



LO03 : Programmation Shell sous Unix ou Linux

Rémi COGRANNE

e-mail permanent : remi.cogranne@utt.fr

téléphone en chine : 159 2100 2446

Hiver 2012-2013



Organisation de l'UV

- **Première partie** : architecture et théorie des systèmes d'exploitation
- **Deuxième partie** : cours langage système Unix-Linux
- **Troisième partie** : administration des systèmes



Principaux objectifs de l'UV

- Compréhension théorique du fonctionnement des systèmes d'exploitations (Windows, Linux, Unix, ...).
- Maîtrise de l'utilisation, de la programmation et des systèmes Unix et Linux.
- Être capable d'administrer un système informatique, notamment sous Linux/Unix



Introduction aux scripts

- Quelques livres
- "*Unix utilisateur*" éditions Eyrolles
- "*Unix shell*" éditions Eyrolles
- "*Unix administration*" éditions Eyrolles
- "*Systèmes d'exploitation*" d'Andrew Tanenbaum chez Pearson Education



Introduction au système Unix

- Système d'exploitation multi-utilisateurs (connexion simultanée de plusieurs utilisateurs) et multi-tâches (décomposition d'une application en plusieurs tâches qui s'exécutent simultanément).
- L'environnement standard de programmation est le C et plus récemment le C++.
- L'environnement réseau est construit sur les protocoles TCP/IP.



Les systèmes Unix du marché

- Il n'existe pas un , mais des systèmes Unix :
 - HP-UX (HP)
 - SunOS et Solaris (Sun)
 - AIX (IBM)
 - Digital UNIX (Dec)
 - GNU Linux (Mandriva, Red Hat, Debian, Ubuntu...)
- Linux est un produit dont la licence est gratuite.

Les systèmes Unix du marché

- Aujourd'hui, il existe deux grands types de systèmes Unix

Propriétaires



AIX



HP-UX Tru64 Unix



UnixWare



IRIS



SOLARIS

Libres



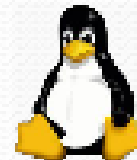
OpenBSD



FreeBSD



Mac OS



Gnu Linux



Unix ou Linux sur votre ordinateur

- Installer Linux sur une partition de votre disque. Double boot Windows-Linux
- Utiliser un boot CD Linux votre PC.
- Télécharger gratuitement un émulateur Unix (e.g : Cygwin).
- Utiliser une clé usb Linux.
- Virtualiser Linux sur votre PC avec Virtuabox par exemple.



Le shell et les commandes

- Le shell est un interpréteur de commandes qui invite l'utilisateur à saisir une commande et la fait ensuite exécuter.
- Il existe plusieurs interpréteurs shell : **Bourne again shell (bash)**, **C shell**, le **Korn shell**, etc...
- Tous les shells peuvent cohabiter à l'intérieur d'un même système Unix. L'administrateur fixe le shell initial de chaque utilisateur dans le fichier de définition des utilisateurs :
/etc/passwd



Quelques commandes

- **Date** : affiche la date et l'heure.
- **Cal** : affiche le calendrier.
- **Who** : affiche la liste des utilisateurs.
- **Whoami** : qui-suis-je ?
- **Uname** : affiche le nom et les caractéristiques du système.
- **Passwd** : modifie son mot de passe.

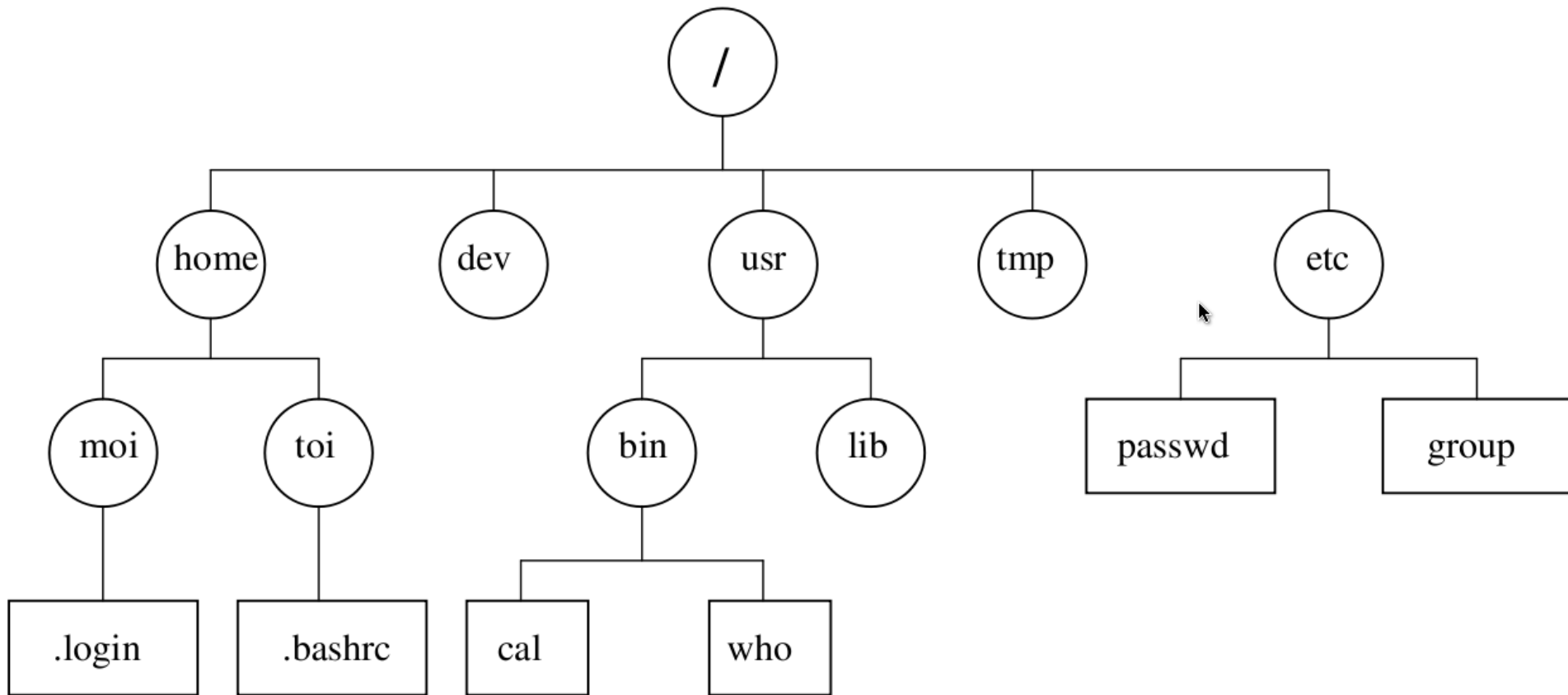


La documentation : Man

- **Name** : décrit en quelques mots l'objet de la documentation.
- **Synopsis** : présente la syntaxe de la commande.
- **Description** : donne la description précise du fonctionnement de la commande et de chacune de ses options.
- **Files** : fournit la liste des fichiers de configuration ou de données que la commande va mettre en œuvre.
- **See also** : renvoie à des éléments connexes de la documentation.
- **Exit status** : renseigne les codes de retour d'exécution, information utilisée pour la programmation Unix.



