# Programmation Shell

# Sommaire

# Bibliographie

- http://fr.wikibooks.org/wiki/Programmation_Bash - En français, introduction à la programmation shell
- http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html - introduction au shell bash, y compris la programmation.
- http://tldp.org/LDP/abs/html/ - Aborde les aspects basiques et avancés de la programmation shell
- www.faqs.org/faqs/unix-faq/shell/shell-differences - historique et particularités des différents shell.
- http://en.wikibooks.org/w/index.phptitle=An_Awk_Primer/ - tutorial Awk.
- http://www.grymoire.com/Unix/Awk.htm

# Vi

# Introduction


- *vi* est un éditeur de texte
- disponible sur l'ensemble des systèmes de type unix / Linux.
- modulaire, il peut servir aussi bien comme éditeur de base, que comme plateforme complète de développement.
- Très puissant, sa maîtrise demande un peu d'habitude et de pratique.

# Question

## Quelle est la 2$^{e}$touche la plus usée d'un clavier d'utilisateur linux ?

# Particularités


*vi* propose 2 modes :

- Un mode *commande* permettant de manipuler le texte dans son ensemble.
- Un mode *insertion* permettant de taper le texte.

Un 3$^{e}$mode *visuel* permet de faire des sélections.

# Pour passer d'un mode à l'autre

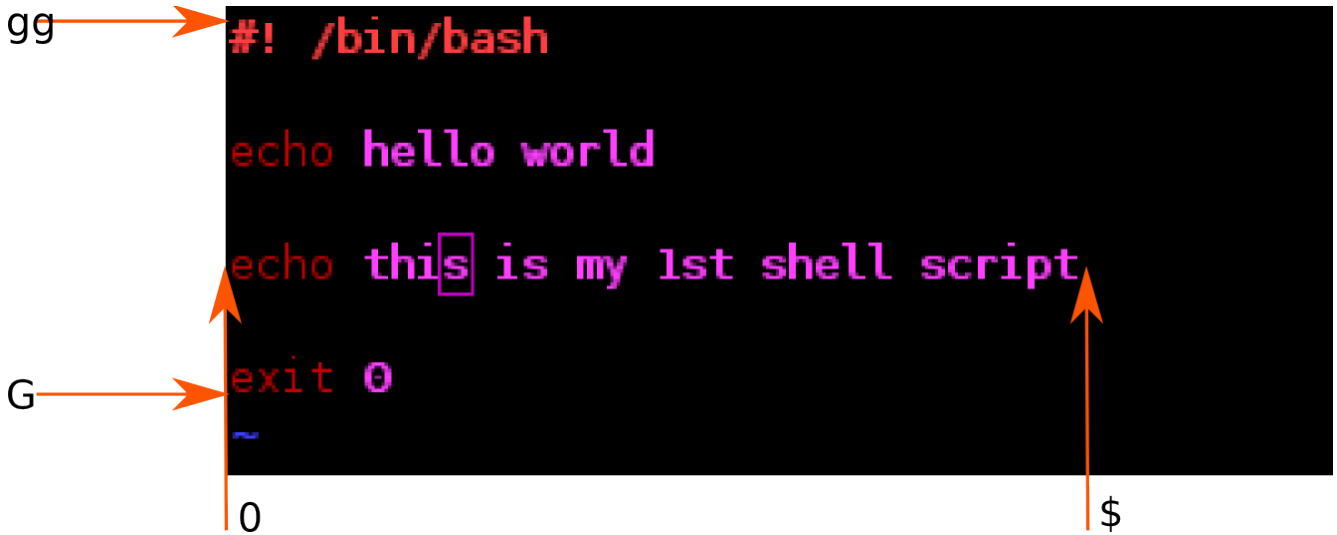

on utilise

- les touches **i, I, o, O, a, A** pour passer du mode *commande* au mode **i**nsertion.
- la touche **echap** pour passer en mode *commande*.
- Attention de ne pas mélanger les 2 modes !

# Édition et navigation

La touche **i** permet de passer en mode *insertion* et d'éditer le texte avec les touches habituelles. Les touches fléchées, Delete, backspace, Tab, PGUp, PGDown, etc. fonctionnent de manière habituelle.

# Navigation en mode commande




- Les éléments de *navigation* sont relatifs à la position du curseur.
- Ils peuvent s'appliquer à des *commandes* afin d'en spécifier *l'étendue*.
- Ils peuvent aussi être *quantifiés*.
- Ainsi, la séquence de touche <Esc>d3w va supprimer les 3 mots suivant le curseur. dG va supprimer toutes les lignes depuis le curseur jusqu'à la fin du fichier.

# Les commandes de suppression

**d**elete - supprimer

**c**orrect - corriger

# Suppression de texte

| | |
|---|---|
| x | supprime le caractère sous le curseur |
| dmouvement | supprime *mouvement* (ex : d0 supprime tout du curseur jusqu'au début de la ligne). |
| cmouvement | est équivalent, mais passe en mode *insertion* après la suppression. |
| dd | supprime l'ensemble de la ligne courante. |
| cc | supprime toute la ligne courante et passe ensuite en mode *insertion*. |
| D | supprime tout jusqu'à la fin de la ligne. |
| C | idem, en passant ensuite en mode *insertion* |

# Couper-Copier-coller

Sous vi, la suppression est en fait une “coupure”. On peut aussi copier de manière explicite :

ymovement *copie* mouvement.

