Cours introduction au shell

Félix HAURI

23 juin 2004

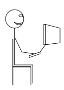

Table des matières

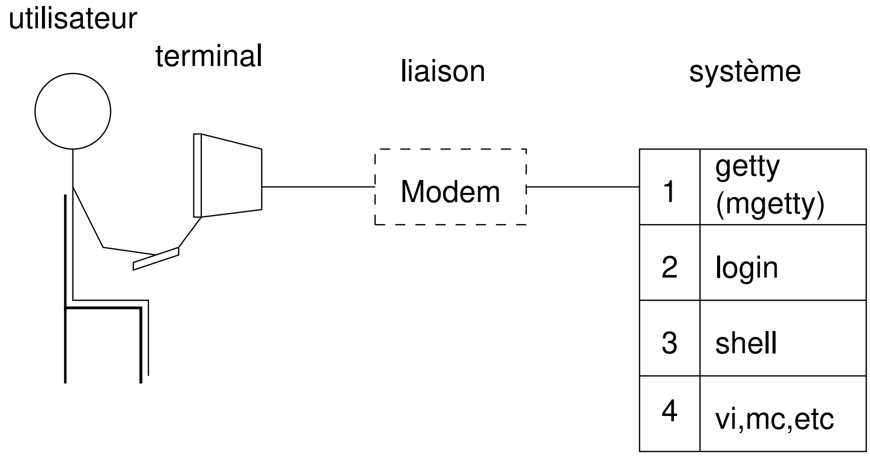

FIG. 1 – Fonction du terminal

1 Comment fonctionne le shell

Le shell est un outil permettant d'interagir avec l'ensemble d'un système Un*x.

2 Introduction

Le shell en lui-même ne sert strictement à rien, sorti de son contexte. C'est pourquoi, je commencerai par ce qu'il y a autour.

2.1 Un terminal

Le terminal est l'outil (texte) qui permet à un utilisateur de dialoguer avec un système informatique. Historiquement, on concevait un ordinateur sans écran dédié, mais pourvu d'un certain nombre de ports séries permettant de raccorder autant de terminaux.

Un terminal ne fait pas grand chose, en fait sa fonction est essentiellement mécanique : il sert à envoyer au système l'information "A" lorsque la touche "A" a été pressée et à afficher à l'écran la lettre "A" quand il la reçoit en provenance du système.

L'une des options standard est par exemple *echo* qui peut être *on* ou *off*. Dans le cas où l'option *echo* est *on*, le terminal enverra les touches pressées simultanément à l'écran ET au système. Dans ce cas, le système ne renverra plus les touches pressées puisque le terminal s'en charge.

Si *echo* est *on* et que le système croit que *echo* est *off*, alors chaque touche pressée aparaitra deux fois à l'écran. Inversement si *echo* est *off* et que le système la croit *on* alors les touches pressées n'apparaitront pas du tout à l'écran. Dans les deux cas les touches pressées seront reçues 1 fois par le système.

La *liaison* sur la figure 1 page précédente était au début un cable série (RS-232) puis un modem pour les connexions distantes et aujourd'hui sous Linux, c'est un module du kernel qui permet d'émuler soixante-trois [1] terminaux virtuels en utilisant l'ensemble clavier-écran. Les douze premiers de ces terminaux accessibles via les combinaisons de touches $\boxed{\text{Alt}}+\boxed{\text{F1}}$ à $\boxed{\text{Alt}}+\boxed{\text{F12}}$, les douze suivants avec les combinaisons $\boxed{\text{AltGr}}+\boxed{\text{F1}}$ à $\boxed{\text{AltGr}}+\boxed{\text{F12}}$. Les terminaux 25 à 63 sont accessibles avec les commandes *open*, *chvt* et *deallocvt*.

2.1.1 Un émulateur de terminal bien pratique : "screen"

Ces terminaux virtuels sont bien utiles, toutefois ils présentent deux inconvénients : ils ne sont réellement utilisables que pour les douze premiers terminaux et sont une particularité du kernel Linux.

Il existe une commande particulièrement pratique lorsque l'on travaille avec des terminaux, c'est la commande *screen*. Il s'agit d'un terminal, qui s'interface entre la console et le système, mais qui est doté d'un véritable panel d'outils très intéressants comme par exemple :
- Plusieurs sessions sur un seul terminal.
- Découpage de la console pour un affichage simultané de plusieurs sessions.
- Détachement de l'affichage d'une session d'un terminal réel pour le rattacher à un autre terminal réel (déplacement).
- Affichage d'une session sur plusieurs terminaux (copie, écolage)
- Ainsi qu'une floppée d'options et possibilités variées... (rtfm ;)

Le lancement habituel de la commande screen :
`screen -D -R`
Se *ratache* à un screen en cours, force le terminal sur lequel il était en train de tourner à quitter, et s'il n'existe pas de session screen en cours, se contente d'en créer une nouvelle.

`screen -x`

[1] ce nombre peut être modifié à la configuration du kernel

S'attache en *copie* à une session de screen en cours, s'il n'en existe pas, quitte avec un message d'erreur. Bien pratique pour l'écolage ; suivre sur plusieurs écrans ce qui se fait dans une session.

Une fois la commande lancée, on est *à l'intérieur* (dans un sous-shell) dès lors, la séquence $\boxed{\text{Ctrl}}+\boxed{\text{A}}$ sert de *séquence d'échappement*, suivie de :

$\boxed{\text{C}}$	Créer un nouveau shell
$\boxed{\text{A}}$	Transmet un *Control-A* au système
$\boxed{\text{Ctrl}}+\boxed{\text{A}}$	Echange l'affichage avec le précédent visible.
$\boxed{\text{Maj}}+\boxed{\text{S}}$	Splite l'affichage verticalement en deux.
$\boxed{\text{Maj}}+\boxed{\text{X}}$	Supprime une découpe de l'affichage.
$\boxed{:}$resize 10	Redimensionne la partie d'affichage courante à 10 lignes de haut.
$\boxed{\text{S}}$	Suspend l'affichage *Scroll-Lock*
$\boxed{\text{Q}}$	Libère l'affichage
$\boxed{\text{D}}$	Détache screen de la console (tty) courante

2.2 Shell interactif/passif

Le shell est par lui-même interactif puisque lancé sans argument, il se contente d'attendre des commandes de l'*entrée standard*.