L'arborescence des fichiers



Sous Linux, toute l'arborescence part de « / », appelé « root » ou racine.
Dans ce répertoire racine se trouvent tous les dossiers élémentaires



Principaux répertoires

/bin	les binaires (exécutables) vitaux pour l'utilisateur
/boot	les fichiers relatifs au bootloader, ainsi que le noyau du système
/dev	tous les périphériques
/etc	les fichiers de configuration et des fichiers nécessaires au démarrage
/home	les répertoires personnels des utilisateurs
/lib	les bibliothèques partagées et les modules du noyau (/lib/modules)
/mnt	les dossiers pour les points de montage temporaires
/proc	accès direct aux paramètres du noyau ainsi qu'aux informations système (processeur, mémoire, ...)
/root	répertoire personnel du super utilisateur (root)
/sbin	les binaires vitaux pour l'administrateur (root)
/tmp	les dossiers et fichiers temporaires. Ce dossier est vidé à chaque démarrage du système.
/usr	Unix System Resources : tout ce qui n'est pas vital au système
/var	les fichiers qui changent fréquemment, tels que : les logs, les mails, queues d'impression



Les chemins

- Le chemin **d'accès absolu** décrit l'itinéraire à emprunter depuis la racine de l'arborescence jusqu'au fichier.
- Le chemin **d'accès relatif** décrit l'itinéraire à emprunter depuis le répertoire de travail courant. Si le fichier à atteindre est situé au dessous du répertoire de travail, l'utilisateur doit mentionner le chemin qu'il reste à parcourir.
- Le symbole « . » désigne le répertoire courant et le symbole « .. » sont père
- Le symbole « . » devant un nom de fichier signifie qu'il est caché (**ls -a** pour le voir).



Exemple de chemins

- Par exemple **/usr/bin/cal** est le chemin absolu du fichier cal.
- Si le répertoire de travail est le répertoire **/home/moi** :
 - **.login** est équivalent au chemin absolu **/home/moi/.login**
 - **./login** est équivalent au chemin absolu **/home/moi/.login**
 - **../toi.kde** est équivalent au chemin absolu **/home/toi.kde**

NB : Le symbole «~» désigne le répertoire de connexion de l'utilisateur. Par exemple, si l'utilisateur est moi, alors **~/.login** correspond au chemin relatif **/home/moi/.login**



Présentation de l'invite de commandes

- La ligne de commande se présente sous forme de texte ayant la signification suivante :

```
cogrannr@cogrannr-laptop:~$ cd /media/SWAP/Cours/L014_UTSEUS/L003
cogrannr@cogrannr-laptop:/media/SWAP/Cours/L014_UTSEUS/L003$ cd ~
cogrannr@cogrannr-laptop:~$ cd /
```

Diagram illustrating the components of a command prompt line:

- `user`: The username part of the prompt (e.g., `cogrannr`).
- `machine`: The machine name part of the prompt (e.g., `cogrannr-laptop`).
- `$`: Indicates the user is a regular user (Utilisateur).
- `#`: Indicates the user is the super-user (Super-utilisateur).
- `répertoire`: The current directory path (e.g., `/media/SWAP/Cours/L014_UTSEUS/L003`).

- Une commande est appelée en tapant son nom.
- Il suffit d'enter une commande pour qu'elle soit exécuter.



Syntaxe d'une commande

- Chaque commande a une syntaxe particulière.

cp [OPTION] ... [-I] SOURCE DEST

- Une commande est constituée d'options et de paramètres
- Les paramètres fournissent les données nécessaires à l'exécution de la commande : **ils sont obligatoires.**
- Les options de spécifier des fonctionnalités supplémentaires / spécifiques : **elles sont optionnels**

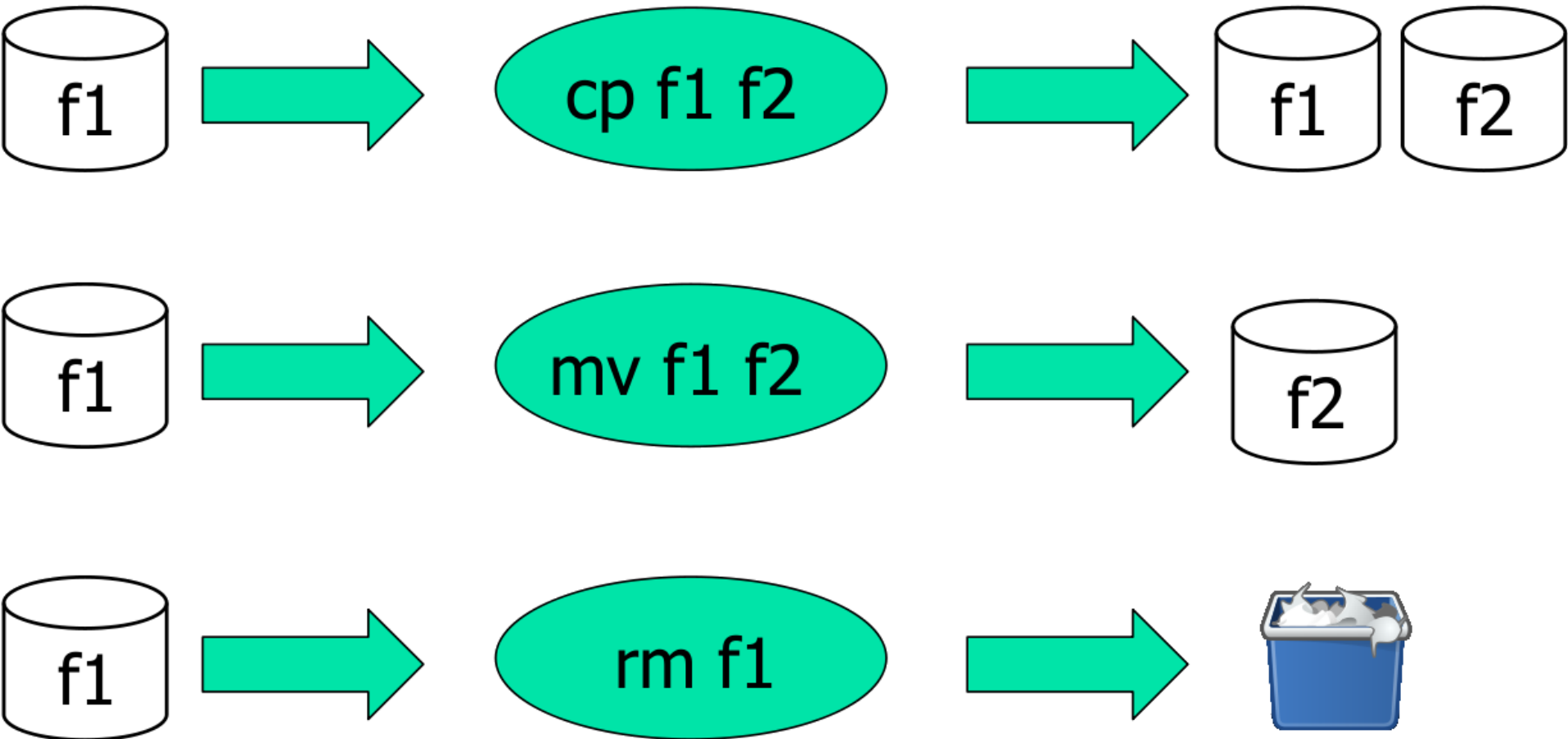
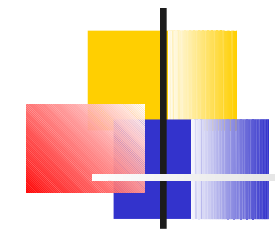
```
cogrannr@cogrannr-laptop:/media/SWAP/Temp$ ls
16_decembre.odt  CIE_Std_A.csv~  demande-achat.doc  formation  qqcmd
CIE_Daylight.csv~ Darmstadt       rs                 pipeline.png  std_D65_interp.txt
cogrannr@cogrannr-laptop:/media/SWAP/Temp$ cp ./qqcmd ./Darmstadt/
cogrannr@cogrannr-laptop:/media/SWAP/Temp$ cp -i ./qqcmd ./Darmstadt/
cp: overwrite './Darmstadt/qqcmd'? n
cogrannr@cogrannr-laptop:/media/SWAP/Temp$ cp -f ./qqcmd ./Darmstadt/
cogrannr@cogrannr-laptop:/media/SWAP/Temp$
```



Quelques commandes pour les fichiers

- **ls** : liste des fichiers dans un repertoire.
- **cp** : copie d'un fichier.
- **rm** : détruit un fichier.
- **mv** : change le nom d'un fichier, déplace un fichier.
- **cat, more** : affiche le contenu d'un fichier .
- **file** : affiche le type du fichier.
- **cmp, comm, diff** : compare des fichiers.