Y ou yy *copie* la ligne entière.

Et pour coller :

p permet de *coller* l’élément précédemment coupé à *droite* du curseur.

P *colle* l’élément à *gauche* du curseur.

# Annuler

vi propose un mécanisme d'annulation :

- u permet d'annuler la dernière opération
- . permet de répeter la dernière opération

Control+r revient dans l'historique des opérations (inverse de *u*).

# Quitter vi

Pour finir :

:w écrit les modifications apportées au fichier.

:q quitte vi

:q ! quitte vi en abandonnant les modifications.

:wq combine les 2 1$^{res}$commandes.

# Configuration

La configuration de *vim* est centralisée :

- dans */etc/vim/vimrc* pour la configuration globale.
- dans ∼/.vimrc pour la configuration personnelle
- Le 2$^{e}$fichier étant interprété *après* le 1$^{er}$.

# Les principales directives du fichier personnel

```
filetype plugin on
syntax on
set showmode
set splitbelow
set splitright
set expandtab
set shiftwidth=3
set tabstop=3
set softtabstop=3
set smarttab
set smartindent
set foldmethod=indent
set nohlsearch
set showmatch
set binary noeol
set backspace=indent,eol,start
set laststatus=2
set nocompatible
set visualbell
set ruler
set autochdir
set background=dark
set cursorline
set linebreak

set modeline
set modelines=5
" F2 permet de passer en mode (no)paste
" avec affichage dans la barre de statut
nnoremap <F2> :set invpaste paste?<CR>
inoremap <F2> <C-O><F2>
```

# La commande find

*find* permet de rechercher des fichiers en fonctions de leurs *attributs*.

**find chemin_depart expression**

chemin_depart : dossier à partir duquel la recherche est faite.

expression : est composée d'*options*, d'*actions* et de *tests*

# Find : tests

Les *tests* permettent de limiter l'étendue d'une recherche aux fichiers répondant à des critères particuliers :

```
tom@cafeine$ find ~ -name "*.pdf"
```

| | |
|---|---|
| -type d,f,l,. . . | recherche sur le type de fichier. |
| -name nom | recherche sur le nom du fichier. |
| -perm permissions | recherche sur les permissions. |
| -size taille(bkM) | recherche sur la taille. |
| -user nom : | recherche sur le propriétaire |

# Find : éléments numériques pour les tests

lorsqu'un test prend une valeur *numérique*, celle-ci peut être spécifiée de la manière suivante :

```
tom@cafeine$ find . -cmin -60 -ls
```

+num : supérieur à *num*

-num : inférieur à *num*

num : égal à *num*

# Find : actions

Les *actions* s'appliquent à chaque fichier correspondant au résultat d'une recherche.

```
tom@cafeine$ find . -not -user  tom -okdir mv
{} /tmp/ \;
< mv ... ./fichier2 > ? y
tom@cafeine$
```

-print : action par défaut. Affichage du nom de fichier.

-ls : Affichage détaillé du fichier et de ses attributs.

-delete : Suppression du fichier.

-execdir command {} : exécution de *commande* pour chaque fichier du résultat.

-okdir command {} : idem ci-dessus, en demandant confirmation.

# Find : erreurs courantes

find : les chemins doivent précéder l'expression : le *point de départ* de la recherche n'a pas été spécifié.

find : paramètre manquant pour exec : la commande *-exec* ne se termine pas par *;*

# Traitements des fichiers textes

# Programmation Shell

1

1. xkcd

# Introduction

L'administration sous GNU/Linux c'est :

- La manipulation de fichiers textes
- Au moyen de différents outils spécialisés
- qu'il convient de combiner entre eux
- pour obtenir le résultat escompté.

# Les filtres

Les commandes de filtre permettent de modifier l'affichage d'un fichier :

| | |
|---|---|
| more, less et pg | affichage *paginé* d'un fichier texte. |
| head et tail | affiche le début ou la fin d'un texte. |
| sort | tri |
| cut | extraction de certaines parties d'un texte de longueur fixe |
| grep | sélection de certaines lignes suivant un critère. |
| sed | applique différents traitements au texte |

Ces commandes travaillent

- sur le fichier passé en paramètre (less /etc/passwd).
- sur les données fournies sur l'entrée standard (cat /etc/passwd | less)

# more et less

more permet d'afficher le contenu d'un fichier *page par page*

less propose les même fonctionnalités, avec en plus :

- La possibilité de revenir en arrière
- La possibilité de lancer des recherches (/recherche)
- La possibilité d'ouvrir le fichier pour édition

