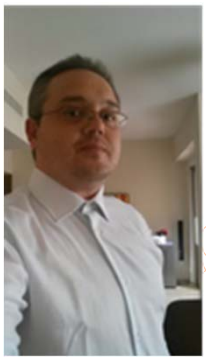
BASH
Scripting Shell

Elies Jebri

Décembre 2018

Version 4.1

Votre formateur…



Elies Jebri

- **Titre:** Consultant en OSS.
- **Certification:**

RHCE CCNA
DCTS NCLA MCTS
LPIC-1 LPIC-2
LPIC-301 LPIC-303
LPIC-304
TSPM

- **Contact:**
elies.jebri@opt2open.com

Sommaire

1. LES INTERPRÉTEURS DE COMMANDES COURANTES (SHELL)

Les fonctions du Shell en général

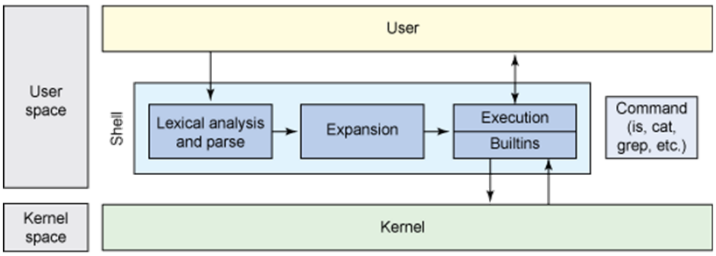
- Le Shell UNIX interprète les commandes de l'utilisateur, qui sont soit directement entrées par celui-ci, ou lues depuis un fichier appelé un script shell.
- Ces scripts sont interprétés, donc non compilés.
- Permet de passer des commandes au noyau,
- Permet de mettre en place un environnement utilisateur qui peut être configuré individuellement par le biais de fichiers de configuration.

Architecture du Shell

L'architecture de base ressemble à un pipeline dans lequel les entrées sont analysées et parcourues,

Les symboles sont développés (à l'aide de diverses méthodes telles que l'accolade, le tilde, le développement et la substitution de variables et de paramètres et la génération de noms de fichiers),

Et enfin les commandes sont exécutées (utilisant des commandes intégrées au shell ou des commandes externes).



Familles de Shell

Le système UNIX généralement offre une variété de types de Shell :

- **sh** ou Bourne Shell : le Shell originel toujours en vigueur sur les systèmes UNIX. C'est le Shell de base, avec peu de possibilités.
- **bash** ou Bourne Again shell : le Shell standard GNU Linux, intuitif et souple. Réputé être une extension du Bourne Shell par l'ajout d'extensions.
- **csh** ou C shell : la syntaxe de ce Shell ressemble à celle du langage de programmation C. Parfois demandée par les programmeurs.
- **tcsh** ou TENEX C Shell : un surensemble du Shell C, convivial et rapide. Certains l'appellent aussi le Turbo Shell C.
- **ksh** ou le Korn shell : quelques fois apprécié des gens venant du monde UNIX. Un sur-ensemble du Bourne Shell ; avec une configuration standard.

Le fichier /etc/shells donne un aperçu des Shells connus du système Linux :

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

Votre Shell par défaut est déclaré dans le fichier /etc/passwd , comme cette ligne pour l'utilisateur mia :

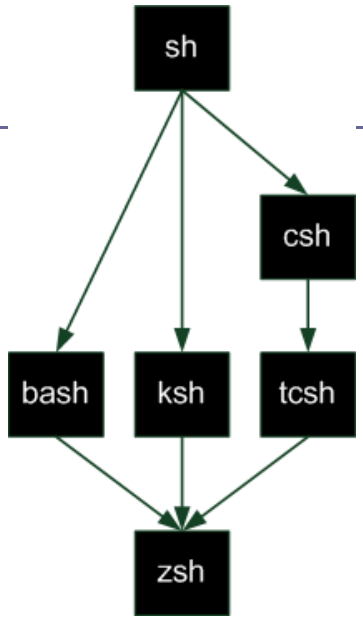
```
mia:L2NOfqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

Pour permuter d'un Shell à un autre, simplement entrez le nom du nouveau Shell actif dans le terminal. Le système trouve le répertoire où le nom apparaît au moyen des paramètres de PATH, et puisqu'un Shell est un fichier exécutable (programme), le Shell courant l'active et il s'exécute. Une nouvelle invite est souvent affichée, du fait que chaque Shell a une interface propre :

```
mia:~> tcsh
[mia@post21 ~]$
```

sh

- L'ancêtre de tous les shells est le sh (*Bourne Shell*).
- C'est le plus vieux et il est installé sur tous les OS basés sur Unix.
- Pauvre en fonctionnalités par rapport aux autres shells.
- Le bash (*Bourne Again Shell*) est le shell par défaut de la plupart des distributions Linux mais aussi celui du terminal de Mac OS X.
- C'est celui que vous utilisez en ce moment sous Linux.
- Le bash est une amélioration du sh.

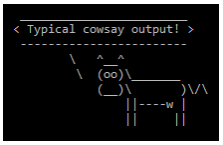


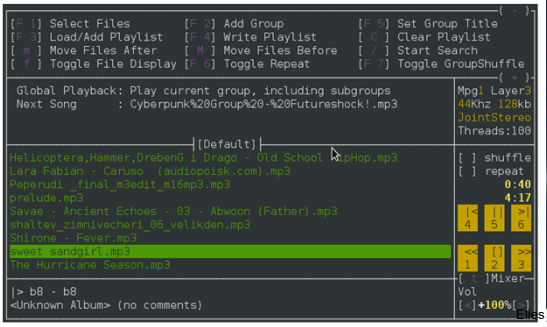
Un environnement de travail

Un Shell doit fournir un environnement de travail agréable et puissant. Par exemple, **bash** permet (entre autres) :

- le rappel des commandes précédentes (gestion de l'historique) ; cela évite de taper plusieurs fois la même commande
- la modification en ligne du texte de la commande courante (ajout, retrait, remplacement de caractères) en utilisant les commandes d'édition de l'éditeur de texte **vi** ou **emacs**
- la gestion des travaux lancés en arrière-plan (appelés *jobs*) ; ceux-ci peuvent être démarrés, stoppés ou repris suivant les besoins
- l'initialisation adéquate de variables de configuration (chaîne d'appel de l'interpréteur, chemins de recherche par défaut) ou la création de raccourcis de commandes (commande interne **alias**).

Rendre son environnement agréable





Cowsay
MP3Blaster

Un langage de programmation

Les Shells sont également de véritables langages de programmation. Un Shell comme **bash** intègre :

- les notions de *variable*, *d'opérateur arithmétique*, de *structure de contrôle*, de *fonction*, présentes dans tout langage de programmation classique, mais aussi
- des opérateurs spécifiques (ex : **|**, **&**)


```
$ a=5  => affectation de la valeur 5 à la variable a
$
$ echo $((a +3 ))  => affiche la valeur de l'expression a+3
8
$
```

L'opérateur |, appelé tube, est un opérateur caractéristique des shells et connecte la sortie d'une commande à l'entrée de la commande suivante.

```
$ lpstat -a
HP-LaserJet-2420 acceptant des requêtes depuis jeu. 14 mars 2013 10:38:37 CET
HP-LaserJet-P3005 acceptant des requêtes depuis jeu. 14 mars 2013 10:39:54 CET
$
$ lpstat -a | wc -l
2
$
```

Atouts des shells

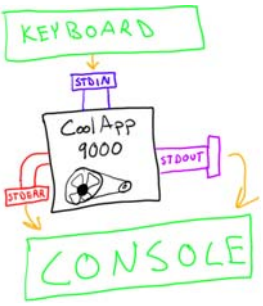
L'étude d'un shell tel que **bash** en tant que langage de programmation possède plusieurs avantages :

- c'est un langage interprété : les erreurs peuvent être facilement localisées et traitées ; d'autre part, des modifications de fonctionnalités sont facilement apportées à l'application sans qu'il soit nécessaire de recompiler et faire l'édition de liens de l'ensemble
- le shell manipule essentiellement des chaînes de caractères : on ne peut donc pas construire des structures de données complexes à l'aide de pointeurs, ces derniers n'existant pas en Shell. Ceci a pour avantage d'éviter des erreurs de typage et de pointeurs mal gérés. Le développeur raisonne de manière uniforme en termes de chaînes de caractères
- le langage est adapté au prototypage rapide d'applications : les tubes, les substitutions de commandes et de variables favorisent la construction d'une application par assemblage de commandes préexistantes dans l'environnement Unix

Atouts des shells

C'est un langage « glu » : il permet de connecter des composants écrits dans des langages différents. Ils doivent uniquement respecter quelques règles particulièrement simples. Le composant doit être capable :

- de lire sur l'entrée standard,
- d'accepter des arguments et options éventuels,
- d'écrire ses résultats sur la sortie standard,
- d'écrire les messages d'erreur sur la sortie standard dédiée aux messages d'erreur.



GLU : Programme permettant de faire fonctionner ensemble d'autres programmes.

Perl est un exemple typique de « langage de glu » (mais il sait faire des tas d'autres choses).

Inconvénients des shells

Cependant, **bash** et les autres shells en général ne possèdent pas que des avantages :

- issus d'Unix, système d'exploitation écrit à l'origine par des développeurs pour des développeurs, les shells utilisent une syntaxe « ésotérique » d'accès difficile pour le débutant
- suivant le contexte, l'oubli ou l'ajout d'un caractère **espace** provoque facilement une erreur de syntaxe
- **bash** possède plusieurs syntaxes pour implanter la même fonctionnalité, comme la substitution de commande ou l'écriture d'une chaîne à l'écran. Cela est principalement dû à la volonté de fournir une compatibilité ascendante avec le *Bourne shell*, shell historique des systèmes Unix
- certains caractères spéciaux, comme les parenthèses, ont des significations différentes suivant le contexte ;
  - en effet, les parenthèses peuvent introduire une liste de commandes, une définition de fonction ou bien imposer un ordre d'évaluation d'une expression arithmétique.

Shell utilisé

- La manière la plus simple de connaître le shell que l'on utilise est d'exécuter la commande **ps** qui liste les processus de l'utilisateur :

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

- Ou aussi

```
$ echo $SHELL
/bin/bash
```

- Comme il existe plusieurs versions de bash présentant des caractéristiques différentes, il est important de connaître la version utilisée. Pour cela, on utilise l'option --version de bash.

```
$ bash --version
GNU bash, version 4.2.24(1)-release (i686-pc-linux-gnu)
Copyright (C) 2011 Free Software Foundation, Inc.
Licence GPLv3+ : GNU GPL version 3 ou ultérieure <http://gnu.org/licen ses/gpl.html>
    ...
```

Syntaxe d'une commande

- La syntaxe générale d'une commande (unix ou de bash) est la suivante :

```
[chemin/]$nom_cmd [ option ... ] [ argument ... ]
```

- C'est une suite de mots séparés par un ou plusieurs blancs. On appelle blanc un caractère tab (tabulation horizontale) ou un caractère espace.
- Un mot est une suite de caractères non blancs. Cependant, plusieurs caractères ont une signification spéciale pour le shell et provoquent la fin d'un mot : ils sont appelés méta-caractères (ex : |, <).

Mots clés de Bash

Bash utilise également des opérateurs

(ex : **(**, ||) et des *mots réservés* :

- !
- case
- do
- done
- elif
- else
- esac
- fi
- for
- function
- if
- in
- select
- then
- time
- until
- while
- {
- }
- [[
-]]

Syntaxe d'une commande

- Bash distingue les caractères majuscules des caractères minuscules.
- Le *nom* de la commande est le plus souvent le premier mot.
- Une *option* est généralement introduite par le caractère **tiret** (ex : **-a**) ou dans la syntaxe GNU par deux caractères **tiret** consécutifs (ex : **--version**). Elle précise un fonctionnement particulier de la commande.
- La syntaxe *[elt ...]* signifie que l'élément *elt* est facultatif (introduit par la syntaxe *[elt]*) ou bien présent une ou plusieurs fois (syntaxe *elt ...*). Cette syntaxe ne fait pas partie du shell ; elle est uniquement descriptive (méta-syntaxe).
- Les *arguments* désignent les objets sur lesquels doit s'exécuter la commande.

Ex : ls -l RepC RepD : commande ls avec l'option l et les arguments RepC et RepD

Lorsque l'on souhaite connaître la syntaxe ou les fonctionnalités d'une commande cmd (ex : ls ou bash) il suffit d'exécuter la commande man cmd (ex : man bash). L'aide en ligne de la commande cmd devient alors disponible.

L'exécution de commandes

- Bash détermine le type de programme qui doit être exécuté.
- Bash segmente la ligne en mots (tokens), le premier mot est généralement une commande :
  1. Alias ou fonction
  2. commande interne (enable)
  3. binaire (variable PATH, which)
- Les autres arguments sont des options ou paramètres de la commande.

Les catégories de commandes

Dans le shell bash, on peut distinguer 3 catégories de commandes :

- Les commandes définies par une fonction shell ou par un alias (par exemple définies dans .bashrc).
- Les commandes primitives (builtin en anglais) du bash qu'on peut appeler aussi commandes internes. Le code de ces commandes est encapsulé directement dans l'exécutable /bin/bash et un appel d'une telle commande ne crée pas de processus fils du shell courant.
- Les commandes externes qui sont des programmes exécutables stockés dans l'un des répertoires du PATH (par exemple find qui se trouve dans le répertoire /usr/bin/). Lors de l'appel de ces commandes, le shell courant crée un processus fils dans lequel la commande en question sera exécutée.

Alias

- Un alias permet de substituer un mot à une chaîne de caractère quand il est utilisé comme premier mot d'une commande simple.
- Le Shell maintient une liste d'alias qui sont déclarés ou invalidés avec les commandes intégrées **alias** et **unalias** .
- Saisir **alias** sans options pour afficher une liste des alias connus du Shell courant.
- Les alias sont utiles pour spécifier une version par défaut d'une commande qui existe en plusieurs versions sur le système,
- Pour spécifier les options par défaut d'une commande.
- Permettre la correction des fautes de frappes.

Commandes internes

- Commandes dont le code est implanté au sein de l'interpréteur de commande.
- Cela signifie que, 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.
- Exemples de commandes internes : **cd** , **echo** , **for** , **pwd**
- Sauf dans quelques cas particuliers, l'interpréteur ne crée pas de processus pour exécuter une commande interne.

Les commandes internes de **bash** peuvent se décomposer en deux groupes :

- les commandes simples (ex : **cd**, **echo**)
- les commandes composées (ex : **for**, **((**, **{**).

Commandes simples

- Parmi l'ensemble des commandes internes, echo est l'une des plus utilisée :
- echo :
- Cette commande interne affiche ses arguments sur la sortie standard en les séparant par un espace et va à la ligne.