En tant que langage de programmation les *scripts shells* peuvent être de simples séries de commandes invariables, à exécuter à la suite, sans intervention extérieure, on parlera d'un *script passif*. Si le script modifie son comportement en fonction de parametres extérieurs mais sans intervention humaine, on parle de *scripts semi-interactif*, enfin un script qui réagit en fonction d'un dialogue avec un utilisateur et un *script interactif*.

Cependant *bash* démarre en *mode interactif* lorsqu'il est lancé sans arguements ou avec l'option `-i` et démarre en *mode passif* lorsqu'il est lancé avec un script. La différence est l'utilisation des librairies readline et history, certains paramétrages et le lancement des scripts d'initialisation *.bashrc* etc.

2.3 Entrées de répertoires (*fichiers*)

L'un des concepts de base sous Un*x est de considérer toutes choses comme une entrée dans un répertoire. A tel point que j'utiliserais abusivement du terme "*fichier*" pour parler d'*entrée de répertoire*.

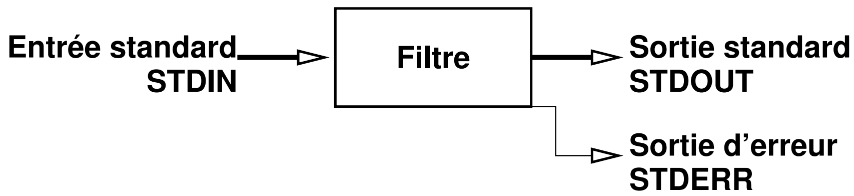

Fig. 2 – Entrée - sorties

Ceci est valable bien sûr pour un fichier, mais aussi pour un disque dur, une bande magnétique, une souris, etc.

Linux dans ses versions actuelles utilise un répertoire */proc* pour y faire figurer ses variables.

Certains *fichiers* n'ont ni début, ni fin (terminal, tty, port série, audio, video...)

! Attention, des accès même en lecture sur certains *fichiers* peuvent avoir un effet fâcheux sur le fonctionnement du système.

2.4 Entrées - sorties

Conçus pour traiter des flux de données provenant de ou allant vers des fichiers tels que précédement définis, les *executables* sous Linux agissent comme des *filtres*, recevant (éventuellement) un flux de données en entrée et générant un flux de données en sortie, en fonction de son entrée.

Une commande Linux se présente souvent comme une succession de filtres. Par exemple pour transformer une image jpeg en tiff, on peut utiliser une commande telle que :
`cat monfichier.jpg | djpeg | pnmtotiff > monfichier.tif`

On peut rediriger un ou plusieurs des trois flux avec les symboles plus-grand-que ou plus-petit-que :

Entrée standard	STDIN	<	Le symbole “<” suivi d’une entrée de répertoire valide y fera pointer l’entrée standard.
Sortie standard	STDOUT	>	Le symbole “>” suivi d’une entrée de répertoire valide y fera pointer la sortie standard. Si l’entrée n’existe pas mais que le répertoire est accessible en écriture, le fichier sera alors créé.
Sortie d’erreur	STDERR	2>	Sortie d’erreur (Le symbole “2>” aurra le même effet sur la sortie d’erreur que “>” sur la sortie standard.

Des commandes comme *dd*, *cat* ou encore *grep*, *sed*, etc. fonctionnent comme des filtres à part entière.

Bien que les trois flux existent, certaines commandes ne les utilisent pas tous, comme par exemple *ls*, *cp* ou *uptime* n’utilisent pas d’entrée et d’autres comme *rm*, *lpr* ou *umount* n’utilisent en prinicipe pas de sortie standard. Toutes, cependant utiliseront la sortie d’erreur pour signaler un disfonctionnement.

La sortie d’erreur sert à communiquer des états déstinés à la surveillance du déroulement d’une commande. Bien que par défaut, la sortie d’erreur *pointe* sur le même périphérique que la sortie standard, elle constitue un flux de données distinct afin de ne pas perturber les données que l’on cherche à obtenir.

A noter que de se débarrasser des messages d’erreurs connus en tapant
`... 2>/dev/null` est source d’ennuis. On préfèrera une syntaxe du style :

```
( ... ) 2>&1 | grep -v 'message connu\|autre message' >&2
```

Attention la ligne étant lue séquentiellement de gauche à droite, l’ordre d’apparition des commandes de redirections est important.

Par exemple :

```
cat </dev/xconsole 2>&1 >/dev/lp0 | logger
```

La commande *cat* sera lancée mais avant, son entrée standard sera pointée sur le *pipe nommé* “/dev/xconsole”, sa sortie d’erreur sera renvoyée sur sa *sortie standard*, c’est à dire le *pipe* “|” qui figure à la fin de la commande *cat* et enfin sa sortie standard sera envoyée directement sur le périphérique */dev/lp0* (le *port imprimante*). La commande *logger* sera ensuite lancée en utilisant le *pipe* “|” créé par la commande précédente (la sortie d’erreur de la commande *cat*) comme entrée standard.

La commande :

```
cat </dev/xconsole >/dev/lp0 2>&1 | logger
```

n'aura pas le même effet : la sortie standard pointant sur */dev/lp0* au moment de la lecture de la commande de redirection *2>&1*, la sortie d'erreur sera également redirigée sur */dev/lp0*. La commande *logger* ne recevra donc rien.

3 Les bases (La page man)

Ce cours est axé sur GNU bash (Bourne-Again SHell), il s'agit d'un choix personnel, je trouve CE shell sympathique facile, pratique et très largement utilisé dans le monde GNU.