# tail et head

```
tom@workine:~$ tail -f /var/log/syslog
Jul 30 09:18:20 workine mountd[2726]: authenticated mount request from 192.168.
10.100:1016 for /home/Maison (/home)
Jul 30 09:18:20 workine mountd[2726]: authenticated mount request from 192.168.
10.100:768 for /home/Donnees (/home)
Jul 30 09:31:43 workine -- MARK --
Jul 30 09:51:43 workine -- MARK --
Jul 30 10:02:01 workine /USR/SBIN/CRON[21074]: (logcheck) CMD (   if [ -x /usr/
sbin/logcheck ]; then nice -n10 /usr/sbin/logcheck; fi)
```

**tail** affiche les 10 dernières lignes d'un fichier.

**tail -f** affiche ces 10 dernières lignes *en continu*.

**head** affiche les 10 premières lignes.

l'option *-n X*, commune aux 2 commandes, permet d'afficher X lignes.

# sort

```
tom@workine:~$ sort /etc/passwd
apt-mirror:x:110:121::/var/spool/apt-mirror:/bin/sh
avahi:x:108:116:Avahi mDNS daemon,,,:/var/run/avahi-daemon:/bin/false
backup:x:34:34:backup:/var/backups:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
games:x:5:60:games:/usr/games:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
haldaemon:x:102:104:Hardware abstraction layer,,,:/var/run/hal:/bin/false
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
```

*sort* permet de trier un flux de données.

Ses options sont :

- -n tri numérique
- -f ignore la casse de caractères
- -r inverse l'ordre du tri

# cut

*cut* est utilisé pour extraire un ou plusieurs *champs*.

- un champs est un sous-ensemble d'une phrase divisée par un *séparateur*
- qui est le caractère "tabulation" par défaut.
- ce délimiteur peut être modifié par l'option "*-d*".
- le champ demandé est référencé par l'option "*-f position*" avec position :

  nombre : le champs dont la position est *nombre*.

  nombre1-nombre2 tous les champs dont la position est comprise dans l'intervalle nombre1 $\sim$ nombre2

  n1,n2,n3 : tous les champs correspondant aux positions données

# cut : exemples

```
cut -f1,3 -d : /etc/passwd
```

```
0
```

# Recherches textuelles

**Question**

Comment rechercher et remplacer de multiples occurrences d'une expression dans un ou plusieurs textes de manière efficace ?

# Grep : recherche dans le contenu

*grep* permet de faire des recherches *à l'intérieur* des fichiers texte.

```
tom@workine:~$ grep tom /etc/passwd
tom:x:1234:1234::/home/tom:/bin/zsh
```

Ses principales options sont :

- -r : recherche récursive
- -l : affiche le nom du fichier contenant le terme recherché
- -i : recherche insensible à la casse.
- -v : inverse la condition
- -o : n'affiche que la partie correspondant au critère
- -c : affiche le nombre d'occurence
- –color affiche le résultat en couleur

# Sed : Modification du contenu

*sed* est un éditeur de flux.

- Il est principalement utilisé pour réaliser des "*recherche / remplacements*" répétitifs
- Il travaille ligne par ligne
- Il propose 3 commandes principales :

  p : affiche
  d : efface
  s : remplace

- ces commandes peuvent s'appliquer à 1 ou plusieurs lignes du fichier :

| | |
|---|---|
| n | sur la neme ligne |
| n,m | toutes les lignes entre la nieme et la mieme. |
| n,$ | depuis la neme ligne jusqu'à la dernière. |
| /regexp/ | sur les lignes correspondant à l'expression. |

# Sed : exemples

sed -ne ’3p’ exemple.txt

```
tom@workine:~$ sed -ne '3p' exemple.txt
Vous serez capable de gérer vos fichiers à partir de Windows et lancer les appl
ications bureautiques présentes sur votre poste.
```

L’option *-n* force sed à n’afficher que les lignes demandées. sed -e ’1,3d’ exemple.txt

```
tom@workine:~$ sed -e '1,3d' exemple.txt

Public : Vous venez d’acquérir un ordinateur et vous souhaitez démarrer.

Niveau requis : Il serait préférable que vous ayez une première pratique du cla
vier et de la souris.
```

sed -ne ’5,$p’ exemple.txt

```
tom@workine:~$ sed -ne '5,$p' exemple.txt
Public : Vous venez d’acquérir un ordinateur et vous souhaitez démarrer.

Niveau requis : Il serait préférable que vous ayez une première pratique du cla
vier et de la souris.
```

Notez l’utilisation de l’option *-n* sed -e ’/^ Objectif/d’ exemple.txt

```
tom@workine:~$ sed -e '/^Objectif/d' exemple.txt
```

# Sed : Recherche et remplacement

À l'aide de la commande *s*, on peut spécifier un *critère de recherche*, suivi de la chaîne de substitution sed –e 's/[Ww]indows/Linux/'exemple.txt

```
tom@workine:~$ sed -e 's/[Ww]indows/Linux/' exemple.txt
Objectif : A l'issue de ce cours, vous maîtriserez les fonctions élémentaires d
e Linux.

Vous serez capable de gérer vos fichiers à partir de Linux et lancer les applic
ations bureautiques présentes sur votre poste.

Public : Vous venez d'acquérir un ordinateur et vous souhaitez démarrer.

Niveau requis : Il serait préférable que vous ayez une première pratique du cla
vier et de la souris.
```

