

**Notre vocation,
votre réussite**

FORMATIONS ORSYS

VOTRE SUPPORT DE COURS

Séminaires
Cours de synthèse
Stages pratiques
Certifications
Cycles certifiants
e-Learning

Ce support pédagogique vous est remis dans le cadre d'une formation organisée par ORSYS. Il est la propriété exclusive de son créateur et des personnes bénéficiant d'un droit d'usage. Sans autorisation explicite du propriétaire, il est interdit de diffuser ce support pédagogique, de le modifier, de l'utiliser dans un contexte professionnel ou à des fins commerciales. Il est strictement réservé à votre usage privé.

Cours Shell

DAVID PLANTROU




Copyright



Cette présentation est mise à disposition selon les termes de la Licence Creative Commons Attribution - Pas d'Utilisation Commerciale - Partage dans les Mêmes Conditions 4.0 International (CC BY-NC-SA 4.0).

<http://creativecommons.org/licenses/by-nc-sa/4.0/>

Vous êtes autorisé à partager et adapter ce document, selon les conditions suivantes :

-  Attribution — Vous devez créditer l'œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'œuvre.
-  Pas d'Utilisation Commerciale — Vous n'êtes pas autorisé à faire un usage commercial de cette œuvre, tout ou partie du matériel la composant.
-  Partage dans les mêmes conditions — Dans le cas où vous reprenez ou effectuez une modification de cette œuvre, vous devez la diffuser dans les mêmes conditions, c'est à dire avec la même licence d'utilisation.

Présentation

Programme

Formation Shell :

- Historique
- Rappels
- Présentation shell Unix : scripting
- Robustesse, debuggung
- Extensions du Korn Shell et Bash
- Outils supplémentaires : vi, grep, find, sort, sed, awk

PAUSE – REFLEXION

Avez-vous des questions ?



UNIX

UNIX fait partie de la famille des systèmes d'exploitation

Il est multitâche et multi-utilisateur

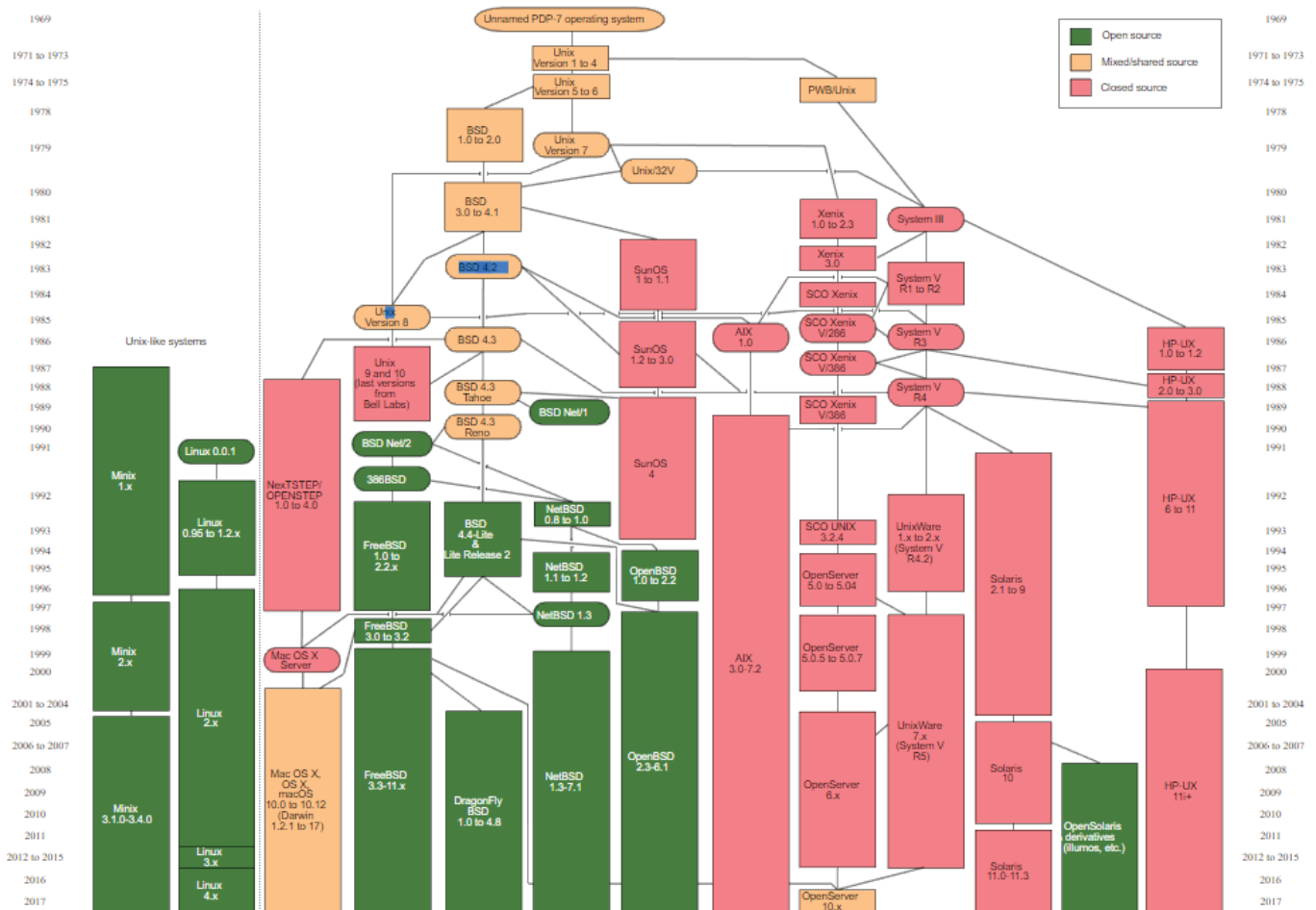
Créé en 1969 par Ken Thompson qui travaillait alors aux Laboratoires Bell

Dennis Ritchie se joint ensuite à lui pour développer UNIX



Ken Thompson et
Dennis Ritchie

Avec le temps de nombreuses branches d'UNIX vont apparaître



PLANTROU DAVID - SHELL - 2019

7

Les grandes familles

HP-UX

- Implémentation propriétaire de Hewlett Packard créée en 1984 – sources non disponibles

AIX

- Implémentation propriétaire de IBM créée en 1986 – sources non disponibles

Famille BSD (Berkeley Software Distribution)

- Sources non disponibles au départ (1977) mais mises à disposition depuis 1991

Oracle Solaris

- Implémentation développée initialement par Sun Microsystems en 1992
- Acquis en janvier 2010 lors du rachat de Sun par Oracle

Linux

- Créé en 1991 par Linus Torvalds – système open-source
- Nombreuses distributions : Debian, Ubuntu, Mint, Red Hat, Fedora, SUSE,...
- Utilisé dans de nombreux systèmes (serveurs, ordinateurs personnels, embarqué - Android)

MAC OS X

- Créé en 2001 par Apple



Linus Torvalds

PLANTROU DAVID - SHELL - 2019

8

PAUSE – REFLEXION

Avez-vous des questions ?



Rappels

Le système de fichiers (1/6)

Le système de fichier est la façon dont le contenu du support mémoire est organisé (disque dur, carte SD, clé USB,...)

Chaque système d'exploitation a développé son propre système de fichier

- FAT16, FAT32 ou encore NTFS pour Microsoft

Un système de fichier est utilisé par partition sur le support de stockage. Ainsi, plusieurs systèmes de fichiers peuvent cohabiter sur le même support

Le système de fichiers (2/6)

Les système de fichiers généralement utilisés sous UNIX sont :

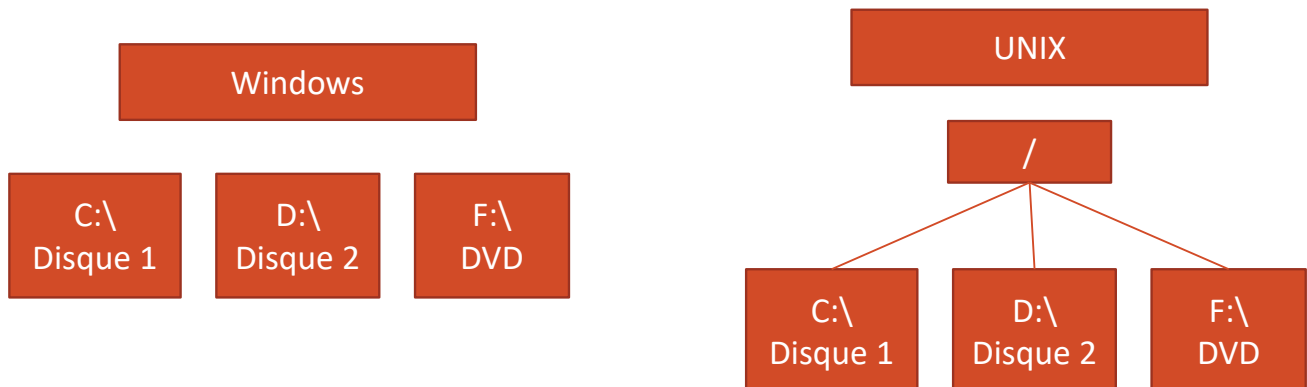
- Unix file system (UFS) sous les systèmes UNIX
- XFS
- JFS
- ReiserFS
- Extended file system 3 (ext3) pour Linux
- Extended file system 4 (ext4) pour Linux

Les partitions FAT et NTFS peuvent aussi être utilisés

Le système de fichiers (3/6)

Dans un système UNIX, tout est représenté par un fichier dans le système

Ainsi, un périphérique, en particulier un support de stockage (disque dur, cd-rom, carte SD,...) est rattaché à l'arborescence des fichiers qui est unique



Le système de fichiers (4/6)

Ainsi, le système de fichiers est représenté par une arborescence UNIX

Un utilisateur ne voit donc pas directement à quel support il s'adresse réellement

Un système de fichiers d'un périphérique est attaché à un endroit de l'arborescence. On dit qu'il est monté

De même, il peut être détaché. On dit qu'il est démonté

Lors de l'installation, une partition doit être désignée comme racine de l'arborescence unique

C'est dans cette arborescence que seront montées les autres partitions / lecteurs

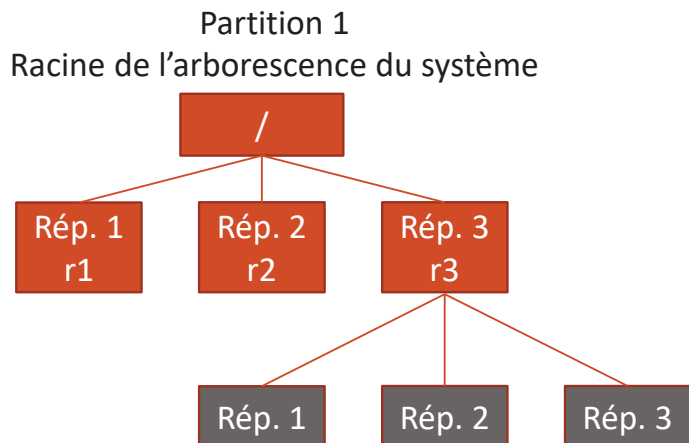
Le système de fichiers (5/6)

Pour illustrer



Le système de fichiers (6/6)

On monte la partition 2 dans le répertoire 3 (r3) de la partition principale : `mount <adresse_peripherique> /r3`



Organisation du système de fichiers

Il n'y a pas de règle absolue sur l'organisation du système de fichier

Chaque distribution UNIX possède ses variantes

Néanmoins, une organisation standard est à peu près respectée

Organisation du système de fichiers

Emplacement	Description
/bin	Programmes / commandes de base
/boot	Noyau du système
/dev	Contient les périphériques du système
/etc	Fichiers de configuration
/home	Répertoires des utilisateurs
/lib	Librairies du système et modules noyau
/mnt	Utilisé pour monter divers supports de stockage
/sbin	Binaires destinés à l'administration (root)
/tmp	Fichiers temporaires
/usr	Programmes utilisateurs, librairies,...
/var	Fichiers de log, fichiers de cache, fichiers divers

Organisation du système de fichiers

Lorsqu'un utilisateur se connecte, il se retrouve à l'emplacement de son répertoire personnel :

- /home/jack pour l'utilisateur jack
- /home/patrick pour l'utilisateur patrick

Il y a une seule exception à la règle :

- L'utilisateur root (administrateur) se retrouve à l'emplacement /root suite à sa connexion

Pour travailler

Pour gérer le système, UNIX propose en général deux possibilités :

- L'interface graphique (KDE, GNOME,...)
- La console

La console est l'outil de base de tout système UNIX et c'est le plus puissant.

Il permet de saisir des commandes manuellement.

Le prompt

Le prompt est l'invite de commande de la console

```
david@debian:/var/log$
```

C'est à la suite du prompt que les commandes sont saisies

Le prompt est personnalisable

```
PS1="\t] \u@\h:\w\$ "
```

Les utilisateurs

Un utilisateur se connecte au système via son nom d'utilisateur et son mot de passe

Il existe dans un système UNIX un utilisateur root qui est l'administrateur de la machine

L'utilisateur root peut tout faire. Il a tout les droits

On peut passer à l'utilisateur root via la commande `su`

```
david@debian:/var/log$ su
```

```
Mot de passe :
```

```
root@debian:/var/log#
```

Pour des raisons évidentes de sécurité, l'utilisateur root est utilisé **UNIQUEMENT** lorsque cela est nécessaire

Les commandes

Les commandes

Shell utilisé :

La manière la plus simple de connaître le shell que l'on utilise est d'exécuter la commande unix

ps qui liste les processus de l'utilisateur :

```
PID TTY TIME CMD
6908 pts/4 00:00:00 bash => l'interpréteur utilisé est bash
6918 pts/4 00:00:00 ps
$
```

Pour connaître la version utilisée : bash --version

Les commandes

Shell utilise 2 types de commandes :

- Les commandes internes
 - Une commande interne est une commande dont le code est implanté au sein de l'interpréteur de commande.
 - Lorsqu'on change de shell courant ou de connexion, par exemple en passant de bash au C-shell, on ne dispose plus des mêmes commandes internes.
 - Ces commandes sont plus rapides.

