UPMC
PARIS UNIVERSITAS

# LPI

# Exam 101

## 103.1 Ligne de commande

# 103.1 – Domaines de connaissances

- Poids : 4
- Utilisation de la ligne de commande
- Modification de l'environnement d'exécution du shell : variables
- Historique des commandes
- Exécution de commandes : dans et hors des chemins définis dans la variable d'environnement PATH

- Commandes internes et externes
  - Les commandes internes font partie du shell (“builtin commands”) :
  - pas de fichier exécutable correspondant
  - Les commandes externes ... c'est l'ensemble des programmes de votre système
  - ... mais il existe des commandes internes et externes qui portent le même nom : `echo`
- Comment reconnaître le type d'une commande
  - `type -a commande`

```
[aoi@test]$ type cd
cd is a shell builtin
[aoi@test]$ type more
more is /bin/more
```

```
[aoi@test]$ type -a echo
echo is a shell builtin
echo is /bin/echo
```

- Le shell (« coquille ») est un programme qui permet la saisie et l'interprétation de ce qui est tapé. C'est l'interface entre le système et l'utilisateur.
- Ici pas d'interface graphique : l'interaction utilisateur se fait par l'intermédiaire de mots frappés au clavier
- C'est à la fois...
  - Un interpréteur de commandes (un programme qui boucle en attente des commandes utilisateur)
  - Un langage de programmation (interprété) offrant les structures de base comme tout autre langage
- Sous Unix, le shell est un programme au même titre qu'un autre. On peut donc changer de shell si besoin.

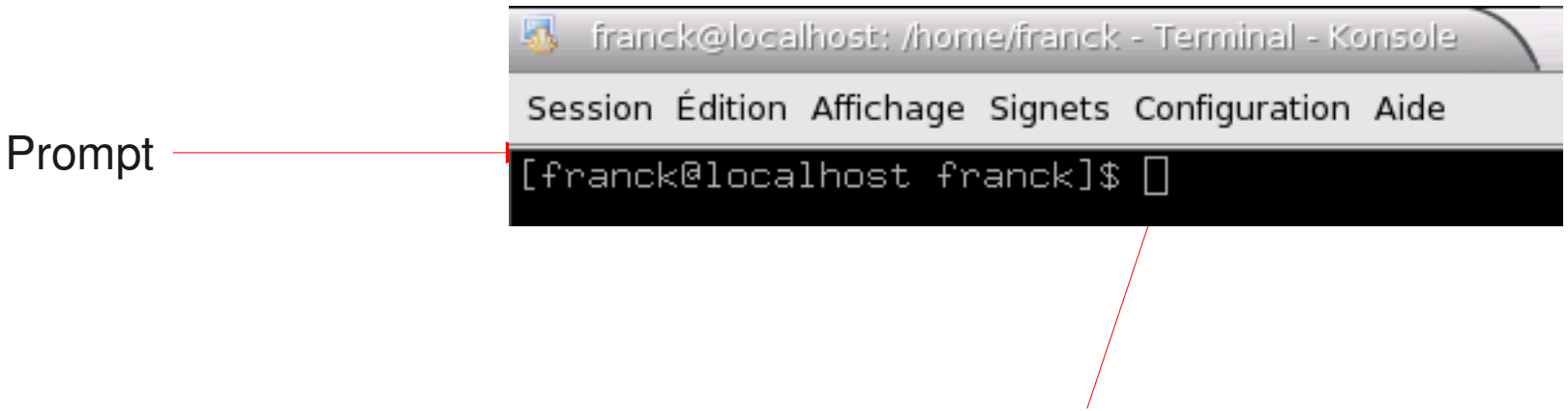
- shells les plus répandus :
  - sh (bourne shell) : disponible sur toute plate-forme Unix
  - bash (bourne Again shell) : Distribué sur la majeure partie des Linux; version améliorée de sh et csh
  - ksh (korn shell) : Bourne shell amélioré par AT&T
  - csh (C shell) : shell développé pour BSD
  - Tcsh (Tom's C shell) : variante du C chell
- Sur les Linux récents
  - Lien symbolique : /bin/sh -> /bin/bash
- Un shell est associé par défaut à chaque utilisateur. Il est définit dans le fichier /etc/passwd

# 103.1 – shell bash

- La liste des shells valides sur un systèmes est stockée dans le fichier `/etc/shells`
- La commande `chsh` permet de modifier le shell par défaut d'un utilisateur
- `/bin/nologin` : commande refusant la connexion d'un utilisateur lorsqu'elle remplace son shell par défaut.
  Le message d'information standard peut être remplacé par le contenu du fichier `/etc/nologin.txt` si ce fichier existe.

```
$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
```

- Tous les shells se présentent sous la même forme :
  - une chaîne de caractères affiche que le shell attend que l'utilisateur tape une commande au clavier : le prompt
  - Par convention
    - $ : identifie un utilisateur standard
    - # : identifie une session root
  - un curseur qui va se déplacer au fur et à mesure de la saisie des commandes



- La composition du prompt est gérée via des variables d'environnement
  - PS1 : prompt par défaut
  - PS2 : prompt affiché lorsque la ligne de commande continue sur la ligne suivante (souvent '>')
  - PS3 : prompt affiché lors d'une boucle select d'un script shell
  - PS4 : prompt utilisé lorsque le shell est en mode debug (souvent : '+')
  - Utilisation de séquences d'échappement pour modifier ces variables (extrait)
    - \d : date
    - \h et \H : nom de machine (avec et sans nom de domaine)

- ....