```
$ echo bonjour     tout  le                    monde
bonjour tout le monde
$
```

- Dans cet exemple, echo est invoquée avec quatre arguments : bonjour, tout, le et monde.

Commandes composées

- Les commandes composées de bash sont :
  - case ... esac
  - if ... fi
  - for ... done
  - select ... done
  - until ... done
  - while ... done
  - [[...]]
  - (...)
  - { ... }
  - ((...))
- Seuls le premier et le dernier mot de la commande composée sont indiqués. Les commandes composées sont principalement des structures de contrôle.

Les fonctions

- Les fonctions Shell sont un moyen de grouper des commandes pour une exécution ultérieure par l'appel d'un nom pour le groupe.
- Elles sont exécutées tout comme des commandes « régulières ».
- Quand le nom de la fonction Shell est employé comme le nom d'une commande simple, la liste des commandes associées à cette fonction est exécutée.
- Les fonctions Shell sont exécutées dans le contexte en cours du Shell, elles ne sont pas interprétées dans un nouveau processus.

Appel de alias, cmd interne et fonction

Les fonctions, les alias et les commandes internes s'appellent directement par leur nom. Les commandes externes, quant à elles, peuvent s'appeler par leur nom ou par leur chemin absolu :

```
1  ~$ mon_alias arg1 arg2
2  ~$ commande_interne arg1 arg2
3  ~$ commande_externe arg1 arg2
4  ~$ /usr/bin/commande_externe arg1 arg2
```


- Avec un chemin absolu (ligne 4 dans l'exemple ci-dessus), il n'y a pas d'ambiguïté : on sait exactement que la commande appelée est une commande externe et on sait aussi où elle se trouve exactement dans le système de fichiers.

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.
- La localisation du code d'une commande externe doit être connue du shell pour qu'il puisse exécuter cette commande.
- A cette fin, bash utilise la valeur de sa variable prédéfinie PATH (chemin de recherche des fichiers exécutables).
- Celle-ci contient une liste de chemins séparés par le caractère : (ex : /bin:/usr/bin).
- **Remarque :** pour connaître le statut d'une commande, on utilise la commande interne **type**.

Résolution d'appel de commande

```
~$ ma_commande arg1 arg2
```

Dans ce cas, la résolution se fait en plusieurs étapes, dans cet ordre :

1. le shell courant regarde dans la liste des alias, puis dans la liste des mots clé si ma_commande existe, s'il ne trouve rien il passe à l'étape suivante;
2. le shell courant regarde dans la liste des fonctions shell si ma_commande existe, s'il ne trouve rien il passe à l'étape suivante;
3. le shell courant regarde dans la liste des commandes internes si ma_commande existe, s'il ne trouve rien il passe à l'étape suivante;
4. le shell courant regarde dans le PATH si ma_commande correspond à un programme, s'il ne trouve rien il passe à l'étape suivante;
5. le shell provoque une erreur du genre "ma_commande: commande introuvable".

https://www.cyberciti.biz/tips/an-example-how-shell-understand-which-program-to-run-part-ii.html

Les tables HASH et PATH ne sont pas la première méthode de localisation de votre programme ou de fichiers exécutables sur des systèmes Linux ou Unix. Votre programme peut être une fonction shell, une commande intégrée ou un alias. Voici la séquence complète adoptée par le shell BASH pour exécuter votre commande:
Avant qu'une commande soit exécutée, REDIRECTION est effectuée. Puis séquence suivante utilisée par SHELL
ALIASES
Expansion des paramètres, substitution de commande, expansion arithmétique et suppression des devis avant d'être affectée à la variable
Shell Fonction
Commande BUILTIN
Tables HASH
Variable PATH
Si tout échoue, vous voyez un message d'erreur de commande introuvable.
Exemples

Essayez les exemples simples suivants pour comprendre la séquence SHELL.
a) Créez votre propre répertoire bin et ajoutez-le à un PATH:
$ mkdir ~/bin
$ export PATH=$PATH:~/bin
b) Créez une fonction shell appelée hello:
$ function hello() { echo "Hello from function()" ; }
c) Créez un alias appelé hello:
$ alias hello='echo Hello from alias'
d) Créez un script shell appelé hello dans le répertoire / home / you / bin (qui est ajouté à votre PATH)
$ echo 'echo Hello from script' > ~/bin/hello
$ chmod + x ~ / bin / hello
e) Laissez-nous le tester:
vous avez maintenant trois commandes différentes portant le même nom (c.-à-d. bonjour). Lequel va exécuter en premier? Tapez les commandes suivantes à l'invite de commande: Exemples de sorties:
$ type hello
$ hello

Bonjour de aliasf) Supprimez un alias et lancez à nouveau hello: Exemples de sorties:
$ unalias hello
$ hello

Bonjour de la fonction ()g) Supprimez une fonction shell hello et exécutez à nouveau hello:
$ unset hello
Exemples de sorties:
Bonjour pour le scripth) Supprimez un script shell hello et exécutez à nouveau hello: Exemples de sorties:
$ rm ~/bin/hello
$ hello

bash: / home / monk / bin / hello: Aucun fichier ni répertoire de ce typeVous venez de supprimer le script hello mais le shell Bash cherche toujours un programme hello à l'emplacement / home / monk / bin / hello. Qu'est-ce qui se passe ici? La réponse est assez simple: «Table HASH. Le shell recherche une entrée PATH en cache pour le script hello. Pour vérifier cela, tapez simplement la commande hash pour voir le tableau HASH
$ hash
:
frappe la commande 1 / bin / rm 1 / bin / chat 2 / usr / bin / print 3 / home / moine / bin / bonjour 1 / usr / bin / manVotre SHELL a retenu les noms des paramètres de

chemin dans la table HASH. Il existe une solution simple: effacez la table de hachage et visualisez l'effet à nouveau:
$ hash -r
$ hello
Exemples de sorties:
bash: bonjour: commande non trouvéeJ'espère que cette explication de base aidera tous les nouveaux utilisateurs. En conséquence, la prochaine fois que vous exécuterez une commande, le shell procédera à une séquence d'opérations assez compliquée pour traiter votre demande.

~$ type toto
bash: type: toto : non trouvé
~$ type pdflatex
pdflatex est /usr/bin/pdflatex

~$ type echo
echo est une primitive du shell
~$ which echo
/bin/echo

Voici une dernière série d'exemples :
~$ type if
if est un mot-clé du shell
~$ type ls
ls est un alias vers « ls --color=auto »
~$ type cd
cd est une primitive du shell
~$ type find
find est /usr/bin/find
~$ type sudo
sudo est haché (/usr/bin/sudo)

Modes d'exécution d'une commande

- Exécution séquentielle
  - `cmd1 ; cmd2 ; … ; cmdn`
- Exécution en arrière-plan.
  - `cmd &`
- Exécution dépendante
  - `cmd1 && cmd2`
- Exécution optionnelle
  - `cmd1 || cmd2`

Exécution séquentielle

- Le mode d'exécution par défaut d'une commande est l'exécution séquentielle : le shell lit la commande entrée par l'utilisateur, 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.
- Il faut 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*.
- Pour terminer l'exécution d'une commande lancée en mode synchrone, on appuie simultanément sur les touches *CTRL* et *C* (notées **^C**).


```
$ 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.


```
$ sleep 5 &
[1] 696
$ ps
```

Exécution dépendante

- Lorsque deux commandes sont séparées par le symbole &&, l'exécution de la seconde est assujettie à la valeur de retour de la première.
- Si la première commande renvoie une valeur nulle, signifiant par convention « réussite », alors le Shell exécute la seconde ; sinon, cette dernière est ignorée.
- La valeur de retour d'une liste liée par && est celle renvoyée par la dernière commande exécutée.

Exécution optionnelle

- Symétriquement, il existe une connexion **||** avec laquelle la seconde partie de la liste de commandes n'est exécutée que si la première partie a échoué (en renvoyant une valeur non nulle).
- La liste s'interrompt donc dès qu'une opération réussit.
- Le code de retour renvoyé est celui de la dernière commande exécutée.

Invocation du Bash

- Invoqué pour être le Shell d'interaction, ou avec l'option `--login'
- Invoqué comme Shell interactif sans étape de connexion
- Invoqué non interactivement
- Invoqué avec la commande sh
- Mode POSIX
- Invoqué à distance
- Invoqué alors que UID est différent de EUID
  - Fichiers de démarrage de Bash

Invoqué pour être le Shell d'interaction

- Interactif signifie que vous pouvez entrer des commandes.
- Le Shell n'est pas lancé parce qu'un script a été activé.
- Un Shell de connexion vous donne accès au Shell après authentification.
- Fichiers lus :
  - /etc/profile
  - ~/.bash_profile, ~/.bash_login ou ~/.profile : le premier fichier lisible trouvé est lu
  - ~/.bash_logout à la déconnexion.
- Des messages d'erreur s'affichent si les fichiers de configuration existent mais sont illisibles. Si un fichier n'existe pas, Bash cherche le suivant.

Invoqué comme Shell interactif sans connexion

- Un Shell sans connexion signifie que l'accès ne nécessite pas d'authentification par le système.
- Par exemple, quand vous ouvrez un terminal par le biais d'une icone, ou d'un menu.
- Fichiers lus :
  - ~/.bashrc
- Ce fichier est habituellement référencé dans ~/.bash_profile :
  - if [-f ~/.bashrc]; then . ~/.bashrc; fi

Invoqué non interactivement

- Tous les scripts utilisent un Shell non-interactif.
- Ils sont programmés pour faire certaines tâches et ne peuvent être utilisés pour faire autre chose que ce pour quoi ils ont été prévus.
- Fichiers lus :
  - définis par BASH_ENV
- PATH n'est pas utilisé pour la recherche de ces fichiers, donc mettre le chemin complet dans la variable si vous souhaitez en faire usage.

Create the following text file and save it in /etc/global
Code:
echo "'sayItAsItIs' function available!" sayItAsItIs () { echo "I'm just saying: $@" }
Notice that there is **no** shebang! If you set your BASH_ENV like
Code:
BASH_ENV=/etc/global
Then all your scripts will be able to call the function 'sayItAsItIs', e.g.
Code:
$ /bin/bash -c 'sayItAsItIs This script has no purpose!'
This will output
Code:
'sayItAsItIs' function available! I'm just saying: This script has no purpose!
Real scripts will also be able to call 'sayItAsItIs':
Code:
$ cat script.bash #!/bin/bash sayItAsItIs No purpose $ ./script.bash 'sayItAsItIs' function available! I'm just saying: No purpose $
So BASH_ENV holds the location of a "substitute" for .bashrc if the shell is non-interactive. The variable ENV comes into the picture when you call bash as 'sh'. Read the according section of the manpage again for more details.

Invoqué avec la commande sh

- Bash essaye de se comporter comme le programme historique Bourne sh tout en se conformant à la norme POSIX.
- Fichiers lus :
  - /etc/profile
  - ~/.profile
- Quand il est invoqué de façon interactive, la variable ENV peut pointer vers des informations de démarrage suplémentaires.

Mode POSIX

- Cette option est activée soit en employant l'intégrée *set* :
  - set -o posix
- ou en appelant le Bash avec l'option --posix
- Bash essayera alors de respecter autant que possible la norme POSIX des Shell.
- Déclarer la variable POSIXLY_CORRECT fait la même chose.
- Fichiers lus :
  - définis par la variable ENV

Invoqué à distance

- Fichiers lus quand le Shell est invoqué par rshd :
  - ~/.bashrc
- Eviter l'usage d'outils à distance
- Ayez à l'esprit les dangers de ces outils tels que rlogin, telnet, rsh et rcp.
- Leur usage présente des risques pour la confidentialité et la sécurité de par leur mode d'accès parce que des données non cryptées parcourent le réseau.
- Si vous avez le besoin d'outils à distance, transfert de fichiers et autres, utilisez une version de Secure Shell.