Il est aussi possible de spécifier une étendue sur laquelle appliquer la substitution sed –e '1s/[Ww]indows/Linux/'exemple.txt

```
tom@workine:~$ sed -e '1s/[Ww]indows/Linux/' exemple.txt
Objectif : A l'issue de ce cours, vous maîtriserez les fonctions élémentaires d
e Linux.

Vous serez capable de gérer vos fichiers à partir de Windows et lancer les appl
ications bureautiques présentes sur votre poste.

Public : Vous venez d'acquérir un ordinateur et vous souhaitez démarrer.
```

# Les expressions régulières

**Question :**

Comment exprimer le critère de recherche suivant de manière “formelle” ?

*“Une phrase de caractères commençant par une MAJUSCULE suivie de 5 lettres, d’un tiret et de 3 chiffres”*

# Expressions régulières

Les *expressions régulières* permettent de représenter un *motif* de recherche de manière formelle.

- Appelées aussi *expressions rationnelles*, les *regex* permettent d'affiner les critères de recherche.
- Utilisée notamment par *grep* et *sed*. Elles sont aussi disponibles via *man*, *vim*, *less*...
- Elles peuvent aussi servir de *conditions* dans les structures de tests.

# Caractères et classes

- a : détecte le caractère donné
- [abc] correspond au caractère a, b ou c
- [a–d] n'importe quel caractère dans l'intervalle spécifiée.
- . correspond à n'importe quel caractère.
- a* le caractère "a" 0 ou plusieurs fois
- a+ le caractère "a" 1 ou plusieurs fois
- a{n,m} le caractère "a" répété n à m fois
- ^ correspond au début d'une ligne
- $ correspond à une fin de ligne

# Autres :

- ch(at|ien) -> chat ou chien
- [^a] tout, sauf un a

# Classes de caractères POSIX

- [[: alnum :]] : tous les caractères alphanumériques, minuscules et majuscules.
- [[: blank :]] : tous les caractères d'espacement, tabulation, sauf les fins de ligne / paragraphes.
- [[: space :]] : tous les caractères de type "espace", y compris les fins de ligne
- [[: digit :]] : tous les chiffres.
- ...

# Capture

Il est possible d'extraire certaines parties d'une chaîne à l'aide des expressions régulières.
En effet, les parties entre parenthèses sont stockées dans des variables spéciales

- <img src="([^"]+)"
- Cet exemple, appliqué à un fichier html, détecte toutes les images présentes sur la page.
- À l'aide des parenthèses, leur URL sera stockée dans la variable 1

# Shell et expressions régulières

Il est possible d'utiliser des regexp sans outil externe :

- * correspond à n'importe quelle chaîne
- ? correspond à n'importe quel caractère
- [*abc*] soit a, b ou c
- [*a* – *c*] un caractère de l'intervalle a - c
- [^*x*] tout sauf x

Ex : lister tous les fichiers commençant par un chiffre, suivi de n'importe quelle lettre, sauf un m :

```
ls -l [0-9][^m]*
```

# Shell et expressions régulières

## Globbing étendu

```
shopt -s extglob

ls -l !(*.tex|*.pdf)
```

# Également dans un script

```
#! /bin/bash

if [ $# -lt 1 ] ; then
    read -p "gimme a string" foo
else
    foo="$@"
fi



if [[ "$foo" =~ ^T ]] ; then
    echo "this string start with a T"
else
    echo fail
    exit 5
fi
```

# Utilisation avec vi

Vi propose l'utilisation d'expressions régulières pour rechercher et modifier du texte.

```
:%s/^echo "\(.*\)"$/printf( \1 ) ; /gc
```

```
[range]s/MOTIF/Remplacement/[options]
```

range : spécifie l'étendue de la recherche (%, .,$)

MOTIF : l'expression régulière à rechercher.

Remplacement : la chaîne qui remplacera le motif recherché.

Options : options de recherche et remplacement, notamment ***g****lobal* et ***c****onfirm*.

# Utilisation avec grep

**grep –E "motif" fichier**

motif : expression régulière représentant la chaîne de caractères à rechercher.

fichier : fichier dans lequel effectuer la recherche.

Autres options : –color, –recursive, –ignore-case, –only-matching

# Expressions régulières : exemples

Syntaxe : grep –-color –E regexp fichier(s)

- ligne vide : /^$/
- toutes les lignes commençant par "un" et se terminant par "it" : ^un.*it$
- tous les mots de 8 lettres ou plus : \<[a–zA–Z]{8,}\> ou \<[[: alpha :]]{8,}\>
- toutes les lignes commençant par "univ" et ne se terminant pas par "t" ou "y" : ^univ.*[^ty]$

# Introduction

Awk est un utilitaire de *traitement* de texte.