Pour la bonne bouche, cet aveu : J'ai généré le présent cours en effectuant la commande suivante :

```
$ (IFS=$'\n';
    enlst=($(man -L en -Pcat bash |col -b |
                  grep -i ^[a-z][^1]*$));
    i=0;
    printf "\\%s{%s}\n" subsection $(
              man -Pcat -L fr bash |
              col -bp |
              grep -i ^[a-z][^1]*$ |
              while read line ;do
                  echo "$line \\small{"${enlst[$i]}"}";
                  i=$((i+1))
                  done))
```

Cela me permettra de faire mes commentaires pour chaque point de la page de man.

Au fait, la lecture de la page man est recommandée, mais avais-je besoin de le rappeler ?

```
$ man bash
```

```
$ man -Lfr bash
```

et pour l'imprimer (58pages) :

```
$ man -Lfr -Tps bash | lpr
```

(pour en faire un brochure je peux vous proposer un petit script :
`http://f-hauri.ch/impressionsouslinux/pstobroch.sh`
ou pour un imprimante recto-verso :
`http://f-hauri.ch/impressionsouslinux/pstobroch_rv.sh`

3.1 NOM NAME

Pas grand chose à dire pour ces trois premières sections...

3.2 SYNOPSIS SYNOPSIS

Remarque : Pour lancer un script shell il faut soit qu'il soit exécutable (bit x), soit qu'il soit lisible (bit r), dans ce cas il pourra être lancé en argument à la commande *sh*.

3.3 COPYRIGHT COPYRIGHT

GNU :-)

3.4 DESCRIPTION DESCRIPTION

c.f. : *IEEE POSIX Shell and Tools specification*

3.5 OPTIONS OPTIONS

Parmi les options on relèvera d'abord :
-c "string" Exécutiun de la commande *string*.
-x Affichage des commandes en cours d'execution (utile pour la recherche d'erreurs).
–login Execute un shell initialisé tel qu'issu d'une commande login (test de présence de nouveaux courriels etc.)
–posix Execute un shell strictement conforme aux recommandations POSIX.

3.6 ARGUMENTS ARGUMENTS

Après traitement des options, les arguments restants sont transmis au shell sous la forme de variables d'environnement $1, $2, etc. La variable $0 contient le nom du script lancé en tant que commande (bit x) ou le nom de l'executable lancé (bash).

3.7 INVOCATION invocation

Le comportement de *bash* varie selon qu'il est lancé via un lien nommé *sh*, via la commande *login*, via la commande `sh -l` ou via le système depuis un fichier dont l'entête est : `#!/bin/bash`, `#!/bin/sh` ou encore pour l'exemple : `#!/bin/bash -x`.

3.8 DÉFINITIONS definitions

Ce chapitre définit des termes utilisés tout au long de la man page.

mot , *jeton* ou *token* : suite de caractères alphanumériques, sans *blanc* ou alors encadrés par des délimitateurs (voir protection, section 3.12 page 13).

blanc , *séparation* ou *délimitateur*. Espace, tabulation, saut de ligne ou autres (voire variable IFS 3.13.3 page 14).

nom ou *identificateur*. Suite de caractères alphanumériques ou de soulignement "_" désignant une variable. (Attention à la casse !)

méta-caractère caractère spécial ayant pour effet de délimiter les mots s'ils ne sont pas *protégés* (voir 3.12 page 13).

```
| & ; ( ) < > espace tabulation
```

opérateur de contrôle caractère spécial ayant pour effet de délimiter les commandes si non *protégé*.

```
|| & && ; ;; ( ) | <fin-de-ligne>
```

3.9 MOTS RÉSERVÉS reserved words

Les expressions suivantes sont interprétées pas le shell. Attention à ne pas utiliser ces termes sans protection (guillemets, etc.)

```
! case do done elif else esac fi for function

 if in select then until while { } time [[ ]]
```

3.10 GRAMMAIRE DU SHELL SHELL GRAMMAR

3.10.1 Commandes simples

Une affectation de variable et/ou une commande, éventuellement suivie d'options, paramètres et/ou redirections et terminée par un *opérateur de contrôle*

3.10.2 Pipelines

Suite de commandes se passant les sorties standard des premières aux entrées standard des commandes suivantes, séparées par le caractère spécial *pipe* : "|".

L'interprêteur attend la fin du dernier processus et renvoie le code de retour du dernier processus.

3.10.3 Listes

Séquence d'une ou plusieurs commandes ou pipelines, séparés par l'un des opérateurs `;`, `&`, `&&`, ou `||`, et terminée éventuellement par `;`, `&`, ou *fin de ligne*.

3.10.4 Commandes composées

Succession de une ou plusieurs listes groupées de la manière suivante :
- (*liste*)
- { *liste* ; }
- ((*expression*)
- [[*expression*]]
- **for** *nom* [**in** *mot*] ; **do** *liste* ; **done**
- **select** *nom* [**in** *mot*] ; **do** *liste* ; **done**
- **case** *mot* **in** [*motif* [| *motif*] ...) *liste* ; ;] ... **esac**
- **if** *liste* ; **then** *liste* ; [**elif** *liste* ; **then** *liste* ;] ...
- **while** *liste* ; **do** *liste* ; **done**
- **until** *liste* ; **do** *liste* ; **done**

3.11 COMMENTAIRES COMMENTS

Lorsqu'un caractère "#" est rencontré par l'interpréteur, le reste de la ligne sera ignoré.

3.12 PROTECTIONS QUOTING

Les *méta-caractères* et *opérateur de contrôle* peuvent êtres *protégés* afin de ne pas être interprétés par le shell.

