Cours Unix 2
Procédures shell
sort, sed, grep, awk et autres outils

A Programmation Shell sous Unix (Bourne Shell)

Le langage **shell** est le langage de l'interpréteur de commandes d'UNIX qui dans d'autres systèmes d'exploitation pourra être désigné du nom assez général de JCL : Job Control Language, ou langage de contrôle des travaux.

La première utilisation de ce type de langage est d'enchaîner de façon automatique un certain nombre de commandes, correspondant le plus souvent à des demandes d'exécution de programmes (voir les fichiers .BAT sous MSDOS). Mais dans UNIX ce langage, interprété, offrira des possibilités très évoluées qui permettent de le considérer comme un véritable langage de programmation.

Sous Unix existent plusieurs interpréteurs de commandes:

- Bourne shell : **sh**, qui fut le shell des premières versions d'Unix, conçu par **Bourne**.
- Korn shell : **ksh**, qui est le shell préféré des environnements **AIX** (IBM, BULL etc…)
- C shell : **cshell**, dont le langage ressemble beaucoup au langage C (en mode interprété), et qui est le shell préféré des environnement **Solaris**
- « Bourne Again Shell », **bash** qui ressemble beaucoup au Bourne Shell, mais avec des extensions intéressantes, et qui est le plus souvent le **shell** des environnements **Linux**.

Chacun de ces interpréteurs de commandes a un langage un peu spécifique, mais il y a surtout 2 grandes catégories : le C Shell dont le langage nous l'avons dit est proche du C, et les autres, dont le langage reste assez proche du Bourne Shell d'origine.

Notre présentation se limitera au langage Bourne Shell.

1 . Commentaires

Dans un programme shell, tout ce qui suit le signe #, jusqu'à la fin de la ligne sera considéré un commentaire.

2 . Les variables et expressions du shell

Une variable n'a pas à être déclarée : elle est définie au moment ou on lui affecte une valeur.

Elle est **toujours de type chaîne de caractères**: si une variable est utilisée sans avoir été définie, elle a valeur de chaîne vide.

Les constantes n'ont pas besoin d'être mises entre guillemets: on distinguera la chaîne “alpha” du contenu de la variable “alpha” en écrivant **$alpha** lorsqu'il s'agit de la variable.

Une variable pourra être déclarée non modifiable par **readonly**
exemple : readonly **a** (interdira toute modification ultérieure de $a)

a) Opérations sur les variables

L’affectation : opérateur **=**

beta = bonjour $beta a pour valeur “bonjour”
alpha = beta $alpha a pour valeur “beta”
alpha = $beta $alpha a pour valeur “bonjour”

La concaténation : par **juxtaposition** des expressions

a= 20 $a vaut “20”
a= $a + 20 $a vaut “20 + 20”

Les expressions arithmétiques : utiliser **expr**

a = 20 $a vaut “20”
expr $a+ 20 affiche 40 à l'écran

pour affecter à $a cette valeur calculée par **expr**,
il faudra encadrer la commande par les caractères **'** (altgr + 7) : a = 'expr $a + 20 ' **# $a vaut “40”**

expr reconnaît les opérateurs **+ , - , * , / , %**

Il est aussi possible d'utiliser les parenthèses doubles **:**

a=20 ; a=$((a+20)); echo $a # $a vaut "40"

Les expressions logiques : utiliser **test** ou **[ ]**

- comparaisons arithmétiques **-eq, -ne, -gt, -ge, -lt, -le**
pour equal,not equal, greater than, greater or equal, less than, less or equal

if **test** $a -eq 20 ou if **[** $a –eq 20 **]**
if **test** $a -lt 30 ou if **[** $a -lt 30 **]**

- comparaisons de chaînes de caractères **=, !=, -z, -n**
pour *égal, non égal, chaîne vide, chaîne non vide*

if [$a = 20].... if [$a != vingt] .... if [-n $a]

- opérateurs de relations : **-a , -o , !**
pour and, or, not

if [$a = vingt **-o** $a -eq 20] ...

b) Variables prédéfinies

Le shell définit de façon standard les variables suivantes

$HOME :nom du répertoire d'accueil
$PATH :chemin d'accès aux commandes
$PS1 :prompt primaire
$PS2 :prompt secondaire
$LOGNAME :nom de l'utilisateur (donné au login)
$TERM :identificateur du type de terminal
$EXINIT :paramètres d'initialisation de **vi**

$0 ... $9 : paramètres de la ligne de commande ($0 est le nom de la commande)

S'il y a plus de 9 paramètres, (bien qu'il soit aussi possible d'utiliser **${10} ${11}**...) **shift** permet de réaliser un décalage sur les numéros de paramètres : $**0** reste inchangé, mais $**2** devient $**1** et le l0ème paramètre devient $**9**.

Il est possible de réaliser ainsi plusieurs décalages successifs, et même de demander par exemple un décalage unique de 9 positions dans lequel le dixième paramètre devient $ 1. Mais, attention, il n'est pas possible de décaler en arrière: il est donc nécessaire de mémoriser dans des variables auxiliaires les paramètres perdus par **shift,** si on doit plus tard les réutiliser.