Invoqué alors que UID est différent de EUID

- Aucun fichier de démarrage n'est lu dans ce cas.
- Real ID
- Effective ID

Permissions are determined by the *effective* user ID, not the *real* one.
Command ./foo.sh outputs 0, and the command sudo ./foo.sh outputs 1

```
#!/bin/bash
if [[ $EUID -ne 0 ]]
 then echo "This script must be run as root"
 exit 1
fi
```

Avant d'aller plus loin

2. OBTENIR DE L'AIDE SOUS LINUX

Trop d'infos tuent l'info

- Les sources d'information dans le monde de l'Open Source sont extrêmement nombreuses.
  - Avantage : vous cherchez de l'info, vous allez trouver pléthore de documents.
  - Inconvénient : vous risquez de vous noyer dans toute cette masse, surtout en débutant.

man

- Chaque commande dispose d'une page de manuel en ligne (appelée manpages).
- Pour accéder à cette aide en ligne, il suffit de taper `man <la commande>`.
- Les pages de manuel sont réparties en sections. `$ man man`

```
...
 1 Programmes exécutables ou commandes du shell
 2 Appels système (Fonctions fournies par le noyau)
 3 Appels de bibliothèque (fonctions fournies par les
   bibliothèques des programmes)
 4 Fichiers spéciaux (situés généralement dans /dev)
 5 Formats des fichiers et conventions. Par exemple /etc/passwd
 6 Jeux
 7 Divers (y compris les macropaquets et les conventions).
 8 Commandes de gestion du système (généralement réservées au
   superutilisateur)
 9 Sous-programmes du noyau [hors standard]
```

Rechercher une page de manuel

- Que faire si vous ne connaissez qu'un mot-clé du manuel que vous recherchez ?
  - **whatis**: recherche sur les noms exacts de page
  - **apropos**: recherche sur les noms et les descriptions,
- Utilitaires fournis avec man, permettent d'effectuer rapidement une recherche à l'aide d'un mot clé, avec ou sans jokers, ou bien à l'aide d'une expression rationnelle.
- whatis utilise la recherche par mot-clé sans joker,
- apropos utilise les expressions rationnelles.

Which
Whereis
Type

Autres

- **info**: documentation organisée au moyen de liens d'hypertextes.
- La plupart des commandes disposent d'une option **--help.**
- /**usr/share/doc**: documentation des logiciels installés.
- TLDP
- …

3. CONSTRUCTION DE BLOCS

La syntaxe Shell

- Si la saisie n'est pas commentée, le Shell la lit et la divise en mots et opérateurs, selon les règles d'analyse qui déterminent la signification de chaque caractère saisi.
- Alors ces mots et opérateurs sont transformés en commandes et autres constructions, lesquels retournent un statut d'exécution qui peut être exploité.

Parenthèses, crochets, accolades

- PARENTHÈSES
  - Commandes composées :
    - Contient un bloc de commandes.
    - Les parenthèses peuvent être accolées au texte.
    - Exécution dans un sous-shell.
- CROCHETS DROITS
  - Indice de variable tableau
  - Tests

ACCOLADES

- Opérateur variable
- Commandes composées :
  - Nécessite un espace à l'intérieur de l'accolade (ne pas coller au texte)
  - Contient un bloc de commande.
  - Ne change pas le shell courant.
  - Permet de modifier la précédence.
- Développement des chaines de caractère
  - Rien à voir avec les regroupements de commande ci-dessus.
  - Pas d'espace entre les accolades et le texte.
  - Éléments séparés par des virgules (sans espaces)

Exécution par le shell courant

{commande; commande; commande;}

Exécution par un sous-shell

(commande; commande; commande;...)

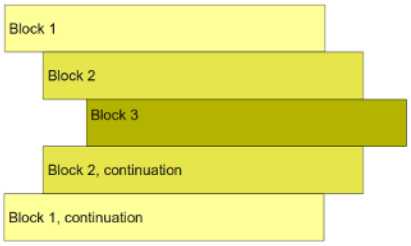
{<START>..<END>}
{<START>..<END>..<INCR>} (Bash 4)
<PREAMBLE>{.....}
{.....}<POSTSCRIPT>
<PREAMBLE>{.....}<POSTSCRIPT>

```
$ echo {5..12}
 5 6 7 8 9 10 11 12
$ echo {c..k} c d e f g h i j k
```

Exécuter un programme dans un script

- Quand le programme en exécution est un script Shell, Bash créera un nouveau processus Bash en activant un *fork*.
- Ce sous-Shell lit les lignes du script une par une.
- Les commandes de chaque ligne sont lues, interprétées et exécutées comme si elles avaient été entrées au clavier.
- Tandis que le sous-Shell opère sur chaque ligne du script, le Shell parent attend que le processus fils ait fini.
- Quand il n'y a plus de ligne à lire dans le script, le sous-Shell se termine.
- Le Shell parent s'active et affiche l'invite de nouveau.

EXERCICE 1



Scripting with style

4. ECRIRE DE BONS SCRIPTS

Caractéristiques d'un bon script

1. Un script devrait s'exécuter sans erreurs.
2. Il devrait accomplir la tâche pour laquelle il a été conçu.
3. La logique du programme est clairement définie et apparente.
4. Un script n'exécute pas des instructions inutiles.
5. Les scripts devraient être réutilisables.

Structure

- La structure d'un script est très flexible.
- Même si Bash permet beaucoup de liberté, vous devez mettre en œuvre une logique rigoureuse, un contrôle des données, une efficacité qui permet à l'utilisateur qui exécute le script de le faire facilement et correctement.
- Au moment d'écrire un nouveau script, posez-vous les questions suivantes :
  - Aurai-je besoin d'informations de la part de l'utilisateur ou de son environnement ?
  - Comment vais-je mémoriser ces données ?
  - Des fichiers doivent-ils être créés ? Où et avec quel propriétaire et quelles permissions ?
  - Quelles commandes utiliserais-je ? Si le script est exécuté sur différents systèmes, est-ce que tous ces systèmes ont les commandes dans la version requise ?
  - L'utilisateur a-t-il besoin que le script lui renvoie des informations ? Quand et pourquoi ?

Directives d'indentations

- L'indentation n'est rien qui influence techniquement un script, ce n'est que pour nous les humains.
  - Je suis habitué à utiliser *deux caractères d'espace*
  - facile et rapide de taper
  - Ce n'est pas TAB - qui s'affiche différemment dans différents environnements
  - Il est assez large pour donner un aspect visuel et assez petit pour ne pas perdre trop d'espace sur la ligne

Briser des lignes

- Chaque fois que vous devez diviser les lignes de code long, vous devez suivre l'une de ces deux règles:
- Indentation à l'aide de la largeur de la commande:

```
Commande Some_very_long_option \
                 Some_other_option
```

- Indentation en utilisant deux espaces: (préféré)

```
Commande Some_very_long_option \
  Some_other_option
```

Divers

- Comme toutes les variables réservées sont MAJ, le moyen le plus sûr est d'utiliser uniquement des noms de variables MIN.
- Cela est aussi vrai pour les fonctions,
- Si vous utilisez des noms MAJ, préfixez le nom avec (MY_ par exemple)
- Si possible, utilisez un shebang.
- Il spécifie l'interpréteur à utiliser lorsque le fichier de script est appelé directement: si vous codez en Bash, spécifiez /bin/bash
- Commentez tout ce que vous pouvez.

5. ECRIRE ET CORRIGER DES SCRIPTS

Le shebang

- Le *shebang* (#!) en tête de fichier, indique à votre système que ce fichier est un script pour l'interpréteur indiqué.
- Tout de suite après le *shebang* se trouve le chemin vers le programme qui interprète les commandes de ce script, qu'il soit un shell, un langage de programmation ou un utilitaire.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/usr/bin/expect
#!/bin/sed -f
#!/bin/awk -f
```

Appel du script

- Après avoir écrit un script, vous pouvez l'appeler avec la commande
  - sh nom_script, ou avec
  - bash nom_script (différents)
- Rendre le script directement exécutable avec un chmod.
  - chmod +rx nom_script
  - vous pouvez le tester avec ./nom_script ou /<chemin>/nom_script
- Enfin, vous pouvez le déplacer dans /usr/local/bin (en tant que root), ou $HOME/bin (RedHat) pour le rendre utilisable par tous les utilisateurs du système ou par vous seulement.
- Le script pourra alors être appelé en tapant simplement nom_script [ENTER] sur la ligne de commande.

Hello World

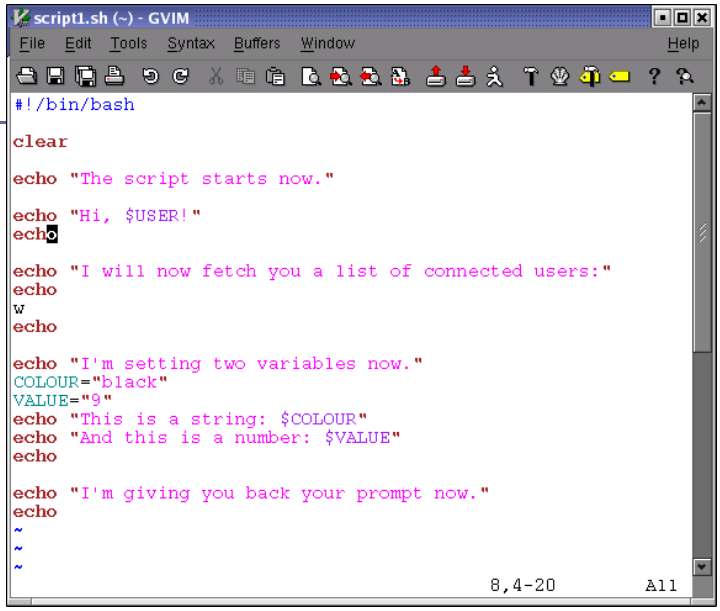
```
#!/bin/bash
# script0.sh
echo "Hello World"
exit
```

Exécuter de plusieurs façons

$chmod u+x script1.sh
$ls -l script1.sh
$script1.sh
$echo $COLOUR
$echo $VALUE
$rbash script_name.sh
$sh script_name.sh
$bash -x script1.sh
$source script1.sh
$echo $VALUE

source = .

L'intégrée Bash « source » est un synonyme de la commande Bourne shell « . » (dot).



Ajout de commentaires

- Les lignes commençant avec un # (à l'exception de #!) sont des commentaires et ne seront pas *exécutées*.

```
# Cette ligne est un commentaire.
```

- Peuvent apparaître après la fin d'une commande.

```
echo "Un commentaire va suivre." # Un commentaire ici.
# Notez l'espace avant le #
```

- On peut inclure les commentaires à l'intérieur d'un tube.

```
initial=( `cat "$startfile" | sed -e '/#/d' | tr -d '\n' |\
# Supprimer les caractères '#' de commentaire.
          sed -e 's/\./\. /g' -e 's/_/_ /g'` )
# Extrait du script life.sh
```

Débugger les scripts Bash

- Bash fournit divers moyens pour débugger.
- La plus commune est de lancer le sous-Shell avec l'option -x ce qui fait s'exécuter le script en mode débug.
- Une trace de chaque commande avec ses arguments est affichée sur la sortie standard après que la commande ait été interprétée mais avant son exécution.

```
bash -x script1.sh
```

Débugger une partie du script

- Avec l'intégrée « set » vous pouvez exécuter en mode normal ces portions de code où vous êtes sûr qu'il n'y a pas d'erreurs, et afficher les informations de débuggage seulement pour les portions douteuses.