Le caractère spécial "\" sert à protéger le caractère suivant. la séquence "\\" produira "\"

Les protection par guillemets """ diffèrent des protection par apostophe "'" par le fait que les variables "*$nom*" et les commandes imbriquées "*'liste'*" ou "*$(liste)*" seront développées entre guillemets mais ne le seront pas entre apostrophes

A l'inverse, l'échapement *apostrophe inversée* ainsi que *$(liste)* provoqueront l'éxecution de la liste de commandes imbriquées et le passage de sa sortie standard en *paramètre* à la ligne de commande courante, AVANT sont exécution.

De même, l'échapement *$'sequence ANSI'* provoquera l'interprétation de la séquence ANSI avant le passage de paramètres à la commande en cours.

3.13 PARAMÈTRES PARAMETERS

Symbolisés avec le "$", un paramètre est une entité permettant de stocker une valeur. Peut être soit un *nom*, dans ce cas il s'agit d'une *variable* soit un nombre, dans ce cas il s'agit d'un *Paramètre Positionnel.*

3.13.1 Paramètres Positionnels

$0 prend pour valeur la commande en cours. Les options sont stoquées dans $1 à $n. Il s'agit des options ou arguments de la ligne qui a appelé la commande (*script* ou *fonction*) ou de *parsing* séparation des champs à l'aide de la commande *set* –.

3.13.2 Paramètres Spéciaux

Une série d'informations non modifiables (a priori)
- "$*" Tous les paramètres positionnels depuis $1, séparés par ${IFS :0 :1}
- "$@" Idem
- "$#" Nombre de paramètre positionnels (décimal)
- "$?" Code de retour de la dernière commande
- "$-" Liste des options du shell en cours
- "$$" PID du shell en cours
- "$!" PID du dernier processus *fils* démarré depuis le shell en cours
- "$0" Nom du script en cours d'execution
- "$_" Dernier arguement de la dernière ligne de commande tapée.

3.13.3 Variables du Shell

Une petite sélection de variables intéressante :

- **PPID** L'ID du processus parent.
- **PWD** Répertoire courant
- **OLDPWD** Dernier répertoire quitté.
- **UID** User ID
- **EUID** Effective User ID
- **BASH_VERSION** 2.05a.0(1)-release ;-)
- **SHLVL** Niveau généalogique du shell en cours
- **RANDOM** Entier positif entre 0 et 32767 au *hasard*
- **SECONDS** Nombre de secondes écoulées depuis le démarrage du shell en cours.
- **HOSTNAME** Et d'autres variables du même métal : HOSTTYPE, OSTYPE, MACHTYPE...
- **IFS** Input Field Separators. en principe `IFS=$' \t\n'`
- **HOME** Le répertoire personnel de l'utilisateur
- **COLUMNS** et **LINES**. Voir la commande *tput*
- **LANG** Détermine la localisation par défaut
- **LC_ALL** Supplante **LANG** et les autres **LC_**
- **LC_TIME** Supplante **LANG** pour l'affichage du temps.
- **PS1** Prompt de commande un truc comme : `PS1='\u@\h:\w\$ '`
- **PROMPT_COMMAND** Permet de spécifier une commande qui sera exécutée avant chaque affichage du prompt.
- **TIMEFORMAT** Permet de formater la sortie de la commande *time*
- **HISTFILESIZE** Nombre de ligne du fichier d'historiques `~/.bash_history`

3.13.4 Les Tableaux

Bash propose un mécanisme simple permettant de réaliser des tableaux à une dimension.

Pour stoquer des variables dans un tableau nommé FOO, on fera :

`FOO=(valeur1 valeur2 "valeur trois")`

Pour récupérer les valeur : `$FOO` donnera *valeur1*, tandis que `${FOO[2]}` donnera *valeur trois*

Pour modifier le tableau : `unset ${FOO[1]};echo ${FOO[@]}` donnera : *valeur1 valeur trois.*

Il existe un tableau sans nom... C'est le tableau des parametres positionnels.

3.14 EXPANSION expansion

3.14.1 Expansion des accolades

Permet de spécifier plusieurs variations sur une chaine de paramètres :

`a{b,c,e}` sera étendu en *ab ac ae.*

3.14.2 Développement du Tilde

Les caractère ˜ est utilise pour des répertoires :

- **˜** : Répertoire personnel $HOME
- **˜username** : Répertoire de l'utilisateur *username*
- **˜+** : =PWD
- **˜-** : =OLDPWD

3.14.3 Remplacement des paramètres

Remplacement des parametres identifiés par le symbole $. Traitement des variables dans le même temps :
- **${paramètre :-mot}** Utilisation d'une valeur par défaut.
- **${paramètre :=mot}** Attribution d'une valeur par défaut.
- **${paramètre :?mot}** Erreur si paramètre non défini.

- **${paramètre :+mot}** Utilisation d'une valeur alternative si paramètre défini.
- **${paramètre :début :longueur}** Partie de paramètre.
- **${#paramètre}** Longueur du paramètre.
- **${paramètre#mot}** Suppression 1x du motif correspond à *mot*, au début du paramètre.
- **${paramètre##mot}** Suppression de tous les motifs correspond à *mot*, au début du paramètre.
- **${paramètre%mot}** Idem # mais à la fin du paramètre.
- **${paramètre%%mot}** Idem ## mais à la fin du paramètre.
- **${paramètre/motif/chaîne}** Remplacement de *motif* par *chaîne*.

3.14.4 Substitution de commandes

Permet d'effectuer une *liste* et de renvoyer comme paramètre à la commande courante la sortie standard de la liste.

`echo ‘date‘` ou `echo $(date)`

3.14.5 Évaluation Arithmétique

Le format d'évaluation arithmétique est :

```
$((expression))
```

3.14.6 Séparation des mots

Les mots sont séparés en fonction de la variable $IFS.

Les espaces en début de ligne et consécutifs entre les paramètres sont ignorés.

3.14.7 Développement des noms de fichiers

Remplacement des caractères spéciaux "*", " ?" et "[...]" par des noms de fichiers existants.

3.14.8 Suppression des protections

Toutes les occurences non protégées des caractères "\", "´", et """ qui ne résultent pas d'un développement sont supprimées.