$# nombre de paramètres
$* tous les paramètres "$l $2.....
$@ tous les paramètres "$I" "$2" ....

$$ numéro du processus shell en cours
$! numéro de la dernière commande lancée en arrière plan (par **commande &)**

$? code retour de la dernière commande

*Il est possible de redéfinir les arguments du shell courant par la commande **set**. Exemple :*

```
ptregouet@forum:~> set a b c d
ptregouet@forum:~> echo $*
  a b c d
  ptregouet@forum:~> echo $#
  4
  ptregouet@forum:~> echo $@
  a b c d
```

De la même façon qu'il est possible de passer des paramètres à une procédure, il est possible de passer des paramètres au shell initial au moment du **login :** ces paramètres sont intégrés à l'environnement sous les noms **$L0, $L1 etc...**

Il est aussi possible au moment du **login** de définir, ou redéfinir une variable d'environnement,

par exemple :

login : toto olive marius HOME=/users

cette connexion crée
$LOGNAME=toto $L0=olive $Ll=marius $HOME=/users

Il est possible **d'afficher une liste des variables définie** par la commande **set.** Par ailleurs **env** affiche les variables qui seront transmises à un processus fils, donc qui ont fait l'objet d'une commande **"export".**

c) Application

vérifier ce que produit l'enchaînement de commandes suivant

```
env
a=bonjour
env
echo $a
export a
env
```

3 . Le dialogue opérateur

Les messages à l'attention de l'opérateur seront en général produits par la commande **echo.**
Il est aussi possible de demander une valeur à l'opérateur en utilisant la commande **read .**

exemple :
```
echo " âge du capitaine "
read age
```

4 . les instructions conditionnelles if et case

a) La structure générale d'un bloc if

Le bloc **else** est facultatif.

```
if   condition
then  action 1; action 2
else   action 3;  action 4
fi
```

b) La structure générale d'un bloc case

```
case  $variable  in
            v1 )        action 1    ; ;
          v2 | v3 )     action 2    ; ;
             * )        action_autres_cas ; ;
esac
```

c) Expressions conditionnelles

Une expression conditionnelle est de la forme **test expression** ou **[expression]**
dans laquelle **expression** est

- soit une comparaison entre valeurs numériques ou de type chaînes de caractères (voir ci-dessus opérations sur les variables)
- soit une condition relative aux fichiers et répertoires

Dans ce dernier cas, on trouvera par exemple ***if test -d $1*** qui va permettre de savoir si le premier paramètre passé à la procédure est bien le nom d'un répertoire.

Il sera possible d'utiliser les tests suivants sur les répertoires et les fichiers :

-f $x	pour savoir si $x est un fichier ordinaire
-d $x	pour savoir si $x est un répertoire
-s $x	pour savoir si $x existe et est non vide
-r $x	pour savoir si $x existe et est accessible en lecture
-w $x	pour savoir si $x existe et est accessible en écriture
-x $x	pour savoir si $x existe et est accessible en exécution

- soit le nom d'une commande UNIX qui sera exécutée

La condition sera vraie si la commande renvoie un code retour à 0, elle sera fausse dans tous les autres cas.

exemple

```
if cc prog.c
then echo exécution correcte de la compilation
else echo Dommage, il y avait des erreurs
```

- Une condition peut toujours être niée par !
- Plusieurs conditions peuvent être combinées en utilisant -o (or) ou -a (and).

5 . Les boucles de répétition

a) Boucles For

- La première boucle de répétition est celle qui permet d'exécuter le même traitement pour tous les paramètres passés à la procédure. Elle a la forme suivante

```
for   variable
do      commande 1
        commande 2
done
```

- Une autre forme de la boucle for est plus traditionnelle, elle ressemble à une boucle for avec incrémentation d'une variable de contrôle, mais comme ici toutes les valeurs sont de type chaîne de caractères, il n'y aura pas de valeur début et de valeur fin, il faudra tout simplement énumérer l'ensemble des valeurs possibles.

```
for i in jean alex paul
do
        write $i <message.txt;
        echo message.txt envoyé à $i
done
```

b) Boucles While et until

- Les boucles **while** et **until** permettent de réaliser des boucles conditionnelles, contrôlées

  - soit par le résultat d'une expression **test**
  - soit par le compte-rendu d'exécution d'une commande UNIX (vrai si $? = 0)

  ... exactement comme nous l'avons vu pour l'instruction **if.**

Boucle **while :** cette boucle sera exécutée tant que le résultat de commande-condition sera 0 (vrai)

```
while     commande-condition

do commandel
   commande2
done
```

Boucle **until** : cette boucle sera exécutée tant que le résultat de commande-condition sera 1 (faux)

```
until     commande-condition
do commandel
   commande2
done
```

c) Commandes true, false et test

- la commande **true** rend une valeur vraie; la commande **false** rend une valeur fausse
- **test** rend une valeur vraie si la condition est vraie

Exemple : attente du spool (répertoire /usr/spool/lp vide)

```
until test -f /usr/spool/lp/*
do sleep 10
done
```

6 . Les mécanismes de substitution de chaînes de caractères

Le shell fournit plusieurs mécanismes permettant de substituer une expression par le résultat de son évaluation. Le plus simple est celui qui permet de remplacer une variable par son contenu ($LOGNAME remplacé par la valeur du de la variable LOGNAME par exemple), mais il en existe d'autres, en voici la liste :

Substitutions de variables

expression textuelle	remplacée par ...
$variable	valeur de la variable (ou chaîne vide)
${variable-xx}	**valeur de la variable** ou "xx" si variable non définie
${var?}	valeur de la variable message d'erreur si variable non définie
${var?mes}	valeur de la variable message "mes" si variable non définie

:${varl?} ${var2?} ${var3?} teste si var l ... var3 sont définies

exemples :

```
demo@forum:~> echo ${varl-xx}

xx

demo@forum:~> :${PATH?} ${HOME?} ${var3?}

-bash: var3: parameter null or not set

demo@forum:~> :${PATH?Path non défini} ${HOME?Home non défini} ${var3?
var3 non défini}

-bash: var3: var3 non défini
```

Noms de fichiers et expressions case multivaluées

- Dans **les noms de Fichiers et les expressions "case",** il est possible de définir des noms ambigus en utilisant un ou plusieurs des caractères * et ?