- Il permet de récupérer diverses informations à partir d'un fichier,
- et de présenter ces informations sous forme de rapports.
- de faire du traitement mathématiques sur des fichiers contenant des données numériques.

# Syntaxe de base

```
awk option '/pattern/{instructions}' fichier
```

awk va parcourir *fichier* à la recherche de *pattern* et appliquer *instructions* au(x) résultat(s).

- Si *pattern* est null, l'ensemble du fichier sera traité.
- Si *instructions* est null, l'action *print* est effectuée par défaut.

Lors du traitement du fichier, awk découpe chaque ligne du résultat en *mot* et stocke chaque mot dans les variables $1, $2, $3, etc.

# Structure d'un programme Awk

```
BEGIN{
instruction1 ;
instruction2 ;
...
}
<pattern1>{instruction1 ;instruction2;
    instruction3... }
<pattern2>{instruction1 ;instruction2;
    instruction3... }
END{ instruction1; instruction2; ...}
```

# Invocation

```
awk [-Fseparateur] prog | -f fichierProg [
    variables] -|fichier
```

-F permet de spécifier le caractère utilisé pour séparer les champs.

-f indique le script awk

fichier représente le fichier à traiter. On peut utiliser - pour traiter l'entrée standard.

variables permet de passer des variables au script, sous la forme nom=valeur

# Patterns

Les “*patterns*” permettent de sélectionner les lignes du fichier sur lesquelles travailler. Ils peuvent prendre la forme :

- D’une expressions régulière.
- des mot-clés *BEGIN* et *END*
- d’un test (awk ’NR==5,NR== 10{print NR, $0}’/usr/share/dict/french
- d’un ensemble de patterns, combinés par des opérateurs logiques

# Variables

Awk dispose de variables réservées, et autorise les variables définies par l'utilisateurs, qui peuvent être passées au programme.

- awk –f script –v name=value fichier
- awk –f script –– name=value fichier

Les variables Awk ne sont pas typées. Les tableaux sont autorisés (var[1])

# Variables reservées

| | |
|---|---|
| FILENAME | nom du fichier utilisé. |
| FNR | numéro de la ligne courante |
| FS | séparateur de champs |
| IGNORECASE | booléen permettant de modifier le comportement vis à vis de la casse des caractères. |
| NF | nombre de champs dans la ligne courante. |
| NR | nombre de lignes déjà traitées. |
| PROCINFO | tableau contenant des informations sur le processus *awk* actuel. |

# Opérations

Awk propose les opérateurs de tests et arithmétiques classiques.

- Ces opérations obéissent à la même syntaxe que le C
  (awk 'BEGIN{a=10 ; print a+=10 }')
- La seule opération possible sur les chaînes est la *concaténation*
  (awk 'BEGIN{print str1  str2  ;} ' )

# Structures de contrôle

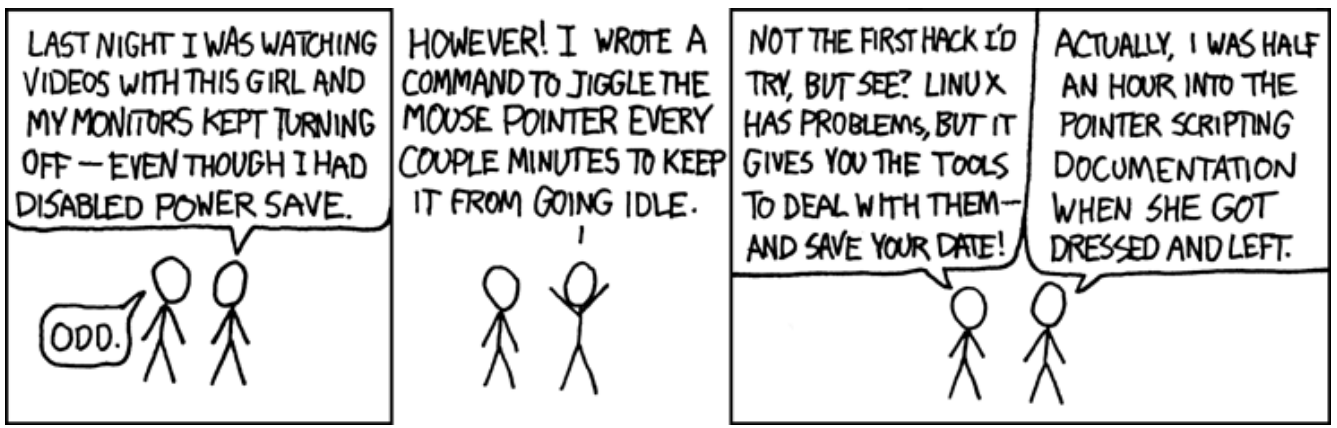
De la même manière qu'en C, on a les structures suivantes :

```
awk –F: '{ if (  $3>500){print $1 ;}} '   /etc/passwd
```

- if else if else
- while
- for
- Les boucles peuvent être controlées via les instructions *break* et *continue*.
- L'exécution d'un script awk est contrôlable via *next* et *exit*

# Programmation Shell

# Programmation Shell



2

2. xkcd

# Introduction

Un script shell permet de réaliser des tâches complexes et répétitives en combinant plusieurs commandes unix.

```
#! /bin/bash

echo "hello world"
exit 0
```

- Créer des comptes utilisateurs définis dans un fichier texte.
- Réaliser une sauvegarde personnalisée
- Apporter la même modification à un grand nombre de fichiers
- etc.

