ou autre
- $ curl url ; pour envoyer un requete plus évoluer que wget (POST, PUT, GET, …)
- $ zip unzip ; des outils pour lire le contenu zipé dezipper compatible vers windows
-

Monde Unix : quelques outils communs

- sshd ssh scp outils d’accès à distance
- Nginx, apache, serveur web
- Mail, smtp, pop, … outils de messagerie
- rsync sftp samba NFS outils de serveur de fichiers
- …