  Ainsi, dans un nom de fichier, ou une constante **case,** en lieu et place de...

*	: toute suite de caractères convient
?	: tout caractère (un seul) convient
[abcd]	: l'un quelconque des caractères a,b,c,d convient
[A-K]	: l'un quelconque des caractères entre A et K convient..

Récupération du résultat affiché par une commande

- Pour obtenir comme valeur de substitution le **résultat** (normalement envoyé sur la console) de l'exécution d'une commande, on placera cette commande entre 2 caractères' **(Alt Gr + 7).** Entre ces 2 caractères, $ **et** * sont interprétés par le shell.

  exemple : A = **`** **date +%Y** **`** affecte à A l'année de la date du jour

- Une autre écriture permet d'obtenir le même résultat avec **bash** : A= **$(date +%Y)**

Annulation des mécanismes de substitution

Il est quelque fois nécessaire de retirer à un caractère comme $ son effet de substitution, ne serait-ce que pour afficher ce caractère sur l'écran. Plusieurs mécanismes sont ici encore disponibles

\ annule l'effet spécial du caractère qui le suit
exemples: **\$** \\ **\?** \{ **\"** **\'** **\[**

ou donne un sens particulier à un caractère (voir printf en C)
exemples : **\t** est un caractère de tabulation, **\n** saut de ligne

" entre guillemets, certains caractères sont interprétés :

$** **$xx est interprété comme la valeur de la variable xx
` **(AltGr + 7) 'pwd'** est remplacé par le nom du répertoire courant
\ **\t** est interprété comme le caractère de tabulation

' entre apostrophes (différent de altgr+7) tous les caractères sont considérés comme de simples caractères (aucune interprétation n'est réalisée par le shell)

7 . Mécanismes complémentaires interprétés par le shell (voir man sh)

Il est possible de contrôler le déroulement d'une boucle **until** ou **while** en utilisant les instructions **break** et **continue** avec le même sens qu'en langage C : break termine la boucle en cours, alors que continue permet de passer à l'itération suivante sans exécuter les instructions qui suivent.

Il est aussi possible de terminer l'exécution d'une procédure avant la dernière instruction en utilisant **exit N** où N est la valeur du code retour qui sera envoyé à la procédure appelante. Enfin, il est possible de définir des fonctions à l'intérieur d'une procédure **shell**

Une **fonction etape ( )** pourra par exemple être définie et utilisée comme suit:

```
#!/bin/bash

#Variable "globale"
init=10

    # définition de la fonction
    etape()
    {
    somme=`expr $init + $1 + $2`
    echo la somme de $init, $1 et $2 est
    $somme
    }

# utilisation de la fonction etape( )
# apres sa declaration

etape   1    2
```

```
ptregouet@forum:~/procedures-shell>
./fonction.sh

la somme de 10, 1 et 2 est 13
```

8 . Créer et exécuter une procédure shell

Les programmes écrits dans ce langage sont communément appelés des ***scripts shell.*** Ce sont des fichiers de texte qui peuvent être manipulés par toutes sortes d'éditeurs de texte, **ed** et **vi** en particulier, mais aussi **pico**, **joe, kedit**, **kwrite, kate, gnome-editor** etc...

Pour lancer l'exécution d'un tel programme on disposera des méthodes suivantes :

Si le fichier s'appelle **test1**

1. **source** test1 # exécution des commandes par le shell courant
2. **bash** test1 (ou sh test1) # exécution dans un shell enfant
3. **chmod +x** test1 ; **./**test1 # exécution dans un shell enfant
4. **chmod +x** test1 ; **. ./**test1 # exécution des commandes par le shell courant

Le nom de la procédure, **test1**, dans ces différents cas d'exécution, peut être suivi d'une liste de paramètres.

a) Le shell et les processus UNIX

Dans les **cas 2 et 3** ci-dessus de demande d'exécution d'une procédure, un nouveau processus **shell** est créé, et le **shell** initial se met en attente de la fin du processus fils.

Le processus **fils** hérite de toutes les variables qui ont fait l'objet d'une commande **export.** Leur liste peut être affichée par la commande **env,** ces variables constituent l'environnement du processus.

Si à l'intérieur d'un programme shell, on appelle un autre programme shell, il y a de nouveau création d'un processus. Pour éviter la création d'un nouveau processus, il faut faire précéder le nom du fichier appelé par "**.**" le contenu de ce fichier sera alors exécuté par le **même shell,** exactement comme si l'on avait inclus le texte de la deuxième procédure à l'intérieur de la première (cf. cas 4 ci-dessus).

Intérêts ?

- exécution plus rapide (pas de duplication de processus)
- modification possible de l'environnement de la procédure appelante (par exemple modification du répertoire courant par **cd,** ou modification des variables).En effet, il n'y a pas de commande *import*: les variables peuvent être exportées vers le processus fils (recopiées dans son environnement), mais si celui-ci les modifie, le processus père ne peut en aucune façon récupérer les nouvelles valeurs!