```
set -x              # active le mode débug
<votre code>
set +x              # stoppe le mode débug
```

- On peut basculer du mode activé à désactivé autant de fois que l'on veut dans le script.

Débugger (exécuter) ligne par ligne

```
#!/bin/bash
set -x
trap read debug

< YOUR CODE HERE >
```

```
#!/bin/bash
set -x
trap 'echo $? ; read' debug

< YOUR CODE HERE >
```

Exercice 2

- Les administrateurs système écrivent souvent des scripts pour automatiser certaines tâches. Donnez quelques exemples de situations où de tels scripts sont utiles.
- Écrivez un script qui, lors de son exécution, donne la date et l'heure, la liste de tous les utilisateurs connectés et le temps passé depuis le lancement du système (uptime). Pour terminer, votre script doit enregistrer cette information dans un journal.
- Ajoutez des commentaires.
- Ajoutez des informations à destination de l'utilisateur.
- Exécuter le script en mode normal puis en mode débug. Il doit s'exécuter sans erreurs.
- Faites que le script fasse une erreur : voyez ce qui arrive si la première ligne n'est pas correcte ou si vous libellez mal une commande ou une variable - par exemple déclarez une variable par un nom en majuscule et référencez-la avec le nom en minuscule. Voyez ce que les commentaires de débug affichent dans ce cas.

6. L'ENVIRONNEMENT DU BASH

Introduction

- Une fonctionnalité principale de Bash est de gérer les **paramètres**.
- Un paramètre est une entité qui stocke des valeurs et est référencée par un **nom** , un **numéro** ou un **symbole spécial**.
- Les paramètres référencés par un nom sont appelés **variables** (cela s'applique également aux tableaux)
- Les paramètres référencés par un nombre sont appelés **paramètres de position** et reflètent les arguments donnés à un shell,
- Les paramètres référencés par un **symbole spécial** sont des paramètres auto-définis qui ont des significations et des utilisations différentes.

Les variables

- Une variable est identifiée par un nom, suite de lettres, de chiffres ou de caractères espace souligné ne commençant pas par un chiffre. Les lettres majuscules et minuscules sont différenciées.
- Les variables peuvent être classées en trois groupes :
  - les variables utilisateur (ex : **a**, **valeur**)
  - les variables prédéfinies du shell (ex : **PS1**, **PATH**, **REPLY**, **IFS**)
  - les variables prédéfinies de commandes unix (ex : **TERM**).
- En général, les noms des variables utilisateur sont en lettres minuscules tandis que les noms des variables prédéfinies (du Shell ou de commandes unix) sont en majuscules.
- L'utilisateur peut affecter une valeur à une variable en utilisant l'opérateur d'affectation **=** ou la commande interne **read**.

Les variables Bash ne sont pas typées

- Contrairement à de nombreux autres langages de programmation, Bash ne classe pas ses variables par « type ».
- Pour l'essentiel, *les variables Bash sont des* ***chaînes*** *de caractères* mais, suivant le contexte, Bash autorise les opérations arithmétiques et les comparaisons sur ces variables, le facteur décisif étant la présence de chiffres uniquement dans la variable.

Types (classes) de variables

- variables **locales**
  - variables visibles uniquement à l'intérieur d'un bloc de code ou d'une fonction
- variables **d'environnement**
  - variables qui affectent le comportement du shell et de l'interface utilisateur

Les variables Globales

- Chaque fois qu'un shell démarre, il crée les variables shell correspondantes à ses propres variables d'environnement.
- Mettre à jour ou ajouter de nouvelles variables d'environnement force le shell à mettre à jour son environnement, et tous les processus fils (les commandes qu'il exécute) héritent de cet environnement.

Exporter les variables

- Quand un script déclare des variables, il faut ensuite les « exporter », pour les ajouter à l'environnement du script. C'est le rôle de la commande **export**.
- Les variables Globales ou variables d'environnement sont disponibles dans tous les Shells.
- Les commandes **env** ou **printenv** peuvent être employées pour afficher les variables d'environnement.
- Les variables exportées sont appelées les variables d'environnement. Définir et exporter sont souvent 2 actions faites en même temps.

```
export VARNAME="valeur"
```

Note

- Un script peut exporter des variables seulement vers ses processus fils, c'est-à-dire seulement vers les commandes ou processus que ce script en particulier initie.
- Un script invoqué depuis la ligne de commande ne peut pas ré-exporter des variables à destination de l'environnement de la ligne de commande dont il est issu.
- Les processus enfants ne peuvent pas réexporter des variables vers les processus parents qui les ont fait naître.

Variables locales

- Les variables locales ne sont visibles que dans le Shell courant.
- La commande intégrée **set** sans aucune option fait afficher une liste de toutes les variables (y compris les variables d'environnement) et les fonctions.
- L'affichage sera trié et dans un format réutilisable.

```
diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
```

Variables typées selon leur contenu

- A part distinguer les variables selon leur portée - locale/globale -, nous pouvons aussi les distinguer par catégories selon ce qu'elles contiennent.
- De ce point de vue, les variables se distinguent en 4 types :
  - Variables de chaîne de caractères
  - Variables d'entier
  - Variables de constantes
  - Variables tableau

Créer des variables

- **Affectation directe**
  - Syntaxe : *nom=[valeur] [nom=[valeur] ...]*
- Il est impératif que le nom de la variable, le symbole **=** et la valeur à affecter ne forment qu'une seule chaîne de caractères.
- Il est conseillé d'entourer la valeur avec des guillemets lors de l'assignation : ça réduit les risques d'erreurs.
- **Affectation par lecture**
- Elle s'effectue à l'aide de la commande interne **read**. Celle-ci lit une ligne entière sur l'entrée standard.
  - Syntaxe : *read [var1 ...]*

Variable vs Valeur

- Si *variable1* est le nom d'une variable, alors ***$****variable1* désigne une référence à sa valeur, la donnée qu'elle contient.
- Une variable non initialisée a une valeur « vide (en anglais 'NULL') » - pas de valeur assignée du tout (mais *pas* zéro !).

Guillemets et apostrophes

- Lors du référencement d'une variable, il est généralement conseillé de placer son nom entre guillemets ("…").
- Ceci empêche la réinterprétation de tous les caractères spéciaux à l'intérieur de la chaîne -- sauf pour $, ` (apostrophe inversée) et \ (échappement).
- Comme $ reste interprété comme un caractère spécial, cela permet de référencer une variable entre guillemets ("$variable"), c'est-à-dire de remplacer la variable par sa valeur.

Guillemets et apostrophes

- Encadrer une chaîne de caractères par des apostrophes (' ... ') a pour effet de protéger les caractères spéciaux de la chaîne en empêchant leur réinterprétation ou leur expansion par le shell ou par un script shell.
- Un caractère est « spécial » s'il a une autre interprétation possible en plus de sa valeur littérale.
- Par exemple, l'astérisque * représente le caractère joker dans le remplacement et dans le expressions rationnelles.

Attributs des variables : declare ou typeset

- Les commandes internes **declare** et **typeset**, qui sont des synonymes exacts, permettent de modifier les propriétés des variables.
- Options pour declare/typeset
  - -r en lecture seule
    - (declare -r var1 fonctionne de la même façon que readonly var1)
  - -i entier
    - Certaines opérations arithmétiques sont permises pour des variables déclarées entières
  - -a tableau (array)
  - -f fonction(s)
  - -x export
  - -x var=$value

6.1 EXPANSION DES PARAMÈTRES

http://wiki.bash-hackers.org/syntax/pe

L'expansion des paramètres

- C’est la procédure pour obtenir la valeur de l'entité référencée, comme l'expansion d'une variable pour imprimer sa valeur.
- Au moment de l'expansion, vous pouvez faire des choses très intéressantes avec le paramètre ou sa valeur.

Utilisation simple

```
$PARAMETER
${PARAMETER}
```

- La forme la plus simple est d'utiliser le nom d'un paramètre dans les accolades.
- L'avantage c'est qu'il peut être immédiatement suivi par des caractères qui seraient interprétés comme faisant partie du nom du paramètre.

```
WORD="voiture"
Echo "Le pluriel de $WORD est probablement $WORDs"
Echo "Le pluriel de $WORD est très probablement ${WORD}s"
```

Indirection

`${!PARAMETER}`

- Le paramètre référencé n'est pas PARAMETER lui-même, mais le paramètre dont le nom est stocké comme valeur de PARAMETER.


```
read -p 'Quelle variable souhaitez-vous inspecter?' Look_var
printf 'La valeur de "%s" est: "%s" \n' "$look_var" "${!Look_var}"
```

Modification de la casse

```
${PARAMETER^}          ${PARAMETER^^}
${PARAMETER,}          ${PARAMETER,,}
${PARAMETER~}          ${PARAMETER~~}
```

- L'opérateur (^) modifie le premier caractère en majuscule, l'opérateur (,) en minuscule.
- Lorsque vous utilisez la double forme (^^ et ,,), tous les caractères sont convertis.
- Les opérateurs ~ et ~~ inversent la casse du texte donné (dans PARAMETER).
- ~ inverse la casse de la première lettre de mots dans la variable
- ~~ inverse la casse pour tous.

Suppression de sous-chaine

`${PARAMETER#PATTERN}` `${PARAMETER##PATTERN}`
`${PARAMETER%PATTERN}` `${PARAMETER%%PATTERN}`

- Expansion d'une partie de la valeur d'un paramètre, compte tenu d'un motif pour décrire ce qu'il faut supprimer de la chaîne.
- L'opérateur "#" tentera d'enlever le texte le plus court correspondant au motif, tandis que "##" essaye de le faire avec la plus longue correspondance de texte à partir du début.
- Avec "%" et "%%" idem, sauf que Bash essaie de faire correspondre le motif à partir de la fin de la chaîne

- Depuis le début
  `${FILENAME#*.}`
  ⇒ ~~`bash`~~`.hackers.txt`
  `${FILENAME##*.}`
  ⇒ ~~`bash.hackers`~~`.txt`

- Depuis la fin
  `${FILENAME%.*}`
  ⇒ `bash.hackers`~~`.txt`~~
  `${PATHNAME%%.*}`
  ⇒`bash.`~~`hackers.txt`~~

Rechercher et remplacer

`${PARAMETER/PATTERN/STRING}` `${PARAMETER//PATTERN/STRING}`

- La première consiste à ne substituer que la première apparition du motif donné, la seconde consiste à remplacer toutes les occurrences du motif.

`${PARAMETER/PATTERN}` `${PARAMETER//PATTERN}`

- Si la chaine de substitution est omise, les correspondances sont remplacées par NULL, c'est-à-dire qu'elles sont supprimées. Cela équivaut à spécifier un remplacement vide.

MYSTRING="Be liberal in what you accept, and conservative in what you send“

First form: Substitute first occurrence

${MYSTRING/in/by} ⇒ Be liberal ~~in~~by what you accept, and conservative in what you send

Second form: Substitute all occurrences

${MYSTRING//in/by} ⇒ Be liberal ~~in~~by what you accept, and conservative ~~in~~by what you send

Longueur de chaine

```
${#PARAMETER}
```

- Lorsque vous utilisez cette syntaxe, la longueur de la valeur du paramètre est étendue.

Expansion de sous-chaine

`${PARAMETER:OFFSET}` `${PARAMETER:OFFSET:LENGTH}`

- étendre une partie de la valeur d'un paramètre, étant donné une position pour commencer et peut-être une longueur.
- Si on omet LENGTH, le paramètre sera étendu jusqu'à la fin de la chaîne.
- Si OFFSET est négatif, il est compté à partir de la fin de la chaîne
- OFFSET et LENGTH peuvent être une expression arithmétique. Attention: les OFFSET débutent à 0, pas à 1

MYSTRING="Be liberal in what you accept, and conservative in what you send"

echo ${MYSTRING:34}⇒ ~~Be liberal in what you accept, and~~ conservative in what you send

echo ${MYSTRING:34:13}⇒ ~~Be liberal in what you accept, and~~ conservative ~~in what you send~~

Utilisez une valeur par défaut

`${PARAMETER:-WORD}` `${PARAMETER-WORD}`

- Si le paramètre PARAMETER est désactivé (jamais défini) ou null (vide), celui-ci se développe en WORD,
- sinon il se développe à la valeur de PARAMETER, comme si c'était juste ${PARAMETER}.
- Si vous omettez le « : », (deuxième forme), la valeur par défaut n'est utilisée que lorsque le paramètre est désactivé, et non lorsqu'il est vide.

Affecter une valeur par défaut

`${PARAMETER:=WORD}` `${PARAMETER=WORD}`

- Celui-ci fonctionne comme l' « utilisation de valeurs par défaut », mais le texte par défaut que vous donnez n'est pas seulement affiché, mais également **affecté** au paramètre.

Utilisez une autre valeur

`${PARAMETER:+WORD}` `${PARAMETER+WORD}`

- Renvoie la valeur fournie à droite du symbole :+ si la variable est définie et non vide, sinon elle renvoie une chaîne vide.

Afficher l'erreur si nul ou désactivée

`${PARAMETER:?WORD}` `${PARAMETER?WORD}`

- Si le paramètre PARAMETER est défini/non nul, cette expression l'étendra simplement.
- Sinon, l'extension WORD sera utilisée comme annexe pour un message d'erreur.

L'expansion arithmétique

- L'expansion arithmétique permet l'évaluation d'une expression arithmétique et la substitution par le résultat. Le format pour l'expansion arithmétique est :
- $((EXPRESSION)) # Right value
  echo $((EXPRESSION)) # Résultat Booléen
- ((EXPRESSION)) ; echo $?
- let EXPRESSION ; echo $?


```
z=$(($z+3))
z=$((z+3)                    #  Correct aussi.
                             #  À l'intérieur de parenthèses doubles, le déréférencement
de paramètres est optionnel.

# $((EXPRESSION)) est une expansion arithmétique.
# À ne pas confondre avec une substitution de commande.

# Vous pouvez aussi utiliser des opérations à l'intérieur de parenthèses doubles sans
affectation.

  n=0
  echo "n = $n"                 # n = 0

  (( n += 1 ))                 # Incrément.
# (( $n += 1 )) est incorrect!
  echo "n = $n"                 # n = 1

let z=z+3
```

```
let "z += 3"  #  En présence d'apostrophes doubles, les espaces sont permis dans l'affectation des variables.
#  'let' réalise une évaluation arithmétique, plutôt qu'une expansion à proprement parler.