Exemples de commandes internes : cd , echo , for , pwd

RQ : Sauf dans quelques cas particuliers, l'interpréteur ne crée pas de processus pour exécuter une commande interne.

Les commandes

Shell utilise 2 types de commandes :

- Les commandes externes
 - Une commande externe est une commande dont le code se trouve dans un fichier ordinaire.
 - Le shell crée un processus pour exécuter une commande externe.
 - Parmi l'ensemble des commandes externes que l'on peut trouver dans un système, nous utiliserons principalement les commandes unix (ex : ls, mkdir, cat, sleep) et les fichiers shell.

Pour connaître le statut d'une commande, utiliser la commande interne type.

```
Ex :    $ type -t sleep
        file => sleep est une commande externe
        $ type -t echo
        builtin => echo est une commande interne du shell
```

Les commandes

Les commandes respectent en général un modèle de syntaxe

```
commande [options] paramètres
```

Par exemple

- `ls -la`
- `chmod -R root:root /var/log/*.rtr`

La commande la plus importante à connaître :

```
man commande
```

Par exemple

- `man ls` affichera la documentation de la commande `ls`

On peut aussi utiliser généralement `commande --help`

Les commandes

```
commande [options] paramètres
```

Les options vont permettre de modifier le comportement d'une commande

Pour certaines commandes, il en existe des dizaines qu'il est possible de combiner

C'est le résultat de ces combinaisons qui fait la puissance de la console UNIX

L'exécution d'une commande

Pour exécuter une commande il suffit de la saisir au niveau du prompt et de valider la saisie par la touche ENTREE

Le résultat de la commande (si il existe) s'affiche alors sur la console

La console nous affiche lors un nouveau prompt

Il est possible de saisir ses commandes sur plusieurs lignes grâce au caractère \

Exemples de commandes

`id` : affiche les informations (uid, gid et groupes) sur l'utilisateur courant

```
david@debian:~$ id
uid=1000(david) gid=1000(david)
groupes=1000(david),24(cdrom),25(floppy),29(audio),30(dip),44(video),46(plugdev),108(netdev),110(lpadmin),113(scanner),118(bluetooth)
```

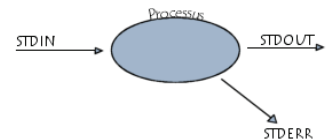
`hostname` : affiche le nom de la machine

```
david@debian:~$ hostname
debian
```

Entrées sorties standard

Lors de l'exécution d'une commande, un processus est créé. Celui-ci va ouvrir 3 flux :

- **stdin** (entrée standard), dans lequel le processus va lire les données d'entrée. Par défaut stdin correspond au clavier ; STDIN est identifié par le numéro 0 ;
- **stdout** (sortie standard), dans lequel le processus va écrire les données de sortie. Par défaut stdout correspond à l'écran ; STDOUT est identifié par le numéro 1 ;
- **stderr** (erreur standard), dans lequel le processus va écrire les messages d'erreur. Par défaut stderr correspond à l'écran. STDERR est identifié par le numéro 2 ;



Exécution séquentielle

- Mode d'exécution par défaut : le shell lit la commande entrée, l'analyse, la prétraite et si elle est syntaxiquement correcte, l'exécute.
- Une fois l'exécution terminée, le shell effectue le même travail avec la commande suivante.
- L'utilisateur doit donc attendre la fin de l'exécution de la commande précédente pour que la commande suivante puisse être exécutée.
- On dit que l'exécution est synchrone.

Exemple : `$ cd /tmp ; pwd; echo bonjour; cd ; pwd`
/tmp => affichée par l'exécution de pwd
bonjour => affichée par l'exécution de echo bonjour
/home/sanchis => affichée par l'exécution de pwd

Exécution en arrière-plan

- L'exécution en arrière-plan permet à un utilisateur de lancer une commande et de récupérer immédiatement la main pour lancer « en parallèle » la commande suivante (parallélisme logique).
- On utilise le caractère & pour lancer une commande en arrière-plan.
- L'utilisateur peut, en parallèle, exécuter d'autres commandes
- Lorsque la commande en arrière-plan se termine, le shell le signale à l'utilisateur après que ce dernier ait appuyé sur la touche entrée.

```
Ex : $ sleep 5 &
      [1] 696
      $ ps
      PID TTY          TIME CMD
      683 pts/0        00:00:00 bash
      696 pts/0        00:00:00 sleep
      697 pts/0        00:00:00 ps
      $      => l'utilisateur a appuyé sur la touche entrée mais sleep n'était pas terminée
      $      => l'utilisateur a appuyé sur la touche entrée et sleep était terminée
      [1]+  Done                  sleep 5
      $
```

PAUSE – REFLEXION

Avez-vous des questions ?



Le shell

PRINCIPES D'EXÉCUTION D'UNE COMMANDE (EXEC, PIPELINE, SOUS-SHELL, BACKGROUND...)

Le shell

Le shell (coquille) est un interpréteur de commande

C'est un programme qui sert d'intermédiaire entre l'utilisateur et le système d'exploitation

Le shell est chargé d'exécuter des programmes :

- Lecture d'une ligne de commande
- Analyse de la ligne de commande
- Lancement du programme concerné avec ses paramètres
- Gestion des redirection entrées / sorties
- Exécution de scripts

Les types de shells

La famille UNIX propose différents types de shells

Le shell historique apparu au début des années 70

- Le « Bourne-Shell »
- `/bin/sh`

Un autre shell apparu à la fin des années 70

- Le « C Shell »
- Syntaxe proche du langage C
- `csh`

Les types de shells

Avec le temps, de nombreux shells, inspirés de ceux existant déjà ont fait leur apparition

Le très utilisé bash

- Bourne-Again shell (par défaut sous de nombreux UNIX – Linux / Mac OSX)

D'autres variantes

- Korn shell (`ksh`)
- Z shell (`zsh`)
- TENEX C Shell - (`tcsh`)

Chaque shell est venu apporté ses modifications et ses ajouts

https://en.wikipedia.org/wiki/Comparison_of_command_shells liste les caractéristiques des différents shells

Le shell

Lorsqu'un utilisateur est créé, il est associé à un shell par défaut :

- Consultable dans le fichier `/etc/passwd`
- Il s'agit souvent du shell `bash (/bin/bash)`

Dès que l'utilisateur se connecte (via un terminal) ou lance une console dans l'interface graphique, le shell est lancé automatiquement

Après son lancement, le shell se configure via des scripts de profil

Une fois opérationnel, il affiche un prompt et attend une commande

Pour quitter le shell, on utilise la commande `exit`

Le shell

- Pour résumer Les fichiers `/etc/profile` et `$HOME/.bash_profile` sont lus quand le shell est invoqué comme shell interactif de connexion.
- Le fichier `$HOME/.bashrc` est lu quand le shell est invoqué comme shell interactif sans fonction de connexion
- Il s'agit d'une idée générale. Le fonctionnement change d'un UNIX à l'autre.
- Il faut consulter le man de son shell
- Pour connaître les options de `bash` : commande `set -o`

Le shell

Le shell bash utilise pour se configurer :

- Le fichier `/etc/profile` : il s'agit du fichier de configuration universel pour tous les utilisateurs, `root` compris – On y trouve par exemple le format du prompt. Parfois ce script se contente d'invoquer `/etc/bash.bashrc`
- Le fichier `$HOME/.bash_profile` (=spécifique à chaque utilisateur) est utilisé en cas de connexion via une console ou à distance via `ssh`
- Le fichier `$HOME/.bashrc` est utilisé en cas de connexion via l'interface graphique ou via la commande `/bin/bash` (= pas une ouverture de session).

Fichiers configuration shell (1/3)

	sh	ksh	csh	tcsh	bash	zsh
<code>/etc/.login</code>			login	login		
<code>/etc/csh.cshrc</code>			yes	yes		
<code>/etc/csh.login</code>			login	login		
<code>~/.tcshrc</code>				yes		
<code>~/.cshrc</code>			yes	yes		
<code>~/etc/ksh.kshrc</code>		int.				
<code>/etc/sh.shrc</code>	int					
<code>\$ENV</code> (typically <code>~/.kshrc</code>) ¹	int	int.			int.	
<code>~/.login</code>			login	login		
<code>~/.logout</code>			login	login		

Fichiers configuration shell (2/3)

	sh	ksh	csH	tcsh	bash	zsh
/etc/profile	login	login			login	login
~/.profile	login	login			login	login
~/.bash_profile					login	
~/.bash_login					login	
~/.bash_logout					login	
~/.bashrc					int.+n/login	

Fichiers configuration shell (3/3)

	sh	ksh	csH	tcsh	bash	zsh
/etc/zshenv						yes
/etc/zprofile						login
/etc/zshrc						int.
/etc/zlogin						login
/etc/zlogout						login
~/.zshenv						yes
~/.zprofile						login
~/.zshrc						int.
~/.zlogin						login

Le shell : les alias (1/2)

On peut spécifier des **alias** de commande pour se faciliter la vie

Par exemple, pour tous les utilisateurs, on peut rajouter dans le fichier `/etc/bash.bashrc` la ligne

```
alias ll="ls -l"
```

Cela permettra à l'utilisateur de pouvoir saisir `ll` au lieu de `ls -l`

Remarque : il y a un ordre de priorité dans l'exécution des commandes, alias et autres fonctions :

- ✓ les alias ;
- ✓ les fonctions ;
- ✓ les commandes internes ;
- ✓ les commandes externes.

Le shell : les alias (2/2)

Il est possible également de créer des alias permanents (ou suivis) en les créant dans le fichier `~/bash_aliases`

Exemple :

```
alias ls='ls --color'
alias ll='ls -l'
alias la='ls -A'
```

Bien vérifier que les lignes suivantes dans le fichier `~/bashrc` sont décommentées :

```
if [ -f ~/.bash_aliases ]; then
  . ~/.bash_aliases
fi
```

Les variables d'environnement

Il s'agit de variables permettant de stocker une information

Cette variable est valable au sein de tout l'environnement utilisateur

On liste les variables d'environnement grâce à la commande `env`

On peut afficher une variable particulière grâce à la commande `echo`

```
david@debian:~$ echo $HOME  
/home/david
```

Les variables d'environnement

La variable `PATH` contient une liste de répertoires séparés par des deux-points « : »

```
david@debian:~$ echo $PATH  
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/u  
sr/games
```

Ce sont les répertoires dans lesquels le shell cherche la commande qu'on écrit au clavier.

La recherche se fait dans l'ordre des répertoires contenus dans la variable `PATH`.

Les variables d'environnement

Pour la recherche :

On peut agrandir cette liste en n'oubliant pas d'inclure ce qui est déjà spécifié

```
david@debian:~$ PATH=$PATH:/home/david (pour le shell uniquement)
```

```
david@debian:~$ export PATH=$PATH:/home/david  
(pour les sous-processus également)
```

On peut stocker cette configuration en éditant ses fichiers

`$HOME/.bashrc` ou `$HOME/.bash_profile`

Les variables d'environnement

Pour utiliser un même fichier de ressources personnels on peut écrire dans son `.bashrc`

```
if [ -f ~/.bash_profile ]; then  
    . ~/.bash_profile  
fi
```

Les variables d'environnement

Il est possible de mettre dans une variable le résultat d'une commande pour venir réutiliser ce résultat ensuite

```
david@debian:~$ architecture=$(uname -m)
david@debian:~$ echo $architecture
x86_64
```

Les variables d'environnement

\$HOME	chemin du répertoire personnel de l'utilisateur
\$OLDPWD	chemin du répertoire précédent
\$PATH	liste des chemins de recherche des commandes exécutables
\$PPID	PID du processus père du shell
\$PS1	invite principale du shell
\$PS2	invite secondaire du shell
\$PS3	invite de la structure shell "select"
\$PS4	invite de l'option shell de débogage "xtrace"
\$PWD	chemin du répertoire courant
\$RANDOM	nombre entier aléatoire compris entre 0 et 32767
\$REPLY	variable par défaut de la commande "read" et de la structure shell "select"
\$SECONDS	nombre de secondes écoulées depuis le lancement du shell

PAUSE – REFLEXION

Avez-vous des questions ?



Le shell (avancé)

Le shell

Le shell propose différentes fonctionnalités pour aider à la saisie

- Les touches haut et bas pour naviguer dans l'historique (fichier `$HOME/.bash_history`)
- La commande `history`
- La combinaison CTRL-R permet de faire une recherche dans l'historique
- Relancer une commande avec les mêmes arguments (« ! »)

```
david@debian:~$ ls -l
david@debian:~$ cd Documents/
david@debian:~/Documents$ !ls
```

- `~` représente le répertoire personnel de l'utilisateur
- `~user` représente le répertoire personnel de l'utilisateur `user`
- La tabulation pour compléter une commande automatiquement

Le shell

Le shell permet l'utilisation de caractères spéciaux (métacaractères) :

- `*` remplace une chaîne de longueur quelconque
- `?` remplace un seul caractère
- `[]` un caractère quelconque de la liste ou de l'intervalle

```
david@debian:~$ ls [DI]*
david@debian:~$ ls [A-Za-z]*
```

- `[^]` un caractère quelconque sauf ceux de la liste

Le shell

métacaractères : le shell peut lire des caractères de façon littérale, c'est-à-dire protégés.

Par exemple, le caractère « * » perd son statut de métacaractère pour redevenir une « étoile ».