3.15 REDIRECTION REDIRECTION

Les caractères spéciaux "<" et ">" sont utilisés pour rediriger respectivement les entrées et les sorties.

- < Redirection de l'entrée en provenance de.
- > Redirection de la sortie vers.
- >> Redirection de la sortie à la suite de.
- **2**> Redirection de la sortie d'erreur vers.
- **2**>> Redirection de la sortie d'erreur à la suite de.
- **2**>**&1** Redirection de la sortie d'erreur vers là où pointe la sortie standard.
- **&**> Redirection de la sortie standard **et** de la sortie d'erreur vers...
- << Document en ligne.

Attention à l'ordre sur la ligne :

`ls >fichier 2>\&{}1` n'aura pas le même effet que :

`ls 2>\&{}1 >fichier`

3.16 ALIAS ALIASES

Permet de systématiser une chaine de commande :
`alias myls="/bin/ls -ltrF --color"`

3.17 FONCTIONS FUNCTIONS

Permet de créer des fonctions indépendantes qui se comporteront comme des scripts.
`mycalc() { echo "scale=30;pi=4*a(1);$@"|bc -l; }`

Note : A la différence des scripts shell, les fonctions s'exécutent dans la session courrante, cela permet de créer des fonctions qui affecteront les variables. Un exemple :

Si on considère le script cd2tmp.sh ci dessous :
`#!/bin/sh`

```
cd /tmp
```

L'exécution du script en lançant "`sh cd2tmp.sh`" ou "`./cd2tmp.sh''` n'aura pas l'effet espéré. Pour que le script s'exécute dans le shell courant il faut utiliser la commande "*source*" ou son alias "." :
`source cd2tmp.sh` ou `. cd2tmp.sh`.

Le seul moyen d'obtenir un effet d'une commande sur un shell en cours est de créer une fonction :
`cd2tmp() { cd /tmp; }`

3.18 CALCUL ARITHMÉTIQUE ARITHMETIC EVALUATION

Shell est capable d'effectuer des fonctions arithmétiques sur des entiers. (voir 3.14.5 page 16).

- `-` + plus et moins unaire,
- `!` `~` négations logique et binaire,
- `**` exponentiation,
- `*` / % multiplication, division, reste,
- + `-` addition, soustraction,
- « » décalage arithmétique à gauche et à droite,
- <= >= < > comparaisons,
- == != égalite et différence,
- & ET binaire,
- ^ OU exclusif binaire,
- | OU binaire,
- && ET logique,
- || OU logique.

3.19 EXPRESSIONS CONDITIONNELLES CONDITIONAL EXPRESSIONS

Pour effectuer un test, deux syntaxes : `test` *`expr cond`* ou `[`*`expr cond`*`]`.

par exemple :

```
if test -b /dev/fd0 ; then echo fd0 is a block device;fi

if [ -b /dev/fd0 ] ; then echo fd0 is a block device;fi
```

- **-b fichier** Une entrée de répertoire "fichier" existe et est de type *block device.*
- **-c fichier** Une entrée de répertoire existe et est de type *character device*
- **-d fichier** Une entrée de répertoire existe et est un (sous-)répertoire.
- **-e fichier** Une entrée de répertoire existe.
- **-f fichier** Une entrée de répertoire existe et est un fichier régulier.
- **-g fichier** Une entrée de répertoire existe et a son bit Set-GID positionné.
- **-h fichier** Une entrée de répertoire existe et est un lien symbolique.
- **-k fichier** Une entrée de répertoire existe et a son bit *sticky* positionné.
- **-p fichier** Une entrée de répertoire existe et est un tube nommé (FIFO).
- **-r fichier** Une entrée de répertoire existe et est accessible en lecture avec les droits actuels.
- **-s fichier** Une entrée de répertoire existe, est un fichier et a une taille non nulle.
- **-t [fd]** Vrai si fd est ouvert sur un terminal. Si *fd* n'est pas fourni, les test est effectué sur la sortie standard.
- **-u fichier** Une entrée de répertoire existe et a son bit Set-UID positionné.
- **-w fichier** Une entrée de répertoire existe et est accessible en écriture avec les droits actuels.
- **-x fichier** Une entrée de répertoire existe et est exécutable avec les droits actuels.
- **-O fichier** Une entrée de répertoire existe et appartient à l'ID effectif de l'utilisateur.
- **-G fichier** Une entrée de répertoire existe et appartient au groupe effectif de l'utilisateur.
- **-L fichier** Une entrée de répertoire existe et est un lien symbolique.
- **-S fichier** Une entrée de répertoire existe et est une socket.
- **-N fichier** Une entrée de répertoire existe et a une date de lecture antérieur à sa date de modification.
- **fichier1 -nt fichier2** Vrai si le fichier1 est plus récent que le fichier2, d'après les dates de modification.
- **fichier1 -ot fichier2** Vrai si le fichier1 est plus vieux que le fichier2.
- **fichier1 -ef fichier2** Vrai si le fichier1 et le fichier2 sont sur le même périphérique et ont les mêmes numéros d'i-noeuds (*liens durs*).
- **-o option** Vrai si l'option shell *option* est active. (voir commande set -o section 3.31 page 27)
- **-z chaîne** Vrai si la longueur de la chaîne est nulle.
- **-n chaîne**
- **chaîne** Vrai si la longueur de la chaîne est non-nulle.
- **chaîne1 == chaîne2** Vrai si les deux chaînes sont égales. Le symbole = peut servir à remplacer ==

- **chaîne1 != chaîne2** Vrai si les deux chaînes sont différentes.
- **chaîne1 $<$ chaîne2** Vrai si chaîne1 est *plus petit que* chaîne2 dans le sens alphanumérique du terme.
- **chaîne1 $>$ chaîne2** Vrai si chaîne1 est *plus grand que* chaîne2 dans le sens alphanumérique du terme.
- **arg1 OP arg2** Où arg1 et arg2 sont des nombres entiers, et *OP* est l'un des opérateurs **arithmérique** suivants :
  - **-eq** Arg1 est égal à arg2,
  - **-ne** Arg1 est différent d'arg2,
  - **-lt** Arg1 est inférieur à arg2,
  - **-le** Arg1 est inférieur ou égal à arg2,
  - **-gt** Arg1 est supérieur à arg2,
  - **-ge** Arg1 est supérieur ou égal à arg2,

voir "`help test`".