# Arrêt d'un script et code de retour

- exit int termine le programme en renvoyant la valeur *int*.
- si *valeur* est omis, c'est le code de retour de la dernière commande qui est renvoyé.
- Par convention, un code d'erreur != 0 indique une erreur.
- Le code de retour de la dernière commande est stocké dans la variable $?

# (Quelques) caractères spéciaux

- # pour commencer un *commentaire*.
- $ pour spécifier une *variable*.
- ; pour *chaîner des commandes sur la même ligne*.
- . pour “*sourcer*” un fichier externe.
- : équivalent à “true”.
- || OU logique.
- && ET logique.
- . . .

# Les variables : principes

- Pas de type.
- Pas de déclaration.
- Globale par défaut ( sauf si `local` ).
- Préfixée par $.

# Variables spéciales

- $0 nom de la commande en cours d'exécution.
- $1 1$^{er}$argument de ligne de commande.
- $n n$^{e}$argument de ligne de commande.
- $@ tableau contenant l'ensemble des arguments de ligne de commandes.
- $# nombre d'arguments.
- $$ pid du processus courant.
- $ ? code de retour de la précédente commande.
- ...

# Variables d'environnement

- Commandes env et export
- Elles sont définies au niveau système ( /etc/profile )
- Elles modèlent l'environnement de travail.
- Certaines sont en lecture-seule.

# Tableau

```
declare -a array1
array2=(one two three)
echo ${array1[1]}   # element 1 du tableau
echo ${#array[@]}   # nombre d'elements dans le tableau
```

# “Citation” et échappement

```
login="tom"
nom_complet="thomas      constans"
domaine="opendoor.fr"
echo $login
echo "$nom_complet"
echo "mail: ${nom}@${domaine}"
echo "\$login a pour valeur $login"
```

# Opération sur les variables

- Affectation simple.
- Affectation complexe : `userlist=$(cut -f1 -d: < /etc/passwd)`
- Concaténation. `mail="${user}@{domaine}"`
- Opérations arithmétiques. `let result=$a1 + $a2`

# Substitution de paramètres

- ${value:–default} si value n'est pas définie, utiliser *default*
- ${value:=default} si value n'est pas définie, l'initialiser à *default*
- ${value:?message} si value n'est pas définie, afficher *message*

# Opérations sur les chaînes de caractères

- ${var*#Pattern*} supprime de *var* la plus petite partie de *pattern*, à partir du début de $var
- ${var*##Pattern*} supprime de *var* la plus grande partie de *pattern*, à partir du début de $var : echo "${PWD##*/}"
- ${var%Pattern} supprime de *var* la plus petite partie de *pattern*, à partir de la fin de $var : mv $file ${ file %.c}.cpp
- ${var%%Pattern} supprime de *var* la plus grande partie de *pattern*, à partir de la fin de $var

# Syntaxe

- Les structures de contrôle permettent de combiner entre elles différentes expressions et commandes.
- On distingue notamment :
  1. Les *tests*.
  2. Les *répétitions*.
- Bash est pointilleux sur la syntaxe, et notamment sur les espaces.
- À chaque structure de contrôle correspond une version multiligne, et une version mono-ligne (mot-clés séparés par ;).

# test et [

```
test -d /home && echo "le repertoire /home existe"} }
[ "$LOGNAME" == "root" ] || echo "you are not admin"
```

- La commande interne **test** et son équivalent **[** renvoie 0 ou 1 suivant l'évaluation de *expression*.
- Le résultat du *test* est stocké dans la variable spéciale **$?**.
- On pourra l'utiliser soit par l'intermédiaire de la structure *if ; then ; fi*, soit à l'aide des opérateurs logiques *&&* et //.

# Tests sur chaînes de caractères

```
[ chaine1 operateur chaine2 ]
```

*operateur* peut prendre les valeurs suivantes :

- [ chaine1 == chaine2 ] vrai si chaine1 et chaine2 sont égales.
- [ chaine1 != chaine2 ] vrai si chaine1 est différente de chaine2.
- [ chaine1 \> chaine2 ] vrai si chaine1 se place devant chaine2 (alphabétiquement).
- [ chaine1 \< chaine2 ] vrai si chaine1 se place après chaine2.
- [ –z chaine1 ] vrai si chaine1 est nulle.
- [ –n chaine1 ] faux si chaine1 est nulle.

Il est nécessaire d'échapper les caractères spéciaux.