3 moyens pour protéger un caractère ou une chaîne de caractères :

- \ : empêche l'interprétation spéciale d'un métacaractère ;
- ` : les guillemets inversés (backquotes, touches AltGr+7 sur un clavier français) provoquent une interprétation de la chaîne de caractères incluse entre deux de ces caractères comme une commande.
- ' : tous les caractères inclus entre deux de ces caractères sont interprétés comme du texte.

Le shell

Un programme retourne toujours une valeur de retour

Par convention, cette valeur est de 0 si tout s'est bien déroulé

On peut connaître la valeur retour d'une commande via `echo $?`

```
david@debian:~/Documents$ commande_nulle
-bash: commande_nulle : commande introuvable
```

```
david@debian:~/Documents$ echo $?
127
```

```
david@debian:~/Documents$ ls
fichier2.txt  fichier.txt
```

```
david@debian:~/Documents$ echo $?
0
```

La ligne de commande

Une ligne de commande est composée de plusieurs mots séparés par des espaces

- Le premier mot est le nom de la commande
- Les autres mots sont les paramètres de la commande

Pour empêcher cette séparation avec les espaces ou utiliser une chaîne complète, on peut utiliser les guillemets

```
david@debian:~/Documents$ ls "Documents Persos"
```

Pour plusieurs caractères spéciaux (espace par exemple ou un guillemet, on peut utiliser le caractère d'échappement \

```
david@debian:~/Documents$ ls Documents\ Persos
```

```
david@debian:~/Documents$ mkdir rep\"Perso
```

La ligne de commande

Une ligne est considérée comme une commande entière

On peut utiliser le séparateur **inconditionnel** ; pour écrire plusieurs commandes sur une seule ligne – Tout sera exécuté

```
david@debian:~/Documents$ mkdir repertoire; ls repertoire
```

On peut utiliser le séparateur conditionnel &&

- `commande1 && commande2`
- `commande2` ne sera exécutée que si le code retour de `commande1` est 0 (=réussi)

On peut utiliser le séparateur conditionnel ||

- `commande1 || commande2`
- `commande2` ne sera exécutée que si le code retour de `commande1` est différent de 0 (`commande1` ne termine pas correctement)
- `cd $HOME/images || mkdir $HOME/images`

Gestion des processus

Commandes	Opérations	Explications
A &		Crée un nouveau processus en lançant la commande A (dépendant de la console), permettant d'entrer d'autres commandes dans la même console.
(A &)		Même chose que la commande ci-dessus, mais cette fois le processus créé est indépendant de la console.
A && B	<u>ET logique</u> (&&)	Exécute B, si A réussit . B si A OK
A B		Exécute B uniquement si A échoue . B si A KO
A `B`	paramètres dynamiques	A utilise les résultats de l'exécution de B
A \$(B)		

Caractère échappement

Échappement par antislash	Transformation par Bash
\a	Bip
\b	Espacement arrière
\e	Échappement
\f	Saut de page (le nom anglais de ce caractère est <i>form feed</i>)
\n	<u>Saut de ligne</u>
\r	<u>Retour chariot</u>
\t	Caractère de tabulation horizontale
\v	Caractère de tabulation verticale
\\	Anti-slash
\'	Une apostrophe (le nom anglais de ce caractère est <i>quote</i>)
\nnn	Le caractère 8 bits dont la valeur en <u>octal</u> est nnn
\xHH	Le caractère 8 bits dont la valeur en <u>hexadécimal</u> est HH
\cx	Le caractère contrôle-X

Redirection des entrées-sorties

Le shell utilise 3 canaux de communication :

- L'entrée standard stdin pour la lecture des données
- La sortie standard stdout pour la sortie des résultats
- La sortie erreurs stderr

Par défaut ces canaux utilisent l'écran et le clavier

On peut faire une redirection des entrées et des sorties

- < redirige la sortie standard depuis un fichier
- > redirige la sortie standard vers un fichier

```
ls > resultat.txt
```

Attention, le fichier est écrasé sans avertissement

- >> redirige la sortie standard vers un fichier mais concatène ce dernier avec le résultat

Redirection des entrées-sorties

Enfin, on peut utiliser le pipe (« | ») pour rediriger la sortie d'une commande vers l'entrée d'une autre.

- Les tubes (en anglais « pipes », littéralement tuyaux) constituent un mécanisme de communication propre à tous les systèmes UNIX.
- Un tube, symbolisé par une barre verticale (caractère « | »), permet d'affecter la sortie standard d'une commande à l'entrée standard d'une autre, comme un tuyau permettant de faire communiquer l'entrée standard d'une commande avec la sortie standard d'une autre.

Redirection des entrées-sorties

PIPE

- La sortie de la première commande est redirigée vers l'entrée de la deuxième commande
- Exemple

```
ls -l | more
```
- La valeur de retour `$?` de la ligne de commande est celle de la dernière commande (more dans l'exemple précédent)

Redirection des entrées-sorties

Commandes	Opérations	Explications
A > fichier	sortie (>)	Exécute la commande A et redirige sa sortie standard (stdout) dans <i>fichier</i> en écrasant son contenu ou en créant <i>fichier</i> si celui-ci n'existe pas
A >> fichier	sortie (>>)	Exécute la commande A et redirige sa sortie standard à la fin de <i>fichier</i>
A 2> fichier	sortie (2>)	Exécute la commande A et redirige sa sortie standard des erreurs (stderr) dans <i>fichier</i> en écrasant son contenu ou en créant <i>fichier</i> si celui-ci n'existe pas
A 2>> fichier	sortie (2>>)	Exécute la commande A et redirige sa sortie standard des erreurs à la fin de <i>fichier</i>
A 2>&1	sortie (2>&1)	Exécute la commande A et redirige sa sortie standard des erreurs dans sa sortie standard
A < fichier	entrée (<)	Exécute la commande A en lui passant le contenu de <i>fichier</i> dans son entrée standard (stdin)
A B	sortie, entrée()	Exécute A et envoie le contenu de sa sortie standard dans l'entrée standard de B

PAUSE – REFLEXION

Avez-vous des questions ?



Les calculs

Quelques calculs

Plusieurs commandes permettent d'effectuer des calculs arithmétiques

Il est possible de déclarer une variable locale simplement

```
david@debian:~$ n=5
david@debian:~$ m=3
david@debian:~$ let o=6/3
david@debian:~$ echo $o
2
```

La commande `let` permet quant à elle d'effectuer des opérations et d'assigner le résultat

```
david@debian:~$ o=2
david@debian:~$ let o=$o+2
david@debian:~$ echo $o
4
```

Dans certains shell on peut faire simplement

```
david@debian:~$ o=2
david@debian:~$ o=o+2
david@debian:~$ echo $o
4
```

Quelques calculs

La commande `expr` permet d'évaluer une opération (ne pas oublier les espaces autour de l'opérateur) :

```
david@debian:~$ o=2
david@debian:~$ $o + 5
bash: 2 : commande introuvable
david@debian:~$ expr $o + 5
7
```

Les opérateurs disponibles :

- `+`, `-`, `*`, `/`, `%` : Opérations classiques
- `>`, `>=`, `<`, `<=`, `=`, `!=` : Opérations de comparaison
- `&`, `|` : ET / OU
- `~` : NON
- `(` `)` : Regroupement

PAUSE – REFLEXION

Avez-vous des questions ?



La manipulation des fichiers

Les types de fichiers

Le système UNIX présente différents types de fichiers

- Les fichiers physiques enregistrés sur le disque dur. C'est la notion habituelle d'un fichier
- Les répertoires qui sont des fichiers nœud de l'arborescence
 - . Lui-même
 - .. répertoire parent
- Les liens qui permettent de faire une référence vers un autre fichier
 - Lien symbolique
 - Pointeurs virtuels vers des fichiers réels (fichier non supprimé si lien supprimé – le lien peut être cassé si on supprime sa cible)
 - Pointeurs physiques vers des fichiers réels (le fichier est physiquement détruit quand son nombre de référence atteint 0)
- Les fichiers virtuels
 - N'ont pas d'existence réelle (/proc)
- Les fichiers de périphériques
 - Représentent les périphériques (/dev)

Les caractères

Dans les commandes de manipulation de fichier, généralement on peut utiliser les caractères de substitution

- * remplace une chaîne de longueur quelconque
- ? remplace un seul caractère
- [] un caractère quelconque de la liste ou de l'intervalle

```
david@debian:~$ ls [DI]*
```

```
david@debian:~$ ls [A-Za-z]*
```

- [^] un caractère quelconque sauf ceux de la liste

La manipulation des fichiers

La commande pwd

Pour connaître l'emplacement courant, on utilise la commande `pwd`

```
david@debian:/var/log$ pwd  
/var/log
```

La manipulation des fichiers

La commande ls

La commande `ls` permet de lister les fichiers du répertoire passé en paramètre

Sans répertoire passé en paramètre, la commande `ls` liste les fichiers du répertoire courant

Certaines options permettent de personnaliser l'affichage des résultats

La manipulation des fichiers

```
root@debian:/var/log# ls
```

```
alternatives.log  faillog      speech-dispatcher
apt               firebird     syslog
auth.log          fontconfig.log user.log
btmtp             fsck         vboxadd-install.log
cups              gdm3         vboxadd-install-x11.log
daemon.log        hp           VBoxGuestAdditions.log
debug            installer    VBoxGuestAdditions-uninstall.log
dmesg             kern.log     wtmp
dpkg.log          lastlog      Xorg.0.log
exim4             messages     Xorg.0.log.old
```

La manipulation des fichiers

La commande `ls` propose de nombreuses options intéressantes d'affichage ou de tri

Option	Résultat
-l	Affiche les détails de chaque fichier
-a	Affiche tous les fichiers, y compris les fichiers cachés
-R	Affiche une liste récursive du contenu
-t	Trier selon la date de modification

Les options sont combinables entre elles

```
ls -lta
```

Voir la page du man pour toutes les options

La manipulation des fichiers

La commande cd

La commande `cd` permet de changer de répertoire de travail

Par exemple

```
cd /var/www pour accéder au répertoire /var/www
```

Remarques

- `..` Représente le répertoire parent
- `.` Représente le répertoire courant
- `~` Représente le répertoire personnel de l'utilisateur

Ainsi, pour l'utilisateur `patrick`, il pourrait saisir

```
cd ~/comptabilite
```

Pour accéder au répertoire `/home/patrick/comptabilite`

La saisie de la commande `cd` seule permet d'accéder au répertoire de l'utilisateur

La manipulation des fichiers

La commande du

Grâce à la commande `du` (Disk Usage) il est possible d'obtenir des informations sur l'occupation de l'arborescence principale

```
root@debian:/var/log# du
4      ./hp/tmp
8      ./hp
4      ./speech-dispatcher
4      ./gdm3
14512  ./installer/cdebconf
15800  ./installer
8      ./cups
4      ./firebird
548    ./apt
12     ./fsck
8      ./exim4
19048  .
```

La manipulation des fichiers

De même la commande `du` propose de nombreuses options (voir le `man`)

`-d, --max-depth=NIVEAU` pour limiter la profondeur de l'analyse

`-h, --human-readable` afficher les tailles dans un format lisible

```
root@debian:~# du -h -d 1 /var/log
8,0K    /var/log/hp
4,0K    /var/log/speech-dispatcher
4,0K    /var/log/gdm3
16M     /var/log/installer
8,0K    /var/log/cups
4,0K    /var/log/firebird
548K    /var/log/apt
12K     /var/log/fsck
8,0K    /var/log/exim4
19M     /var/log
```

La manipulation des fichiers

La commande `df`

Grâce à la commande `df` (Disk Free) il est possible d'obtenir des informations sur l'occupation d'un disque

```
root@debian:~# df -h
Sys. de fichiers Taille Utilisé Dispo Uti% Monté sur
/dev/sda1          7,7G   3,9G   3,4G   54% /
udev              10M         0    10M    0% /dev
tmpfs             403M   5,9M   397M    2% /run
tmpfs            1006M   216K  1005M    1% /dev/shm
tmpfs             5,0M    4,0K    5,0M    1% /run/lock
tmpfs            1006M         0  1006M    0% /sys/fs/cgroup
/dev/sda6          22G    49M    21G    1% /home
tmpfs            202M    8,0K   202M    1% /run/user/118
tmpfs            202M    12K   202M    1% /run/user/1000
/dev/sr0           57M    57M         0  100% /media/cdrom0
```

La manipulation des fichiers

La commande touch

Il est possible de créer un fichier vide en utilisant la commande `touch`

```
david@debian:~/Documents$ ls
david@debian:~/Documents$ touch fichier.txt
david@debian:~/Documents$ ls -l
total 0
-rw-r--r-- 1 david david 0 févr. 15 10:48 fichier.txt
```

La manipulation des fichiers

La commande mkdir

On utilise la commande `mkdir` (make directory) pour créer un nouveau répertoire dans l'arborescence

```
david@debian:~/Documents$ ls -l
total 0
-rw-r--r-- 1 david david 0 févr. 15 10:48 fichier.txt

david@debian:~/Documents$ mkdir comptabilite

david@debian:~/Documents$ ls -l
total 4
drwxr-xr-x 2 david david 4096 févr. 15 10:49 comptabilite
-rw-r--r-- 1 david david 0 févr. 15 10:48 fichier.txt
```

La manipulation des fichiers

La commande cp

La commande `cp` (copy) permet de copier un fichier ou un répertoire

```
david@debian:~/Documents$ ls
comptabilite  fichier.txt
```

```
david@debian:~/Documents$ cp fichier.txt fichier_copy.txt
```

```
david@debian:~/Documents$ ls
comptabilite  fichier_copy.txt  fichier.txt
```