3.20 DÉVELOPPEMENT DES COMMANDES SIMPLES SIMPLE COMMAND EXPANSION

Les commandes simples sont déchiffrées de gauche à droite en interprétant dans l'ordre :

1. Les redirections et noms d'affectation sont mémorisées.
2. Les mots sont développés, le premier mot restant est considéré comme *la* commande, le reste est passé en tant qu'arguments.
3. Les redirections sont appliquées.
4. Les expressions suivants des "=" sont développées, tilde et autres variables, substitutions des commandes, évaluations mathématiques.

3.21 EXÉCUTION DES COMMANDES COMMAND EXECUTION

Le premier mot de la ligne est une commande... S'il contient un "/" il s'agit d'un chemin complet, l'interpréteur cherchera un fichier exécutable correspondant et l'executera ou affichera une erreur.

S'il ne contient pas de "/" c'est qu'il s'agit soit d'une fonction, soit d'une fonction interne, soit d'un alias, soit d'une commande présente dans l'un des répertoires listés dans la variable *PATH*, soit enfin d'une erreur.

Si la commande est un fichier exécutable dont la première ligne commence par "# !", le reste de la première ligne indique le chemin, le nom et les paramètres de l'interpréteur à lancer pour exécuter (interprèter) le fichier.

3.22 ENVIRONNEMENT D'EXÉCUTION DES COMMANDES COMMAND EXECUTION ENVIRONMENT

Lorsqu'un shell démarre, il comprend :
- Entrée standard, sortie standard, sortie d'erreurs,
- un répertoire courant,
- un masque de création de fichier,
- une gestion des signaux,
- les paramètres du shell tels que transmis par le parent ou positionnés par la commande set (voir 3.31 page 27),
- des fonctions,
- des options fournies au démarrage par défaut, sur la ligne de commande ou par la commande interne set,
- les options de shell activées par la commande interne shopt,
- des alias
- des variables telles que les ID de divers processus, y compris ceux à l'arrièreplan,
- la valeur de $$, et la valeur de $PPID.

Lorsqu'une commande - qui n'est ni une fonction interne, ni une fonction définie - est executée, un nouvel environnement *fils* est généré, il recevra en héritage les informations suivantes :
- les I/Os,
- le répertoire courant,
- le masque de création de fichiers,
- les variables *exportées*,
- les réactivités aux signaux d'interruption.

Une commande invoquée dans un environnement séparé ne pourra pas affecter l'environement du shell qui l'a invoquée.

Les substitutions, commandes imbriquées ou asynchrones sont invoquées dans un sous-shell dont l'environnement sera une copie de l'environnement du shell qui invoque, à l'exception des réactivités aux signaux d'interruption qui reprendront les comportements prévus au lancement du shell.

Lorsque une commande est invoquée avec "&" et que le contrôle des jobs n'est pas actif (voir section 3.26 page suivante), le périphérique virtuel */dev/null* est utilisé comme entrée standard. Si le contrôle des jobs est actif, alors les entrées-sorties courantes sont utilisées.

3.23 ENVIRONNEMENT ENVIRONMENT

Quand un programme est invoqué, il reçoit un tableau de chaînes que l'on appelle environnement. Il s'agit d'une liste de paires nom-valeur, de la forme nom=valeur.

Les commandes *declare -x* et *export* permettent de préparer les variables ou fonctions pour l'exportation vers les sous-shells, les commandes *declare +x* et *export -n* interdisent l'exportation de variables ou fonctions vers des sous-shells.

Les variables peuvent êtres supprimées avec la commande *unset*

3.24 CODE DE RETOUR EXIT STATUS

Le code de retour d'une commande peut être accédé via la variable spéciale d'environement *$?*.

Lorsqu'une commande se termine normalement, le code de retour (result code ou RC) vaut "0".

Lorsqu'un processus fils ne trouve pas une commande, il retourne 127, s'il trouve une commande qu'il ne peut pas exécuter, il retourne 126...

La plupart des commandes disposent d'un codage particulier des significations des codes d'erreur, en fonction de leur application.

3.25 SIGNAUX SIGNALS

Pour ce chapitre, je vous renvoie à la page de man et à `man 7 signal`.

A priori pour commencer, la chose à noter est que le "SIGTERM" termine un script shell, mais pas un shell interactif.

3.26 CONTRÔLE DES JOBS JOB CONTROL

Possibilité de suspendre un job en cours, de l'envoyer en exécution en *arrière plan*, de le ramener en *avant plan*, d'en lancer plusieurs, de les tuer...

Lors de l'exécution d'un process ou une commande, la pression de la combinaison des touches Ctrl+Z suspendra instantanement l'exécution. La commande *fg* relancera l'exécution normalement, tandis de *bg* relancera l'exécution, mais en *arrière plan.*

On utilise l'alias "%" pour manipuler et adresser les jobs, la commande *jobs* pour lister et vérifier leur états, les commandes *fg* et *bg* servent respectivement à amener un job en *avant plan* ou en *arrière plan* et enfin la commande *kill* permettra d'envoyer des signaux à une commande en cours.

3.27 SYMBOLE D'ACCUEIL (PROMPT) PROMPTING

En mode interactif (ligne de commande), le shell affiche une chaine d'information ayant pour principale fonction de signifier que le shell est prêt à recevoir une commande. Cette chaine se nomme le prompt et peut être configurée à loisir.