  - \u : nom utilisateur
  - \w et W\ : répertoire courant et basename
  - \$ : # pour root et $ pour les autres utilisateurs
  - \n : newline

```
$ echo $PS1
[\u@\h \W]\$
[moi@ma-machine ~]$ PS1="Mon prompt\
$"
Mon prompt$
```

```
$ echo "Commande qui se poursuit \
> sur la seconde ligne"
Commande qui se poursuit sur la seconde ligne
```

- commande unix = un ensemble de mots séparés par des caractères blancs (white spaces) :espace, tabulation
  - premier mot : nom de la commande
  - reste des mots : paramètres de la commande
- parmis ces paramètres, les options, modifient le comportement de la commande
- format d'une commande type :

```
$ commande [options] parametres
```

  - les crochets [ et ] encadrent les éléments facultatifs de la commande
  - les crochets ne doivent pas apparaître dans la ligne de commande

- La syntaxe pour spécifier une option a été définie dans le programme de la commande
- Il existe pourtant des notations que l'on retrouve dans la plupart des commandes (mais il existe des exceptions)
  - une option est introduite par le signe – et est souvent constituée d'une lettre

    $ cp -r -v source dest
  - l'ordre des options n'a pas d'importance et on peut les cumuler

    $ ls -a -l = $ ls -l -a = $ ls -al
  - les options constituée d'un mot entier sont souvent introduite par le signe --
    $ urpmi --auto-update

- Erreur classique : oubli de l'espace entre la commande, les options et les paramètres
  - le shell ne reconnaît pas la commande et affiche un message d'erreur
  - les paramètres ou options ne sont pas valides pour la commande : la commande affiche une erreur
  - résultat inattendu de la commande

```
[franck@localhost ~]$ ls -al
total 6011
drwxr-xr-x 63 franck franck    3072 mar  7 21:33 ./
drwxr-xr-x  6 root   root      1024 fév 20 12:30 ../
-rw-rw-r--  1 franck franck   95684 déc 13 16:57 24488-
linuxevolution.tar.bz2
.....

[franck@localhost ~]$ ls-al
bash: ls-al: command not found
```

- Historique des commandes
  - Chaque commande validée est enregistrée chronologiquement avant son interprétation et substitutions
  - Fonctionnalité du shell activée par défaut : `set -o history`
  - A la sortie du shell, les commandes sont écrites dans le fichier `.bash_history`
  - Par défaut, le shell enregistre 1000 commandes
    Paramétrable via la variable `HISTSIZE`

# 103.1 – Historique des commandes

- bash peut rechercher et rappeler (et exécuter !) des commandes déjà saisies grâce aux flêches de navigation  et
  -  : rappelle la commande précédente (dans l'ordre chronologique)
  -  : rappelle la commande suivante (dans l'ordre chronologique)
  - `!!` : rappelle la dernière commande
  - `!chaîne` : rapelle la dernière commande débutant par chaîne
  - `!?chaîne` : rapelle la dernière commande contenant chaîne
  - `!-n` : rappelle la nième dernière commande
  - `!n` : rappelle la nième commande

# 103.1 – Historique des commandes

- Intérêts :
  - Gain de temps et évite les erreurs de saisie en particulier pour les commandes longues
  - Effet mémoire des commandes saisies sur une machine
    `# history | grep mount`
- Commande `history` : affiche l'historique complet des commandes
  - `history N` : affiche les N dernières commandes
  - `history -d num-ligne` : efface la ligne à la position num-ligne
  - `history -a` : met à jour le fichier .bash_history

# 103.1 – Historique des commandes

```
$ history 5
 1002  ls
 1003  history 5
 1004  echo $HISTORY
 1005  echo $HISTSIZE
 1006  history 5

$ tail -5 .bash_history
ssh toto@titi.com
exit
ssh toto@titi.com
uname -a
exit
```

```
$ history -a

$ tail -5 .bash_history
echo $HISTORY
echo $HISTSIZE
history 5
tail -5 .bash_history
history -a
```

```
$ ls c*.txt
centre2.txt  centre3.txt  centre4.txt  centre.txt

$ !!
ls c*.txt
centre2.txt  centre3.txt  centre4.txt  centre.txt

$ !ls
ls c*.txt
centre2.txt  centre3.txt  centre4.txt  centre.txt

$ !ls:s/txt/py/
ls c*.py
centre1.py  centre2.py  centre3.py  check.py  clean.py  colorcal.py
```

# 103.1 – Raccourcis

- Quelques raccourcis utiles (mode emacs)
  - **[Ctrl]+[a]** : positionne le curseur au début de la commande
  - **[Ctrl]+[e]** : positionne le curseur à la fin de la commande
  - **[Alt]+[f]** : positionne le curseur au début du mot suivant
  - **[Alt]+[b]** : positionne le curseur au début du mot précédant
  - **[Ctrl]+[t]** : inverse deux lettres
  - **[Ctrl]+[u]** : efface (coupe) la ligne
  - **[Ctrl]+[k]** : efface (coupe) de la position courante jusqu'à la fin de ligne
  - **[Ctrl]+[y]** : colle le texte coupé

# 103.1 – Raccourcis

- Navigation dans la commande
  - Utiliser les flêches de navigation et pour déplacer le curseur d'insertion dans la ligne de commande
- Edition de la commande
  - **[Backspace]** : supprime la caractère précédent le curseur d'insertion
  - **[Suppr]** : supprime la caractère suivant le curseur d'insertion