Inconvénients ?

- la récursivité n'est possible que s'il y a création d'un nouveau processus shell.

b) Applications

Application 1 : créer une procédure de compilation

Créer une procédure compil admettant en paramètres :

1. le nom du programme à compiler par cc (sans l'extension.c)
2. (paramètre facultatif) **ex** pour demander l'exécution du programme en cas de succès de **cc.**
3. (paramètre facultatif) nom du fichier UNIX vers lequel sera redirigé le flux des messages d'erreurs de **cc** (utiliser **cc... 2>fichier).**

La procédure doit aussi satisfaire les spécifications suivantes

a) vérifier que le fichier source existe dans le répertoire courant
b) produire un fichier exécutable ayant pour nom la valeur du premier paramètre.
c) se conformer à la table de décision suivante

Conditions		**Actions**
compilation correcte	**exec demandée**	
faux		lancer vi pour modifier le source
vrai	faux	afficher “compilation Ok”
vrai	vrai	lancer l'exécution du programme

Application 2 : créer un fichier ~/.profile (ou ~/ .bash_profile)

Ce fichier d'initialisation de la session UNIX devra...

1. redéfinir PS1 et PS2 (voir man **bash** pour la signification de PS1 et PS2)

2. ajouter à la liste des chemins d'accès (**$PATH**), le répertoire utilisateur **“ ~/bin ”**. Ce répertoire contiendra les utilitaires définis par l'utilisateur comme **compil** etc.

3. on admettra au **login** un paramètre : le nom du répertoire de travail, nommé relativement au répertoire d'accueil. La procédure d'initialisation devra activer ce répertoire et afficher “Votre répertoire de travail est : .....”

B Le traitement automatique des fichiers textes

9 . Redirection des entrées sorties

De nombreuses commandes UNIX permettent de traiter des fichiers de texte : ces commandes sont souvent utilisées dans des scripts shell sous forme de filtres.

Elles sont souvent utilisées aussi avec la redirection des entrées ou des sorties standard. Rappelons que

>	permet de rediriger les sorties sur un fichier
>>	redirige les sorties en complémentarité sur un fichier
|	définit un filtre pour un enchaînement de commandes
<	définit un nouveau fichier comme entrée standard
<<FIN	définit le fichier de commande courant, jusqu'au mot FIN, comme entrée standard pour la commande à exécuter.

exemple

```
vi x.c     <<STOP  2> /dev/null
:1,$s/xl/x2/g
:wq
STOP
echo  remplacement de xl par x2 dans x.c OK
```

Un processus Unix est "connecté" à 3 flux de données

0 : entrées standard rédirigée par < ou <<
1 : sortie standard redirigée par > , >> ou de manière équivalente par 1> ou 1>>
2 : sortie des erreurs redirigée par 2> ou 2 >>

Il est possible de rediriger les 2 flux de sortie (standard et d'erreurs) vers la même destination en utilisant
&1 pour désigner la redirection courante de la sortie standard
&2 pour désigner la redirection courante de la sortie des erreurs

Exemples :

```
find /etc -name "*.conf" 2> /dev/null
find /etc -name "*.conf"   > /tmp/find.etc.txt 2> /dev/null
find /etc -name "*.conf"  1> /tmp/find.etc.txt 2> &1
find /etc -name "*.conf"    > /tmp/find.etc.txt 2> &1
find /etc -name "*.conf"  2 >&1   > /tmp/find.etc.txt  # les erreurs restent sur l'écran
```

10 . Quelques commandes de traitement des textes

Nous ne ferons ici que citer quelques-unes unes de ces commandes: d'autres pourront être trouvées dans l'index permuté (mot clé file...) et leur description sera trouvée dans le manuel en ligne: man

wc [-1] permet de compter le nombre de mots [ou de lignes] dans un texte

colrm permet de retirer une ou plusieurs colonnes d'un texte disposé en tableau

cut permet de ne retenir qu'une ou plusieurs colonnes d'un tableau, les champs étant délimités par un séparateur commun

head donne les premières lignes d'un fichier de texte

tail permet d'obtenir les dernières lignes d'un fichier de textes

sort permet de trier les lignes d'un texte (voir ci-après)

grep (global regular expressions print) permet de rechercher et d'afficher les lignes d'un texte contenant un mot ou une expression donnée.

sed (sequential editor) permet de modifier un texte par des mécanismes de recherche et remplacement (commandes de substitution dans la ligne de commande ou mémorisées dans un fichier)

awk fournit un véritable langage de programmation pour la modification d'un texte ou l'édition de rapports sur imprimante (awk travaille sur des textes dont les lignes contiennent des séparateurs permettant de considérer ces textes comme des tableaux)

11 . Expressions régulières

Le traitement des fichiers de texte fait souvent appel aux expressions régulières, qui permettent de décrire formellement une chaîne de caractère et sa position dans une ligne ou un texte.