(( a = 23 ))  #  Initialise une valeur, style C, avec un espace  de chaque côté du signe "=".
echo "a (valeur initiale) = $a"

(( t = a<45?7:11 ))   # opérateur à trois opérandes, style C.

echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "        # Oui!
```

Les opérateurs arithmétiques

- Les mêmes que dans la plupart des langages de programmation :
- +, -, *, et / pour les quatre opérations de base ; % pour le modulo ; << et >> pour les décalages binaires à gauche et à droite ; &, |, et ^ pour les opérateurs binaires Et, Ou et Ou Exclusif ; et finalement ~ pour la négation binaire.
- <, >, <=, >=, !=, &&, || (uniquement avec double parenthèses)
- Les opérateurs peuvent être regroupés entre parenthèses pour des questions de priorité. On peut placer des blancs (espaces) à volonté pour améliorer la lisibilité du code.

let

Substitution de commande

- Elle a la forme suivante : **$(commande)**.
- La forme **`commande`**, est peu recommandée, car moins lisible et difficile à imbriquer.
- La commande qui se trouve entre les parenthèses est exécutée, et tout ce qu'elle a écrit sur sa sortie standard est capturé et placé à son emplacement sur la ligne de commande.

```
liste=$(ls)
echo $(date)
```

Substitution de commande

- Lorsque la commande interne n'est qu'une redirection d'entrée, et rien d'autre, par exemple

```
$(<FICHIER)
```

- Alors Bash tente de lire le fichier donné et d'agir comme si la commande donnée était « cat FICHIER ».

Paramètres de position

- Un paramètre de position est référencé par un ou plusieurs chiffres : 8 , 0 , 23
- L'assignation de valeurs à des paramètres de position s'effectue :
  - soit à l'aide de la commande interne **set**
  - soit lors de l'appel d'un fichier Shell ou d'une fonction Shell.
- $1 $2 … $9 représentent respectivement le premier, deuxième …argument du script ou fonction.
- ATTENTION : on ne peut utiliser ni le symbole =, ni la commande interne read pour affecter directement une valeur à un paramètre de position.

shift

- La commande shift réaffecte les paramètres de position, ce qui produit le même effet que de les déplacer d'un cran vers la gauche.
  $1 <--- $2, $2 <--- $3, $3 <--- $4, etc.
- Le vieux $1 disparaît mais $0 (le nom du script) ne change pas.
- La commande **shift** peut prendre un paramètre numérique indiquant le nombre de décalages.
- Si vous faites usage d'un grand nombre de paramètres positionnels dans un script, shift vous permet d'accèder à ceux au-delà de 10, bien que la notation {entre accolades} le permette également.

Paramètres spéciaux

- $* Est remplacé par la liste des paramètres **"$1 $2 ... "**, sauf le premier $0.
- $@ Est remplacé par les éléments **"$1" "$2" ...** , sauf le premier $0.
- $# Renvoie le nombre de paramètres positionnels en décimal.
- $? Renvoie le statut d'exécution de l'instruction la plus récemment exécutée en avant-plan dans un tube.
- $- Renvoie les options déclarées lors de l'invocation de l'intégrée **set**, ou celles positionnées par le Shell lui-même (tel que -i).
- $$ Renvoie l'identifiant du process du Shell. (BASH_SUBSHELL,BASHPID)
- $! Renvoie l'identifiant du process de la commande la plus récemment exécutée en tâche de fond (asynchrone).
- $0 Renvoie le nom du Shell ou du script Shell actif.

Différence entre $* et $@
Dans un shell script, $* et $@ référencent tous les arguments
Exemple

```
homel% cat all1          homel% cat all2
#! /bin/sh               #! /bin/sh
for i in $*              for i in $@
do                       do
  echo $i                  echo $i
done                     done

homel% all1 a b c        homel% all2 a b c
a                        a
b                        b
c                        c
homel% all1 'a b' c      homel% all2 'a b' c
a                        a
b                        b
c                        c
```

Les arguments sont réexaminés au moment de la boucle
L'espace dans le premier argument crée deux arguments !
Protection par " des $* et $@

```
homel% cat all3          homel% cat all4
#! /bin/sh              #! /bin/sh
for i in "$*"            for i in "$@"
do                     do
  echo $i                echo $i
done                   done

homel% all3 a b c        homel% all4 a b c
a b c                    a
                       b
                       c
homel% all3 'a b' c      homel% all4 'a b' c
a b c                    a b
                       c
```

Règles :
$@ º $* : les arguments de la commande sont réexaminés
"$*" est un seul mot composé de tous les arguments
"$@" fournit exactement les arguments

Tableaux

- Une déclaration indirecte peut se faire avec la syntaxe suivante de déclaration de variable :
  ARRAY[INDEXNR]=value
- Le INDEXNR est traité comme une expression arithmétique qui doit être évaluée comme nombre positif.
- Une déclaration explicite d'un tableau est faite avec l'intégrée declare :
  declare -a ARRAYNAME
- Les variables de tableau peuvent aussi être créées avec une affectation composée selon ce format :
  ARRAY=(value1 value2 ... valueN)


```
ARRAY=(one two three)

echo ${ARRAY[*]}
one two three

echo $ARRAY[*]
one[*]

echo ${ARRAY[2]}
three

ARRAY[3]=four

echo ${ARRAY[*]}
one two three four

unset ARRAY[1]

echo ${ARRAY[*]}
```

one three four

unset ARRAY

echo ${ARRAY[*]}
<--no output-->

eval

eval *expr1 expr2 ..... exprn*

- La commande **eval** permet de faire subir à une ligne de commande une double évaluation.
- Permet de construire de toutes pièces une expression, puis de l'évaluer comme s'il s'agissait d'une partie du programme en cours.

$b	contient a
\$$b	contient $a
eval echo \$$b	contient le contenu de la variable a.

- En fait, eval permet d'outre passer les signes "$".

$ **nom=toto**
Initialisation de la variable **var** avec le nom de la variable définie ci dessus
$ **var=nom**
Affichage des valeurs des variables
$ **echo $nom**
toto
$ **echo $var**
nom
$ **echo \$$var**
$nom
$

eval echo \${$n} runs the parameters passed to eval.
After expansion, the parameters are echo and ${1}.
So eval echo \${$n} runs the command echo ${1}.

Equivalent to bash syntax

```
echo ${!n}
```

Valeur temporaire de variable

- Lorsqu'une affectation de variable précède la commande (sur la même ligne), le contenu de la variable ne prend la valeur indiquée que pendant l'exécution de la commande.
- Ce mécanisme, bien que rarement employé, permet de modifier une variable d'environnement pendant l'exécution d'une commande, sans changer sa valeur pour le reste du script.


```
$ echo $LANG
fr_FR
$ date
dim oct 7 08:55:51 CEST 2007
$ LANG=en_US date
Sun Oct 7 08:55:54 CEST 2007
$ LANG=de_DE date
So 7. Okt 08:55:57 CEST 2007
$ LANG=it_IT date
dom ott 7 08:56:00 CEST 2007
$ echo $LANG
fr_FR
```

: no op

- : est exactement équivalent à true

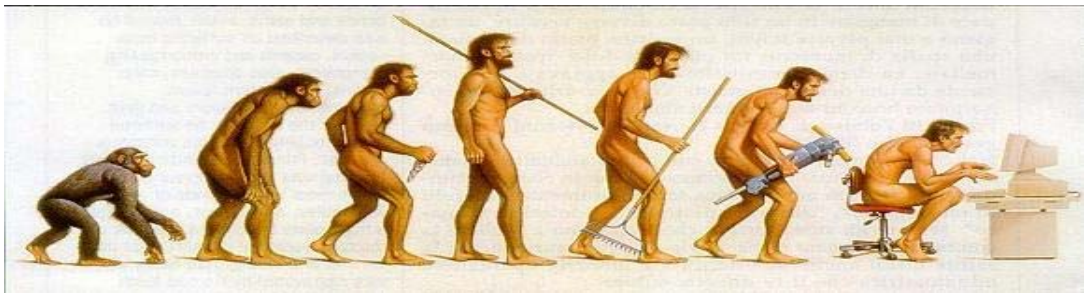
```
while keep_waiting; do
  : # busy-wait
done
```