- Commande `exec`
- Exécute le programme passé en paramètre et quitte immédiatement (en fait remplace le processus en cours) le programme appelant
- Si appelé depuis un script shell : arrêt immédiat du script

```
$ echo $$
5116
[franck@port-135 ~]$ bash
[franck@port-135 ~]$ echo $$
5597
[franck@port-135 ~]$ exec bash
[franck@port-135 ~]$ echo $$
5597
[franck@port-135 ~]$ exit
exit
[franck@port-135 ~]$ echo $$
5116
```

- Commande `source`
- Permet d'exécuter un script dans le processus courant. Cela permet de conserver les affectations ou modifications de variables valides dans le processus courant.
- Deux syntaxes (commandes `source` ou « . »)
  - `# . /opt/appli/etc/config.sh`
  - `# source /opt/appli/etc/config.sh`

- Commande `echo`
- Affiche les arguments passés en paramètre sur la sortie standard
- Options
  - -n  : pas de newline
  - -e : interprétation des caractères d'échappement
    - \n : newline
    - \b : backspace
    - \t : tabulation horizontale

- Commande `uname`
- Affiche les informations sur la version de noyau et le système en cours d'éxécution
- Options
  - -a  : affiche tout
  - -s : nom du noyau
  - -r : release du noyau
  - -v : version du noyau
  - -m : nom de la machine (CPU)
  - -o : nom de l'OS
  - -i : plateforme matérielle
  - -p : processeur

```
$ uname -a
Linux port-135.ipsl.jussieu.fr 2.6.30.5-43.fc11.i686.PAE #1 SMP Thu Aug 27
21:34:36 EDT 2009 i686 i686 i386 GNU/Linux
[franck@port-135 exo-lpi]$ uname -s
Linux
[franck@port-135 exo-lpi]$ uname -v
#1 SMP Thu Aug 27 21:34:36 EDT 2009
[franck@port-135 exo-lpi]$ uname -r
2.6.30.5-43.fc11.i686.PAE
[franck@port-135 exo-lpi]$ uname -m
i686
[franck@port-135 exo-lpi]$ uname -o
GNU/Linux
[franck@port-135 exo-lpi]$ uname -i
i386
[franck@port-135 exo-lpi]$ uname -p
i686
```

- **tabulation, espace** : délimiteurs des élements d'une commande
- **retour chariot** : fin et demande d'exécution de la commande
- « **;** » : séparateur de commandes sur une seule ligne
- « **( )** » : exécution de commande dans un sous-shell
- « **{ }** » : exécution des commandes en série
- « **&** » : lancer une commande en arrière plan
- « **'** », « **"** », « **** » : quote characters, altèrent la manière dont le shell interprète les caractères spéciaux
- « **<** », « **>** », « **<<** », « **>>** », « **|** » : redirections des entrées-sorties
- « ***** », « **?** », « **[ ]** », **[^]** » : expansion des noms de fichiers
- « **$** » : valeur d'une variable

- « ; » permet de lancer plusieurs commandes sur la même ligne
  - utile pour grouper plusieurs commandes dans un sous-shell

```
[aoi@test]$ date;ls exemple.txt
mer mar 28 23:24:06 CEST 2007
exemple.txt

[aoi@test]$ (ls ; whoami ; id ) > result.txt
```

- « ( ) » : délimitent un bloc de commandes exécutées dans un sous shell

```
[aoi@test]$ pwd ; (cd rep; pwd) ; pwd
/home/franck/UNIX
/home/franck/UNIX/rep
/home/franck/UNIX
```

- « ' » : (apostrophe – single quote) le shell n'interprète aucun caractère spécial pour tout les texte encadré par deux apostrophes
- « '' » : (guillemet – double quote) le shell n'interprète aucun caractère spécial pour tout les texte encadré par deux guillemets à l'exception de « $ », « ` » et « \ ».
- « \ » : (antislash – backslash) le shell n'interprète pas le caractère spécial sui suit immédiatement l'antislash.

- Exemples :

```
[aoi@test]$ echo un     deux
un deux
[aoi@test]$ echo "un    deux"
un    deux
[aoi@test]$ echo 'un    deux'
un    deux
```

```
[aoi@test]$ echo "aujourd'hui"
aujourd'hui
[aoi@test]$ echo aujourd\'hui
aujourd'hui

[aoi@test]$ echo aujourd'hui
> '
aujourdhui
```

- Exemples :

```
[aoi@test]$ echo '"today"'
"today"
[aoi@test]$ echo "today"
today
[aoi@test]$ echo '"'
"
[aoi@test]$ echo "'"
'

[aoi@test]$ echo $HOME
/home/franck
[aoi@test]$ echo "$HOME"
/home/franck
[aoi@test]$ echo '$HOME'
$HOME
[aoi@test]$ echo "\$HOME"
$HOME
[aoi@test]$ echo '\$HOME'
\$HOME
```

- Exemples :

```
[aoi@test]$ find . \( -name '*.jpg' -o -name
'*.jpeg' \) -atime +7 -exec rm {} \;
```

```
$ echo trois\>deux
trois>deux

$ echo "la variable \$HOME
contient
> mon rep perso : $HOME"

la variable $HOME contient
> mon rep perso : /home/franck
```

Seul le caractère spécial '**>**' n'est pas interprété

# 103.1 – shell bash

- Fichiers d'initialisation du shell bash
- Il existe en fait plusieurs modes d'exécution du shell
  - shell de login : shell lancé par la commande « `login` »
  - shell interactif : les commandes sont saisies au terminal
  - shell non interactif : les commandes sont lues et exécutées à partir d'un fichier
- En fonction du mode d'exécution du shell, des fichiers d'initialisation du shell seront exécutés ou non
  - shell de login interactif :
    - exécution de « `/etc/profile` »
    - exécution du premier fichier parmi
      « `$HOME/.bash_profile` »,
      « `$HOME/.bash_login` », « `$HOME/.profile` »