Une **expression régulière** pourra contenir

\c : banalisation du caractère spécial c

^ : début de ligne

$: fin de ligne

. : n'importe quel caractère

c : le caractère c

[a-z] : tout caractère entre a et z

[abc] : soit a, soit b, soit c

[^abc] : ni a ni b ni c

c* : 0,1 ou n occurrences du caractère c

c+ : 1 ou n occurrences du caractère c

c? : 0 ou 1 occurrence du caractère c

e1e2 : e1 et e2 concaténées

el|e2 : e1 ou e2

( ) : groupage d'expressions

12 . Commande cut : un exemple

liste=`cut -f1 -d" "  users/FIEP01`

Cette commande appliquée au fichier **users/FIEP01 ci-dessous**
affectera à la variable **liste** la première colonne (-f**1**) de ce fichier,
le délimiteur de champs étant l'espace (-fd" ")
comme le montre le résultat de la commande **echo $liste** :

Contenu de **users/FIEP01 :**

```
fiep01.bard   bard      Bartakovic Damir
fiep01.funa   funa      Funes Andres
fiep01.ivuv   ivuv      Ivulich Tromer Veronica
fiep01.jimm   jimm      Jimenez Sevilla Maite
fiep01.ruoa   ruoa      Ruoff Alexander
fiep01.schp   schp      Schraps Patrick
fiep01.tarj   tarj      Tarrago Joan
```

Résultat de **echo $liste** :

fiep01.bard **fiep01.funa** fiep01.ivuv **fiep01.jimm** fiep01.ruoa **fiep01.schp** fiep01.tarj

13 . Commande sort : trier un fichier

La commande sort trie le contenu d'un fichier en fonction d'une ou plusieurs clés.

L'élément de base est la ligne, elle-même composée d'un ensemble de champs.

U**n champ** est une chaîne de caractères encadrée d'espaces ou de tabulations (séparateurs par défaut) ou d'un autre séparateur choisi par le paramètre -t de sort

exemple : **sort -t:** définit ":" comme séparateur de champs

Une clé est un ensemble de champs utilisés pour servir de critère de tri. Une clé est indiquée après le **paramètre -k**. Il peut y avoir plusieurs critères de tri. Il sera possible d'utiliser un point décimal pour préciser que l'on ne retiendra que tel ou tel caractère du champ dans la clé.

Les différents champs seront désignés par un chiffre représentant leur numéro d'ordre dans la ligne, à partir de 1.

Les numéros des caractères composant la clé, si la clé est seulement une partie d'un champ, sont aussi numérotés à partir de 1.

Une clé sera considérée de type chaîne de caractères, sauf pour les champs dont le numéro est suivi de **n** (numérique).

Pour chaque clé de tri on pourra préciser "r" (reverse) si les lignes doivent être triées du pls grand au plus petit selon cette clé

L'ordre des clés est important : la première clé nommée est la plus significative.

exemples

cat /etc/passwd | sort -t: -k 3nr

- cette commande trie le contenu du fichier /etc/passwd (la liste des utilisateur)
- Le séparateur de champs est le caractère ":"
- le critère de tri est 3ème champ c'est à dire le numéro d'utilisateur, considéré comme numérique (n)
- et l'ordre du tri est du plus grand au plus petit (r=reverse) comme le montre le résultat ci-après :

```
uucp:x:10:14:Unix-to-Unix CoPy system:/etc/uucp:/bin/bash
news:x:9:13:News system:/etc/news:/bin/bash
mail:x:8:12:Mailerdaemon:/var/spool/clientmqueue:/bin/false
lp:x:4:7:Printing daemon:/var/spool/lpd:/bin/bash
daemon:x:2:2:Daemon:/sbin:/bin/bash
bin:x:1:1:bin:/bin:/bin/bash
root:x:0:0:root:/root:/bin/bash
```

ll | sort -k5n

Cette commande trie la liste des fichiers du répertoire courant d'après la taille des fichiers (champ numéro 5 considéré comme valeur numérique).

Le résultat est le suivant:

```
-rwxr-xr-x 1 ptregouet users  83 2005-10-18 16:36 for.test
-rwxr-xr-x 1 ptregouet users  99 2007-06-27 11:12 awk-print
-rwxr-xr-x 1 ptregouet users 113 2005-10-18 16:37 incrementer
drwxr-xr-x 2 ptregouet users 144 2007-02-23 10:39 controle
-rwxr-xr-x 1 ptregouet users 179 2007-06-27 10:33 controle2
```

cat /etc/passwd | sort -t: -k4n -k1.7,1.8r -k3n

- cette commande trie le fichier /etc/passwd
  le séparateur de champs est le caractère ":"
- le premier critère de tri est le champ 4 : le numéro de groupe
- le deuxième critère de tri est composé des caractères 7 et 8 du champ1 (utilisateur)
- ce critère est traité dans l'ordre décroissant
- le troisième critère de tri est le numéro de l'utilisateur, numérique, dans l'ordre normal, croissant

Résultat partiel :

```
irt01.bejg:x:1169:999::/serveur/IRT01/irt01.bejg:/bin/bash
irt01.barf:x:1082:999::/serveur/IRT01/irt01.barf:/bin/bash
irt01.barh:x:1083:999::/serveur/IRT01/irt01.barh:/bin/bash
irt01.batm:x:1084:999::/serveur/IRT01/irt01.batm:/bin/bash
irt01.bags:x:1168:999::/serveur/IRT01/irt01.bags:/bin/bash
irt01.aydi:x:1167:999::/serveur/IRT01/irt01.aydi:/bin/bash
```

14 . Commande grep : recherche d'expressions régulières

grep utilise des expressions régulières du même type que celles utilisées par **ed** ou **sed**

(**fgrep et egrep** sont 2 autres utilitaires ayant sensiblement le même fonctionnement: voir documentation UNIX)

Exemples

grep -n "^#" *.c

recherche dans les fichiers *.c du répertoire courant les lignes contenant le caractère # en premier caractère (les lignes trouvées seront affichées avec leur numéro de ligne: option –n)

Les différentes **options** de grep sont les suivantes

- **-v** : donne les lignes ne contenant pas la chaîne citée
- **-c** : donne le nombre de lignes trouvées
- **-1** : donne le nom des fichiers dont une ligne au moins satisfait la recherche
- **-n** : affiche le numéro des lignes trouvées
- **-i** : ne distingue pas entre majuscules et minuscules
- **-s** : (silent) minimise les messages d'anomalies (accès interdit etc.)