# Tests arithmétiques

```
[ a operateur b ]
```

*operateur* peut prendre les valeurs suivantes :

- [ a –lt b ] vrai si a < b
- [ a –le b ] vrai si a <= b
- [ a –gt b ] vrai si a > b
- [ a –ge b ] vrai si a >= b
- [ a –eq b ] vrai si a == b
- [ a –ne b ] vrai si a != b

# Tests sur les fichiers

```
[ operateur fichier ]
```

*operateur* peut prendre les valeurs suivantes :

- [ –d file ] vrai si *file* est un répertoire
- [ –e file ] vrai si *file* existe
- [ –f file ] vrai si *file* est un fichier régulier
- [ –r file ] vrai si *file* est accessible en lecture
- [ –w file ] vrai si *file* est accessible en écriture
- ...

# Combinaison et inversion de test

```
[ ! -n "$var" -o -r /etc/conf ]
```

| | |
|---|---|
| ! test | inverse le test. |
| test1 -o test2 | permet de lier les 2 tests par un *OU* logique. (renvoie vrai si l'un ou l'autre des tests est vrai). |
| test1 -a test2 | lie les 2 tests par un *ET* logique. (renvoie vrai si les 2 tests renvoient vrai). |

# Inconvénients

Les tests basés sur les commande **test** et **[** souffrent de 2 lacunes :

1. Les *expressions* peuvent être soumises à interprétation par le shell, et entraîner des problèmes
2. Les opérateurs dépendent du type de données à traiter.

La construction [[ test ]] permet de corriger ces limites, en proposant une syntaxe similaire aux autres langages.

```
[[ -z "$1" ]] && echo "argument manquant"
```

# Opérateurs logiques : utiliser les résultats d'un test

2 possibilités d'utiliser le résultat d'un test, en testant sa valeur de retour **?$** :

1. Utiliser un opérateur de test : `[ test ] && echo vrai || echo false`
2. Utiliser la structure de contrôle `if ( command ); then echo success ; fi`

# if ; then ; elif ; else ; if

```
if [[ $# < 2 ]] ; then echo "sage: $0 arg1 arg2" ; exit 1 ; fi
if [[ $1 == "--help" ]]
then
echo "Usage: $O arg1 arg2"
exit 0
elif [[ $1 == "root" ]]
then
echo "arg1 is root"
else
echo "arg1 is $1"
fi

if [[ (-e $2) && (-r $2) ]]; then echo "le fichier $2 existe et
    on peut le lire" ; fi

exit 0
```

# case

- La structure de contrôle case mot in motif1 | motif2) action ;; ... esac permet d'associer l'exécution d'un bloc de code en fonction de la valeur de *mot*.
- C'est l'équivalent du *switch* en C.
- Une fois qu'un *motif* a été repéré, le traitement s'arrête.

```
case "$1" in
"start"|"reload") echo starting
;;
"stop") echo stopping
;;
*) echo needing help ?
;;
esac
```

# Boucle while

- La boucle *while* permet d'exécuter le bloc d'instructions encadré par les mots-clés *do* et *done* tant que *test* est vrai.

```
while test
do
cmd1
cmd1
cmd1
done
```

# Boucle for et séquences

- Permet de "parcourir" une liste élément par élément.
- Une liste est une suite de mots séparés par des espaces.
- Il est possible de générer une suite de nombres à l'aide de la commande seq start [step] last
- Ex :
  - for file in *.c ; do gcc –Wall $file –o ${ file %.c}.exe ; done
  - for i in $(seq 1 5 100) ; do echo $i ; done
  - for word in $(cat fichier . txt ) ; do echo $word ; done

# loop control

Les instructions *break* et *continue* permettent de contrôler le déroulement d'une boucle

- break : provoque la sortie de la boucle.
- continue : saute automatiquement en début de boucle.

```
#! /bin/bash
i=0
while [[ $i -lt 20 ]] ; do
let "i+=1"
[[ $i == 10 ]] && continue
[[ $i -gt 14 ]] && break
echo -ne "$i "
done
echo $i
```

# Menu avec “select var in liste”

- Cette structure permet de construire rapidement un menu composé des éléments numérotés de *liste*.
- Le choix de l’utilisateur sera stocké dans la variable *var*.

```
select serveur in cafeine cocaine adrenaline morphine
do
echo Going to work on $serveur today
break
done

ssh $serveur
```

# Fonctions

```
function myFunction { command; }
myFunction () { command; }
```

- Une fonction permet de regrouper un bloc d'instructions qui sera exécuté plusieurs fois.
- Elle se termine à la fin du bloc d'instruction, ou à l'invocation de l'instruction *return*.
- les accolades doivent être séparées du nom de la fonction

# Fonctions et variables

- Toutes les variables sont par défaut *globales*
- Il est possible de réduire la portée d'une variable au corps de la fonction à l'aide du mot clé *local*
- Une fonction ne peut pas retourner autre chose qu'un *entier*.
- Pour utiliser le résultat d'une fonction, on peut utiliser la syntaxe suivante :

```
#! /bin/bash
function mySquareFunction {
echo $(($1*$1))
}
square=$(mySquareFunction $1)
```

# Fonctions et paramètres

- une fonction récupère ses paramètres par l'intermédiaire des *variables positionnelles*
- Les variables $1, $2, etc. contiennent les arguments passées à la fonction.
- La variable $0 reste inchangée.