La manipulation des fichiers

La commande cp

Pour copier un répertoire on utilise l'option `-R` ou `-r`

```
david@debian:~/Documents$ cp comptabilite/ comptabilite2
cp: omission du répertoire « comptabilite/ »
david@debian:~/Documents$ cp -r comptabilite/ comptabilite2
david@debian:~/Documents$ ls
comptabilite  comptabilite2  fichier_copy.txt  fichier.txt
david@debian:~/Documents$ cp fichier_copy.txt comptabilite
david@debian:~/Documents$ ls -R
.:
comptabilite  comptabilite2  fichier_copy.txt  fichier.txt

./comptabilite:
fichier_copy.txt
```

La manipulation des fichiers

Il est possible d'utiliser les jokers *

```
david@debian:~/Documents$ ls -R
.:
comptabilite  fichier2.txt  fichier.txt

./comptabilite:
david@debian:~/Documents$ cp *.txt comptabilite/
david@debian:~/Documents$ ls -R
.:
comptabilite  fichier2.txt  fichier.txt

./comptabilite:
fichier2.txt  fichier.txt
```

La manipulation des fichiers

Dans les commandes de manipulation de fichiers, on peut utiliser les raccourcis `..` (répertoire parent), `.` (répertoire courant) et `~` (répertoire de l'utilisateur)

```
david@debian:~/Documents/comptabilite$ ls
fichier2.txt  fichier.txt
david@debian:~/Documents/comptabilite$ cp *.txt ..
david@debian:~/Documents/comptabilite$ ls ..
comptabilite  fichier2.txt  fichier.txt
```

La manipulation des fichiers

La commande mv

La commande `mv` (move) permet de déplacer un fichier ou un répertoire

```
david@debian:~/Documents$ ls
comptabilite  fichier2.txt  fichier.txt
david@debian:~/Documents$ mv fichier2.txt fichier2.bak
david@debian:~/Documents$ ls
comptabilite  fichier2.bak  fichier.txt
```

De même on peut utiliser les jokers

```
david@debian:~/Documents$ ls
comptabilite  fichier2.txt  fichier.txt
david@debian:~/Documents$ mv *.txt comptabilite/
david@debian:~/Documents$ ls comptabilite/
fichier2.txt  fichier.txt
david@debian:~/Documents$ ls
comptabilite
```

La manipulation des fichiers

La commande rm

La commande `rm` (remove) supprime des fichiers ou des répertoires

```
david@debian:~/Documents/comptabilite$ ls
fichier2.txt  fichier.txt
david@debian:~/Documents/comptabilite$ rm fichier.txt
david@debian:~/Documents/comptabilite$ ls
fichier2.txt
```

Pour supprimer un répertoire on utilise l'option `-r`

```
david@debian:~/Documents$ ls
comptabilite  fichier2.txt  fichier.txt
david@debian:~/Documents$ rm comptabilite/
rm: impossible de supprimer « comptabilite/ »: est un dossier
david@debian:~/Documents$ rm -r comptabilite/
```

La manipulation des fichiers

L'option `-i` permet de rentrer dans un mode interactif

```
david@debian:~/Documents$ ls -l
total 4
drwxr-xr-x 2 david david 4096 févr. 15 11:12 comptabilite
-rw-r--r-- 1 david david 0 févr. 15 11:07 fichier2.txt
-rw-r--r-- 1 david david 0 févr. 15 11:07 fichier.txt
```

```
david@debian:~/Documents$ rm -ri *
rm : supprimer répertoire « comptabilite » ? y
rm : supprimer fichier vide « fichier2.txt » ? n
rm : supprimer fichier vide « fichier.txt » ? N
```

```
david@debian:~/Documents$ ls -l
total 0
-rw-r--r-- 1 david david 0 févr. 15 11:07 fichier2.txt
-rw-r--r-- 1 david david 0 févr. 15 11:07 fichier.txt
```

La manipulation des fichiers

La commande cat

Pour afficher le contenu d'un fichier à l'écran on utilise la commande `cat`

La commande `cat` n'interprète pas le fichier. Elle se contente d'afficher son contenu

```
root@debian:/var/log# cat auth.log
Feb 15 09:32:32 debian systemd-logind[410]: Watching system buttons on /dev/input/event2 (Power Button)
Feb 15 09:32:32 debian systemd-logind[410]: Watching system buttons on /dev/input/event3 (Sleep Button)
Feb 15 09:32:32 debian systemd-logind[410]: Watching system buttons on /dev/input/event4 (Video Bus)
Feb 15 09:32:32 debian systemd-logind[410]: New seat seat0.
Feb 15 09:32:37 debian gdm-launch-environment]: pam_unix(gdm-launch-environment:session): session opened for
user Debian-gdm by (uid=0)
Feb 15 09:32:37 debian systemd-logind[410]: New session c1 of user Debian-gdm.
Feb 15 09:32:37 debian systemd: pam_unix(systemd-user:session): session opened for user Debian-gdm by (uid=0)
Feb 15 09:32:49 debian polkitd(authority=local): Registered Authentication Agent for unix-session:c1 (system
bus name :1.24 [gnome-shell --mode=gdm], object path /org/freedesktop/PolicyKit1/AuthenticationAgent, locale
fr_FR.UTF-8)
Feb 15 09:32:59 debian gdm-password]: pam_unix(gdm-password:session): session opened for user david by
(unknown) (uid=0)
Feb 15 09:32:59 debian systemd: pam_unix(systemd-user:session): session opened for user david by (uid=0)
Feb 15 09:32:59 debian systemd-logind[410]: New session 1 of user david.
...
...
```


La manipulation des fichiers

La commande more

La commande `more` propose la même fonctionnalité que la commande `cat` sauf que le système attendra une interaction de l'utilisateur pour défiler chaque page

Avec le `cat`, le fichier était affiché une fois pour toute en bloc

```
root@debian:/var/log# more auth.log
```

La manipulation des fichiers

La commande head

La commande `head` permet d'afficher par défaut les 10 premières lignes d'un fichier

```
root@debian:/var/log# head auth.log
```

La commande propose plusieurs options (voir le `man`)

- `-n <param>` : spécifier le nombre de lignes à afficher (par défaut 10)
- `-b <param>` : spécifier le nombre d'octets à afficher

La manipulation des fichiers

La commande tail

La commande `tail` permet d'afficher par défaut les 10 dernières lignes d'un fichier

```
root@debian:/var/log# tail auth.log
```

Cette commande est très utilisée pour l'analyse des fichiers de log afin de consulter les derniers événements

La commande propose plusieurs options (voir le `man`)

- n <param> : spécifier le nombre de lignes à afficher (par défaut 10)
- b <param> : spécifier le nombre d'octets à afficher

La recherche de fichier

La commande find

La commande `find` permet de rechercher un fichier dans l'arborescence

```
find repertoire(s) critères option(s)
```

Options disponibles

- -print : affiche le chemin d'accès de chaque fichier
- -name : recherche par nom de fichier
- -type : recherche par type de fichier
- -user : recherche par propriétaire
- -size : recherche par taille
- -atime, -amin : recherche par date de dernier accès (jour, minute)
- -mtime, -mmin : recherche par date de dernière modification
- -ctime, -cmin : recherche par date de création
- -perm : recherche par autorisation d'accès

La recherche de fichier

La commande find

Quelques exemples

```
find . -type f -name "f*"
```

Recherche tous les fichiers dans le répertoire courant qui commencent par f

```
find / -type f -size +10M -mtime -10
```

Recherche tous les fichiers de l'arborescence supérieurs à 10Mo modifiés il y a moins de 10 jours

```
find / -perm 666
```

Recherche tous les fichiers de l'arborescence accessibles à tout le monde

On peut utiliser l'option `-maxdepth 1` pour limiter la recherche au chemin cité dans ses sous-répertoires

Le filtre de fichier (1/5)

La commande grep

grep est un filtre. Il peut trouver un mot dans un fichier, par exemple :

```
grep malloc *.c
```

cherche la chaîne de caractères malloc dans tous les fichiers dont le nom se termine par .c (*.c).

- ✓ Grep : Cette commande signifie : « rechercher **g**lobalement les correspondances avec l'expression rationnelle (en anglais, **regular expression**), et imprimer (**p**rint) les lignes dans lesquelles elle correspond ».

Équivalent de grep sous Windows : la commande findstr
Équivalent de grep sous IOS : la commande include

Le filtre de fichier (2/5)

La commande grep

grep est un filtre. Il peut trouver un mot dans un fichier, par exemple :

```
grep malloc *.c
```

cherche la chaîne de caractères malloc dans tous les fichiers dont le nom se termine par .c (*.c).

On peut insérer une variable dans le critère de recherche :

```
grep "^[^:]*:[^:]*:$1:" /etc/passwd
```

```
grep "^$utilisateur
```

```
grep "^$utilisateur" > /dev/null 2>&1
```

ou utiliser un tube pour filtrer la sortie d'une commande :

```
locate Quick | grep dosemu
```

Le filtre de fichier (3/5)

La commande grep

Caractères spéciaux servant de modèle pour grep (= Regex)

Caractère	Signification
[...]	Plage de caractères permis.
[^...]	Plage de caractères interdits.
^	Début de ligne.
.	Un caractère quelconque, y compris un espace.
*	Caractère de répétition, agit sur le caractère placé avant l'étoile. Accepte également l'absence du caractère placé devant lui.
\$	Fin de ligne.
\{...\}	Répétition.
\{Nombre\}	Répétition de <i>Nombre</i> exactement.
\{Nombre,\}	Répétition de <i>Nombre</i> au minimum.
\{Nombre1 Nombre2\}	Répétition de <i>Nombre1</i> à <i>Nombre2</i> .

Le filtre de fichier (4/5)

La commande grep

Options de la ligne de commande

Option	Signification
-c	Nombre de ligne trouvées (sans les afficher).
-i	Ne fait pas la différence entre majuscule et minuscule.
-n	Affiche le numero de la ligne.
-l	Affiche le nom du fichier contenant la ligne (et pas la ligne).
-v	Affiche toutes les lignes qui ne contiennent pas le mot en question.

Le filtre de fichier (5/5)

La commande grep – Exemples :

[a-z][a-z]*

cherche toutes les lignes contenant au minimum une lettre en minuscule. Le critère avec grep aurait été.

^[0-9]\{3\}\$

cherche toutes les lignes contenant uniquement un nombre à 3 chiffres.

:[0-9]\{2,\}:

cherche toutes les lignes contenant des nombres de minimum 2 chiffres avant les deux points (``:``).

^[0-9]\{1,5\}:

cherche toutes les lignes commençant par des nombres de minimum 1 à 5 chiffres suivis par deux points (``:``).

(une|deux) fois

cherche toutes les lignes contenant la chaîne ``une fois`` ou ``deux fois``.

PAUSE – REFLEXION

Avez-vous des questions ?



Script Shell

Script shell

Programme bash = texte bash dans un fichier texte

- Interprétable par bash au lancement par l'utilisateur
- Modifiable par un éditeur de texte (Ex: emacs, vi, mais pas word !)
- Un programme bash doit être rendu exécutable avec :

`chmod u+x mon_script.sh`

- Par convention, les noms de script sont suffixés par l'extension « .sh »

Exemple : `mon_script.sh`

Script shell

Invocation script bash `mon_script.sh` :

- `./mon_script.sh`
- Avec ses arguments :

`./mon_script.sh arg1 arg2`

Structure script shell

- Première ligne : `#!/bin/bash`
 - `#!` : indique au système que ce fichier est un ensemble de commandes à exécuter par l'interpréteur dont le chemin suit :
Exemple : `/bin/sh`, `/usr/bin/perl`, `/bin/awk`, etc.
 - `/bin/bash` lance bash
- Puis séquence structurée de commandes shell :

```
#!/bin/bash  
  
commande1  
commande2  
...  
mon_script.sh
```

- ✓ Sortie implicite du script à la fin du fichier
- ✓ Sortie explicite avec la commande `exit`

Variables bash

- Déclaration/affectation avec `=` (exemple `ma_var=valeur`)
- Consultation en préfixant du caractère `$` (exemple `$ma_var`)
- Saisie interactive : `read var1 var2 ... varn`
 - Lecture d'une ligne saisie par l'utilisateur (jusqu'au retour chariot)
 - Le premier mot va dans `var1`
 - Le second dans `var2`
 - Tous les mots restants vont dans `varn`
 - Remarque : pas d'espace dans les mots saisis (=séparateur)

Variables bash

```
$ a=42
$ echo $a
42
$ s='Bonjour, monde!!!'
$ echo $s
Bonjour, monde!!!
$ read x
Ceci est une phrase
$ echo $x
Ceci est une phrase
$ read x y
Ceci est une phrase
$ echo $x
Ceci
$ echo $y
est une phrase
```

Le mécanisme qui différencie la variable de son contenu est nommé substitution.

Saisi par l'utilisateur

Saisi par l'utilisateur

Premier mot

Tous les mots qui suivent

Variables bash

- À l'inverse de nombreux langages de programmation, Bash ne regroupe pas ses variables par « type ».
- Essentiellement, les variables bash sont des chaînes de caractères mais, suivant le contexte, Bash autorise des opérations entières et des comparaisons sur ces variables.
- Le facteur décisif étant la seule présence de chiffres dans la variable.