Copier, détruire, renommer un fichier

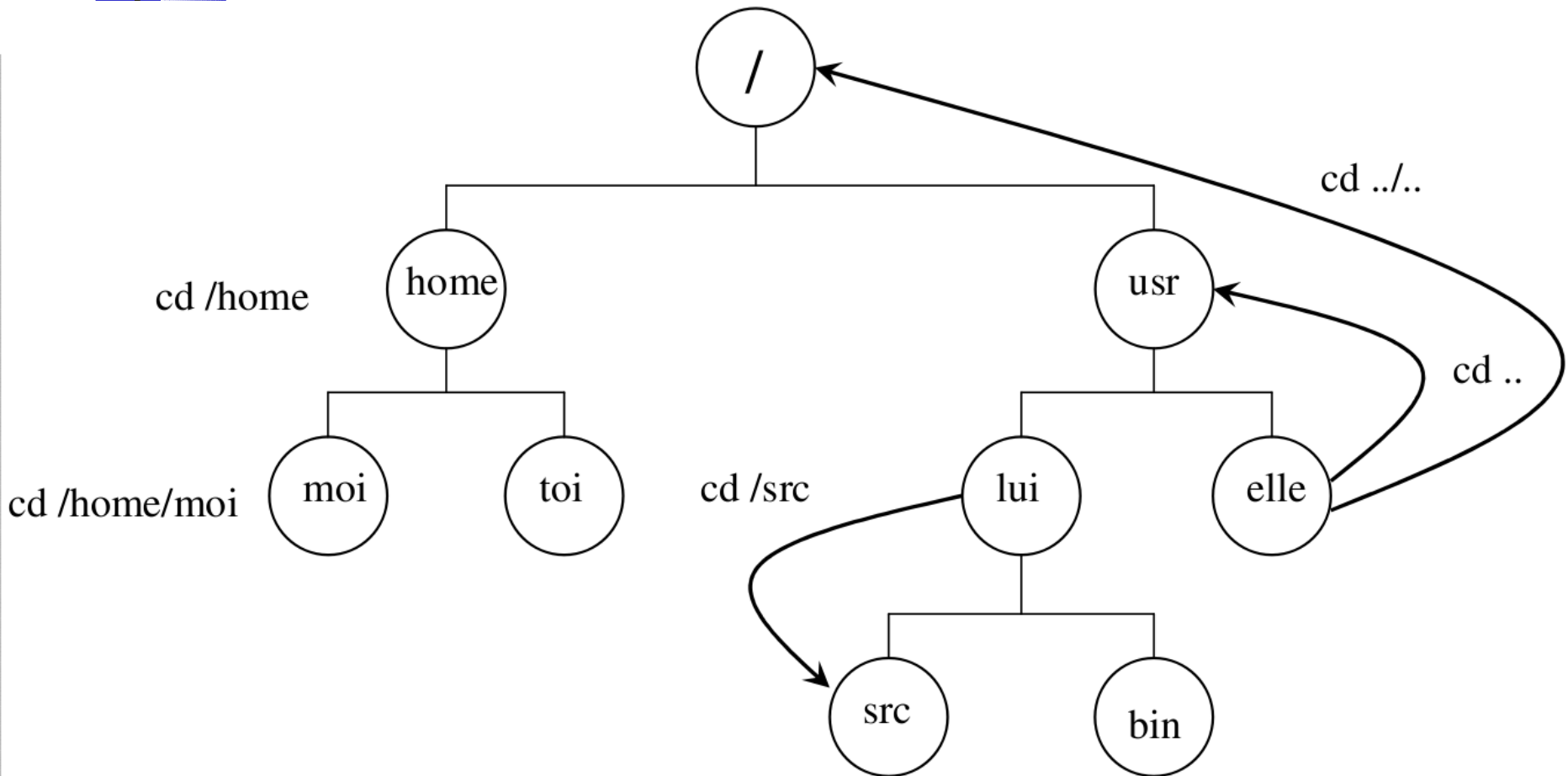




Quelques commandes pour les répertoires

- **pwd** : affiche le répertoire courant.
 - **cd** : change de répertoire.
 - **mkdir** : crée un répertoire.
 - **rmdir** : supprime un répertoire.
 - **du** : affiche la taille d'une arborescence.
- find** : recherche des fichiers dans une arborescence.
- locate** : recherche des fichiers dans une arborescence.

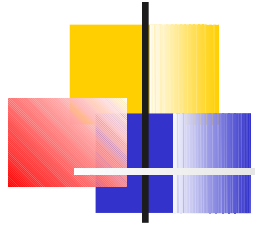
La commande `cd`





Le **shell** : généralité

- Le **shell** interprète les commandes.
- Le **shell** est un langage de programmation et les procédures de commandes s'appellent des *scripts*.
- Le **shell** possède des caractères spéciaux : les jockers, des caractères de protection et des caractères de redirection



Introduction aux scripts

- L'environnement d'un utilisateur (type de terminal, choix du prompt, définition des touches d'effacement des caractères,...) est construit à partir de deux scripts :
 - le script `/etc/profile` commun à tous les utilisateurs.
 - le script `~/.bash_profile` qui se trouve dans le répertoire de connexion de l'utilisateur.



Les commentaires

- Dans une ligne de commande, le caractère « # » indique le début d'un commentaire qui se poursuit jusqu'à la fin de la ligne.
- Le pseudo commentaire « #! », placé en début de script, permet de désigner explicitement le shell qui doit exécuter le script.
- `#!/bin/ksh` # ce script est exécuté par le Korn shell
Ce script affiche la liste des processus
`ps`



Exécution d'un script

- Exécution d'un script

\$ bash lescript

\$ bash <lescript

\$ chmod +x ; ./lescript

- Activation/désactivation du mode trace dans le script :
set -x

- Exécution d'un script avec l'affichage de la trace des commandes
\$ bash -xv lescript



Les variables

- Création d'une variable :
variable=valeur
- Valeur d'une variable :
\$variable
- Liste des variables :
La commande « set » liste l'ensemble des variables
- Suppression d'une variables :
unset variable



Les variables d'environnement

```
$ var_locale=alpha
```

```
$ var_environ=gamma
```

```
$ export var_environ
```

- La commande « `export` » permet à un shell de placer une variable dans l'environnement.

Syntaxe `$ export variable[=valeur]`

- La commande « `env` » permet de lister l'ensemble des variables exportées.



Les paramètres

- Un script est à l'image d'une commande. Il est possible de fournir, sur la ligne de commandes, les arguments nécessaires à son exécution.
- Les arguments sont affectés aux variables 0,1,...,9,10,11,...
- La variable « * » désigne l'ensemble des paramètres sous la forme d'un seul argument.
- La variable « @ » désigne l'ensemble des paramètres, un argument par paramètre



Les paramètres

La variable « # » désigne le nombre de paramètres passés au script.

- La commande « shift » permet de décaler les paramètres : la valeur de la variable 1 est remplacée par celle de la variable 2, celle de la variable 2 par celle de la variable 3, etc.

shift [n] : décale de n paramètres

- La commande interne « set » permet de remplacer tous les paramètres.

```
$ set un deux trois ; echo $*
```

```
un deux trois
```



L'instruction read

- L'instruction « read » permet de renseigner des variables à partir de l'entrée standard, le clavier par défaut :

\$ read variable

- La commande « readonly » permet de créer des variables qu'on ne peut plus supprimer et dont on ne peut plus modifier la valeur:

\$ readonly variable=valeur

- L'option -a (array) permet de renseigner des tableaux.