```
[mandy@root]$ a=11
[mandy@root]$ b=20
[mandy@root]$ c=30
[mandy@root]$ echo $a; : echo $b ; echo $c
10
30
```

Exercice

- Ecrire un script dans lequel 2 entiers sont assignés à 2 variables. Le script doit calculer la surface d'un rectangle à partir de ces valeurs. Il devrait être aéré avec des commentaires et générer un affichage plaisant.

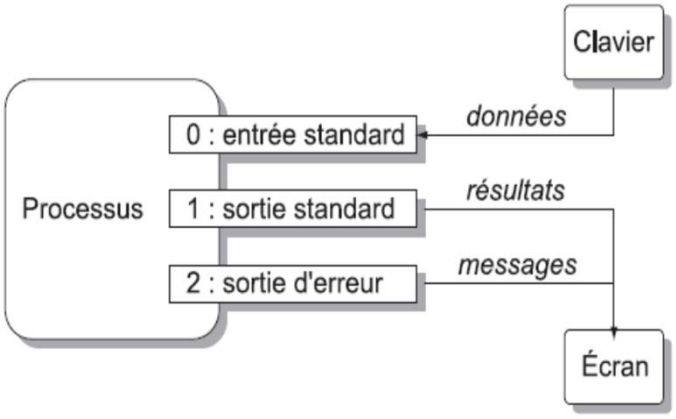


7. ÉLÉMENTS DE PROGRAMMATION SHELL

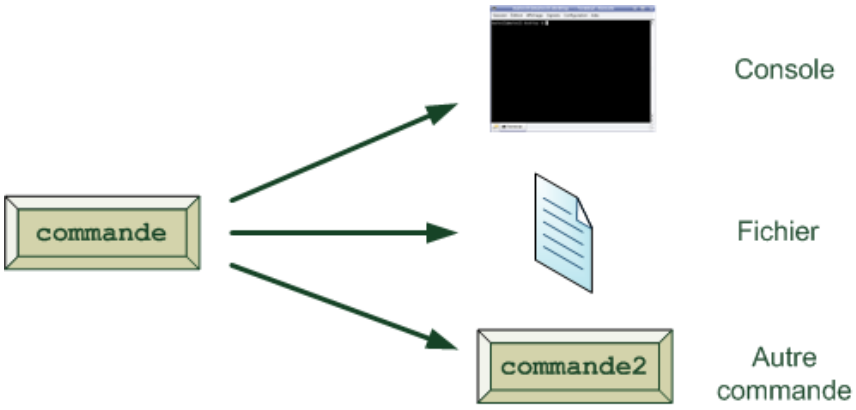
7.1 REDIRECTIONS D'ENTRÉES-SORTIES

Entrées-sorties standards

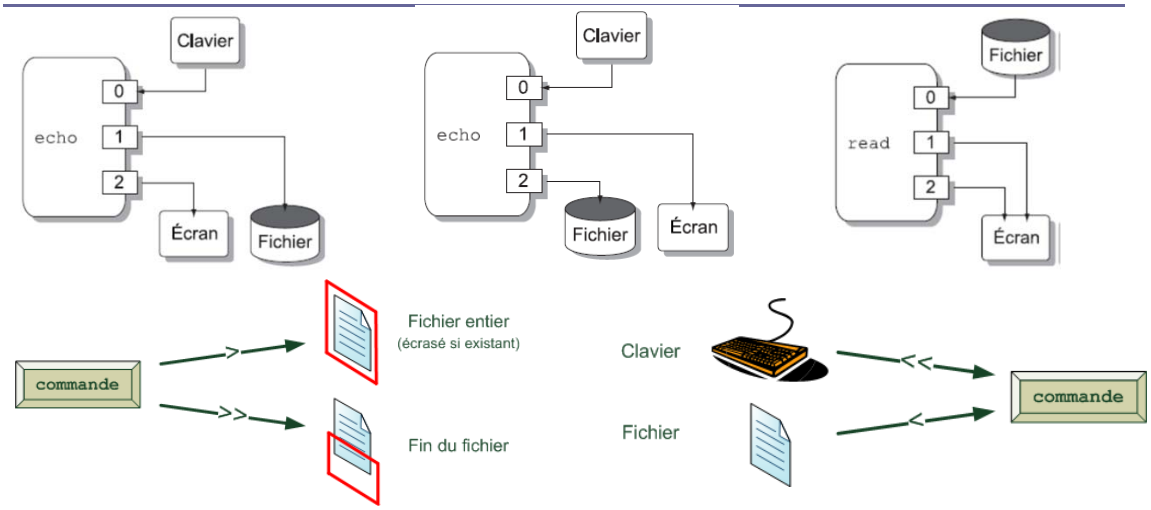
Nombre entier	Nom	Flux
0	Entrée standard	stdin
1	Sortie standard	stdout
2	Erreur standard	stderr



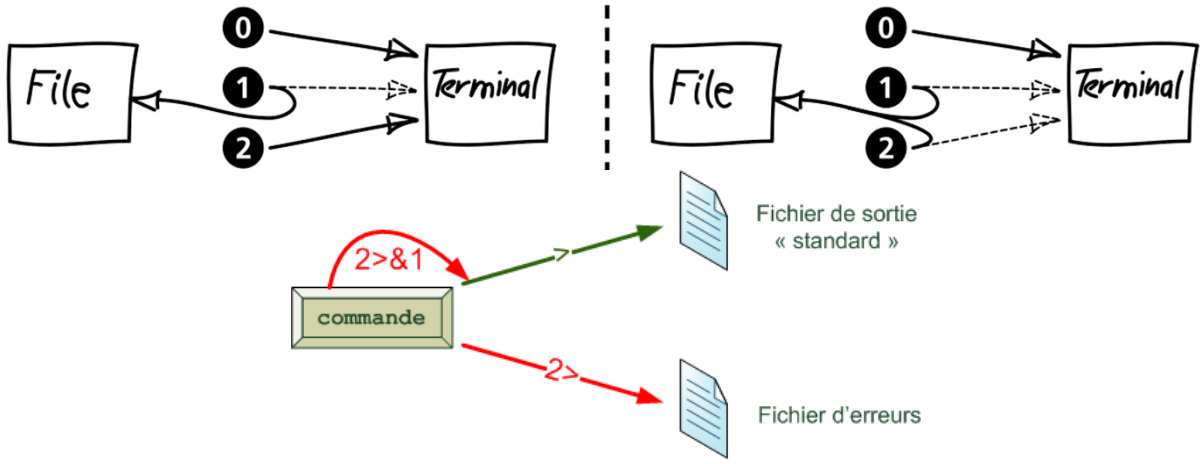
Redirections



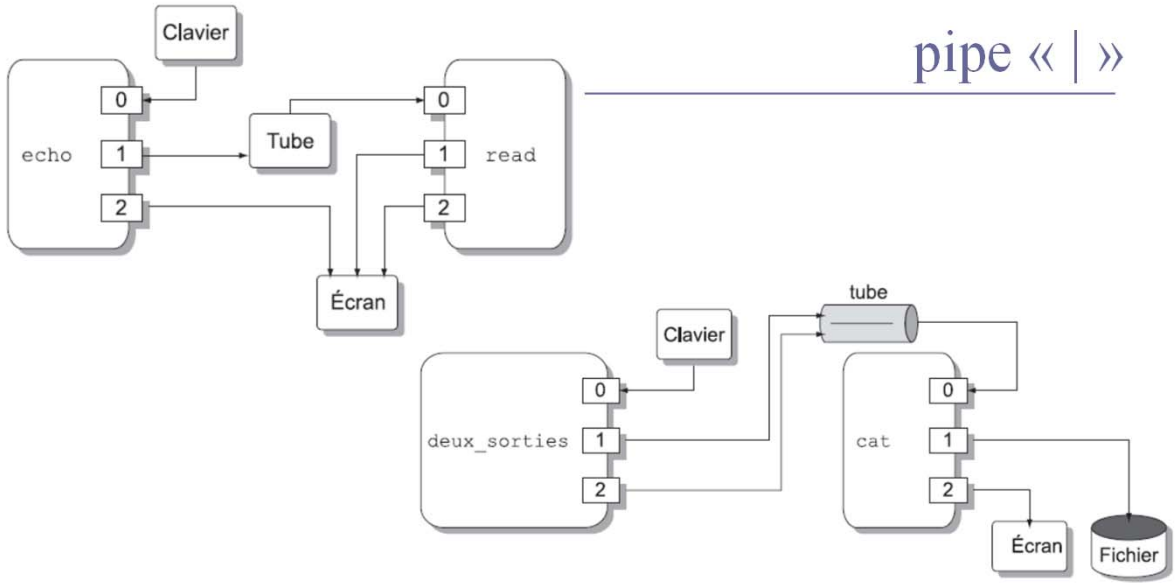
Redirection des entrées et sorties standards >, >>, 2>, 2>>, <, <<



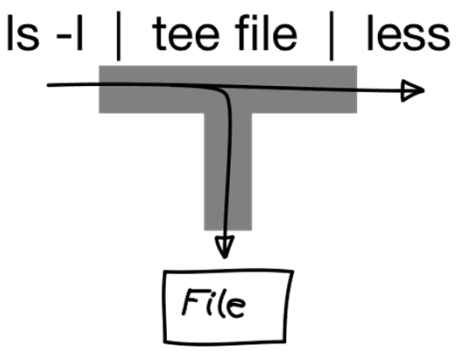
Redirections avancées
<<, &>, 2>&1, >&2

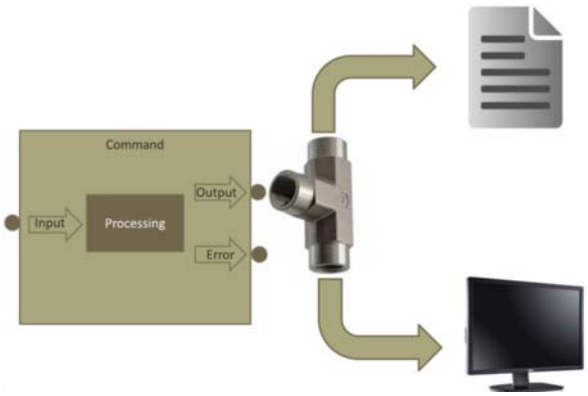


pipe « | »

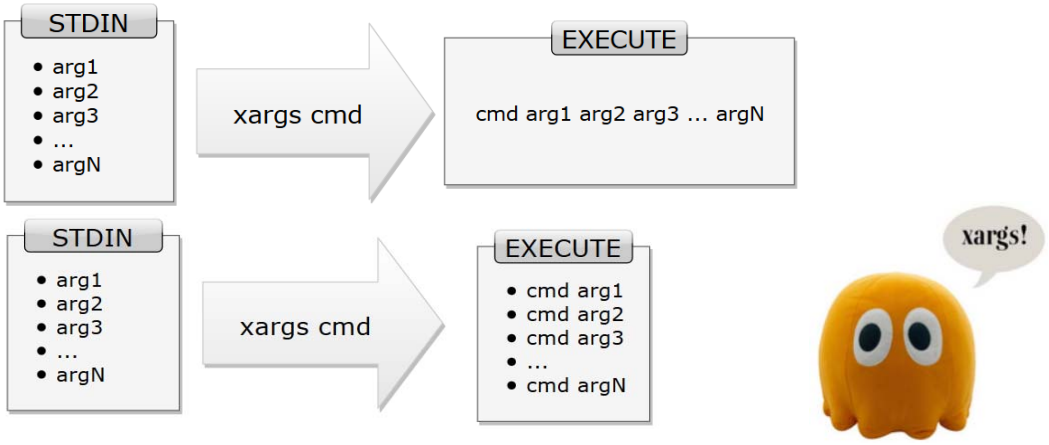

Clavier
0
1
2
echo
Tube
0
1
2
read
Écran
tube
Clavier
0
1
2
deux_sorties
0
1
2
cat
Écran
Fichier

tee

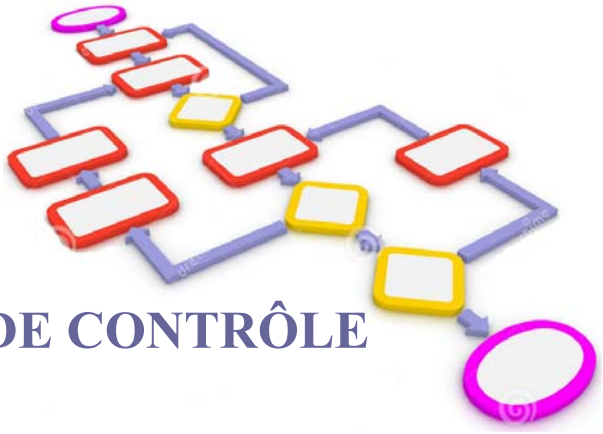

ls -l | tee file | less
File


Command
Input
Processing
Output
Error

xargs



https://marcin-chwedczuk.github.io/how-to-use-xargs



7.2 STRUCTURES DE CONTRÔLE

LES TESTS

Présentation

- Tous les langages de programmation raisonnablement complets sont capables de tester des conditions, puis d'agir en fonction du résultat du test.
- Bash dispose de la commande ***test***, de différents opérateurs à base de crochets et de parenthèses, ainsi que de contrôles if/then.

Les constructions de test

- Un contrôle if/then teste si l'état de sortie d'une liste de commandes vaut 0 (exit code 0 indique le succès) et dans ce cas, fait exécuter une ou plusieurs commandes.
- Il existe une commande dédiée appelée **[** , synonyme de la commande ***test***.
- Cette commande considère ses arguments comme des expressions de comparaison ou comme des tests de fichiers et renvoie un état de sortie correspondant au résultat de la comparaison (0 pour vrai, 1 pour faux).
- Bash a introduit la commande étendue de test **[[...]]**. Cette commande réalise les comparaisons d'une façon qui est familière aux programmeurs venant d'autres langages. Remarque : [[en lui-même est un mot clé, pas une commande.
- Bash considère l'expression [[$a -lt $b]] comme un seul élément, et il renvoie un état de sortie.
- Les expressions ((...)) et let ... renvoient un code de sortie qui dépend du fait que les expressions arithmétiques qu'elles évaluent se résolvent en une valeur non nulle. Cette syntaxe avec évaluations arithmétiques peut par conséquent être utilisée pour effectuer des comparaisons arithmétiques.

Commande test / […]

- Deux syntaxes équivalentes permettent de tester des expressions **[expression]** et **test expression**.
- Elles renvoient toutes les deux un code de retour valant 0 si l'expression est vraie et 1 si l'expression est fausse.
- Il y a beaucoup d'opérateurs disponibles pour réaliser des tests sur les fichiers, sur du texte ou sur des valeurs (arithmétique).

La commande ‘test’ […] cas des fichiers

- -e ce fichier existe
- -f ce fichier est un fichier normal
- -s la taille de ce fichier est supérieure à zéro
- -d ce fichier est un répertoire
- -b ce fichier est un périphérique de type bloc
- -c le fichier est un périphérique de type caractères
- -L ce fichier est un lien symbolique
- -S ce fichier est une socket

La commande ‘test’ […] cas des entiers

- -eq est égal à : if ["$a" -eq "$b"]
- -ne n'est pas égal à : if ["$a" -ne "$b"]
- -gt est supérieur à : if ["$a" -gt "$b"]
- -ge est supérieur ou égal à : if ["$a" -ge "$b"]
- -lt est inférieur à : if ["$a" -lt "$b"]
- -le est inférieur ou égal à : if ["$a" -le "$b"]
- < est inférieur à (entre doubles parenthèses) : (("$a" < "$b"))
- <= est inférieur ou égal à (entre doubles parenthèses) (("$a"< "$b"))
- > est supérieur à (dans une expression entre doubles-parenthèses) (("$a" >"$b"))
- >= est supérieur ou égal à (dans une expression entre doubles-parenthèses) (("$a" >= "$b"))

La commande ‘test’ […] cas des chaines

- = ‘est égal à’ : if ["$a" = "$b"]
- == ‘est égal à’ : if ["$a" == "$b"] Ceci est un synonyme de =.
- != ‘n'est pas égal à’ : if ["$a" != "$b"]
- -n ’la chaîne de caractères n'est pas « vide »’
- -z ‘la chaîne de caractères est « vide », c'est-à-dire qu'elle a une taille nulle’

La construction [[…]]

- Est la version plus souple de [ ] dans Bash.
- Utiliser la construction [[...]], au lieu de [...] peut vous permettre d'éviter des erreurs de logique dans vos scripts.
- Par exemple, les opérateurs &&, ||, < et > fonctionnent à l'intérieur d'un test [[ ]] bien qu'ils génèrent une erreur à l'intérieur d'une construction [ ].

Exemples l'opérateur == dans [[…]]

- L'opérateur de comparaison == se comporte différemment à l'intérieur d'un test à double crochets qu'à l'intérieur de crochets simples.
- [[$a == z*]] # Vrai si $a commence avec un "z" (correspondance de modèle).
- [[$a == "z*"]] # Vrai si $a est égal à z* (correspondance littérale).
- [$a == z*] # Correspondance de fichiers
- ["$a" == "z*"] # Vrai si $a est égal à z* (correspondance littérale).

A = inside a [[is exactly equivalent to:
A == inside a [[does both byte-by-byte and glob matching.
If the string or variable on the right side of the == is quoted, a byte comparison is made. If all the bytes are equal, the result of the [[is "good" (0).
If the string, or preferable in all cases: a variable, is unquoted, the match is performed as in a filename glob.
$ [[aaaa == "aaaa"]] && echo yes yes $ a='aaaa' $ [[aaaa == "$a"]] && echo yes yes $ a='a*' $ [[aaaa == "$a"]] && echo yes $ $ a='a*' $ [[aaaa == $a]] && echo yes yes
It is interesting to note that the unquoted aaaa also work:
$ a='aaaa' $ [[aaaa = $a]] && echo yes yes

Exemples l'opérateur =~ dans [[…]]

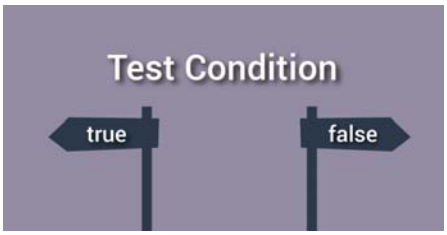
- Faire des correspondances d'expressions régulières

- Avec [vous pourriez écrire

  if ["$answer" = y -o "$answer" = yes]

- Avec [[vous pouvez écrire ceci comme
  - if [[$answer =~ ^y(es)?$]]