# Signaux

- Les signaux sont un moyen de communication entre processus.
- Ils sont principalement utilisés pour le *contrôle de tâches*
- À l'arrivée d'un signal, un processus (et donc un script) peut choisir :
  - de l'ignorer (sauf SIGKILL et SIGSTOP)
  - d'exécuter une action particulière
  - d'exécuter le *comportement par défaut* lié au signal

# Envoi de signaux par le clavier

Il est possible d'envoyer des signaux à l'application en avant-plant à l'aide du clavier : Il y a différentes manières d'envoyer un signal à un processus :

- control + C envoie SIGINT
- control + Z envois SIGSTP
- control + \ envoie SIGKILL

# Envoi de signaux

On peut envoyer n'importe quel signal à n'importe quel processus (sous réserve d'en avoir le droit).

```
kill -SIGNUM pid
killall -SIGNUM nom
pkill -SIGNUM nom
```

numsignal : signal à envoyer. SIGTERM par défaut.

nom|pid : nom ou identifiant du processus cible.

# Interception de signaux

- Il peut être nécessaire d'empêcher l'interruption d'un script par l'arrivée d'un signal.
- l'instruction *trap* permet d'associer la réception d'un signal à l'exécution d'une commande ou d'une fonction.

trap command SIG1 SIG2 ...

command : instructions à exécuter à la réception du signal. Peut être *null*.

SIG1 SIG2 : liste des signaux à intercepter. Peut prendre les valeurs *EXIT* ou *DEBUG*.

# Contrôle de tâches

Au niveau du shell, une commande peut être lancée :

en avant-plan : (foreground). Ne rend pas la main pendant son exécution. Sensible aux interruptions clavier.

en arrière-plan : (background). Elle est alors insensible au signaux émis par le clavier et rend la main. Pour lancer une tâche en arrière-plan, il faut la suffixer par **&**.

- Il est possible changer l'état d'une commande par l'intermédiaire des instructions *bg* et *fg*.
- La liste des commandes en arrière-plan est disponible via la commande *jobs*.

# Contrôle de tâches

Les signaux permettent de modifier l'état d'un processus :

- En envoyant un signal *SIGSTP*, la commande en cours d'exécution est stoppée.
- En envoyant *SIGCONT*, elle est "réveillée" et poursuit son exécution.
- En envoyant les signaux *SIGTERM*, *SIGINT* (control+C) ou *SIGKILL* (control+\), la commande sera terminée (plus ou moins violemment).

# Cas pratique : passer une tâche en arrière-plan

1. Mise au point d'un script shell : vi script.sh, édition.
2. Test du script : un control+z met l'éditeur en arrière-plan et permet d'avoir la main sur le shell.
3. Retour à l'éditeur via la commande bg

# Affichage de messages

```
echo –e –n message
```

-e permet d'interpréter les séquences de caractères $\backslash n$, $\backslash t$, ...

-n pas de retour chariot à la fin du message.

# Récupérer une entrée utilisateur

La commande *read* récupère les informations entrées par l'utilisateur dans différentes variables.

```
read option VAR1 VAR2 VAR3
```

Avec *option* :

-p message : affiche *message* comme invite.

-s : l'entrée utilisateur n'est pas répétée sur la console

-t limite : attendre une entrée utilisateur pendant *limite* secondes.

VAR1 VAR2, ... : chaque mot de la ligne rentrée par l'utilisateur sera stocké dans chacune de ces variables.

...

ex : read –t 5 –p "entrez une chaine:" mot1 mot2 reste va afficher l'invite et attendre une entrée utilisateur pendant 5 secondes. Le 1$^{er}$mot de cette entrée sera stocké dans *$mot1*, le 2$^{e}$dans *$var2*, le reste de la phrase dans *$reste*.

# HereDocs

La syntaxe “heredocs” permet de spécifier une chaîne de caractère de manière *literale* en préservant les *espaces* et les *sauts de ligne*.

```
$ tr a-z A-Z << EOF
>ceci sera converti en majuscule
>EOF
CECI SERA CONVERTI EN MAJUSCULE
$
```

# Déboggage de script shell

- L'option *-x* de l'interpréteur permet d'afficher chaque commande au fur et à mesure du déroulement du script.
- En donnant à une variable l'attribut *trace* et en "*trappant*" les messages *DEBUG*, il est possible de détecter l'utilisation d'une variable donnée au cours de l'exécution du script :

```
declare -t variable
trap "echo $variable is in use here" DEBUG
...
```

# Bibliographie

- Bash wikibooks - En français, introduction à la programmation shell
- Bash for beginner - introduction au shell bash, y compris la programmation.
- Advanced Bash-Scripting Guide - Aborde les aspects basiques et avancés de la programmation shell

# Licence


## CC-BY-NC-SA

Ce support est mis à disposition selon le *Contrat Paternité - Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique 2.0 France* disponible en ligne ici ou par courrier postal à Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.