Le bash propose pour cette configuration une variable d'environnement *PS1* qui sert à configurer le prompt :
`export PS1='\u@\h:\w\$'`
fera afficher avant chaque commande, le nom d'utilisateur, un *at*, le nom d'hôte de la machine, un *deux-points*, le chemin de répertoire courant et un *dollard* ou un *dièse* selon que l'utilisateur courrant et *root* ou non.

3.28 BIBLIOTHÈQUE READLINE READLINE

La librairie *readline* est l'ensemble de routines chargées de gérer les manipulations en lignes d'un utilisateur. Il s'agit essentiellement de fonctions d'édition de lignes...

On distingue essentiellement deux modes de fonctionnement interactif : le mode *vi* et le mode *emacs* auxquels on accède respectivement avec `set -o vi` ou `set -o emacs`.

En mode emacs, le mode par défaut, les combinaisons suivantes sont disponibles (entre autres) :

Ctrl+A	Début de ligne
Ctrl+E	Fin de ligne
Ctrl+L	Effacer l'ecran
Ctrl+_	fonction *undo*

La fonction *undo* correspond à la combinaison de touches : Ctrl+_ (avec le clavier suisse-romand, les touches Ctrl+Shift+_).

La réactivité aux séquences de commandes est entièrement *customizable.*

3.29 HISTORIQUE HISTORY

Par défaut, le shell garde une trace des dernières commandes entrées. Cette trace est confinée dans le fichier `$HOME/.bash_history`

Petit survol des principales variables y relatives :

HISTIGNORE	Liste de motifs séparés par des " :"	unset
HISTCONTROL	Fonction de l'un des trois mots clef : *ignorespace*, *ignoredups* ou *ignoreboth.*	unset
HISTSIZE	Nombre de ligne gardées en mémoire.	500
HISTFILE	Nom du fichier de sauvegarde de l'historique.	`$HOME/.bash_history`
HISTFILESIZE	Taille maximum du fichier de sauvegarde.	500

3.30 EXPANSION DE L'HISTORIQUE HISTORY EXPANSION

On utilisera le point d'exclamation " !" pour rappeler des lignes de commandes de l'historique :

Un " !" suivit de quelque chose d'autre qu'un espace, retour de chariot ou ouverture de parenthèse provoquera une recherche dans l'historique.

!n	Numéro de ligne dans l'historique.
!-n	Nième avant-dernière ligne dans l'historique.
!chaîne	Recherche de la ligne la plus récente commençant par "chaîne".
! ?chaîne[?]	Recherche de la ligne la plus récente contenant "chaîne".
^chaîne_1^chaîne_2^	Substitution rapide : répétition de la dernière ligne après substitution de "chaîne_1" par "chaîne_2".

3.31 COMMANDES INTERNES DU SHELL SHELL BUILTIN COMMANDS

Il s'agit de commandes internes, servant habituellement à modifier le compordement du shell lui-même.

:	[arguments]	Ne fait rien, mais avec succès.
.	fichier [arguments]	Incorpore (exécute) *fichier* dans l'environnement courrant.
source	fichier [arguments]	Idem "."
alias	[-p] [nom[=valeur] ...]	Définition de *raccourcis*
bg	[job]	BackGround, fait tourner *job* en arrière-plan.
bind	[-m jeu_de_touches] [-lpsvPSV]	Manipulation des réactivités des touches du clavier via la librairie *readline*
bind	[-m jeu_de_touches] [-q nom] [-u fonction] [-r sequence]	
bind	[-m jeu_de_touches] -f fichier	
bind	[-m jeu_de_touches] sequence :fonction	
break	[n]	Sortir de n niveaux de boucles imbriquées.
builtin	commande_interne [arguments]	Effectue une commande interne même si une fonction ou un alias a été défini avec un nom identique.
cd	[rep]	Change de répertoire. Modifie les variables *PWD* et *OLDPWD*.
command	[-pVv] commande [arguments ...]	Effectue un commande interne ou une commande présente dans le *PATH* même si un alias ou une fonction à été défini avec un nom identique.
compgen	[option] [mot]	Génère une complétion possible pour le mot, en fonction des options fournies.
complete	-pr [name ...]	Configure les completions possibles. Les programmation de completions ne seront pas abordées dans ce cours.

continue	[n]	Saute l'exécution courante d'une boucle pour passer à sa prochaine itération.
declare	[-afFirx] [-p] [nom[=valeur]]	Configure une variable comme étant une valeur entière, une fonction, une liste, une variable *read-only* et/ou une variable à exporter.
typeset	[-afFirx] [-p] [nom[=valeur]]	Idem declare.
dirs	[-clpv] [+/-n]	Affiche et manipule les répertoires empilés avec *pushd* ou dépilés avec *popd*.
disown	[-ar] [-h] [job ...]	Libère un sous-job pour le rendre orphelin.
echo	[-neE] [arguments ...]	Retourne les arguments...
enable	[-adnps] [-f fichier] [nom ...]	Autorise ou empêche l'accès à des commandes internes.
eval	[arguments ...]	Exécute les arguments comme une commande et transfère le code de retour à la commande eval.
exec	[-cl] [-a nom] [commande [arguments]]	Exécute une commande avec le process courant. Le shell quitte et dès la fin de la commande, le process termine.
exit	[n]	Fin de script. La valeur servira de code de retour.
export	[-nf] [nom[=mot]] ...	Exporte une variable de sorte que les processus fils en héritent.
export	-p	Liste les variables exportées
fc	[-e éditeur] [-nlr] [premier] [dernier]	Ré-édite une ou plusieurs lignes de commande et les execute.
fc	-s [motif=nouveau] [commande]	Ré-execute la dernière ligne dont la commande est *commande* après avoir remplacé *motif* par *nouveau*
fg	[job]	ForeGroung, ramène le job suspende *jpb* en avant-plan.
getopts	chaîne_d_options nom [arguments]	Lecture d'option pour création de scripts.
hash	[-r] [-p fichier] [nom]	Mémorise des chemins d'accès pour des commandes courantes.
help	[motif]	Aide.
history	[-c] [n]	Affiche n lignes d'historique (-c détruit le fichier)
history	-anrw [fichier]	Manipule l'historique de ou vers un fichier