Variables bash

```
c=BB34
echo "c = $c"           # c = BB34
d=${c/BB/23}            # Substitue "23" à "BB".
                        # Ceci fait de $d un entier.

echo "d = $d"           # d = 2334
let "d += 1"            # 2334 + 1 =
echo "d = $d"           # d = 2335
echo

# Et à propos des variables nulles ?
e=""
echo "e = $e"           # e =
let "e += 1"            # Les opérations arithmétiques sont-elles permises sur
                        # une variable nulle ?

echo "e = $e"           # e = 1
echo                    # Variable nulle transformée en entier.

# Et concernant les variables non déclarées ?
echo "f = $f"           # f =
let "f += 1"            # Opérations arithmétiques permises ?
echo "f = $f"           # f = 1
echo                    # Variable non déclarée transformée en entier.
```

Variables bash

• Manipulation de variables simples

var=val ou var="a b"	affectation de la variable "var"
\$var ou \${var}	contenu de la variable "var"
\${#var}	longueur de la variable "var"
export var ou declare -x var	exportation de la variable "var" vers les shells fils
set	affichage de l'ensemble des variables définies dans le shell
unset var	suppression de la variable "var"

• Tableaux

tab[0]=val	affectation du premier enregistrement du tableau "tab"
\${tab[0]} ou \$tab	contenu du premier enregistrement du tableau "tab"
\${tab[11]}	contenu du douzième enregistrement du tableau "tab"
\${tab[*]}	ensemble des enregistrements du tableau "tab"
\${#tab[11]}	longueur du douzième enregistrement du tableau "tab"
\${#tab[*]}	nombre d'enregistrements du tableau "tab"

Visibilité variables

```
#!/bin/bash
# Variables globales et locales à l'intérieur d'une fonction.

fonc ()
{
    local var_local=23      # Déclaré en tant que variable locale.
    echo                    # Utilise la commande intégrée locale.
    echo "\"var_local\" dans la fonction = $var_local"
    var_global=999          # Non déclarée en local.
                           # Retour en global.
    echo "\"var_global\" dans la fonction = $var_global"
}

fonc

echo
echo "\"var_loc\" en dehors de la fonction = $var_loc"
                           # "var_loc" en dehors de la fonction =
                           # Non, $var_local n'est pas visible globalement.
echo "\"var_global\" en dehors de la fonction = $var_global"
                           # "var_global" en dehors de la fonction = 999
                           # $var_global est visible globalement.

echo
```

Au contraire de C, une variable Bash déclarée dans une fonction n'est locale #+ que si elle est déclarée ainsi.

Visibilité variables

```
#!/bin/bash

func ()
{
    var_globale=37          # Visible seulement à l'intérieur du bloc de la fonction
                           #+ avant que la fonction ne soit appelée.
}                           # FIN DE LA FONCTION

echo "var_globale = $var_globale" # var_globale =
                                # La fonction "func" n'a pas encore été appelée,
                                #+ donc $var_globale n'est pas visible ici.

func
echo "var_globale = $var_globale" # var_globale = 37
                                # A été initialisée par l'appel de la fonction.
```

Avant qu'une fonction ne soit appelée, toutes les variables déclarées dans la fonction sont invisibles à l'extérieur du corps de la fonction, et pas seulement celles déclarées explicitement locales.

Schéma algorithmique séquentiel

Suite de commandes les unes après les autres :

- Sur des lignes séparées
- Sur une même ligne en utilisant le caractère point virgule (;) pour séparateur

Schéma algorithmique alternatif

Schéma alternatif simple :

- Si alors ... sinon (si alors ... sinon ...)
- elif et else sont optionnels

```
if cond; then
  cmds
elif cond; then
  cmds
else
  cmds
fi
```

Schéma algorithmique alternatif

Tests sur des valeurs numériques :

- [n1 -eq n2] : vrai si n1 est égal à n2
- [n1 -ne n2] : vrai si n1 est différent de n2
- [n1 -gt n2] : vrai si n1 supérieur strictement à n2
- [n1 -ge n2] : vrai si n1 supérieur ou égal à n2
- [n1 -lt n2] : vrai si n1 inférieur strictement à n2
- [n1 -le n2] : vrai si n1 est inférieur ou égal à n2

Tests sur des chaînes de caractères :

- [mot1 = mot2] : vrai si mot1 est égale à mot2
- [mot1 != mot2] : vrai si mot1 n'est pas égale à mot2
- [-z mot] : vrai si mot est le mot vide
- [-n mot] : vrai si mot n'est pas le mot vide

Schéma algorithmique alternatif

Syntaxe :

- [cond] est un raccourci pour la commande test cond
- test est une commande renvoyant vrai (valeur 0) ou faux (valeur différente de 0) en fonction de l'expression qui suit

```
if [ $x -eq 42 ]; then
    echo coucou
fi
```

Équivaut à

```
if test $x -eq 42; then
    echo coucou
fi
```

Schéma algorithmique alternatif

Exemple :

```
if cond; then
  cmds
elif cond; then
  cmds
else
  cmds
fi
```



```
x=1
y=2
if [ $x -eq $y ]; then
  echo "$x = $y"
elif [ $x -ge $y ]; then
  echo "$x >= $y"
else
  echo "$x < $y"
fi
```

Schéma alternatif multiple (case)

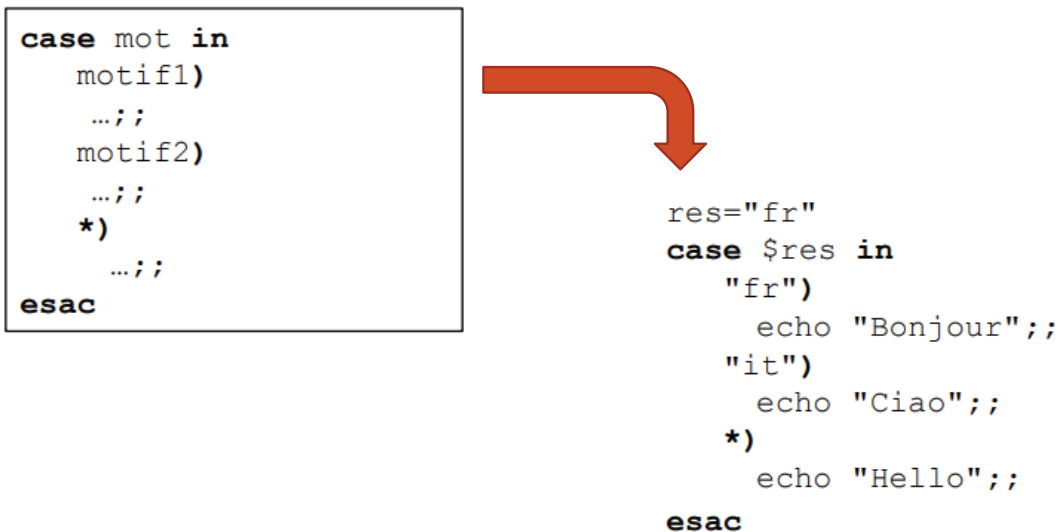
- Si mot vaut motif1 ...
Sinon si mot vaut motif2 ...
Sinon ...

```
case mot in
  motif1)
    ...;;
  motif2)
    ...;;
  *)
    ...;;
esac
```

- Motif : chaîne de caractères pouvant utiliser des méta-caractères
- *) correspond au cas par défaut

Schéma alternatif multiple (case)

Exemple :



```
case mot in
    motif1)
        ...;;
    motif2)
        ...;;
    *)
        ...;;
esac
```

```
res="fr"
case $res in
    "fr")
        echo "Bonjour";;
    "it")
        echo "Ciao";;
    *)
        echo "Hello";;
esac
```

Schéma algorithmique alternatif

Expressions primitives

Primitives	sens
[-a FICHIER]	Vrai si FICHIER existe.
[-b FICHIER]	Vrai si FICHIER existe et est un fichier de type bloc.
[-c FICHIER]	Vrai si FICHIER existe et est un fichier de type caractère.
[-d FICHIER]	Vrai si FICHIER existe et est de type répertoire.
[-e FICHIER]	Vrai si FICHIER existe.
[-f FICHIER]	Vrai si FICHIER existe et est un fichier régulier.
[-g FICHIER]	Vrai si FICHIER existe et son bit SGID est positionné.
[-h FICHIER]	Vrai si FICHIER existe et est un lien symbolique.
[-k FICHIER]	Vrai si FICHIER existe et son bit collant est positionné.
[-p FICHIER]	Vrai si FICHIER existe et est un tube nommé (FIFO).
[-r FICHIER]	Vrai si FICHIER existe et est lisible.
[-s FICHIER]	Vrai si FICHIER existe et a une taille supérieure à zéro.
[-t FD]	Vrai si le descripteur de fichier FD est ouvert et qu'il se réfère à un terminal.
[-u FICHIER]	Vrai si FILE existe et son bit SUID (Set User ID) est positionné.
[-w FICHIER]	Vrai si FICHIER existe et est en écriture.
[-x FICHIER]	Vrai si FICHIER existe et est exécutable.

Schémas itératifs

Boucles while :

- ✓ Tant que ... faire ...
- ✓ Mot clé break pour sortir de la boucle

```
while cond; do
  cmds
done
```



```
x=10
while [ $x -ge 0 ]; do
  read x
  echo $x
done
```

```
CONDITION_FINALE=fin
until [ "$var1" = "$CONDITION_FINALE" ]
# Condition du test ici, en haut de la boucle.
do
  echo "Variable d'entrée #1 "
  echo "($CONDITION_FINALE pour sortir)"
  read var1
  echo "variable #1 = $var1"
done
```

Schémas itératifs (à la C)

Boucles while :

```
LIMITE=10
((a = 1))      # a=1
# Les doubles parenthèses permettent l'utilisation des espaces pour initialiser
#+ une variable, comme en C.

while (( a <= LIMITE )) # Doubles parenthèses, et pas de "$" devant la variable.
do
  echo -n "$a "
  ((a += 1))      # let "a+=1"
  # Oui, en effet.
  # Les doubles parenthèses permettent d'incrémenter une variable avec une
  #+ syntaxe style C.
done
```


Schémas itératifs

Boucles for :

- ✓ Pour chaque ... dans ... faire ...
 - ✓ var correspond à la variable d'itération
 - ✓ liste : ensemble sur lequel var itère

```
for var in liste; do  
  cmds  
done
```



```
for var in 1 2 3 4; do  
  echo $var  
done
```

Schémas itératifs (boucle for à la C)

Boucles for :

```
LIMITE=10  
  
for ((a=1; a <= LIMITE ; a++)) # Double parenthèses, et "LIMITE" sans "$".  
do  
  echo -n "$a "  
done  
# Une construction empruntée à 'ksh93'.
```

```
for ((a=1, b=1; a <= LIMITE ; a++, b++)) # La virgule chaîne les opérations.  
do  
  echo -n "$a-$b "  
done
```

Schémas itératifs

Break, continue :

- La commande break termine la boucle (en sort).
- break peut de façon optionnelle prendre un paramètre. Un simple break termine seulement la boucle interne où elle est incluse mais un break N sortira de N niveaux de boucle.
- Continue fait un saut à la prochaine itération (répétition) de la boucle, oubliant les commandes restantes dans ce cycle particulier de la boucle.
- Un continue N termine toutes les itérations à partir de son niveau de boucle et continue avec l'itération de la boucle N niveaux au-dessus.

Arguments d'une commande

`mon_script.sh arg1 arg2 arg3 arg4 ...`

⇒ chaque mot est stocké dans une variable numérotée

<code>mon_script.sh</code>	<code>arg1</code>	<code>arg2</code>	<code>arg3</code>	<code>arg4</code>	<code>...</code>
<code>"\$0"</code>	<code>"\$1"</code>	<code>"\$2"</code>	<code>"\$3"</code>	<code>"\$4"</code>	<code>...</code>

- `"$0"` : toujours le nom de la commande
- `"$1"` ... `"$9"` : les paramètres de la commande
- `$#` : nombre de paramètres de la commande
- `"$@"` : liste des paramètres : `"arg1" "arg2" "arg3" "arg4" ...`
- `shift` : décale d'un cran la liste des paramètres

Arguments d'une commande

Exemple :

```
#!/bin/bash
for i in "$@"; do
    echo $i
done
```

mon_echo.sh



```
$/mon_echo.sh
$/mon_echo.sh toto titi
toto
titi
$/mon_echo "fin de" la demo
fin de
la
demo
$
```

Imbrication de commandes

Pour récupérer le texte écrit sur le terminal par une commande dans une chaîne de caractères :

- `$(cmd)`
- Attention à ne pas confondre avec `$cmd` qui permet l'accès à la valeur de la variable `cmd`

```
$ date
lundi 27 juillet 2015, 12:47:06 (UTC+0200)
$ echo "Nous sommes le $(date). "
Nous sommes le lundi 27 juillet 2015, 12:47:06
(UTC+0200).
$ echo "Nous sommes le $date."
Nous sommes le .
$
```

Attention, récupère la variable date et non le résultat de la commande date

Variables prépositionnées d'un script

`$@` : tous les paramètres vus comme des mots séparés : "`$@"`" équivaut à "`$1`" "`$2`" ...

`$#` : nombre de paramètres sur la ligne de commande

`$-` : options du shell

`$?` : code de retour de la dernière commande

`$$` : PID du shell

`$!` : PID du dernier processus lancé en arrière-plan

`$_` : dernier argument de la commande précédente

PAUSE – REFLEXION

Avez-vous des questions ?



Programmation avancée

Notion de sous-shell

- Exécuter un script shell lance un nouveau processus, un sous-shell.
- Définition : un sous-shell est un processus lancé par un shell (ou script shell).
- Un sous-shell est une instance séparée du gestionnaire de
- Chaque script shell en cours d'exécution est un sous-processus (processus enfant) du shell parent shell.
- Un script shell peut lancer lui-même des sous-processus. Ces sous-shells permettent au script de faire de l'exécution en parallèle, donc d'exécuter différentes tâches simultanément.

Notion de sous-shell

- Les variables utilisées dans un sous shell ne sont pas visibles en dehors du code du sous-shell.
- Elles ne sont pas utilisables par le processus parent, le shell qui a lancé le sous-shell.
- Elles sont en réalité des variables locales.
- Le changement de répertoire effectué dans un sous-shell n'a pas d'incidence sur le shell parent.
- La commande **exit** termine seulement le sous-shell dans lequel il s'exécute, mais il ne termine pas le shell ou le script parent.
- Une commande externe appelée avec un `exec` ne lance pas (habituellement) un sous-processus / sous-shell.