15 . Commande awk : traitement des tableaux

La commande awk permet, comme **sed** de substituer des chaînes de caractères dans un texte, en utilisant la fonction sub ou la fonction gsub.

Mais awk est surtout efficace lorsqu'il s'agit de traiter un fichier dont chaque ligne peut être considérée comme un enregistrement d'une table; les différents champs de l'enreistrement étant séparés par un délimiteur de 1 carctère.

Beaucoup des fichiers d'administration d'Unix utilisent un tel format, en particulier le fichier /etc/passwd qui contient la liste des utilisateurs d'un système Unix, les différents champs d'un enregistrement étant séparés par le caractère ":".

Exemple :

Contenu de /etc/passwd :

```
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:
daemon:x:2:2:daemon:/sbin:
```

La commande awk permettra très facilement de considérer ce fichier comme un tableau dont les différents champs de la ligne courante seront désignés par $1, $2 … $7. L'enregistrement courant dans son ensemble étant désigné par $0.

$1	$2	$3	$4	$4	$6	$7
root	**x**	**0**	**0**	**root**	**/root**	**/bin/bash**
bin	**x**	**1**	**1**	**bin**	**/bin**	
daemon	**x**	**2**	**2**	**daemon**	**/sbin**	

a) Syntaxe :

La syntaxe générale de awk est la suivante :

awk [-Fc] ['{ programme }' | -f fichier_programme] [paramètres] [FichiersTraités]

-Fc où c est un caractère quelconque permet de redéfinir le séparateur de champs par exemple **–F:** définit ":" comme séparateur. Les séparateurs par défaut sont espace et tabulation.

Le programme pour awk peut être donné directement dans la ligne de commande, entre les caractères **'{** et **}'**, ou écrit dans un fichier dont on donnera le nom à **awk** dans la ligne de commande

D'autres paramètres peuvent être passés à awk (voir **man** awk)

On indiquera en dernier le ou les fichiers traités. Si aucun nom de fichier n'est donné, **awk** prendra les données à traiter dans l'entrée standard.

La sortie de **awk** sera la sortie standard, que l'on pourra rediriger dans un fichier si nécessaire.

b) Forme générale d'un programme awk

La commande **awk** exécute les commandes pour toutes les lignes du fichier qui **vérifient la condition énoncée** ou pour toutes les lignes du fichier, si aucune condition n'est énoncée.

Les conditions spéciales BEGIN et END correspondent à des initialisations et des actions finales à exécuter avant ou, respectivement, après le traitement des lignes du (ou des) fichier(s).

BEGIN { instructions à exécuter avant de traiter les données }
Condition 1 { instructions 1, exécutée pour les lignes vérifiant la condition 1 }
Condition 2 { instructions 2, exécutée pour les lignes vérifiant la condition 2 }
… etc ….
END { instructions à exécuter après le traitement des données }

c) Quelques variables awk

NR	numéro de l'enregistrement courant
NF	nombre de champs dans l'enregistrement courant
FS	séparateur de champ en entrée
OFS	séparateur de champ en sortie
RS	séparateur d'enregistrement en entrée
ORS	séparateur d'enregistrements en sortie
FILENAME	nom du fichier courant
$0	contenu de l'enregistrement courant
$1, $2 …	contenu des champs 1, 2 … de l'enregistrement courant
OFMT	format des nombres en sortie

d) Fonctions prédéfinies :

print	impression sur la sortie standard
printf	impression formatée (cf langage C)
length	taille d'un enregistrement ou d'un champ (ex length ($2))
sub	substitution d'une chaîne par une autre (expressions régulières)
gsub	idem que **sub**, mais plusieurs substitutions possibles s'il y a plusieurs occurrences de la chaîne à substituer; **sub** ne remplaçant que la première occurrence.
substr (s,m,n)	extraction d'une sous-chaîne
index (s1,s2)	position d'une sous-chaîne dans une chaîne
sprintf(FORMAT,s1,s2…)	formatage dans une chaîne (cf langage C)
split (s,tableau)	éclatement d'une chaîne en ses champs (séparateur)
sqrt	racine carrée
log	logarithme népérien
exp	exponentielle
int	partie entière

exemple d'utilisation de split :

```
n=split (s, t)                  # séparateur espace ou tabulation par défaut
n=split (s,t,separateur)
```

n est le nombre de champs trouvés par split
s est la chaîne explorée par split
t est un tableau qui contiendra les différents champs extraits de s par split

ainsi **n=split ($0,t)**
permet d'obtenir t[1] identique à $1, t[2] identique à $2 etc…

exemple d'utilisation de substr et index

```
substr (abc", 2,1)      donne "b"
index ("abc", "bc")     donne 2
```

e) Opérateurs reconnus par awk

Tous les opérateurs arithmétiques et de comparaison du langage C

```
+ - * / % ++ -- += -= *= /= %= < <= > >= == !=
```

Deux opérateurs spécifiques :