Définition de séparateur de champs : IFS



- La commande « read » lit une ligne entière et utilise la variable « IFS » pour décomposer la ligne en mots qu'elle va affecter aux différentes variables de lecture qui lui sont passées en argument.
- Par défaut, la variable « IFS » est initialisée avec les caractères <Espace>, <tabulation> et <Saut de ligne>.
- Modification de la variable « IFS » :
 - IFS=,



Les tableaux

```
$ tableau[0]=un; tableau[1]=deux; tableau[2]=trois
```

```
$ echo ${tableau[0]}
```

```
un
```

```
$ tableau[1]=six
```

```
$ echo ${tableau[*]}
```

```
un six trois
```

```
$ echo ${#tableau[*]}
```

```
3
```

Alternative:

```
$ tableau_OS=(Linux Windows MacOSX freeBSD)
```




Les tableaux (avec shell et bash)

- Les shells sont des exécuteurs de commandes.
Malgré la proximité entre les différents shells des différences existent.
(*bash* : bourne again shell , *csh* : c shell , *ksh* : korn shell)
- L'exemple de la gestion des tableau du shell (sh) illustre cela ;
en tapant la ces commandes sous on obtient :

```
$ tableau=(1 2 3)
```

```
sh: Syntax error: "(" unexpected
```

```
$ tableau[0]=1
```

```
sh: tableau[0]=1: not found
```



Les instructions de contrôle

- Comme les langages de programmation, le langage shell possède des instructions de contrôle telles que par exemples:
 - L'alternative : l'instruction if.
 - Les structures itératives : les instructions for, while, until.
 - Le choix multiple : l'instruction case.



if : l'alternative

```
■ if condition
    then
        bloc de commandes
    elif condition
        then
            bloc de commandes
    else bloc de commandes
fi
```



L'alternative : exemple

```
If test -f $1
```

```
    then
```

```
        file $1
```

```
    else    echo " le fichier $1 n'existe pas "
```

```
fi
```

- Si le fichier dont le nom est passé en paramètre existe, alors il affiche le type de son contenu.
- Il est possible de positionner les mots clés « if » et « then » sur la même ligne en les séparant par un point virgule :
 - if test -f \$1 ; then file \$1



Le code retour

- Une commande Unix ou un script qui se termine émettent un code de retour numérique.
- La variable prédéfinie « ? » est automatiquement initialisée avec le code retour de la dernière commande exécutée.

```
$ who | grep thomas
```

```
$ echo $?    # thomas est-il connecté ?
```

```
0           # thomas est connecté
```

```
1           # thomas n'est pas connecté
```



Le code retour et l'alternative

- Il est possible de spécifier le code retour d'un script en utilisant la commande :
`exit xx` `#xx` est un entier
- L'alternative *if*, évalue la condition et exécute les commandes du bloc *then* si le code retour est 0 (et non pas 1 comme en C)



L'alternative && et ||

- Les opérateurs « && » et « || » permettent de conditionner l'exécution d'une commande par le code retour de celle qui précède.
- `commande1 && commande2`
 - équivaut à : `if commande1 ; then commande2 ; fi`
 - « `commande2` » est exécutée si « `commande1` » a renvoyée 0.
- `commande1 || commande2`
 - Équivaut à : `if commande1 ; then : ; else commande2 ; fi`
 - « `commande2` » est exécutée si « `commande1` » a renvoyée 1.



La commande test

- La syntaxe
 - if **test** expression ; then fi
 - if [expression] ; then fi
- La commande test n'affiche pas de résultat sur la sortie standard, elle renvoie le code retour 0 si l'expression est vraie et 1 si elle est fausse.



Test d'un attribut de fichier

- **-f fichier** : vrai si le fichier existe et qu'il est ordinaire.
- **-e fichier** : vrai si le fichier existe.
- **-r fichier** : vrai si le fichier existe et qu'il est accessible en lecture.
- **-w fichier** : vrai si le fichier existe et qu'il est accessible en écriture.



Test sur des chaines de caractères

- **-z chaine** : vrai si la longueur de la chaine est 0.
- **-n chaine** : vrai si la longueur de la chaine est différente de 0.
- **chaine1 = chaîne2** : vrai si chaine1 est identique à chaîne2
- **chaine1 != chaîne2** : vrai si chaine1 est différente de chaîne2



Test sur les nombres

- **arg1 –eq arg2** : vrai si arg1 est égal à arg2.
- **arg1 –ne arg2** : vrai si arg1 est différent de arg2.
- **arg1 –lt arg2** : vrai si arg1 est strictement inférieur à arg2.
- **arg1 –gt arg2** : vrai si arg1 est strictement supérieur à arg2.
- « **le** » et « **ge** » pour inférieur ou égal et supérieur ou égal.



Opérateurs logiques

- **!expression_logique** : vrai si expression_logique est fausse.
- **expression_logique1 –a expression_logique2** : vrai si les deux expressions sont vraies.
- **expression_logique1 –o expression_logique2** : vrai si l'une des deux expressions est vraie.



La structure case

```
■ case paramètre in
    choix1 )          commandes ;;
    [ choix2 )        commandes ;; ]
    * )              commandes ;;
esac
```

- Chaque valeur du paramètre est terminée par «) ». Si plusieurs valeurs sont à traiter de la même manière, on les sépare par le signe « | ».
- Chaque bloc de commandes se termine par « ;; ». La choix par défaut se marque par « *) » et est toujours situé en dernier.



La structure case : exemple

```
$ cat supprime
```

```
echo "voulez vous supprimer le fichier ?"
```

```
read reponse
```

```
case $reponse in
```

```
    O | o ) rm $1 ;;
```

```
    N | n ) echo "le fichier n'est pas supprimer" ;;
```

```
    * )      echo "réponse incorrecte" ;;
```

```
esac
```



La boucle tant que

- Syntaxe :

```
while condition
do
    commandes
done
```
- Exemple :

```
while [ -r $1 ]
do
    cat $1 >> result
    shift
done
```
- Concatène dans le fichier « result » l'ensemble des fichiers dont les noms sont donnés en argument.



La boucle répéter jusqu'à

- Syntaxe :

```
until condition
do
    commandes
done
```
- Exemple :

```
until [ ! -r $1 ]
do
    cat $1 >> concat
    shift
done
```

- Notez la négation du test grâce à l'opérateur de négation.



La boucle for

- Syntaxe :
for paramètre [**in** liste]
do
commandes

- Exemple :
for i in `ls`
do
cp \$i /tmp/\$i
echo "\$i copié"
done

- « `ls` » signifie le résultat de l'exécution de « ls ».
Recopie chaque fichier du répertoire courant dans « /tmp ».



Les sauts inconditionnels

while ... ; do

....

if ... ; then

continue

fi

if ... ; then

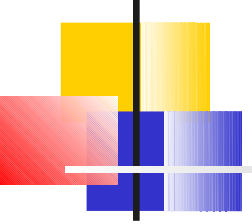
break

fi

done

....

L'arithmétique

- 
- La commande « `expr` » offre la possibilité d'effectuer des opérations arithmétiques sur des variables qui contiennent un nombre entier.

- syntaxe : **`expr`** arg1 opérateur arg2

- `expr 3 + 2 * 5 = 13`

la multiplication est prioritaire sur l'addition !