Notion de sous-shell

Exemple 1 :

```
#!/bin/bash
# subshell-test.sh

(
# Parenthèses à l'intérieur, donc un sous-shell...
while [ 1 ] # Boucle sans fin.
do
    echo "Sous-shell en cours d'exécution..."
done
)

# Le script s'exécutera sans jamais s'arrêter,
#+ au moins tant que vous ne ferez pas Ctrl-C.

exit $? # Fin du script (mais l'exécution ne parviendra jamais jusqu'ici).
```

Notion de sous-shell

Exemple 2 :

```
(cat liste1 liste2 liste3 | sort | uniq > liste123) &
(cat liste4 liste5 liste6 | sort | uniq > liste456) &
# Concatène et trie les 2 groupes de listes simultanément.
# Lancer en arrière-plan assure une exécution en parallèle.
#
# Peut également être écrit :
#   cat liste1 liste2 liste3 | sort | uniq > liste123 &
#   cat liste4 liste5 liste6 | sort | uniq > liste456 &

wait # Ne pas exécuter la commande suivante tant que les sous-shells
      # n'ont pas terminé

diff liste123 liste456
```

Processus fils, parallélisme

- Principe de duplication de processus (fonction de bas niveau UNIX : `fork()`).
- Pour qu'un shell se duplique et se ré-invoque lui-même : faire un appel suivi d'un `&`
- Le paramètre `&0` contient le nom du programme ayant servi à l'invocation initiale. Pour se relancer l'appel `&0 &` suffit.
- Pour passer les paramètres reçus en ligne de commande, on utilisera la syntaxe `$0 "@" &`

Pour lancer un processus (dont un autre shell ou un sous-shell) : rajouter un `&` à la ligne de commande

Processus fils, parallélisme

Exemple :

```
#!/bin/sh

if [ "$MON_PID" != "$PPID" ] ; then
    # Processus Père
    export MON_PID=$$
    echo "PERE : mon PID est $$"
    echo "PERE : je lance un fils"
    $0 &
    sleep 1
    echo "PERE : le PID de mon fils est $!"
    echo "PERE : je me termine"
else
    # Processus FILS
    echo "FILS : mon PID est $$, celui de mon père est $PPID"
    sleep 1
    echo "FILS : je me termine"
fi
```

Processus fils, parallélisme

Exemple amélioré 1 :

Instruction wait :
afin que le père
Attende la fin du fils

```
#!/bin/sh

function compte
{
    local i;
    for i in $(seq 3) ; do
        echo "$1 : $i"
        sleep 1
    done
}

if [ "$MON_PID" = "$PPID" ] ; then
    # Processus FILS
    echo "FILS : mon PID est $$, mon PPID est $PPID"
    compte "FILS"
    echo "FILS : je me termine"
else
    # Processus Père
    export MON_PID=$$
    echo "PERE : mon PID est $$"
    echo "PERE : je lance un fils"
    $0 &
    echo "PERE : mon fils a le PID $!"
    compte "PERE"
    echo "PERE : j'attends la fin de mon fils"
    wait
    echo "PERE : je me termine"
```


Processus fils, parallélisme

Exemple amélioré 2 :

Avec code retour

```
#!/bin/sh

if [ "$MON_PID" = "$PPID" ] ; then
    # Processus FILS
    echo "FILS : mon PID est $$, mon PPID est $PPID"
    echo "FILS : je me termine avec le code 14"
    exit 14
else
    # Processus Père
    export MON_PID=$$
    echo "PERE : mon PID est $$"
    echo "PERE : je lance un fils"
    $0 &
    echo "PERE : mon fils a le PID $!"
    echo "PERE : j'attends la fin de mon fils"
    wait $!
    echo "PERE : mon fils s'est terminé avec le code $?"
fi
```



Signaux avec le shell (1/3)

- L'instruction **trap** permet de capter les signaux et d'exécuter des commandes sur le processus recevant le signal.
 - Cette instruction est une instruction interne au shell utilisé.
 - Exemple : Ex. : trap 'rm /tmp/toto\$\$, exit 1' 2 3
 - Cette commande demande l'interception de l'appui sur les touches CTRL-C et CTRL-D du terminal, d'exécuter rm /tmp/toto\$\$ (\$\$ étant le n° de processus courant) et de sortir (exit 1).
- L'instruction **kill** permet quant-à lui d'envoyer des signaux vers un processus (voir annexe).
 - A l'inverse de l'instruction trap, cette instruction n'est pas une instruction interne au shell.
 - Syntaxe : kill [-s signal | -p] [-a] pid(s)
 - Si vous ne précisez pas le signal, kill enverra par défaut le signal TERM. Tous processus ne capturant pas ce signal seront tués. Pour les processus interceptant le signal TERM, il sera nécessaire d'être plus violent en envoyant la valeur 9.



Signaux avec le shell (2/3)

Exemple 1 :

Dans le script suivant, on vérifie que la chaîne transmise à trap n'est bien évaluée qu'à la réception du signal, en y plaçant une variable dont on modifie le contenu.

Le processus s'envoie lui-même un signal à l'aide du paramètre \$\$ qui contient son propre PID.

RQ : utilitaire nohup pour ignorer le signal SIGHUP et implémenté par un script qui invoque la commande : trap "" HUP

```
#!/bin/sh

function gestionnaire_1
{
    echo " -> SIGINT reçu dans gestionnaire 1"
}

function gestionnaire_2
{
    echo " -> SIGINT reçu dans gestionnaire 2"
}

trap '$GEST' INT

echo "GEST non remplie : envoi de SIGINT"
kill -INT $$

echo "GEST=gestionnaire_1 : envoi de SIGINT"
GEST=gestionnaire_1
kill -INT $$

echo "GEST=gestionnaire_2 : envoi de SIGINT"
GEST=gestionnaire_2
kill -INT $$
```

143



Signaux avec le shell (3/3)

Exemple 2 :

```
USR1_recu=0;
function gestionnaire_USR1
{
    USR1_recu=1;
}
trap gestionnaire_USR1 USR1
```

Et dans le corps du script :

```
# attente du signal
while [ $USR1_recu -eq 0 ] ; do
    sleep 1
done
# le signal est arrivé, on continue...
```

PAUSE – REFLEXION

Avez-vous des questions ?



Debugging

ROBUSTESSE D'UN SHELL-SCRIPT

Debugging

- Pour lutter contre des fautes de frappe, mettre en début de script l'option :

`set -o nounset`

: erreur déclenchée si on fait référence à une variable non définie

- Option `set -v` (ou `set -verbose`) pour afficher les lignes au fur et à mesure de leur exécution
- Option, `set -x` (ou `set -xtrace`) qui indique le suivi pour chaque commande simple

Autre possibilité en ligne de commande : `bash -x <fichier.sh>`

- Possibilité d'utiliser le pseudo-signal `DEBUG` et la variable spéciale `LINENO` qui contient le numéro de la dernière ligne exécutée

Debugging - Exemple

```
#!/bin/bash
_DEBUG="on"
function DEBUG()
{
  [ "$_DEBUG" == "on" ] && $@
}

DEBUG echo 'Reading files'
for i in *
do
  grep 'something' $i > /dev/null
  [ $? -eq 0 ] && echo "Found in $i file"
done
DEBUG set -x
a=2
b=3
c=$(( $a + $b ))
DEBUG set +x
echo "$a + $b = $c"
```

Pour lancer mettre `_DEBUG` à off pour voir la différence

Commande ulimit

- Restreindre la consommation des ressources système par un processus
- Ex 1 : interdire la création d'un fichier core :

```
ulimit -c 0
```

- Ex 2 : restreindre le nombre de processus pour un même utilisateur :

```
ulimit -u 64
```

- Ex 3: restreindre la consommation CPU en limitant le temps d'exécution d'un processus

```
ulimit -t 4
```

Utiliser la commande *times* pour obtenir des stats sur les temps utilisateur et système consommés par un processus et par ses descendants

Commande interne getopts (1/3)

- Permet à un script d'analyser les options passées en argument.
- Chaque appel à la commande *getopts* analyse l'option suivante de la ligne de commande.
- Pour vérifier la validité de chacune des options, il faut appeler *getopts* à partir d'une boucle.
- Pour *getopts*, une option est composée d'un caractère précédé du signe "+" ou "-". Exemple :

```
ls -l *.sh           : une option et un argument ici
```

Commande interne getopt (2/3)

Exemple :

```
while getopt "abcd:e:" option
do
    echo "getopts a trouvé l'option $option"
    case $option in
        a)
            echo "Exécution des commandes de l'option a"
            echo "Indice de la prochaine option à traiter : $OPTARG"
            ;;
        b)
            echo "Exécution des commandes de l'option b"
            echo "Indice de la prochaine option à traiter : $OPTARG"
            ;;
        c)
            echo "Exécution des commandes de l'option c"
            echo "Indice de la prochaine option à traiter : $OPTARG"
            ;;
        d)
            echo "Exécution des commandes de l'option d"
            echo "Liste des arguments à traiter : $OPTARG"
            echo "Indice de la prochaine option à traiter : $OPTARG"
            ;;
        e)
            echo "Exécution des commandes de l'option e"
            echo "Liste des arguments à traiter : $OPTARG"
            echo "Indice de la prochaine option à traiter : $OPTARG"
            ;;
    esac
done
echo "Analyse des options terminée"
exit 0
```

151

Commande interne getopt

Remarques :

- L'appel à la commande getopt récupère l'option suivante et retourne un code vrai tant qu'il reste des options à analyser.
- La liste des options utilisables avec ce script sont définies à la ligne 3 (getopt "abcd:e:" option). Il s'agit des options -a, -b, -c, -d et -e.
- Le caractère ":" inscrit après les options "d" et "e" (getopt "abcd:e:" option) indique que ces options doivent être suivies obligatoirement d'un argument.
- La variable "option" (getopt "abcd:e:" option) permet de récupérer la valeur de l'option en cours de traitement par la boucle while.
- La variable réservée "\$OPTARG" contient l'indice de la prochaine option à traiter.
- La variable réservée "\$OPTARG" contient l'argument associé à l'option.

Commande interne getopts (3/3)

Remarques :

- Lorsque la commande getopts détecte une option invalide, la variable option est initialisée avec la caractère "?" et un message d'erreur est affiché à l'écran. Les options suivantes sont analysées.
- Si le caractère ":" est placé en première position dans la liste des options à traiter (ligne 3), les erreurs sont gérées différemment. En cas d'option invalide :
 - Ex : *while getopts ":abcd:e:" option*
 - getopts n'affichera pas de message d'erreur.
 - la variable OPTARG sera initialisée avec la valeur de l'option incorrecte (ligne 29).

Virgule flottante

- Différence entre Bash et Ksh : pas la possibilité pour Bash d'effectuer des calculs en virgule flottante.
- Solution: appel à la bibliothèque étendue bc (via l'option -l en ligne de commande)

```
#!/bin/sh

X=1.234
Y=5.6789
Z=$( echo "$X * $Y" | bc -l )
echo "Z vaut : $Z"
```

Cron (1/4)

- Programme pour permettre l'exécution de scripts (ou autres) de façon régulière
- Tâches programmées dans un fichier crontab

Commande	Explication
crontab -l	Afficher la liste des actions
crontab -r	Supprimer toutes les actions
crontab -e	Editer les actions

Cron (2/4)

Syntaxe schématique d'un fichier crontab :

```
# Example of job definition:
# .----- minute (0 - 59)
# | .----- hour (0 - 23)
# | | .----- day of month (1 - 31)
# | | | .----- month (1 - 12) OR jan,feb,mar,apr ...
# | | | | .----- day of week (0 - 6) (Sunday=0 or 7) OR sun,mon,tue,wed,thu,fri,sat
# | | | | |
# * * * * * user command to be executed
```

Pour chaque unité, on peut utiliser les notations suivantes :

- 1-5 : les unités de temps de 1 à 5.
- */6 : toutes les 6 unités de temps (toutes les 6 heures par exemple).
- 2,7 : les unités de temps 2 et 7.

Exécution tous les jours à 22h00 d'une commande et rediriger les infos dans sauvegarde.log :

Code BASH :

```
00 22 * * * /root/scripts/sauvegarde.sh >> sauvegarde.log
```

Exécution d'une commande toutes les 6 heures :

Code BASH :

```
00 */6 * * * /root/scripts/synchronisation-ftp.sh
```


Cron (3/4)

Commande pour exécuter backup.sh deux fois par jour (à 2h30 et 14h30) du lundi au vendredi

```
30 2,14 * * 1-5 root /usr/local/sbin/backup.sh
```

Minute	Heure	Jour	Mois	Jour de la semaine	Utilisateur qui exécute la commande	Commande à exécuter
30 0-59	2,14 0-23	* 1-31	* 1-12	1-5 0-7	root	/usr/local/sbin/backup.sh

Mois
Janvier = 1 (...) Décembre = 12

Jour de la semaine
Dimanche = 0 ou 7; lundi = 1; mardi = 2; mercredi = 3; jeudi = 4; vendredi = 5; samedi = 6

<http://jhroy.ca>

- Par défaut, cron envoie un message à l'utilisateur qui a planifié sa tâche.
- Pour ne pas envoyer de façon automatique ce message à l'utilisateur (ou utilisateur@domaine.ext si un agent tel que postfix a été configuré) il faut indiquer sur la première ligne du crontab –e :

MAILTO=""

Cron (4/4)

Raccourcis :

Raccourcis	Description	Équivalent
@reboot	Au démarrage	Aucun
@yearly	Tous les ans	0 0 1 1 *
@annually	Tous les ans	0 0 1 1 *
@monthly	Tous les mois	0 0 1 * *
@weekly	Toutes les semaines	0 0 * * 0
@daily	Tous les jours	0 0 * * *
@midnight	Toutes les nuits	0 0 * * *
@hourly	Toutes les heures	0 * * * *

Journalisation : utiliser l'horodatage

Ex : 30 23 * * * df > /tmp/df_date +\%d_\%m_\%Y_\%H_\%M`.log

PAUSE – REFLEXION

Avez-vous des questions ?