```
#!/bin/bash
INT=-5

if [[ "$INT" =~ ^-?[0-9]+$ ]]; then

echo "INT is an integer."

else

echo "INT is not an integer." >&2

exit 1

fi
```



If then else fi case esac

SÉLECTION D'INSTRUCTIONS

Construction if-then-else

```
if condition_1
then
commande_1
elif condition_2
then
commande_2
else
commande_n
fi
```

```
if condition ; then
commande
fi
```

Exercice

- Ecrire un script qui prend en argument le nom d'un fichier et qui affiche son type (fichier, répertoire, bloc, …)
- Le script doit faire les vérifications d'usage, à savoir, vérifier s'il y a ou pas un argument, si le fichier existe ou pas etc.
- Essayez de le faire avec des if en optimisant avec des elif.

Construction case-esac

```
case expression in
    motif_1 ) commande_1 ;;
    motif_2 ) commande_2 ;;
    ...
esac
```

while until for select

7.3 ITÉRATIONS D'INSTRUCTIONS

Répétitions while-do et until-do

while condition
do
    commandes
done

until condition
do
    commandes
done


```
cat $1 | while read ligne
do
 echo $ligne
done
```

Rupture de séquence

- Les ruptures de séquences qui se produisent dans les boucles sont introduites par les mots-clés **break** et **continue**.
- **break** sert à sortir immédiatement de la boucle en cours, en transférant le contrôle après le **done**.
- **continue** permet de renvoyer le contrôle au début de la boucle.
- On revient ainsi immédiatement à l'emplacement du test **while** ou **until** sans exécuter les instructions suivantes.

Construction for-do

Bash format

for variable [**in** liste_de_mots]
do
    commandes
done

The C-style for-loop

Synopsis

```
for (( <EXPR1> ; <EXPR2> ; <EXPR3> )); do
  <LIST>
done

for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) do
  <LIST>
done

for (( <EXPR1> ; <EXPR2> ; <EXPR3> )) {
  <LIST>
}
```



```
for ((x=1; x<=3; x++))
{
  echo $x
}

filename=$1
IFS=$'\n'
for next in `cat $filename`
do
   echo "$next read from $filename"
done
exit 0
```

Choix et itération avec select-do

- La structure **select** permet la génération facile de menus.

```
select variable in liste_de_mots
do
    commandes
done
```

- A l'affichage des tous les éléments ainsi que l'invite PS3 sont affichés,
- La ligne lue est mémorisée dans la variable REPLY.


```
#!/bin/bash

PS3='Choisissez votre légume favori : ' # Affiche l'invite.

echo

select legume in "haricot" "carotte" "patate" "oignon" "choux"
do
  echo
  echo "Votre légume favori est $legume."
  echo
  break  #  Qu'arriverait-il s'il n'y avait pas de 'break' ici ?
       #+ fin.
done

exit 0
```

Un menu

menu.sh :

```
#! /bin/bash
 cat << FIN
 *************************
 * titre du programme *
 *************************
 1 - Première option
 2 - Seconde option
 0 - Fin
 FIN
```

Fonctions

```
function ma_fonction
{
...
}

ma_fonction ()
{
...
}
```

- **local** variable : ne sera visible que dans la fonction où elle est définie.
- renvoie un exit code, celui de la dernière instruction.
- **return** val : permet, de préciser code de retour.

Divers

- **exit** code
- nombre entre 0 et 255, de 126 à 255 réservée,

Code	La description
0	Succès
1-255	Échec (en général)
126	La commande demandée (fichier) ne peut pas être exécutée (mais a été trouvée)
127	Commande (fichier) introuvable
128	Selon l'ABS, il est utilisé pour signaler un argument invalide à la sortie intégrale, mais je n'ai pas pu vérifier cela dans le code source de Bash (voir le code 255)
128 + N	La coque a été terminée par le signal N (également utilisé comme ceci par divers autres programmes)
255	Mauvais argument à la sortie intégrale (voir code 128)

- **$IFS** «Internal Field Separator»,
- Par défaut, "<espace> <tabulation> <nouvelle ligne>"
- echo -n "$IFS" | cat –vTE
- IFS n'est jamais exportée dans l'environnement des sous-processus.
- IFS est réinitialisée à sa valeur par défaut à chaque démarrage du Shell (donc pas importée).


```
IFS=$'\n'; for n in $(ls) ; do echo $n ; done
fichier 1
fichier 2
repertoire avec espace 1

https://www.admin-linux.fr/ifs-le-separateur-standard-du-shell/

-------------------------------------------------
#!/bin/bash
while IFS=':' read -r user passwd uid gid comment home shell;  do
   if [ "$uid" -gt 1000 ] ; then
      echo $user
   fi
done < /etc/passwd
```

Scripts avec options en ligne de commande