- shell non de login interactif ou non interactif:
  - exécution de « `$HOME/.bashrc` »
  - « `$HOME/.bashrc` » peut faire appel au fichier « `/etc/bashrc` » commun à tous les utilisateurs
- pour modifier le comportement du shell pour tous les utilisateurs on modifie les fichiers « `/etc/bashrc` » et « `/etc/profile` »
- pour modifier le comportement du shell pour chaque utilisateur on modifie les fichiers d'initialisation localisés dans le répertoire home

- Application : modifier de façon permanente la variable `PATH` et ajouter le chemin « `$HOME/bin` »
  - ajout dans le fichier « `$HOME/.bash_profile` »
    « `PATH=$PATH:$HOME/bin` »
- Quand le shell de login se termine
  - execution de le « `$HOME/.bash_logout` »

- Pour simplifier
  - « /etc/profile » : valable pour tous les utilisateurs; exécuté au login. Contient généralement les variables d'environnement, PATH, programmes au démarrage
  - « /etc/bashrc » : valable pour tous les utilisateurs; appelé depuis les fichiers « .bashrc » de chaque utilisateur; contient les fonctions et alias
  - « $HOME/.bash_profile », « $HOME/.bash_login », « $HOME/.profile » : spécifique à chaque utilisateur; exécuté au login; exécution de l'un des trois
  - « $HOME/.bashrc » : spécifique à chaque utilisateur; exécution au login ou non (interactif ou non)

- Généralités
  - Les variables définies dans un shell sont définies et accessibles uniquement pour ce shell
  - la commande « `set` » affiche la liste des variables définies dans la session shell
  - Par convention les variables sont écrites en majuscules
  - Les noms de variables sont sensibles à la casse
  - Attention à l'interprétation par le shell du caractère $ en fonction de l'utilisation des apostrophes ou guillemets
  - Un nom de variable doit être composé uniquement de caractère alphanumériques et du caractère undescore : _

- Affectation
  - `variable=valeur`
  - Attention à l'erreur classique : pas d'espaces dans une commande d'affectation ( l'espace est un métacaractère qui possède une signification précise pour le shell)

```
[aoi@test]$ VAR=valeur

[aoi@test]$ VAR =valeur
bash: VAR: command not found

[aoi@test]$ VAR= valeur
bash: valeur: command not
found

[aoi@test]$ VAR = valeur
bash: VAR: command not found
```

# 103.1 – Variables d'environnement

- Consultation du contenu d'un variable
  - `$variable`
  - `${variable}`

```
[aoi@test]$ echo $VAR

[aoi@test]$
VAR=valeur
[aoi@test]$ echo $VAR
valeur
```

- Suppression d'une variable
  - `unset variable`

```
[aoi@test]$ echo $VAR
valeur
[aoi@test]$ unset VAR
[aoi@test]$ echo $VAR
```

- Utilisation des parenthèses : permet d'ajouter au contenu d'un variable une suite de caractères.
  Le shell identifie un nom de variable comme étant une suite de caractères terminées par un espace

```
$ echo $VAR1
Lpi

$ echo $VAR1_2010

$ echo ${VAR1}_2010
lpi_2010
```

- Portée des variables
  - Par défaut, le contenu des variables ne sont pas accessibles depuis les commandes lancées depuis le shell

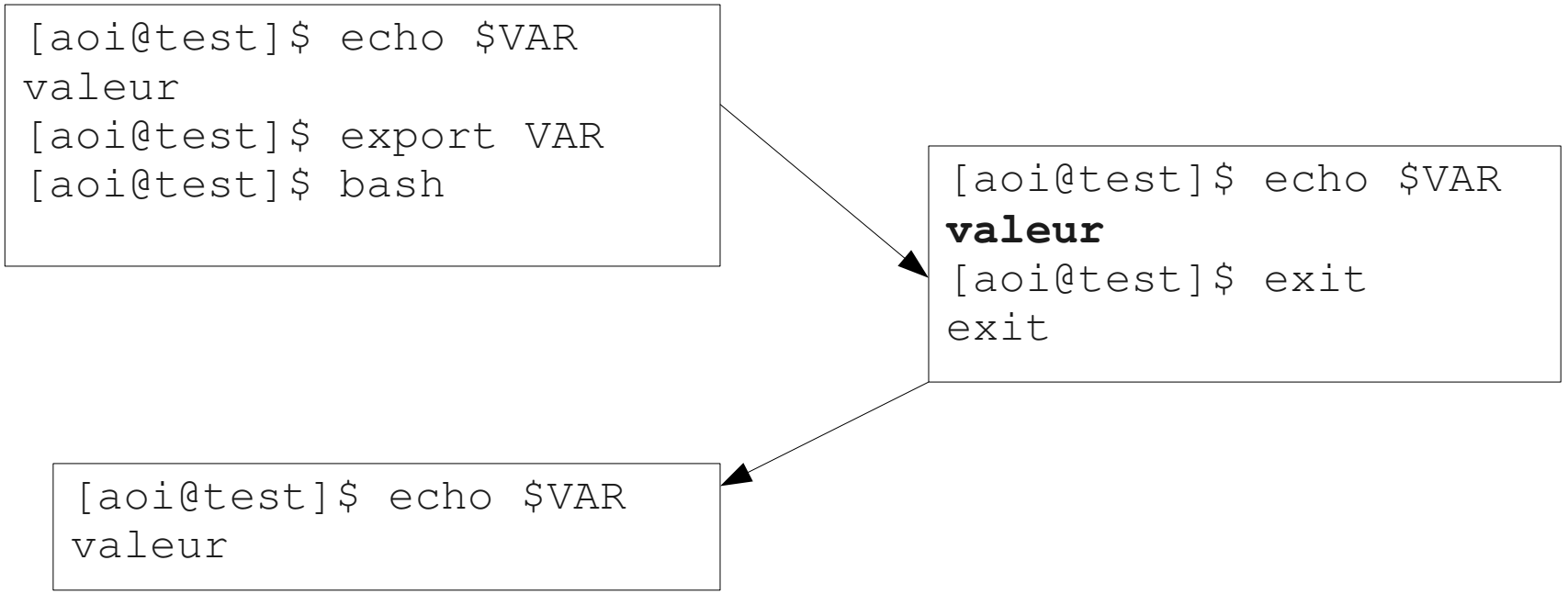
```
[aoi@test]$ VAR=valeur
[aoi@test]$ echo $VAR
valeur
[aoi@test]$ bash
```

```
[aoi@test]$ echo $VAR

[aoi@test]$ exit
exit
```

```
[aoi@test]$ echo $VAR
valeur
```