Sed

Sed

- Editeur de fichiers texte non interactif
- Editeur de flot (Stream Editor) est utilisé pour effectuer des transformations simples sur du texte lu depuis un fichier ou un tube (NdT : pipe).
- Le résultat est envoyé sur la sortie standard.
- La syntaxe de la commande sed ne spécifie pas de fichier de sortie, mais les résultats peuvent être mémorisés dans un fichier par le biais de redirection.
- L'éditeur ne modifie pas le flot d'entrée.

Commandes d'édition

Commande	Effet
a\	Ajoute le texte sous la ligne courante.
c\	Remplace le texte de la ligne courante par le nouveau texte.
d	Supprime le texte.
i\	Insère le texte au dessus de la ligne courante.
p	Imprime le texte.
r	Lit un fichier.
s	Cherche et remplace du texte.
w	Ecrit dans un fichier.

Options	Effet
-e SCRIPT	Ajoute les commandes de SCRIPT aux commandes à exécuter sur le flot en entrée.
-f	Ajoute les commandes du fichier SCRIPT-FILE aux commandes à exécuter sur le flot d'entrée.
-n	Mode Silencieux.
-V	Affiche les informations de version et s'arrête.

Sed - Exemple

```
cat -n exemple
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.
```

Q : toutes les lignes contenant le patron ciblé, dans ce cas « errors » (et n'afficher que ces lignes)

```
sed -n '/errors/p' exemple
```

Q : voir les lignes qui ne contiennent pas la chaîne cible

```
sed '/errors/d' exemple
```

Sed - Exemple

Q : supprimer les lignes 2 à 4.

```
sed '2,4d' exemple
```

Q : afficher le fichier à partir d'une certaine ligne jusqu'à la fin

```
sed '3,$d' exemple
```

Q : chercher et remplacer les erreurs au lieu de seulement sélectionner les lignes contenant la cible.

```
sed 's/errors/errors/g' exemple
```

(la commande g pour indiquer à sed qu'il doit traiter la ligne entière plutôt que de stopper à la première occurrence trouvée)

```
sed 's/$/EOL/' exemple
```

Insérer une chaîne à la fin de chaque ligne

```
sed 's/^/> /' exemple
```

Insérer une chaîne au début de chaque ligne du fichier

```
sed -e 's/errors/errors/g' -e 's/last/final/g' exemple
```

Remplacement multiple

Sed - Exemple

Exemple

David ~> cat txt2html.sh
#!/bin/bash

```
# script simple pour convertir du texte en HTML.  
# D'abord, éliminer tous les caractères de saut de ligne, de sorte que l'ajout  
# ne se produise qu'une fois, puis remplacer les sauts de ligne.
```

```
echo "converting $1..."
```

```
SCRIPT="/home/david/scripts/script.sed"  
NAME="$1"  
TEMPFILE="/var/tmp/sed.$PID.tmp"  
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE  
mv $TEMPFILE $NAME
```

```
echo "done."
```

david ~> txt2html.sh test

```
cat script.sed
```

```
1i\  
<html>\<br>  
<head><title>sed generated  
html</title></head>\<br>  
<body bgcolor="#ffffff">\<br>  
<pre>\<br>  
$a\<br>  
</pre>\<br>  
</body>\<br>  
</html>
```

PAUSE – REFLEXION

Avez-vous des questions ?



Awk

Awk (ou GNU Awk)

- La fonction de base de awk est de chercher dans des fichiers des lignes ou portion de texte contenant un ou des patrons.
- Quand on trouve dans la ligne un des patrons, des actions sont effectuées sur cette ligne.
- Quand vous lancez awk, vous spécifiez un programme awk qui indique à awk quoi faire. Le programme consiste en une série de règles. (Il peut aussi contenir des définitions de fonctions, de boucles, des conditions et autres possibilités que nous ignorerons pour l'instant.)
- Chaque règle spécifie un patron à cibler et une action à effectuer sur les cibles trouvées.
- Syntaxe assez proche du C

Awk

- Ligne de commande :

awk PROGRAM inputfile(s)

- Si de multiples changements doivent être fait, peut-être régulièrement sur de multiples fichiers, il est plus facile de mémoriser les commandes awk dans un script

awk -f PROGRAM-FILE inputfile(s)

Ref : <https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-131/AWK-le-langage-script-de-reference-pour-le-traitement-de-fichiers>

Ref 2 : <https://likegeeks.com/awk-command/>

Awk

Exemples :

```
# $1 est le premier champ, $2 est le second champ, etc.

echo un deux | awk '{print $1}'
# un

echo un deux | awk '{print $2}'
# deux

# Mais quel est le champ #0 ($0)?
echo un deux | awk '{print $0}'
# one two
# Tous les champs !
```

```
awk '{print $3}' $nomfichier
# Affiche le champ 3 du fichier $nomfichier sur stdout.

awk '{print $1 $5 $6}' $nomfichier
# Affiche les champs 1, 5 et 6 du fichier $nomfichier.

awk '{print $0}' $nomfichier
# Affiche le fichier entier the entire file!
# Même effet que : cat $nomfichier . . . or . . . sed '' $nomfichier
```

Awk

Variables :

Variable	Description
ARGC	Nombre d'arguments de la ligne de commandes
ARGV	Tableau des arguments de la ligne de commandes
CONVFMT	Format de conversion des nombres en string (chaîne de caractères)
ENVIRON	Tableau associatif des variables d'environnement
FILENAME	Nom du fichier courant (et son chemin si précisé)
FNR	Numéro de l'enregistrement parcouru dans le fichier courant
FS	Séparateur de champs (par défaut les espaces, tabulations et retours-chariots contigus [\t\n]+)
NF	Nombre de champs de l'enregistrement courant
NR	Numéro de l'enregistrement parcouru (tous fichiers confondus)
OFMT	Format de sortie des nombres
OFS	Séparateur de champs en sortie (un espace)
ORS	Séparateur d'enregistrement en sortie (une nouvelle ligne)
RLENGTH	Longueur du string trouvé par la fonction <code>match()</code>
RS	Séparateur d'enregistrement (une nouvelle ligne)
RSTART	Première position du string trouvé par la fonction <code>match()</code>
SUBSEP	Caractère de séparation pour les routines internes des tableaux (\034)

Awk

Particularités :

- ✓ Fonctions sur les chaînes de caractère, fonctions mathématiques, fonctions définies par l'utilisateur.
- ✓ Utilisation des conditions, des boucles, des tableaux et des expressions régulières.
- ✓ Fonction `printf` pour avoir des sorties formatées (%c, %s, ...)
- ✓ Possibilité de faire du Préprocessing (instruction `BEGIN`) et du postprocessing (instruction `END`)

```
$ awk 'BEGIN{
x = 100 * 100
printf "The result is: %e\n", x
}'
```

```
$ awk 'BEGIN {print "The File Contents:"}
{print $0}
END {print "File footer"}' myfile
```


L'éditeur vi

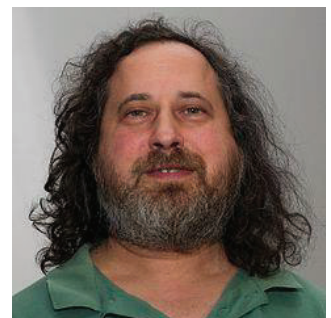
L'édition de documents

Pour éditer des documents au format texte, les systèmes UNIX proposent différents outils d'édition de texte en mode console

Principalement deux acteurs

- emacs

créé dans les années 70 par Richard Stallman
(créateur en 1983 du projet GNU et de la licence GNU)



Richard Stallman

- vi

Créé par Bill Joy en 1976
(cofondateur de Sun Microsystems en 1982)



Bill Joy

L'éditeur vi

L'éditeur vi est un éditeur entièrement en mode texte

Malgré une utilisation très complexe, vi se révèle un éditeur très puissant et bourré de fonctionnalités

Pour lancer vi

```
vi nom_du_fichier
```

A chaque ligne vide de l'écran apparaît au début un ~

2 modes de fonctionnement

- Mode commande : pour saisir des commandes à vi
- Mode insertion : pour insérer du texte dans l'éditeur

L'éditeur vi

Au lancement, vi se place en mode commande

Pour passer en mode insertion

- a pour insérer du texte après le curseur
- i pour insérer du texte avant le curseur
- A pour insérer du texte à la fin de la ligne du curseur

En mode insertion, vi se comporte globalement comme un éditeur classique

Pour passer en mode commande

- Touche échap / escape

L'éditeur vi

Quelques commandes

- `:q` pour quitter l'éditeur
- `:q!` pour quitter sans enregistrer
- `:w` pour enregistrer
- `:wq` pour enregistrer et quitter
- `:nom_fichier` pour enregistrer le fichier sous un autre nom
- `:#` pour placer le curseur à la ligne indiquée
- `/chaîne` et `?chaîne` pour rechercher les occurrences de la chaîne de caractères chaîne après le curseur et avant le curseur respectivement

L'éditeur vi

En mode commande, il est possible d'éditer le texte

- Touches gauche, droite, haut et bas pour naviguer dans le fichier (accompagné des touches page up, page down, line begin, line end)
- `x` pour supprimer le caractère sous le curseur
- `nx` pour effacer n caractères à partir du curseur
- `dd` pour supprimer la ligne du curseur
- `ndd` pour supprimer n lignes à partir de celle du curseur
- `r` pour remplacer le caractère sous le curseur
- `cw` modifie le mot à partir de la position du curseur (passe en mode insertion)
- `cc` modifie la ligne entière (passe en mode insertion)

L'éditeur vi

Pour résumer l'éditeur vi est un éditeur très puissant qui nécessite de la pratique afin d'être bien maîtrisé

Néanmoins, une maîtrise basique de cet outil est indispensable pour prendre en charge des systèmes UNIX

Pour voir toutes ses possibilités

<https://fr.wikipedia.org/wiki/Vi>

PAUSE – REFLEXION

Avez-vous des questions ?

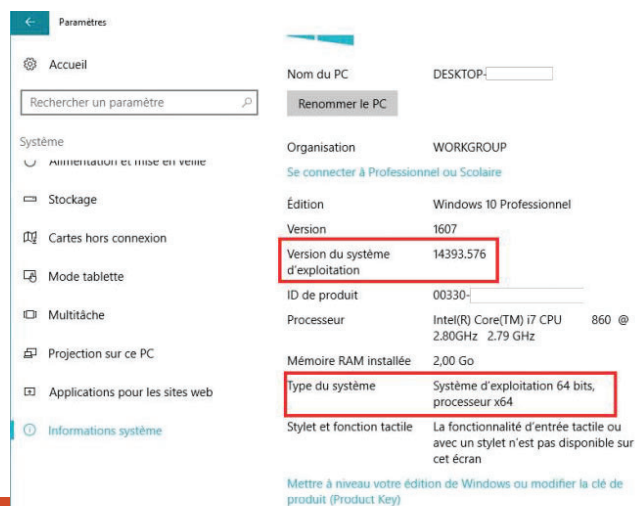


Installation Windows

Bash sous Windows

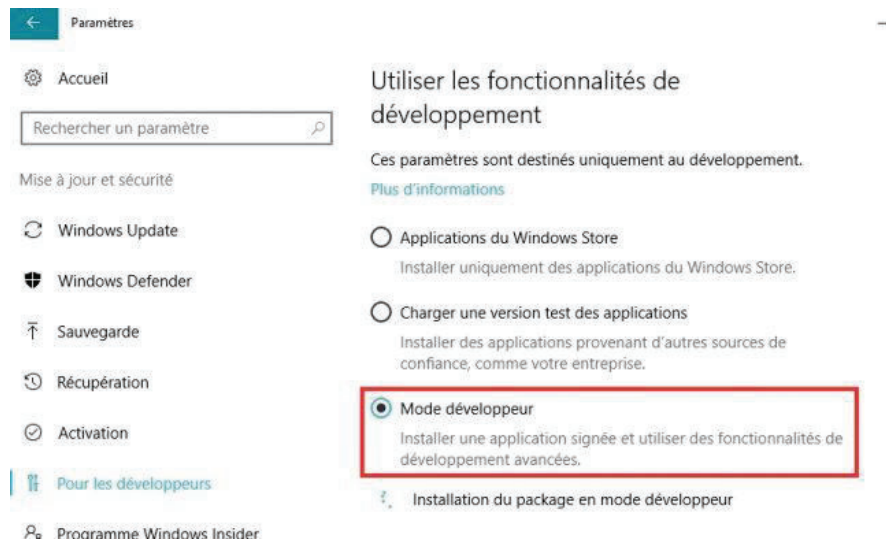
Vérifier que Windows 10 supportera bien Bash : Aller dans les Paramètres -> Système -> Informations Système.

Vous devez avoir une version de Build égale ou supérieure à la 14393 et un Windows 10 en 64 bits.



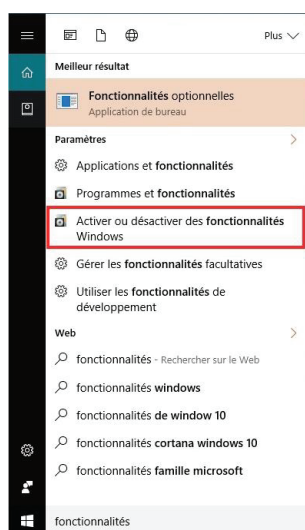
Bash sous Windows

Se rendre dans les Paramètres -> Mise à jour et sécurité et dans le menu « Pour les développeurs », cochez le bouton « Mode développeur ».