~ correspondance avec une expression générique ex `A ~ "*.com"`
!~ Non-correspondance avec une expression générique ex `A !~ "*.com"`

f) Variables de awk

Les variables n'ont pas à d'être déclarées, aucun type n'est à définir pour les variables: **awk** est un langage faiblement typé, le type des variables dépend du contexte

Les tableaux peuvent avoir un indice de type chaîne de caractère (awk utilise un algorithme de hachage pour convertir cette chaîne en valeur entière).

exemple : t ["bleu"] , t["rouge"] , t["vert"]

g) Instructions de contrôle

if (condition) action
else action

while (condition) action

for (expression ; condition ; expression) action

break
continue
next
exit

h) Exemples de programmes réalisés avec awk

- **Exemples simples :**

Imprimer chaque champ ligne par ligne :

```
{               i=1
                while (i <= NF ) { printf "%s \n", $i ; i++ }
}
```

Imprimer toutes les lignes paires, terminer par le nom du fichier traité

```
NR % 2  == 0    ; # action vide, imprime la ligne courante
END             { printf "\n" FILENAME, NR " lignes dans le fichier " }
```

Vérifier l'unicité des noms d'utilisateurs dans le fichier /tmp/passwd

préliminaire :

```
cp  /etc/passwd /tmp/passwd
tail –5 /etc/passwd  >> /tmp/passwd
```

commande : **awk** –F: -f **verif_unicite** /tmp/passwd

contenu de verif_unicite (programme **awk**) :

```
{       if ( utilisateur [$1] )
        {       printf "%s en double \n", $1
        }
        utilisateur[$1] = NR

}
```

- **extraire du fichier /etc/passwd :**

1) les utilisateurs de numero 0
2) les utilisateurs du groupe 120

```
awk   -F:   '

 $3 == "0"     { print "Superutilisateur : ", $1 ,"\t\t", $3,"\t", $6 }
 $4 == "120"   { print "Etudiants       : ", $1 ,"\t\t", $3,"\t", $6 }

' /etc/passwd
```

Le résultat de l'exécution de ce programme pourrait être le suivant :

```
Superutilisateur :  root            0          /
Superutilisateur :  sysadm          0          /
Superutilisateur :  powerdown       0          /usr/admin
Superutilisateur :  setup           0          /
Etudiants        :  bochero         1201        /users/iri93-95/bochero
Etudiants        :  bonnin          1202        /users/iri93-95/bonnin
Etudiants        :  bouchez         1203        /users/iri93-95/bouchez
Etudiants        :  bozec           1204        /users/iri93-95/bozec
Etudiants        :  canonge         1205        /users/iri93-95/canonge
Etudiants        :  cordier         1206        /users/iri93-95/cordier
Etudiants        :  daguin          1207        /users/iri93-95/daguin
Etudiants        :  dilhac          1208        /users/iri93-95/dilhac
Etudiants        :  damien          1209        /users/iri93-95/damien
Etudiants        :  david          1210         /users/iri93-95/david

etc ....
```

- **Envoyer un fichier de texte dans la boite à lettre de tous les utilisateurs connectés ('mail to all users')**

Le principe retenu est le suivant :

1) créer un fichier **who.txt** contenant la liste des utilisateurs connectés
2) exploiter ce fichier par awk, et créer un fichier **mailall.sh** contenant les commandes mail applicables à tous les utilisateurs trouvés

```
who > who.txt
awk -f awk.mail.p message=$1  who.txt > mailall.sh
cat mailall.sh  # controle des commandes mail générées
sh mailall.sh   # exécution des commandes mail
rm mailall.sh   # suppression des fichiers auxiliaires
rm who.txt
```

Le programme awk est le fichier awk.mail.p qui contient les commandes :

```
{
  print "mail   ", $1, " < ", message

}
```

dans ce programme, message est un paramètre initialisé par la commande ci-dessus : **awk ..... message=$1** , $1 étant ici le paramètre de awk.mail

Mise en œuvre :

La fichier **Bonjour** contient le message suivant :

Tu as un "Bonjour" dans ta boite à lettres !

Pour envoyer ce message à tous les utilisateurs on tapera :

```
$ > awk.mail bonjour
mail   oracle  <  bonjour
mail   demo  <  bonjour

$ > mail
From demo Fri Apr 22 11:50 EET 1994
Received: by Isaip32.LOCAL.uucp; Fri, 22 Apr 94 11:50:18 +0200 (MET)
Date: Fri, 22 Apr 94 11:50:18 +0200
From: demo
Message-Id: <9404220950.AA21907@Isaip32.LOCAL.uucp>
Apparently-To: demo

Tu as un "Bonjour" dans ta boite à lettres !
```

16 . Commande sed : éditeur séquentiel

La commande **sed** permet de traiter automatiquement un fichier de texte pour en modifier le contenu par exemple

- remplacer tous les caractères NL (fin) par la séquence CR + NL (\n + \r)
- afficher /etc/passwd dans le format Nom .... UID ..... REP .....
- ... etc ...

sed traite le fichier standard des entrées (stdin) et écrit le résultat dans le fichier standard des sorties (stdout).

Le «fichier» traité par **sed** pourra ainsi être simplement un mot, envoyé sur la sortie standard par la commande echo; et c'est ainsi que l'on pourra voir une utilisation de sed, dans /etc/profile, pour extraire le premier caractère du nom du terminal utilisateur (voir exemples ci après).