- les exemples suivants utilisent le shell comme application mais les principes restent identiques pour n'importe quelle application

- Portée des variables
  - Solution : utiliser la commande « `export` »
  - Attention : le paramètre de `export` est bien le nom de la variable (VAR) et non son contenu ($VAR)



- Attention ! les variables héritées ne sont pas modifiables
  - les variables sont dupliquées au lancement d'une application

```
[aoi@test]$ echo $VAR
valeur
[aoi@test]$ export VAR
[aoi@test]$ bash
```

```
[franck@localhost UNIX]$ echo $VAR
valeur
[franck@localhost UNIX]$ VAR=aoi
[franck@localhost UNIX]$ echo $VAR
aoi
[franck@localhost UNIX]$ exit
exit
```

```
[aoi@test]$ echo $VAR
valeur
```

- Il est possible d'assigner et d'exporter une variable en une seule commande

```
$ export VAR1=lpi

$ bash

$ echo $VAR1
lpi
```

# 103.1 – Variables d'environnement

- Liste des variables d'environnement standard
  - `HOME` : répertoire utilisateur (home)
  - `USER` : login de l'utilisateur
  - `SHELL` : chemin du shell utilisé
  - `PATH` : liste des répertoires de recherche des commandes
  - `TERM` : type de terminal utilisé
  - `PWD` : répertoire courant
  - `$` : process ID du processus courant (ne peut pas être modifiée)
  - `?` : code de retour de la dernière commande (ne peut pas être modifiée)
  - ! : process ID du dernier processus en background

# 103.1 – Variables d'environnement

```
$ echo $PWD
/home/franck

$ echo $$
3031

$ echo $?
0

$ ps -edf | grep bash
franck    3031  3028  0 22:08 pts/0    00:00:00 bash
franck    4301  3031  0 23:40 pts/0    00:00:00 grep bash
```

- Commande `env`
- Sans paramètre, affiche les variables d'environnement courantes. Permet également d'exécuter une commande dans un environnement modifié.
- Options
  - `-i` : lance une commande dans un environnement vide
  - `-u` : lance une commande en supprimant une variable de l'environnement d'exécution

# 103.1 – Variables d'environnement

```
$ env
ORBIT_SOCKETDIR=/tmp/orbit-franck
HOSTNAME=toto.jussieu.fr
IMSETTINGS_INTEGRATE_DESKTOP=yes
TERM=xterm
SHELL=/bin/bash
HISTSIZE=1000
```

```
$ env | wc -l
46

$ env | grep SHELL
SHELL=/bin/bash
```

# 103.1 – Variables d'environnement

```
$ env | grep VAR1
VAR1=lpi

$ env -i bash

$ env | grep VAR1
$ man env
WARNING: terminal is not fully functional


 $ env VAR2=examen bash

 $ echo $VAR2
 Examen

 $ exit

 $ echo $VAR2
```

# 103.1 – Variables d'environnement

- Commande `set`
- Sans paramètre, affiche les variables d'environnement courantes ainsi que les fonctions.
- Son utilisation habituelle consiste à modifier (activer ou désactiver) les options d'exécution du shell.
  - Pour activer une option : `set -f` ou `set -o noglob`
  - Pour désactiver une option : `set +f` ou `set + o noglob`
- La commande set dispose d'un grand nombre d'options
  - `-x` : xtrace ou mode debug
  - Pour le reste : http://www.gnu.org/software/bash/manual/html_node/The-Set-Builtin.html

- La commande set dispose d'un grand nombre d'options. Entre autres :
  - `-x` : (xtrace) : affiche les commandes après les avoir interprétés. Permet de mettre en évidence le travail de substitution (commandes, paramètres) du shell
  - `-v` : (verbose) : affiche les commandes avant de les exécuter
  - `-n` : (noexec) : utilisé pour vérifier la syntaxe de shell script. Ignoré pour les shell interactifs
  - `-f` : (noglob) : désactive le file globbing

```
$ set -v
echo -ne "\033]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}"; echo -ne "\007"
[franck@machine exo-lpi]$ ls
ls
env.txt  fic1  fic2  fic3  fic4  ok.txt  pas-ok.txt  set.txt
echo -ne "\033]0;${USER}@${HOSTNAME%%.*}:${PWD/#$HOME/~}"; echo -ne "\007"
```