- `expr \ (3 + 2 \) * 5 = 25`

- La commande « `let` » permet de faire une opération et une affectation



La commande « expr » : exemple

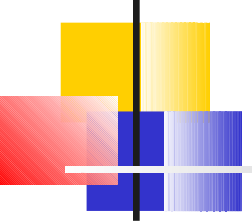
```
a=1
while [ $a -le 10 ]
do
    echo $a
    a=`expr $a + 1`
    #let a="$a+1" alternative possible : attention aux espaces !
done
```

- affichage des nombres entiers de 1 à 10

Les opérateurs

- On retrouve l'ensemble des opérateurs du langage C :
 - « ! » : Non logique
 - « || » : Ou logique
 - « | » : Ou arithmétique (bit à bit)
 - « == » : égalité
 - « != » : différence
 - « <= » : inférieur ou égale
 - +, -, *, /, % (modulo)

La boucle "for" en arithmétique

- 
- Syntaxe : **for** ((expr_init ; expr_arret ; expr_boucle))
do
 commandes
done
 - Exemple : **for** ((x=1,y=10 ; x<4 ; x++,y--))
do
 a=`expr x * y`
done

10
18
24



Les fonctions

- **Syntaxe :** **function** Nom {
 lignes de code ;
 }

ou

```
Nom ( ) {  
lignes de code ;  
}
```

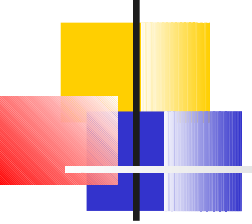
- **Déclaration des variables locales**
 - integer var ou typeset -i var : déclaration d'un entier
 - typeset -r var=valeur : définition d'une constante



Les fonctions

- Une fonction est exécutée en invoquant son nom suivi d'éventuels paramètres qui seront référencés de la même manière qu'un script (\$0, \$1, ...).
- La commande « unset -f » permet de supprimer une fonction
- La commande « return » met fin à une fonction. Le code retour est stocké, comme le retour d'un script, dans la variable « ? » du shell.

Rediriger les entrées-sorties de tout un script

- 
- Dans un script la commande « exec » redirige les entrées-sorties standard de façon permanente pour toutes les commandes qui suivent.
 - `exec > nom_fichier`
redirige la sortie standard vers le fichier nom_fichier (RAZ)
 - `exec >> nom_fichier`
même chose mais la sortie est concaténée avec nom_fichier
 - `exec < nom_fichier`
redirige l'entrée standard depuis le fichier nom_fichier



Lecture d'un fichier dans une boucle

```
$ more programme
```

```
#!/bin/ksh
```

```
while read ligne
```

```
do
```

```
    echo ">$ligne"
```

```
done < fichier
```

```
$ more fichier
```

```
Bonjour
```

```
Hello
```

```
$ programme
```

```
>Bonjour
```

```
>Salut
```



La programmation en parallèle

\$ more programme

#!/bin/ksh

cal

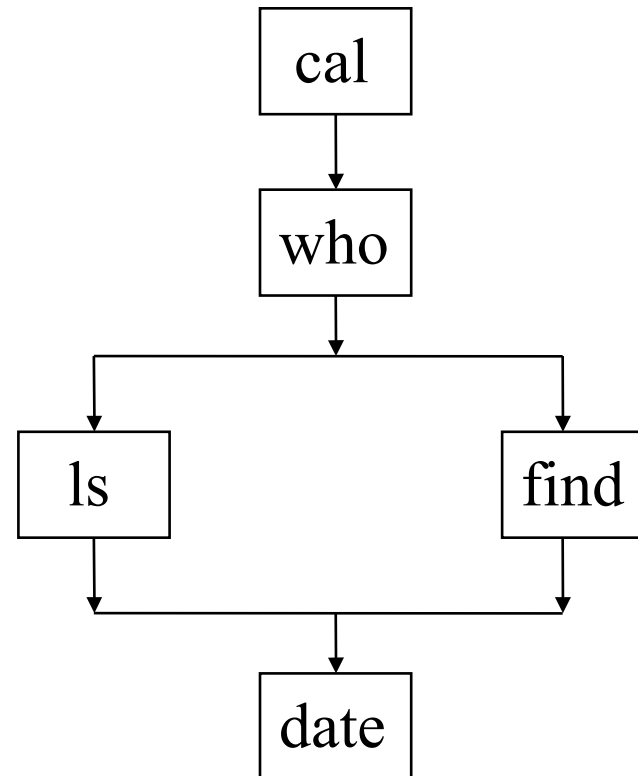
who

ls -lR / &

find / -print &

Wait

date



Le regroupement de commande dans un shell



Le shell permet le regroupement d'un ensemble de commandes.

- Les principales utilisations sont :
 - la redirection d'une séquence de commandes vers ou depuis un même fichier,
 - l'exécution en arrière plan d'un ensemble de commandes,
 - Le conditionnement de l'exécution d'un ensemble de commandes
- { cmd; cmd; ... } exécutées par le shell courant
- (cmd ; cmd; ...) exécutés par un shell fils



Les variables \$ et !

- La variable « ! » contient le numéro de processus de la dernière tâche lancée en arrière plan.
- La variable « \$ » permet de tuer tous les processus lancés par le script.

a=1

(while test \$a -eq 1

#ceci est une boucle "sans fin"

do

sleep 1 ; echo "bonjour"

done) &

sleep 5 ; kill \$!

echo "c'est fini"

Question :

Que fait ce script ?

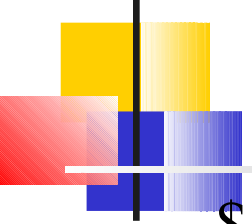
(bravo celui qui trouve)



L'échange de données par tube nommé

- Pour échanger des données, deux processus peuvent utiliser un tube « nommé » : le premier processus écrit ses résultats dans le tube et le deuxième lit ses entrées depuis se tube.
- La commande « exec » permet d'ouvrir un tube pour la lecture ou l'écriture de la même manière que pour un fichier.
- La commande « read » permet la lecture dans un tube.
- `mknod nom_du_tube p` : création du tube `nom_du_tube`

L'échange de données par tube nommé : exemple



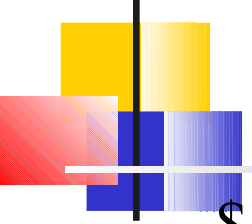
```
$ more ex_p40_p1
exec > mon_tube
while read var
Do
    echo $var
Done
```

```
$ mknod tube p
```

```
$ more ex_p40_p2
exec < mon_tube
while read ligne
do
    echo ">$ligne"
done
```

```
$ bash ex_p40_p2 &
$ bash ex_p40_p1
bonjour
>bonjour
```

L'échange de données par tube nommé : exemple



```
$ more ex_p40_p1
exec > mon_tube
while read var
Do
    echo $var
Done
```

```
$ mknod tube p
```

```
$ more ex_p40_p2
exec < mon_tube
while read ligne
do
    echo ">$ligne"
done
```

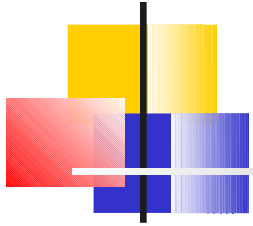
```
$ bash ex_p40_p2 &
$ bash ex_p40_p1
bonjour
>bonjour
```


L'échange de données par le tube

« | »

- Le caractère « | » sert à créer un tube temporaire entre deux commandes :
 - Le résultat de la première commande est écrit dans un tube
 - La seconde commande récupère immédiatement les données écrites dans le tube
- Exemples :
 - `$ ls | wc` → `wc` permet de compter le nombre de ligne, mots ou caractères dans un fichier
 - `$ echo Bonjour toto UTSEUS | cat`