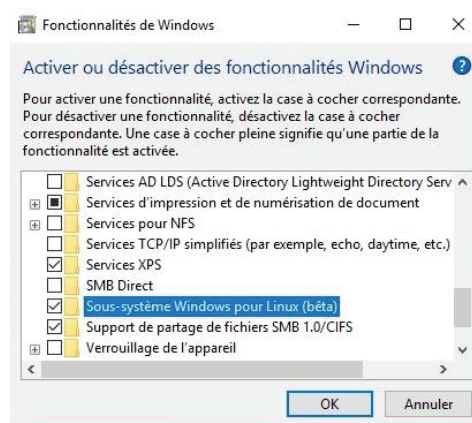


Bash sous Windows

Activer le sous-système Linux de Windows : Pour cela, tapez « fonctionnalités » dans la barre de recherche et cliquez sur « Activer ou désactiver des fonctionnalités Windows ».

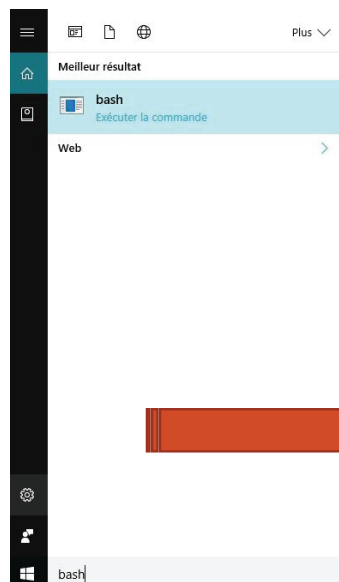


Dans la fenêtre qui s'ouvre, cochez la case « **Sous-système Windows pour Linux** » et faites OK. Votre ordinateur devra ensuite redémarrer.



Bash sous Windows

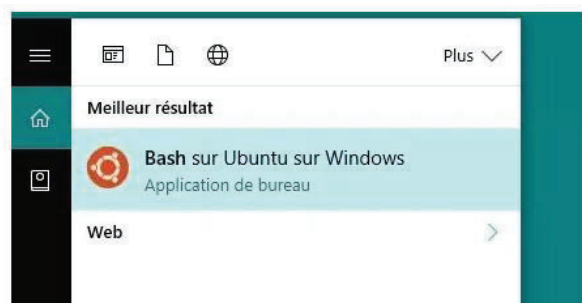
Après le redémarrage, dans la barre de recherches, tapez « **bash** » et lancez-le.



Une fenêtre va s'ouvrir pour demander d'accepter la licence de Canonical (Appuyer sur la touche « o » pour continuer)

Ubuntu se téléchargera depuis le Windows Store (attendre).

Une fois installé, lancer Bash depuis le menu Windows.



Bash sous Windows

- Le C: se trouve dans /mnt/c
- Pour installer des paquets avec la commande « apt-get install ... » (= comme sous Ubuntu)
- Pour les mettre à jour : via « apt-get update » et « apt-get upgrade ».

PAUSE – REFLEXION

Avez-vous des questions ?



Annexes

Substitution et expansion de paramètres

Expression	Signification
<code>\${var}</code>	Valeur de <i>var</i> , identique à <i>\$var</i>
<code>\${var-DEFAULT}</code>	Si <i>var</i> n'est pas initialisé, évalue l'expression <i>\$DEFAULT</i> *
<code>\${var:-DEFAULT}</code>	Si <i>var</i> n'est pas initialisé ou est vide, évalue l'expression <i>\$DEFAULT</i> *
<code>\${var=DEFAULT}</code>	Si <i>var</i> n'est pas initialisé, évalue l'expression <i>\$DEFAULT</i> *
<code>\${var:=DEFAULT}</code>	Si <i>var</i> n'est pas initialisé, évalue l'expression <i>\$DEFAULT</i> *
<code>\${var+AUTRE}</code>	Si <i>var</i> est initialisé, évalue l'expression <i>\$AUTRE</i> , sinon est une chaîne null
<code>\${var:+AUTRE}</code>	Si <i>var</i> est initialisé, évalue l'expression <i>\$AUTRE</i> , sinon est une chaîne null
<code>\${var?ERR_MSG}</code>	Si <i>var</i> n'est pas initialisé, affiche <i>\$ERR_MSG</i> *
<code>\${var:?ERR_MSG}</code>	Si <i>var</i> n'est pas initialisé, affiche <i>\$ERR_MSG</i> *
<code>#!varprefix*</code>	Correspond à toutes les variables déclarées précédemment et commençant par <i>varprefix</i>
<code>#!varprefix@</code>	Correspond à toutes les variables déclarées précédemment et commençant par <i>varprefix</i>

Précédence des opérateurs

Opérateur	Signification	Commentaires
		PLUS HAUTE PRÉCÉDENCE
<code>var++ var--</code>	post-incrément, post-décrément	Opérateurs style C
<code>++var --var</code>	pre-incrément, pre-décrément	
<code>! ~</code>	négation	logique / bit, inverse le sens de l'opérateur qui suit
<code>**</code>	exposant	opération arithmétique
<code>* / %</code>	multiplication, division, modulo	opération arithmétique
<code>+ -</code>	addition, soustraction	opération arithmétique
<code><< >></code>	décalage à gauche et à droite	bit
<code>-z -n</code>	comparaison <i>unitaire</i>	chaîne est/n'est pas <i>null</i>
<code>-e -f -t -x, etc.</code>	comparaison <i>unitaire</i>	<i>fichiers</i>
<code>< -lt > -gt <= -le >= -ge</code>	comparaison <i>composée</i>	string and integer
<code>-nt -ot -ef</code>	comparaison <i>composée</i>	<i>fichiers</i>
<code>== -eq != -ne</code>	égalité / différence	opérateurs de test, chaîne et entier
<code>&</code>	AND	bit
<code>^</code>	XOR	OU <i>exclusif</i> , bit
<code> </code>	OU	bit
<code>&& -a</code>	ET	logique, comparaison <i>composée</i>
<code> -o</code>	OR	logique, comparaison <i>composée</i>
<code>?:</code>	opérateur à trois arguments	C-style
<code>=</code>	affectation	(ne pas confondre avec les tests d' <i>égalité</i>)
<code>*= /= %= += -= <<= >>= &= !=</code>	combinaison d'affectation	égal + multiplication, égal + division, égal + modulo, etc.
<code>,</code>	virgule	lie un ensemble d'opérations
		PLUS BASSE PRÉCÉDENCE

Opérations chaînes de caractère

Expression	Signification
<code>\${#chaine}</code>	Longueur de <i>\$chaine</i>
<code>\${chaine:position}</code>	Extrait la sous-chaîne à partir de <i>\$chaine</i> jusqu'à <i>\$position</i>
<code>\${chaine:position:longueur}</code>	Extrait <i>\$longueur</i> caractères dans la sous-chaîne à partir de <i>\$chaine</i> jusqu'à <i>\$position</i>
<code>\${chaine#sous-chaine}</code>	Supprime la plus petite correspondance de <i>\$sous-chaine</i> à partir du début de <i>\$chaine</i>
<code>\${chaine##sous-chaine}</code>	Supprime la plus grande correspondance de <i>\$sous-chaine</i> à partir du début de <i>\$chaine</i>
<code>\${chaine%sous-chaine}</code>	Supprime la plus courte correspondance de <i>\$sous-chaine</i> à partir de la fin de <i>\$chaine</i>
<code>\${chaine%%sous-chaine}</code>	Supprime la plus longue correspondance de <i>\$sous-chaine</i> à partir de la fin de <i>\$chaine</i>
<code>\${chaine/sous-chaine/remplacement}</code>	Remplace la première correspondance de <i>\$sous-chaine</i> avec <i>\$remplacement</i>
<code>\${chaine//sous-chaine/remplacement}</code>	Remplace <i>toutes</i> les correspondances de <i>\$sous-chaine</i> avec <i>\$remplacement</i>
<code>\${chaine/#sous-chaine/remplacement}</code>	Si <i>\$sous-chaine</i> correspond au <i>début</i> de <i>\$chaine</i> , substitue <i>\$remplacement</i> par <i>\$sous-chaine</i>
<code>\${chaine/%sous-chaine/remplacement}</code>	Si <i>\$sous-chaine</i> correspond à la <i>fin</i> de <i>\$chaine</i> , substitue <i>\$remplacement</i> par <i>\$sous-chaine</i>

Opérations chaînes de caractère

Expression	Signification
<code>expr match "\$chaine" '\$sous-chaine'</code>	Longueur de <i>\$sous-chaine*</i> correspondant au début de <i>\$chaine</i>
<code>expr "\$chaine" : '\$sous-chaine'</code>	Longueur de <i>\$sous-chaine*</i> correspondant au début de <i>\$chaine</i>
<code>expr index "\$chaine" \$sous-chaine</code>	Position numérique dans <i>\$chaine</i> du premier caractère correspondant dans <i>\$sous-chaine</i>
<code>expr substr \$chaine \$position \$longueur</code>	Extrait <i>\$longueur</i> caractères à partir de <i>\$chaine</i> commençant à <i>\$position</i>
<code>expr match "\$chaine" '\(\$sous-chaine\)'</code>	Extrait <i>\$sous-chaine*</i> au début de <i>\$chaine</i>
<code>expr "\$chaine" : '\(\$sous-chaine\)'</code>	Extrait <i>\$sous-chaine*</i> au début de <i>\$chaine</i>
<code>expr match "\$chaine" '.*\(\$sous-chaine\)'</code>	Extrait <i>\$sous-chaine*</i> à la fin de <i>\$chaine</i>
<code>expr "\$chaine" : '.*\(\$sous-chaine\)'</code>	Extrait <i>\$sous-chaine*</i> à la fin de <i>\$chaine</i>

Commandes DOS et équivalent UNIX

Commande DOS	Équivalent UNIX	Effet
ASSIGN	ln	lie un fichier ou un répertoire
ATTRIB	chmod	change les droits d'un fichier
CD	cd	change de répertoire
CHDIR	cd	change de répertoire
CLS	clear	efface l'écran
COMP	diff, comm, cmp	compare des fichiers
COPY	cp	copie des fichiers
Ctl-C	Ctl-C	break (signal)
Ctl-Z	Ctl-D	EOF (end-of-file, fin de fichier)
DEL	rm	supprime le(s) fichier(s)
DELTREE	rm -rf	supprime le répertoire récursivement
DIR	ls -l	affiche le contenu du répertoire
ERASE	rm	supprime le(s) fichier(s)
EXIT	exit	sort du processus courant
FC	comm, cmp	compare des fichiers
FIND	grep	cherche des chaînes de caractères dans des fichiers
MD	mkdir	crée un répertoire
MKDIR	mkdir	crée un répertoire
MORE	more	affiche un fichier page par page
MOVE	mv	déplace le(s) fichier(s)
PATH	\$PATH	chemin vers les exécutables
REN	mv	renomme (ou déplace)
RENAME	mv	renomme (ou déplace)
RD	rmdir	supprime un répertoire
RMDIR	rmdir	supprime un répertoire
SORT	sort	trie un fichier
TIME	date	affiche l'heure système
TYPE	cat	envoie le fichier vers stdout
XCOPY	cp	copie de fichier (étendue)

Signaux gérés par le système

Si le signal est ignoré, provoque un core image.

core: le processus se termine et le contexte est placé dans le fichier core.

ignoré: le processus ne se termine pas.

fin: le processus se termine.

suspension: le processus est suspendu.

Si le signal SIGCLD (mort d'un fils) est intercepté le processus fils ne laissera pas un Zombie. Le signal SIGKILL ne peut être intercepté et provoque toujours la terminaison du processus.

N°	Rem	Nom	Désignation	Effet si signal ignoré
1		SIGHUP	fin de session	fin
2		SIGINT	int: <intr> clavier Ctrl C	fin
3	*	SIGQUIT	int: <quit> clavier Ctrl D	core
4	*	SIGILL	instruction illégale	core
5	*	SIGTRAP	trace ou point d'arrêt	core
6	*	SIGIOT/SIGABRT	instruction IOT ou abort	core
7	*	SIGEMT	émulation trap	core
8	*	SIGFPE	erreur virgule flottante	core
9		SIGKILL	mort du process (non traitable)	fin
10	*	SIGBUS	erreur de bus	core
11	*	SIGSEGV	violation mémoire	core
12	*	SYGSYS	erreur appel système	core
13		SIGPIPE	écriture dans tube sans lecteur	fin
14		SIGALRM	alarme de l'horloge	fin
15		SIGTERM	terminaison du processus	fin
16	+	SIGUSR1	signal 1 pour utilisateurs	fin
17	+	SIGUSR2	signal 2 pour utilisateurs	fin
18	+	SIGCLD/SIGCHLD	mort d'un fils	ignoré
19	+	SIGPWR	coupure/redémarrage	ignoré
20	+	SIGWINCH	changement de taille fenêtre	ignoré
21	+	SIGURG	info. urgentes sur socket	ignoré
22	+	SIGPOLL	occurrence d'événements scruté	fin
23	+	SIGSTOP	demande de suspension	suspension
24		SIGTSTP	demande suspension du terminal	suspension
25		SIGCONT	demande de reprise du processus	ignoré
26		SIGTTIN	lecture terminal en background	suspension
27		SIGTTOU	écriture au terminal en background	suspension
28	+	SIGVTALRM	horloge virtuelle	fin
29		SIGPROF	horloge	fin
30		SIGXCPU	temps cpu maximal écoulé	core
31		SIGXFSZ	taille maximale de fichier atteinte	core

Liens utiles

➤ <https://abs.traduc.org/abs-5.1-fr/index.html>



FIN