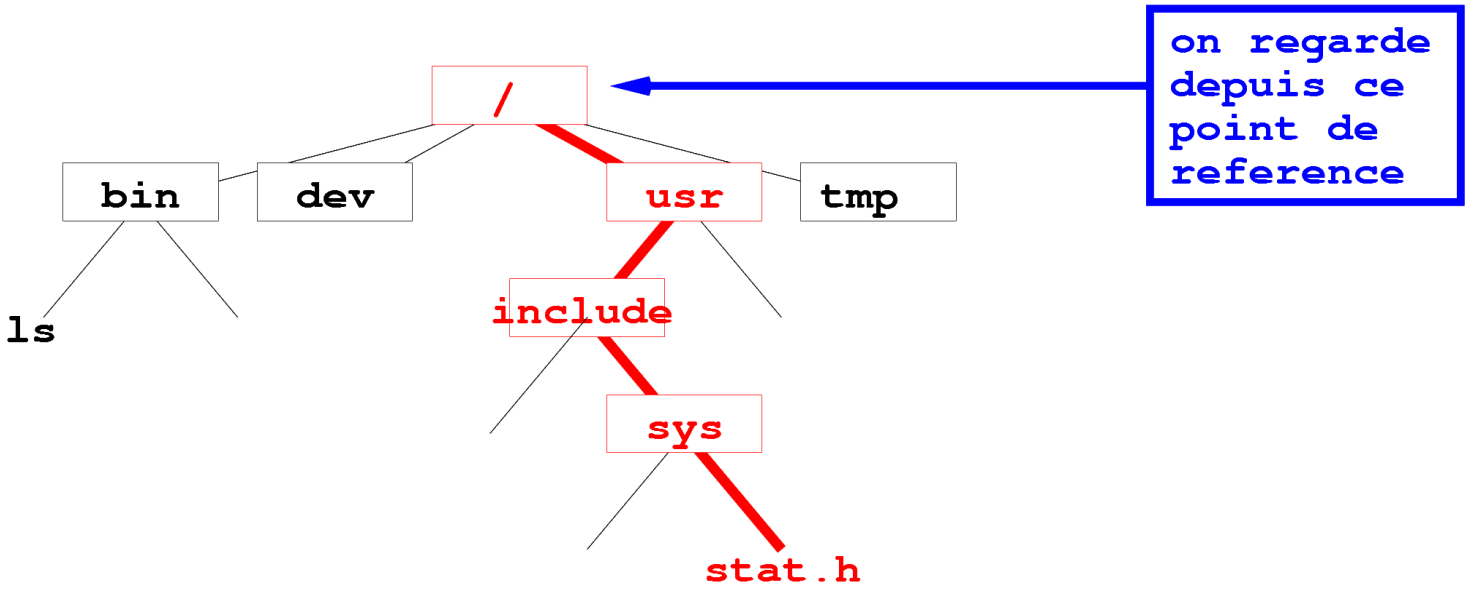
```
$ set -x
++ echo -ne '\033]0;franck@machine:~/src/exo-lpi'

[franck@machine exo-lpi]$ VAR3=linux
+ VAR3=linux
++ echo -ne '\033]0;franck@machine:~/src/exo-lpi'

[franck@machine exo-lpi]$ echo $VAR3
+ echo linux
linux
++ echo -ne '\033]0;franck@machine:~/src/exo-lpi'

[franck@machine exo-lpi]$ kill -HUP `cat /var/run/xinetd.pid
`++ cat /var/run/xinetd.pid
+ kill -HUP 2070
bash: kill: (2070) - Opération non permise
++ echo -ne '\033]0;franck@machine:~/src/exo-lpi'
```

# 103.1 – Path

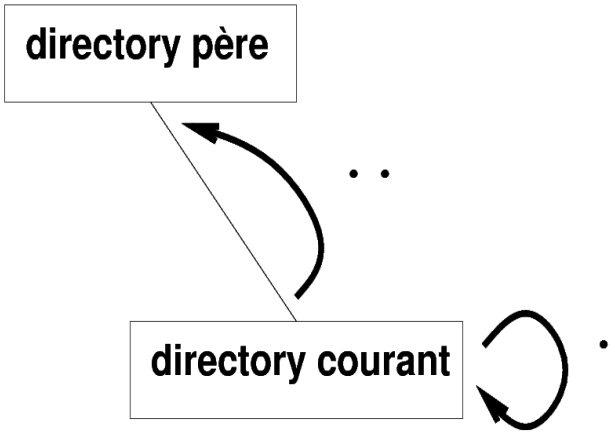
- On accède à un fichier à travers son « chemin » dans l'arborescence
  - le chemin est constitué d'une liste de noms de répertoires et se termine par le nom du fichier
  - le caractère « / » permet de séparer les différents répertoires qui constituent le chemin

```
/repertoire1/répertoire2/fichier
```