Mise en œuvre avec la commande : **cut**



- La commande **cut** permet d'extraire des champs d'un fichier :
 - En utilisant un nombre de caractères : **-c**
 - En utilisant un délimiteur de champs : **-d x**
 - En utilisant un nombre de champs : **-f**
- Exemples :
 - `$ echo "toto.UTSEUS" | cut -d . -f 2` → UTSEUS
 - `$ echo Bonjour | cut -c 1,3,4,6-7` → Bnjur
 - `$ cut -d ":" -f 1-3 /etc/passwd`
 - Extraction des 3 premiers champs séparés par « : »

Protection ou banalisation des caractères spéciaux

- Nous avons vu certains caractères \ * | < > qui ont une utilisation particulière et sont donc interprétés
- Pour que ces caractères soient utilisés comme des caractères simples il faut les protéger ou les banaliser.
 - En utilisant le caractère \ devant le caractère que l'on souhaite utiliser

```
$echo <bonjour
bash: bonjour: No such file or directory
$echo \<bonjour
<bonjour
```

Protection ou banalisation des caractères spéciaux



- En utilisant les quotes `""` pour protéger toute la chaîne de caractères SAUF les quatre suivants `$ \ ' "`

```
$echo <| | -<bonjour
```

```
bash: syntax error near unexpected token ...
```

```
$echo <| | -<bonjour
```

```
<| | -<bonjour
```

- En utilisant les quotes `' '` pour protéger toute la chaîne de caractères

```
$echo $HOME" "
```

```
/home/cogrannr
```

```
$echo '$HOME" " '
```

```
$HOME" "
```



La gestion des menus (ksh)

```
PS3="votre choix ?"
select choix in \
"archive" "restaure" "fin" \
do
    echo "==> $choix"
    case $REPLY in
        1) tar c ;;
        2) tar x ;;
        3) break ;;
    esac
done
```

1) archive

2) restaure

3) fin

votre choix ? 2



==> restaure

fich1

fich2

1) archive

2) restaure

3) fin

votre choix ?



La gestion des menus

- Le contenu de la variable « PS3 » s'affiche et l'entrée standard est lue. Si le numéro d'un des mots de la liste est saisi, le paramètre « choix » prend la valeur.
- Si la ligne est vide, la liste s'affiche à nouveau.
- Le contenu de la ligne lue est sauvegardé dans une variable « REPLY ». La liste est exécutée jusqu'à un caractère d'interruption saisi et qui est proposé dans la liste.



La commande xargs

La commande « xargs » génère les arguments d'une commande depuis l'entrée standard. Elle permet de récupérer les arguments passés par la commande précédente.

```
$ cat >liste
```

```
fich1
```

```
fich2
```

```
$ cat liste | xargs tar -cf mon_zip
```

```
# tar -c fich1 fich2 -f mon_zip
```

équivalent à `tar -c $(cat liste) -f mon_zip`



La commande what

```
$ more lescript
```

```
#!/bin/ksh
```

```
# @(#) lescript du 25/03/2005
```

```
# @(#) version 1.3
```

```
date
```

```
$ what lescript
```

```
lescript du 25/03/2005
```

```
version 1.3
```




Les expressions régulières

- Une expression régulière sert à identifier une chaîne de caractère répondant à un certain critère.
- Par exemple une chaîne contenant des lettres minuscules uniquement.
- Les commandes « grep », « sed », ... utilisent les expressions régulières.

Les expressions régulières



■ Le méta caractère « . »

- remplace dans une expression régulière un caractère quelconque.

■ Les méta caractères « [] »

- désigne des caractères compris dans un certain intervalle de valeur
 - « [Ff]raise » identifie « Fraise » ou « fraise »
 - « [1-3-] » identifie « 1 », « 2 », « 3 » et « - »
 - « [a-cI-K1-3] » identifie « a », « b », « c », « I », « J », « K », « 1 », « 2 », « 3 »
 - « [^0-9] » : identifie autre chose qu'un chiffre

Les expressions régulières

- Les méta caractères « ^ » et « \$ »

- « ^ » identifie un début de ligne
attention à ne confondre « ^x » qui représente x en début de ligne et « [^x] » qui représente tout sauf x
- « \$ » identifie une fin de ligne
 - « ^chaine » identifie les lignes qui commencent par la chaîne « chaîne »
 - « chaîne\$ » identifie une ligne qui se termine par la chaîne « chaîne »

Les expressions régulières

Compter le nombre de fois qu'apparaît un motif

- Deux solutions possible l'utilisation de $\{X\}$ ou de $*$ $?$ $+$
 - « x^* » identifie la chaîne de caractères x répétée 0 ou plusieurs fois – on peut aussi écrire $x\{0,\}$.
 - « x^+ » identifie la chaîne de caractères x répétée 1 ou plusieurs fois – on peut aussi écrire $x\{1,\}$.
 - « $x^?$ » identifie la chaîne de caractères x répétée 0 ou 1 fois – on peut aussi écrire $x\{0,1\}$.
 - « $x\{n\}$ » identifie la chaîne de caractères x répétée n fois.
 - « $x\{n1,n2\}$ » identifie la chaîne de caractères x répétée entre $n1$ et $n2$ fois
 - « $x\{,n\}$ » et « $x\{n,\}$ » identifie la chaîne de caractères x répétée au plus ou au moins n fois

Mise en œuvre avec la commande grep



```
$ grep /sh$ /etc/passwd
```

/sh en fin de ligne

```
$ grep '.*sh$' /etc/passwd
```

la même chose !

```
$ ls -l | grep '^d'
```

le caractère 'd' en début de ligne (dir)

```
$ ls -l | grep '^[^d]'
```

les fichiers qui ne sont pas des directory

```
$ ls -l | grep '[0-9]Mar'
```

Un chiffre entre 0 et 9 suivi
de la chaîne « Mar »

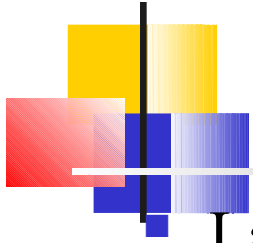
Mise en œuvre avec la commande grep (quelques exercices)

- Chercher les lignes contenant des chaînes de 8 caractères alphanumérique commençant par un nombre (par une lettre)
`$ grep '[0-9][a-zA-Z0-9]{7}' ('[0-9][a-zA-Z0-9]{7}')`
- Afficher les lignes vides
`$ grep '^$' /etc/passwd`
- Afficher les lignes ne contenant que le texte toto
`$ grep '^toto$' /etc/passwd`
- Afficher uniquement le nom des fichiers avec aucune MAJ.
`$ ls | '^^[A-Z]*$'`
- Afficher les fichiers courants du répertoire de plus de 1Mo.
`$ ls -l | grep '^-.*[0-9]\{7,\}'`

La commande tr (transcode)

- La commande tr (transcode) permet de remplacer un groupe de caractères par un autre de la même taille
- **Syntaxe :** tr [option] chaîne 1 chaîne2
- Principales options
 - -s 'x' (squeeze) permet de supprimer les répétitions des caractères de la chaîne 'x'
 - -d 'x' (delete) permet de supprimer les caractères de la chaîne 'x'
- **Exemples :** cat /etc/passwd | tr 'a-z' 'A-Z'
transforme les majuscules en minuscules
- **Exemples :** ls -l | tr -s ' '
supprimer les espaces multiples
- **Exemples :** cat /etc/passwd | tr -d ':/,'
supprimer tous les caractères : / ,

La commande **sort** (pour trier)



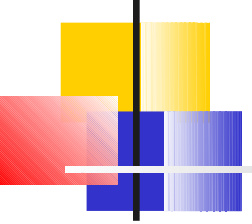
- La commande **sort** permet de trier une liste ou un fichier
- **Syntaxe** : `sort [option] fichier ou cat fichier | sort [options]`
- Principales options
 - `-t` : permet de spécifier le caractère séparateur de champs
 - `-k` : identifie la colonne utilisée pour le tri
 - `-r` : identifie le tri suivant l'ordre inverse
 - `-n` : identifie le tri numérique
- **Exemples** : `sort -t: -k3n /etc/passwd` : Tri le fichier `/etc/passwd` suivant le 3ème champ (uid)
- **Exemples** : `ls -l | tr -s ' ' | sort -rt ' ' -k 9`
trier la liste des fichiers par leurs noms en sens inverse