Les commandes de modification de texte seront trouvées soit dans la ligne de commande après le paramètre -e, soit dans un fichier contenant ces commandes et dont le nom sera donné après le paramètre -f .

a) Syntaxe de sed

sed [-n] [-e commandes] [-f nom fichier] [fichier-1 .... fichier]

Paramètre -n : Dans son mode de fonctionnement normal, sed

- lit une ligne du fichier d'entrée et la stocke dans un tampon
- applique les commandes de modification au texte du tampon
- recopie le tampon modifié dans le fichier de sortie , sauf si le paramètre -n est spécifié, car dans ce cas ne seront recopiées en sortie que les lignes qui auront fait l'objet d'une commande de recopie explicite **p** (print).

b) Syntaxe des commandes d'édition

[adresse1 , adresse2] **fonction** *[arguments de la fonction]*

adresse peut être par exemple

12	: la 12ème ligne
/toto/	**:** toutes les lignes contenant toto
/toto/ , /titi/	**:** toutes les lignes comprises entre la ligne contenant toto et celle contenant *titi.* .

fonction peut être une des fonctions d'édition suivantes

Action sur des lignes entières

d : effacer une ou plusieurs lignes
n : aller à la ligne suivante

a\ : ajouter le texte suivant à la ligne en cours
texte ligne 1 \
texte ligne 2 \
texte dernière ligne

c\ : remplacer la ligne en cours par le texte suivant
texte ligne1 \
texte dernière ligne

i\ : insérer le texte suivant au début de la ligne en cours
texte ligne1 \
texte dernière ligne

Commandes de substitution

s : substitution ; le format général de cette commande est
[adressel,[adresse2]] S <configuration> <remplacement> <options>

<**configuration**> expression régulière délimitée par n'importe quel caractère sauf «espace» et «New line».

<**remplacement**> une expression qui n'utilise pas la syntaxe des expressions régulières mais les conventions suivantes :

& chaîne coïncidant avec <configuration>
\3 3ème expression parenthésée

<**options**> a comme valeurs possibles :

g	global (sur toute la ligne)
p	impression du résultat
w f1	écriture du résultat dans le fichier f1

c) exemples

```
/toto/,/titi/s%[a-z]bc%&def%p
```

remplacement de abc par abcdef , bbc par bbcdef etc ...

```
s/(.*):.*:(.*):.*:.*:(.*):.*/NOM= \1 UID= \2 REP= \3/w passwd1
```

création du fichier **passwd1** à partir de /etc/passwd explicitant Nom, Uid et Rep

```
T=`echo $TERM | sed "1s/^(.).*/\1/"`
```

dans /etc/profile, cette commande affecte à la variable T la première lettre du nom du terminal contenu dans $TERM

Attention : dans les 2 dernières commandes (et) seront remplacées par\(et \) pour éviter leur interprétation par le shell

17 . Exemples de commandes utilisant sed, grep, awk

```
-----------------------------------
commandes avec grep et awk
-----------------------------
cat /etc/passwd|grep root|awk -F: '{print $6}'

ypcat passwd|awk -F: '{print$1}'
ypcat passwd|grep irt01|awk -F: '{print$1"@isaip.uco.fr"}'

liste=`ypcat passwd|grep irt01|awk -F: '{print$1"@isaip.uco.fr"}'`

echo $liste
mail $liste
-----------------------------------
commandes avec sed
-----------------------------------
echo bonjour mon petit canard|sed s/canard/lapin/

echo bonjour bonjour mon petit canard|sed s/bonjour/salut/
echo bonjour bonjour mon petit canard|sed s/bonjour/salut/g
-----------------------------------
commandes avec grep et sed
-----------------------------------
ypcat passwd

ypcat passwd | grep irt01
ypcat passwd | grep irt01 | sed s/irt01/IRT01/g
-----------------------------------
commandes avec grep et sed
-----------------------------------
/sbin/ifconfig

/sbin/ifconfig | grep 'inet adr'
/sbin/ifconfig | grep 'inet adr' | grep -v 127

/sbin/ifconfig | grep 'inet adr' | grep -v 127 | \
     sed "s/^.*inet adr:\([0-9]*.[0-9]*.[0-9]*.[0-9]*\).*/\1/"
```

```
cd /var/named
grep st3 172.16 >172.16.st3
grep st3 isaip.uco.fr >isaip.uco.fr.st3
sed "s/^\(.*\.\)3\(.*st\)3\(.*\)/\14\24\3/" <172.16.st3      >172.16.st4
sed -e s/st3/st4/  -e s/172.16.3/172.16.4/ <isaip.uco.fr.st3     >isaip.uco.fr.st4
less 172.16.st4
less isaip.uco.fr.st4
echo Nombre de lignes de 172.16.st4       : `cat 172.16.st4 | wc -l`
echo Nombre de lignes de isaip.uco.fr.st4 : `cat isaip.uco.fr.st4 | wc -l`
 echo -e "Mettre à jour les fichiers du DNS ? (O/N)\c"
read reponse
if [ $reponse = "O" ]
then
        grep -v st4 172.16 >/tmp/172.16
        grep -v st4 isaip.uco.fr >/tmp/isaip.uco.fr
        cat 172.16.st4 >> /tmp/172.16
        cat isaip.uco.fr.st4 >> /tmp/isaip.uco.fr
        cp 172.16 172.16.old
        cp isaip.uco.fr isaip.uco.fr.old
        cp /tmp/isaip.uco.fr isaip.uco.fr
        cp /tmp/172.16 172.16
        ndc restart
        echo FIN de mise à jour du DNS
else
        echo DNS inchangé
fi
```