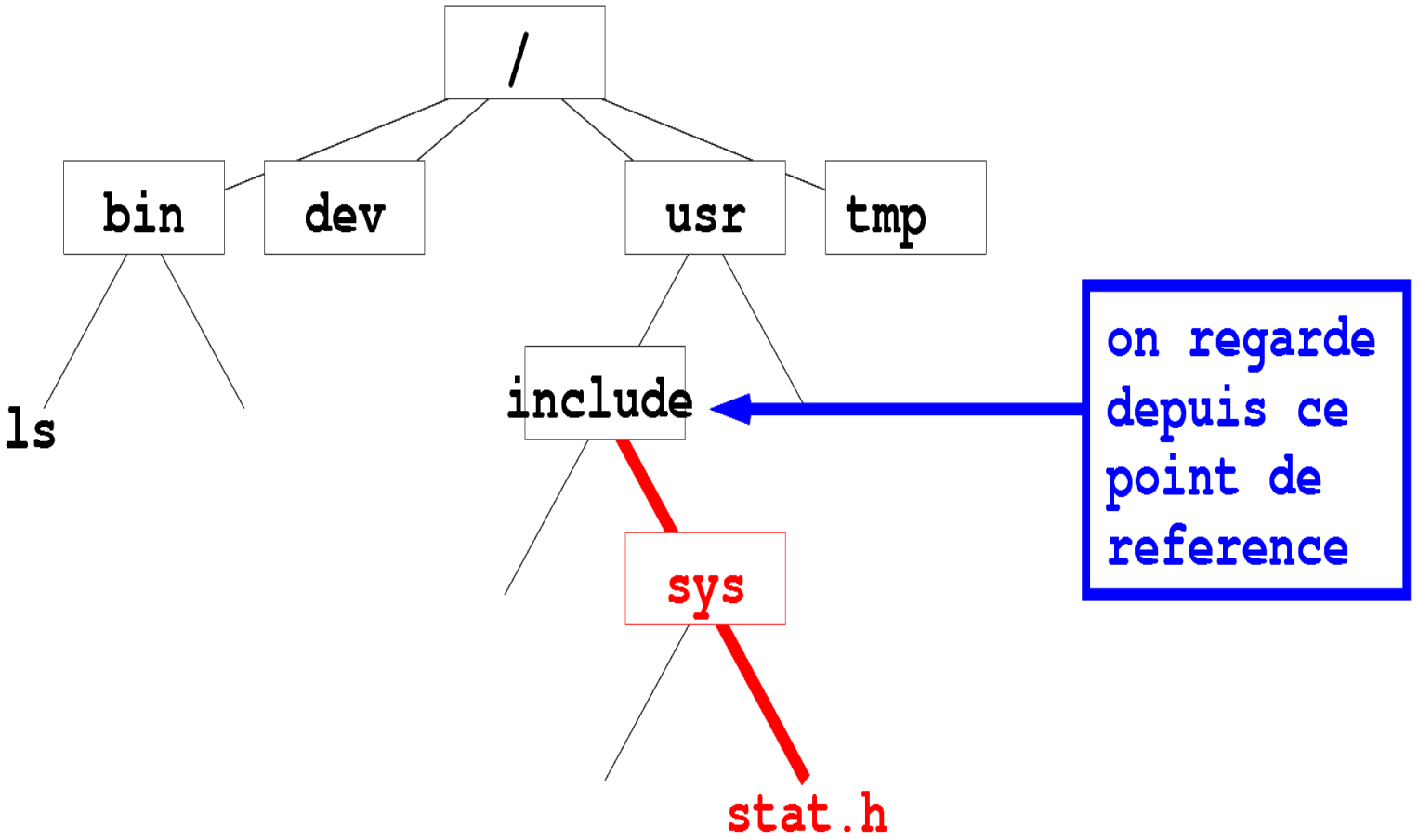
- Chemin d'accès « **absolu** »
  - C'est le chemin qui permet d'accéder à un fichier et **qui commence par la racine de l'arbre**
  - **Un chemin absolu doit toujours commencer par « / »**
  - exemple : « /usr/include/sys/stat.h »