La commande sed

- Le filtre « sed » (stream editor) est un filtre programmable. Les commandes du filtre « sed » sont appliquées à des fichiers, l'entrée standard par défaut ; le résultat du traitement est envoyé sur la sortie standard.
- La commande « sed » consiste à rechercher des instances d'une chaîne de caractères ou d'une expression régulière et à les remplacer par une autre chaîne ou une autre expression régulière. Il ne modifie pas le fichier traité ; il écrit seulement le résultat sur la sortie standard

La commande sed

- 
- Les commandes appliquées au fichier peuvent être incluses dans un fichier : « fichier_programme »
 - Syntaxe :
 - `sed -e 'programme sed' fichier_a_traiter`
 - l'option « -e » n'est pas utile si on a une seule commande à appliquer
 - `sed -f fichier_programme fichier_a_traiter`

Principe de fonctionnement de la commande sed



■ Utilisation de plusieurs instructions sur la ligne de commandes :

\$ more fich

Il fait beau

\$ sed -e 's/u/U/g' -e 's/a/A/g' fich

Il fAit beAU

■ Utilisation d'un fichier de commandes :

\$ more prog

s/u/U/g

s/a/A/g

\$ sed -f prog fich



Les commandes de sed

- Substitution de la chaîne « toto » par la chaîne « tata » :
`$ sed 's/toto/tata/g' fich`
- Afficher les trois premières lignes d'un fichier :
`$ sed '3q' fich`
- Ne pas afficher les lignes qui contiennent la chaîne « toto » :
`$ sed -e '/toto/d' fich`
- Afficher la dernière ligne d'un fichier :
`$ sed -n '$p' fich`
- Afficher les lignes 5 à 10 d'un fichier :
`$ sed -n '5,10p' fich`

Utilisation des expressions régulières dans la commande sed

- `$ sed '/^le/p' fich`

Afficher les lignes qui commencent par « le »

- `$ sed '/filtre$/p' fich`

Afficher les lignes qui se terminent par commencent par « filtre »

- `$ sed 's/^/ligne :/' fich`

Ajouter « ligne : » en début de chaque ligne

- `$ sed 's/filtre/(&)/g' fich`

Mettre le mot « filtre » entre parenthèses

Les sous expressions régulières dans la commande sed

- \$ more fich
un,deux,trois
one,two,three
\$ sed 's/\(.*\),.*,\(.*\)/\1:\2/' fich
un:trois
one:three
- \$ more fich
10000 Troyes
\$ sed 's/^\(.....\)\(.*\)\$/\2\1/' fich
Troyes10000

sed : autres exemples

- `$ sed 's^\([0-9][0-9]*\) /aa\1aa/ '`
 - chaque groupe de chiffre (1 ou plusieurs) sera entouré des caractères aa. La chaîne to2to deviendra toaa2aato
- `$ cat ./liste_users | grep /home | sed '/^db/d'`
 - on supprime toutes les lignes commençant par "db"
- `$ cat ./liste_users | grep /home | sed '/^[^db]/d'`
 - on supprime toutes les lignes ne commençant pas par "db"
- `$ sed '1,10d' fichier`
 - affichage du fichier « fichier » à partir de la onzième ligne



sed : quelques petits exercices

- Supprimez toutes les lignes vides du fichier
 - `$ sed '/^$/d' fichier`
- Supprimez toutes les espaces d'un fichier
 - `$ sed 's/ //g' fichier`
- Réécrivez le fichier en ne laissant que le dernier mot par ligne
 - `$ sed 's/^. * \([^]*\$\)^1/' fichier`

La commande awk

- « awk » est une commande très puissante, c'est un langage de programmation à elle seule qui permet une recherche de chaînes et l'exécution d'actions sur des lignes. Elle est très utile pour récupérer de l'information, générer des rapports et transformer des données.
- Une grande partie de la syntaxe a été empruntée au langage C. « awk » est l'abréviation de ces trois créateurs (k pour kernighan l'inventeur du langage C).
- *Exemple : comment connaître le numéro occupé par une valeur dans un tableau*

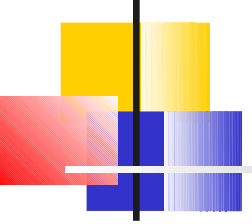
La commande awk



■ Syntaxe

- `$ awk [-F] [-v var=valeur] 'programme' fichier`
- `$ awk [-F] [-v var=valeur] -f fichier_config fichier`
- L 'argument « F » doit être suivi d'un séparateur de champ (« -F: » pour « : » comme séparateur)
- L 'argument « f » est suivi d'un fichier de configuration
- L 'argument « v » définit une variable qui sera utilisée par la suite dans le programme
- Exemple : `awk -F: '{printf $1}' /etc/passwd`

La commande awk

- 
- Enregistrements et champs
 - « awk » scinde les données d'entrée en enregistrements et les enregistrements en champ.
 - Un enregistrement est une chaîne d'entrée délimitée par un retour chariot
 - Un champ est une chaîne délimitée par le caractère de séparation
 - Dans un enregistrement les champs sont référencés par \$1, \$2,, \$NF
 - \$0 désigne l'enregistrement



La commande awk

- « awk » permet décrire des programmes complets, avec des structures itératives, des tests, etc..
- Le programme est passé en paramètre (encadré avec des quotes) à « awk » ou écrit dans un fichier dont le nom est fourni par l'intermédiaire de l'option -f
- Exemples :
 - `awk '{print $1, $3 }' mon_fichier`
 - `{print $1, $3}` est le programme fourni à la commande « awk » et s'exécute sur « mon_fichier »



Structure du programme

- **BEGIN**{ action_begin}
sélection_1 { action_1 }
sélection_2 { action_2 }

...
END{ action_end}
- On peut omettre la séquence **BEGIN**{ } et/ou **END**{ }
- Si on omet « sélection_1 », alors « { action_1 } » est exécutée pour toutes les lignes du fichier
- Si on omet « { action_1 } », alors les lignes pour lesquelles « sélection_1 » est valide sont copiées sur la sortie standard



Séparation des champs

- Pour utiliser des séparateurs de champs différents de « espace » et de « tabulation », on utilise l'option « -F*séparateur* » ou on positionne la variable FS dans la séquence « BEGIN{ } » du programme.

- Exemple :

```
$ echo "05/11/92" | awk -F/ '{ print $3 }'
```

```
92
```

```
$ echo "05/11/92" | awk '{ FS="/"; print $3 }'
```

```
92
```



Séparation des champs : autre exemple

```
$ more fich
```

```
  val=3, val=4,val=5
```

```
110 val=2, val = "valeur"
```

```
$ awk ' BEGIN { FS = "val="} '{ $1 = $1 ; print }' fich
```

```
  3, 4, 5
```

```
110 2, "valeur "
```

- On considère le séparateur de champs constitué de la chaîne « val= »



Variables internes à awk

- \$0 : l'enregistrement (ligne) courant.
- \$i : le ième champ de l'enregistrement courant.
- NF : nombre de champs dans l'enregistrement courant.
- NR : désigne le curseur de l'enregistrement courant (tous fichiers confondus).
- FNR : nombre de l'enregistrement (du fichier courant).
- FILENAME : nom du fichier d'entrée courant.
- OFS : permet de spécifier le séparateur de champs en sortie.