jobs	[-lnprs] [job ...]	Manipulation des processus fils, affichage d'informations.
kill	[-s sigspec | -n signum | -sigspec] [pid | job]	Envoye un signal à un processus.
kill	-l [sigspec | exit_status]	Info sur les signaux.
let	argument [argument ...]	Effectue l'opération arithmétique et retourne 0 (vrai) si l'opération ne donne pas 0.
local	[nom[=valeur] ...]	Défini une variable comme variable locale à l'intérieur d'une fonction. Cette variable retrouvera son état (évent non défini) après la fin de la fonction.
logout		Quitter un shell de login.
popd	[-n] [+n] [-n]	Récupère n entrées de la pile de répertoire et effectue un *cd* sur le premier élément.
printf	format [arguments]	Formate une chaine de sortie en fonction de la chaîne *format*.
pushd	[-n] [rep]	Ajoute un répertoire à la liste pile de répertoires.
pushd	[-n] [+n] [-n]	Manipulation, rotation de la pile avec execution de la commande *cd*
pwd	[-LP]	Print Working Directory, affiche le répertoire courant.
read	[-er] [-a nom_a] [-p msg] [nom ...]	Lit une ligne de l'entrée standard.
readonly	[-afp] [nom ...]	Manipulation de variables en lecture seule.
return	[n]	fin de fonction avec code de retour *n*
set	[-abefhkmnptuvxBCHP] [-o option] [arg ...]	Véritable couteau suisse des variables, cette commande permet de manipuler à peu près tout ce qui constitue le shell.
shift	[n]	Rotation vers la gauche des arguments passés à la commande $2 ... Le premier ($1) est simplement oublié.
shopt	[-pqsu] [-o] [nom_opt ...]	Manipule les paramètres optionels du shell, à savoir : cdable_vars, cdspell, checkhash, checkwinsize, cmdhist, dotglob, execfail, expand_aliases, extglob, histappend, histreedit, histverify, hostcomplete, huponexit, interactive_comments, lithist, mailwarn, nocaseglob, nullglob, promptvars, restricted_shell, shift_verbose et sourcepath

suspend	[-f]	Suspend le shell en cours jusqu'à réception d'un signal SIGCONT.
test	expr	Effectue le test représenté par la condition *expr* et retourne vrai ou faux selon le résultat du test.
times		Affiche les durées cumulées utilisateur et système pour le shell courant et les processus lancés par ce shell
trap	[-lp] [cmd] [sigspec]	Retourne de suite et libère le shell en attendant l'un des signaux proposés par *sigspec*, il effectuera alors la commande *cmd*.
type	[-atp] nom [nom ...]	Donne des informations sur la commande passée en arguement.

ulimit	[-SHacdflmnpstuv [limite]]	Permet de contôler la disponibilité des ressources pour le process courant et ses fils. `ulimit -a` affiche les limites en cours.
		`core file size (blocks, -c) 0`
		`data seg size (kbytes, -d) unlimited`
		`file size (blocks, -f) unlimited`
		`max locked memory (kbytes, -l) unlimited`
		`max memory size (kbytes, -m) unlimited`
		`open files (-n) 1024`
		`pipe size (512 bytes, -p) 8`
		`stack size (kbytes, -s) 8192`
		`cpu time (seconds, -t) unlimited`
		`max user processes (-u) 256`
		`virtual memory (kbytes, -v) unlimited`

umask	[-p] [-S] [mode]	Manipulation du masque de création de fichier
unalias	[-a] [nom ...]	Suppression d'un alias.
unset	[-fv] [nom ...]	Suppression d'une variable et/ou d'une fonction.
wait	[n]	Attendre la fin d'un processus fils.

3.32 SHELL RESTREINT RESTRICTED SHELL

Un shell restreint ne permet pas de
- Changer de répertoire.
- Manipuler les variables *SHELL*, *PATH*, *ENV* ou *BASH_ENV*.
- Faire référence à un fichier ou une commande contenant un "/".
- Importer des fonctions de l'environement père.
- Importer des valeurs de `$SHELLOPTS` de l'environnement père.
- Utiliser des redirections $<$,$>$, etc.
- Utiliser les fonctions internes : exec, enable, command -p, set +r ou set +o restricted.

3.33 VOIR AUSSI SEE ALSO

Mais ne l'ais-je pas déjà dit ?

`man bash`, `info bash`, `man bash`, `zless /usr/share/doc/bash/FAQ.gz` et les exemples dans `cd /usr/share/doc/bash-doc/examples` et aussi :

Bash Reference Manual , Brian Fox and Chet Ramey
The Gnu Readline Library , Brian Fox and Chet Ramey
The Gnu History Library , Brian Fox and Chet Ramey
Portable Operating System Interface (POSIX) Part 2 : Shell and Utilities, IEEE
et sh(1), ksh(1), csh(1), emacs(1), vi(1) et readline(3)
et encore... sed(1), awk(1), grep(1), wc(1), du(1), find(1), etc...

3.34 FICHIERS FILES

Tiens, on ne cite pas */etc/bash.bashrc*... A savoir : selon les distrib (compilation de bash), ce fichier se nomme différement.

3.35 AUTEURS AUTHORS

Une pensée émue pour les auteurs.

Brian Fox, Free Software Foundation <bfox@gnu.org>
Chet Ramey, Case Western Reserve University <chet@ins.CWRU.Edu>

3.36 RAPPORT DE BOGUE BUG REPORTS

A lire avant les complaintes.

3.37 BOGUES BUGS

A lire aussi, sans commentaires.

3.38 TRADUCTION

Un grand merci aux traducteurs de la doc :

Christophe Blaess, Thierry Vignaud.