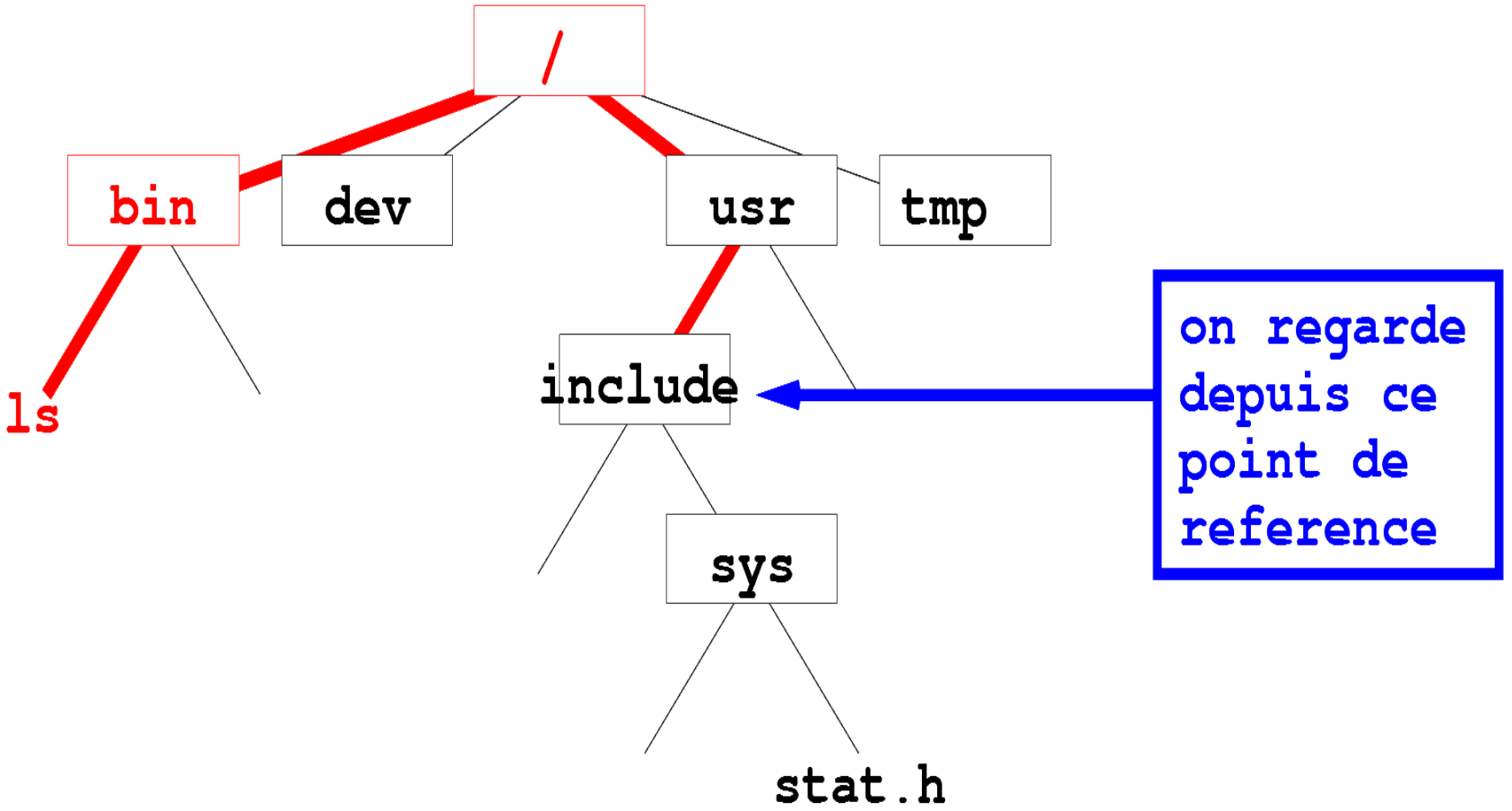
- Chemin d'accès « **relatif** »
  - C'est le chemin qui permet d'accéder à un fichier et **qui peut commencer à n'importe quel endroit de l'arbre excepté la racine**
  - **Un chemin relatif est « relatif » à la position de référence (le répertoire courant)**
  - le répertoire courant est noté « . »
  - le répertoire parent du répertoire courant est noté « .. »



- exemple 1 : depuis « /usr/include/ », le chemin relatif du fichier « stat.h » est « sys/stat.h »



- exemple 2 : depuis « /usr/include/ », le chemin relatif du fichier « ls » est « ../../bin/ls »



- Importance des écritures « . » et « .. »
  - commande « `find` » pour lancer une recherche à partir de l'endroit courant

```
$ find . –name fichier –print
```

  - pour lancer une commande qui se trouve dans le répertoire courant

```
$ ./macommande
```

- Comparaison Windows/Unix
  - Windows : plusieurs volumes « C:, D: »; « \ » comme séparateur de répertoires
  - Unix : une arbre unique; « / » comme séparateur de répertoires

# 103.1 – Path

- `PATH :` liste des répertoires de recherche des commandes séparés par le caractère « : »
- Parcours des répertoires jusqu'à trouver la commande en question
  - Exemple : `PATH=/bin:/usr/bin:/usr/local/bin`
  - `$ ls`
  - recherche dans « `/bin` » si « `ls` » existe. Si oui exécution, sinon chemin suivant
  - recherche dans « `/usr/bin` » si « `ls` » existe. Si oui exécution, sinon chemin suivant etc....
  - si la commande n'est trouvée dans aucun chemin « `command not found` »

- Pour modifier son `PATH` (« `$HOME/bin` » par exemple)
  - ```
    $ PATH=$HOME/bin:$PATH
    $ export PATH
    ```
  - ```
    $ PATH=$PATH:$HOME/bin
    $ export PATH
    ```
- Pour retirer un chemin il faut saisir complètement les chemins et les affecter à la variable `PATH`
- Ne pas mettre « `.` » dans son path
  - exécuter la commande dans le répertoire courant en la précédant de « `./` »si la commande n'est trouvée dans aucun chemin
    ```
    $ ./monscript.sh
    ```

- Différence habituelle du contenu de la variable `PATH` entre un utilisateur standard et root :
  - PATH de root contient les chemins /sbin, /usr/sbin, /sur/local/sbin
- Ce qui fait que
  - Sous l'identité de root, le shell trouve directement la commande ifconfig
  - Sous l'identité d'un utilisateur standard, il faut lui donner le chemin absolu de la commande (/sbin/ifconfig)

# 103.1 – Path

- Commande `pwd`
- Affiche le chemin absolu du répertoire courant (print working directory)
- Chemin présent également dans la variable `PWD`
- Changement de répertoire courant avec la commande `cd`
  - `cd chemin` : chgmt vers chemin
  - `cd ~` : chgmt vers le son répertoire personnel
  - `cd ~user1` : chgmt vers le répertoire personnel de l'utilisateur user1
  - `cd` : idem `cd ~`
  - `cd -` : chgmt vers `OLDPWD`

# 103.1 – Path

```
$ pwd
/home/franck/src/exo-lpi

[franck@port-135 exo-lpi]$ cd /etc

[franck@port-135 etc]$ pwd
/etc

[franck@port-135 etc]$ cd -
/home/franck/src/exo-lpi

[franck@port-135 exo-lpi]$ echo $PWD
/home/franck/src/exo-lpi

[franck@port-135 exo-lpi]$ echo $OLDPWD
/etc
```


- Attention à la syntaxe exacte des commandes