Variables internes à awk

- Remarques:
 - si la variable « A » vaut « 3 », alors « \$A » est le troisième champ de la ligne.
 - « \$NF » est le dernier champ de la ligne, « (\$NF-1) » est l'avant dernier champ de la ligne.
 - Attention : dans un programme « awk » toute chaîne de caractère non encadrée par des double-quotes est considérée comme une variable.



Description d'une sélection

- Sous la forme : `/expression/`
 - toutes les lignes qui vérifient l'expression régulière « expression » vérifient la sélection
- Sous la forme : `$i ~ /expression/`
 - la sélection est vérifiée si le champ « i » vérifie l'expression régulière « expression »
- Exemple :
 - `$1 ~ /^[0-9]/`
 - `$3 == "toto"`
 - `$NR == 10`



Combinaisons des sélections

■ NF == 3

- les lignes contenant exactement trois champs sont sélectionnées

■ /debut/ && \$NF !~ /^[0-9]/

- les lignes qui contiennent « debut » et dont le dernier champ ne commence pas par un chiffre

■ (/un/ && /deux/) || /trois/

- les lignes qui contiennent « un » et « deux », ou les lignes qui contiennent « trois »

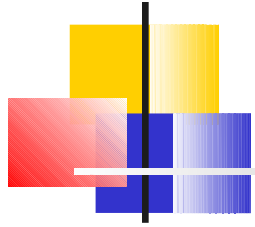
■ ! /exemple/

- les lignes qui ne contiennent pas « exemple »



Les variables et les expressions de awk

- `var = 3` # affectation
- `var = var * 2` # multiplication
- `var += 5` # addition
- `var ++` # incrémentation
- `var --` # décrémentation
- `%` # reste de la division entière
- `==` # égalité
- `!=` # différence
- `!` # négation



Les instructions de awk

- **if** (condition) énoncé [**else** énoncé]
- **while** (condition) énoncé
- **do** énoncé **while** (condition)
- **for** (i=1 ; i<=NF ; i++)
- **for** (indice **in** tableau) énoncé
- **Break** # passe à l'instruction qui suit la boucle.
- **continue** # provoque le passage à l'itération suivante
- **exit** [n] # la variable n alimente la variable « ? » du shell
à partir duquel le « awk » a été lancé.



Fonctions internes à awk

Commande « **PRINT** » :

- permet d'imprimer (afficher) un résultat.
- Chaque commande « print » provoque un retour à la ligne
- Exemple :

```
$ echo "12345 7 9" | awk ' { print ;  
                        print $1, $2 ;  
                        print $1 $2 } '
```

12345 7 9

12345 7

123457



Fonctions internes à awk

Commande « **PRINTF** » :

- Idem « print », mais la sortie est formatée.
- Le premier paramètre est impérativement une chaîne contenant le format des paramètres suivants.
- Exemple :

```
$ echo "123.456 111 un deux" |
```

```
awk ' { printf "%3.2f, %d, %5s, %s\n" , $1, $2, $3, $4 } '
```

```
123.46 111 un deux
```

- La commande « printf » ne réalise un retour chariot que si la séquence « \n » est spécifiée dans le format.



Fonctions internes à awk

■ Commande « **SPRINTF** » :

```
$ echo "123.456 12 12.123456" |
```

```
    awk '{ res = sprintf ("%3.2f, %3.2f, %3.2f" ,$1, $2, $3);  
    print "res=", res }'
```

```
res = 123.46 12.00 12.12
```

■ Commande « **LENGTH** » :

- Donne la longueur totale de la ligne, ou d'un champ si on lui fournit en argument

```
$ echo "12345 789" |
```

```
    awk '{ print length,length($1) }'
```




Fonctions internes à awk

- Commande « **INDEX** » :

- Donne la position d'une sous-chaine à l'intérieur d'une chaine

```
$ echo "T_SPE_CTXP_DF1" |  
    awk '{ print index($1, "SPE") }'  
3
```

- Commande « **SUBSTR** » :

- Extraction d'une partie de chaine

```
$ echo "T_SPE_CTXP_DF1" |  
    awk '{ print substr($1,7,4) }'  
CTXP
```



Fonctions internes à awk

- Commande « **SUB** » :

- Substitution de la première occurrence :

```
$ echo "T_SPE_CTXP_DF1" |  
  awk ' { sub ("DF[012] ", "DFL",$1); print $1 }'  
T_SPE_CTXP_DFL
```

- Commande « **GSUB** » :

- Substitution sur toutes les occurrences (pas seulement la première)



Fonctions internes à awk

- Commande « **MATCH** » :
 - Idem « index », mais on peut spécifier une expression régulière plutôt qu'une chaîne fixe
- Commande « **TOUPPER** » et « **TOLOWER** » :
 - Convertit respectivement en majuscules et en minuscules
 - Exemple : `tolower($1) == " toto "`
teste si le premier champ vaut « toto », « TOTO » ou « ToTo », etc...



Fonctions internes à awk

- Commande « **SPLIT** » :

```
$ echo " T_SPE_CTXP" |  
    awk ' { n = split ($1,tab, "_") ;  
           print "nombre de sous-champs:",n ;  
           for (i=1 ;i<=n ;i++)  
               print "No", i, "=",tab[i] ; } '
```

nombre de sous-champs: 3

No 1 = T

No 2 = SPE

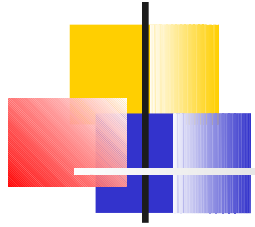
No 3 = CTXP



Fonctions internes à awk

- Commande « **SQRT** » :
 - Calcule la racine carrée de x (exemple de fonction arithmétique)

- Commande « **SYSTEM** » :
 - La fonction exécute une commande donnée en argument et renvoie son code retour. Cela permet de faire exécuter une commande, voir un shell à partir d'un programme awk.
 - Exemple : `system(ls $1)`



Exemples de programme « awk »

- Ecrire un script qui affiche pour les fichier d'extension .c du répertoire courant
nom du fichier : taille = taille du fichier
et qui donne la taille totale des fichiers
- Ecrire un script qui donne la taille des fichiers de chaque groupe contenu dans le répertoire courant



Correction du premier exemple de programme « awk »

```
$ ls -l *.c
```

```
-rw-r- -r- - 1 cogranne users 84 Apr 5 15:38 toto1.c
```

```
-rw-r- -r- - 1 cogranne users 86 Apr 5 15:38 toto2.c
```

```
$ ls -l *.c | awk '{ total += $5 ;
```

```
    printf "%14s: taille = %d \n", $NF, $5 ; }
```

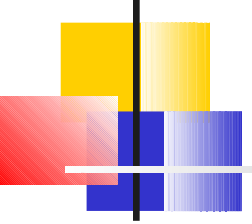
```
END{ print "Total = " total }'
```

```
toto1.c: taille = 84
```

```
toto2.c: taille = 86
```

```
Total = 170
```

Correction du deuxième exemple de programme « awk »



```
$ ls -l | awk 'NF>3 { tabtot[$4] += $5 ;  
                total += $5 ; }  
END{ for ( i in tabtot )  
    printf " total %6s = %7d\n", i, tabtot[i] ;  
    print " \nTotal general = " total ; } '
```

total root = 11532

total sys = 993

total iliade = 1203305

Total general = 1215825

Protection ou banalisation des caractères spéciaux



- Nous avons vu certains caractères \ * | < > qui ont une utilisation particulière et sont donc interprétés
- Pour que ces caractères soient utilisés comme des caractères simples il faut les protéger ou les banaliser.
 - En utilisant le caractère \ devant le caractère que l'on souhaite utiliser

```
$echo <bonjour
bash: bonjour: No such file or directory
$echo \<bonjour
<bonjour
```