```
while getopts liste_d_options option ;
do
  case $option in
    option_1 ) ... ;;
    option_2 ) ... ;;
    ? ) ... ;;
esac
done
```

- liste_d_options : chaîne contenant toutes les lettres acceptées par le script pour introduire une option
- Si une option doit être complétée par un argument, on fait suivre la lettre d'un deux-points.

Exemple : [:]acmr:t:

getopts

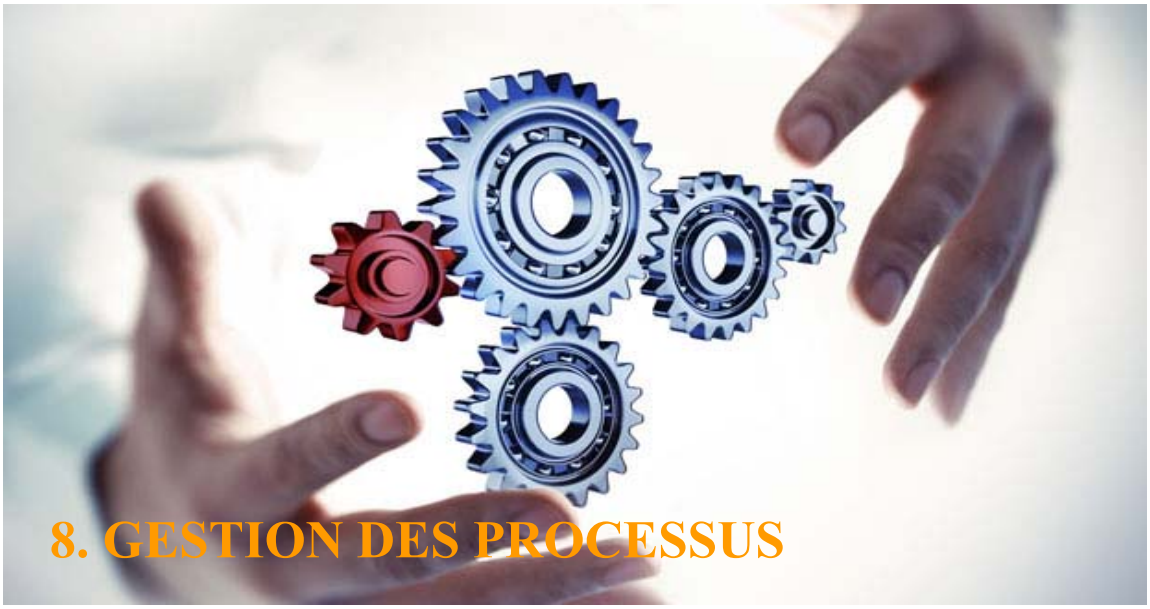
- À chaque appel, la fonction getopts incrémente la variable interne **OPTIND** qui vaut zéro au début du script, et examine l'argument de la ligne de commande ayant ce rang.
- S'il s'agit d'une option simple, sans argument, elle place la lettre correspondante dans la variable à la suite de la chaîne d'options.
- S'il s'agit d'une option qui nécessite un argument supplémentaire, elle en vérifie la présence, et le place dans la variable spéciale **OPTARG**.
- La fonction getopts renvoie une valeur vraie si une option a été trouvée, et fausse sinon.
- Ainsi, après lecture de toutes les options, on peut accéder aux arguments qui suivent, simplement en sautant les OPTIND-1, premiers mots grâce à l'instruction shift.
- Lorsqu'une option illégale est rencontrée, getopts remplit la variable d'option avec un point d'interrogation « ? ».

getopts en cas d'erreurs

- Lorsque le premier caractère de la chaîne contenant les options est un deux-points, getopts n'affiche aucun message d'erreur, mais adopte le comportement suivant :
  - 1. lorsqu'une lettre d'option illégale est rencontrée, le caractère « ? » est placé dans la variable d'option, et la lettre est placée dans OPTARG ;
  - 2. si un argument supplémentaire est manquant, le caractère « : » est placé dans la variable d'option, et la lettre de l'option qui nécessite l'argument est placée dans OPTARG.

Exercices

- Ecrire un script appelé whichdaemon.sh qui vérifie que les démons httpd et init sont lancés sur votre système. Si un httpd est lancé, le script devrait afficher un message comme « Cette machine fait tourner un serveur WEB. » Employez ps pour contrôler les processus.
- Écrire un script qui fait la sauvegarde de votre répertoire HOME sur une machine distante en utilisant scp. Le script devrait écrire son rapport dans un fichier journal, par exemple ~/log/homebackup.log.
- Le script devrait se servir de tar et gzip. Mettre le nom du serveur distant et du répertoire distant dans une variable.
- Le script devrait vérifier l'existence d'une archive compressée. Si elle existe, la supprimer d'abord pour éviter les messages d'erreur.
- Le script devrait nettoyer l'archive compressée avant de se terminer



8. GESTION DES PROCESSUS

Les processus...

- Il existe deux types de processus. Les process, et les jobs.
  - Le process est un programme qui s'exécute dans le système
  - Le job est un process qui est le descendant d'un shell.
- Un processus possède un certain nombre de caractéristiques:
  - PID
  - priorité
  - contexte de mémoire
  - environnement
  - descripteurs de fichiers
  - Références de sécurité
  - exec thread

Comment créer des processus

- Un processus crée (fork) un enfant, pointant vers les mêmes pages de mémoire et marquant la zone en lecture-seule.
- Ensuite, l'enfant exécute (exec) la nouvelle commande, provoquant une erreur 'copy-on-write', copiant alors sur une nouvelle zone de la mémoire
- Un processus peut exécuter, sans créer (recouvrement, exec)
  - L'enfant maintient l'ID de processus du parent

Ascendance de Processus

- **init** est le premier processus lancé au démarrage – il a toujours le PID 1
- À part init, chaque processus a un parent
- Les processus peuvent être à la fois un parent et un enfant

pstree

États des Processus

- **R** : En cours d'exécution (le process est actif et consomme des ressources)
- **D** : En sommeil ininterruptible (Le processus est typiquement en train d'effectuer une tâche non interruptible qui ne peut pas se faire en raison d'une erreur ou d'une autre)
- **S** : En sommeil (le process n'est pas actif mais susceptible d'être réveillé par un appel système)
- **T** : Stoppé ou stracé (Le processus a reçu un signal d'arrêt temporaire - suspension - et attend un SIGCONT)
- **Z** : Zombie (l'état 'Z' correspond à un fils dont le père n'a pas lu la valeur de retour du processus fils. Ce processus ne consomme plus que la place de la structure de description du processus à l'exclusion de toute autre ressource)

Afficher les Processus

- Syntaxe ps:

  ps [options]
- **aux** ou **auxw** sont les options les plus utilisées
  - **a:-** Affiche tous les processus.
  - **x:-** Affiche aussi ceux qui sont sans terminal de contrôle.
  - **u:-** Format User.
- Affiche des informations différentes sur les processus du système
- Contrôle des processus: **top**
- **pgrep, pidof**

Petite définition de chaque partie du résultat

- **User**: le propriétaire du processus, en général l'utilisateur qui a lancé le processus ;
- **PID**: le numéro d'identification unique du processus. Sont attribués dans l'ordre de lancement des processus.
- **%CPU**: pourcentage du temps de CPU consacré à l'exécution de ce processus ;
- **%MEM**: pourcentage de mémoire totale utilisée par ce processus ;
- **VSZ**: superficie totale de la mémoire virtuelle, en blocs de 1k ;
- **RSS**: taille réelle de l'ensemble, le chiffre exact de la mémoire physique allouée à ce processus ;
- **TTY**: terminal associé à ce processus. Le « ? » indique que le processus n'est relié à aucun terminal ;
- **STAT**: état du processus ;
- **START**: indique en heures et minutes depuis quand le programme a été lancé, ou indique depuis combien de jours le processus est lancé s'il fonctionne depuis plus d'une journée ;
- **TIME**: temps CPU utilisé par le processus depuis le début du lancement ;
- **COMMAND**: le nom de la commande.

Petite définition de chaque partie du résultat

- < De haute priorité
- N De faible priorité
- L Pages verrouillées en mémoire
- s Leader de session
- l Multi-thread
- + Est dans le processus de premier plan

Envoyer des Signaux aux Processus

- Syntaxe
  - kill [-*signal*] *pid(s)*
  - kill [-*signal*] %*jobID*
  - Envoie le signal spécifié au processus
  - TERM est le signal par défaut
  - kill -l affiche tous les signaux disponibles
- killall
- pkill

Modifier la Priorité d'Ordonnancement de Processus

- À l'invocation d'un processus

  Syntaxe:

  nice [-n ajustment] commande
- Les processus sont ordonnancés avec la priorité 0 par défaut
- La valeur de priorité peut aller de -20 (priorité la plus haute) à 19 (la plus basse)

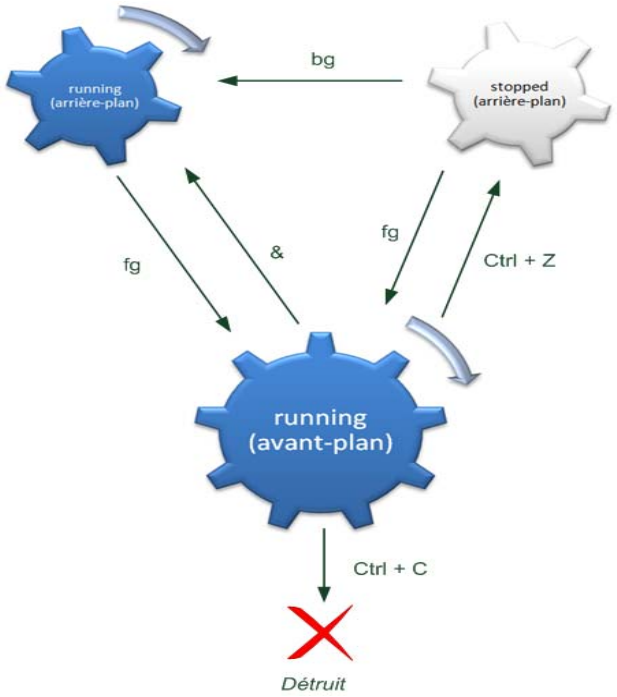
Modifier la Priorité d'Ordonnancement de Processus

- **renice** change la priorité d'un processus en cours d'exécution
  - Syntaxe:

    renice # [[-p|-g] PID] [[-u]user]
- # est la valeur de priorité
- Une fois que la valeur de priorité est augmentée, un utilisateur sans privilèges ne peut pas la diminuer (RedHat)

Suspendre/Reprendre un Processus

- Les travaux commencés au premier plan peuvent être suspendus dans le terminal avec <ctrl-z> ou kill avec **SIGSTOP** ailleurs,
- **Les travaux suspendus peuvent être relancés**
  - Repris en arrière plan (bg)
  - Repris au premier plan (fg)
  - kill avec SIGCONT
- **jobs** affiche tous les processus en exécution ou suspendus en arrière plan
- Le chiffre entre crochets est le numéro de job, utilisé pour tuer ou ramener des travaux au premier plan
- Les numéros de job sont indiqués avec le symbole %num



Résumé des états possibles des processus

157

nohup : détacher le processus de la console

nohup commande

- Lorsqu'un utilisateur se déconnecte, le démon qui détecte cette déconnexion (telnetd, sshd, ...) envoie le signal SIGHUP (signal 1) aux processus enfants, qui provoque par défaut la fin du processus.
- nohup ignore le signal SIGHUP (d'où son nom) puis exécute le programme indiqué en ligne de commande, de sorte que celui-ci sera immunisé contre le signal SIGHUP.
- nohup.out

disown

- The command to separate a running job from the shell (= makes it nohup) is disown and a basic shell-command.
- From bash-manpage (man bash):
- disown [-ar] [-h] [jobspec ...]
- Without options, each jobspec is removed from the table of active jobs. If the -h option is given, each jobspec is not removed from the table, but is marked so that SIGHUP is not sent to the job if the shell receives a SIGHUP. If no jobspec is present, and neither the -a nor the -r option is supplied, the current job is used. If no jobspec is supplied, the -a option means to remove or mark all jobs; the -r option without a jobspec argument restricts operation to running jobs. The return value is 0 unless a jobspec does not specify a valid job.

wait

- Suspend l'exécution du script jusqu'à ce que tous les jobs en tâche de fond aient terminé, ou jusqu'à ce que le numéro de job ou l'identifiant de processus spécifié en option se termine. Retourne l'état de sortie de la commande attendue.
- Vous pouvez utiliser la commande wait pour empêcher un script de se terminer avant qu'un job en arrière-plan ne finisse son exécution (ceci créerait un processus orphelin).
- Optionnellement, wait peut prendre un identifiant de job en tant qu'argument, par exemple, wait%1 ou wait $PPID.

suspend

- Ceci a un effet similaire à Controle+Z, mais cela suspend le shell (le processus père du shell devrait le relancer à un moment approprié).

Réception d'un signal - trap -

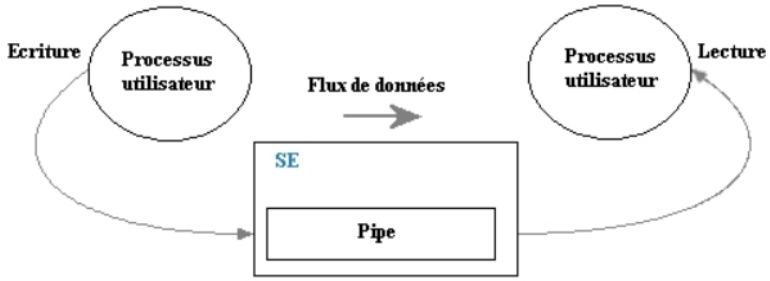
- Syntaxe :

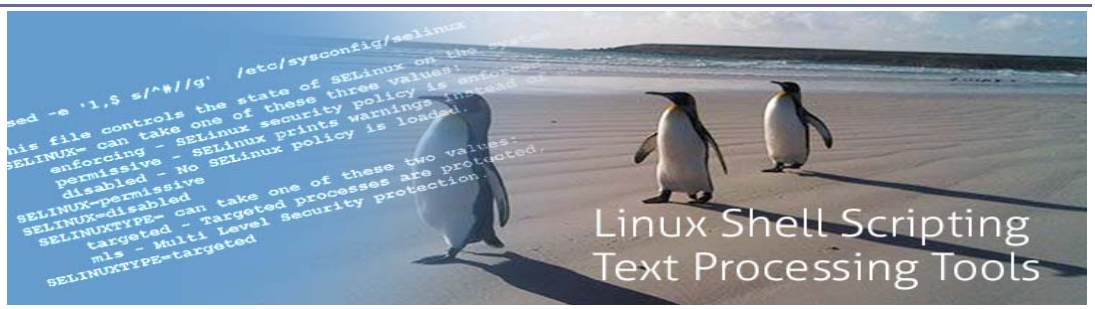
  trap [COMMANDS] [SIGNALS]

- Il peut y avoir des situations ou vous ne souhaitez pas que les usagers de vos scripts quittent abruptement par une séquence de touches du clavier, par exemple parce qu'une entrée est en attente ou une purge est à faire.
- L'instruction trap trappe ces séquences et peut être programmée pour exécuter une liste de commandes à la récupération de ces signaux.

Communication entre processus

- pipe : mkfifo





9. TRAITEMENT DE CHAÎNES - INTRODUCTION

wc

textkit

- Compte les mots ("word count") – mais également compte les lignes et les caractères
  - $ wc story.txt
  - 39 237 1901 story.txt
- Utiliser -l pour le seul compte de lignes
- Utiliser -w pour le seul compte de mots
- Utiliser -c pour le seul compte de caractères

sort

textkit

- Trie du texte en sortie standard – le fichier original n'est pas changé: $ sort [options] [règles] fichier(s)
- Options courantes
- -r Inverse l'ordre en tri descendant (ascendant par défaut)
- -n Tri numérique
- -f Ignore la casse de caractères
- -u Unique (supprime les lignes dupliquées dans la sortie)
- -t 'x' Utilise x en tant que séparateur de champs et non pas les espaces blancs par défaut
- -k POS1 Tri à partir du champ POS1
- -k POS1,POS2 Tri à partir du champ POS1 jusqu'au champ POS2

uniq

textkit

- Supprime les lignes dupliquées successives dans un fichier
- Peut être utilisé en conjonction avec sort pour supprimer toutes duplications (ou utilisez sort -u)
- Utilisez -c pour compter le nombre d'apparitions de données dupliquées

cut

textkit

- Affiche des colonnes de fichiers de données spécifiques
  - $ cut -f4 results.dat
  - -f spécifie champ ou colonne
  - -d spécifie le délimiteur de champ (TAB, par défaut)
    - $ cut -f3 -d: /etc/passwd
  - -c coupe par caractères
    - $ cut -c2-5 /usr/share/dict/words

Autres Outils de Traitement de Chaînes

- paste – fusionne des fichiers
- join – jointure entre deux fichiers
- tr – traducteur de caractères
- split – coupe un fichier en plusieurs autres
- head / tail – entête et fin de fichier

EXPRESSIONS RÉGULIÈRES

ANCIENT EGYPTIAN REGEXP

Les expressions régulières

- Elles ne caractèrisent pas des noms de fichier mais des chaînes de caractères.
- Couramment utilisées sous unix par les commandes vi, sed, awk, find…
- La chaîne renvoyée par une expression régulière est toujours la plus grande pouvant être représentée par cette E.R
- Pour échapper un caractère, utiliser "\"
- Utilisation classique : remplacement

de chaîne : s/RE/chaine_de_remplacement/

Les expressions régulières

- Le point

  .

  Il représente n'importe quel caractère

  Exemple :

  chaîne de départ : azerty…p

  cmd de remplacement : s/…/---/

  chaîne résultat : ---rty…p

Les expressions régulières

- Les suites de caractères [ ]
  - Définies entre crochets.
  - Ils représentent une liste de caractères.
    - Exemple : [abc]
  - Si la liste commence par ^ ça caractérise un caractère qui n'appartient pas à la liste définie entre crochets.
    - Exemple : [^a]
  - Pour définir une suite entre 2 caractères, ils faut utiliser le caractère –
    - Exemple : [0-9]

Les expressions régulières

[abc]	a, b ou c
[a-z]	une lettre minuscule
[0-57]	0, 1, 2, 3, 4, 5 ou 7
[a-d5-8X]	a, b, c, d, 5, 6, 7, 8 ou X
[0-5-]	0, 1, 2, 3, 4, 5 ou –
[^0-9]	pas un chiffre
[012^]	0, 1, 2 ou ^

Les Modificateurs

- **Les modificateurs déterminent le nombre de répétition des caractères précédents**
  - ***** zéro ou plus caractère précédent
  - **\+** un ou plus caractère précédent
  - **\?** zéro ou un caractère précédent
  - **\{i\}** exactement i caractère précédent
  - **\{i,\}** i ou plus caractère précédent
  - **\{i,j\}** de i à j caractère précédent

Les Ancres

- Les ancres comparent le début ou la fin d'une ligne ou d'un mot
  - ^ la ligne commence avec
  - $ la ligne se termine avec
  - \< le mot commence avec
  - \> le mot se termine avec

Combinaisons regex

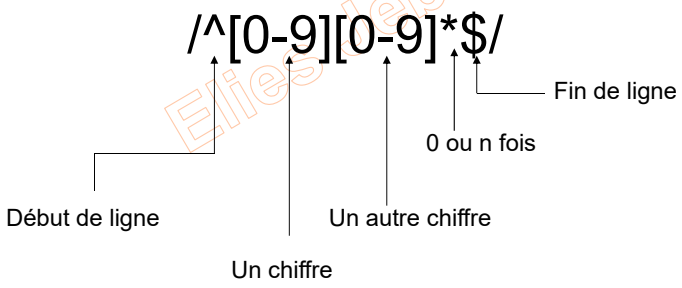
- Les expressions régulières sont les plus utiles en les combinant entre elles
  - .* zéro ou plus tout caractère
  - [a-z]* zéro ou plus lettres
  - \<cat\> le mot 'cat'
  - ab..ef ab et ef séparés de deux caractères
  - .\{32\} 32 fois tout caractère
  - * un astérisque littéral

Expressions Régulières- Exemples

- À quoi correspondent les modèles suivants?
  1. El.es
  2. El[iy]es
  3. www\.redhat\.com
  4. ^#!
  5. \<the
  6. ^[a-z0-9]\{28\}$
  7. ^^Yipes!$$

Les expressions régulières

- Exercice : définir l'expression régulière permettant la caractérisation d'une ligne qui ne contient que des chiffres et non vide.



Les Guillemets et les Expressions Régulières

- Sur la ligne de commande, entourez les les expressions régulières de guillemets
- Les caractères de génération de noms de fichiers doivent rester sans guillemets
- N'utilisez pas de guillemets dans les expressions régulières à l'intérieur de commandes

La commande grep

- GREP = *General Regular Expression Parser*, soit analyseur général d'expression régulière
- Cette commande recherche dans des fichiers ou l'entrée standard les lignes qui correspondent à l'expression régulière.
- Syntaxe

```
grep [options] expreg [fichiers]
```

- Exemples :
  - grep "korn" fichier
  - cat /etc/passwd | grep "*user*"

La commande grep

- Quelques options utiles

-c	donne seulement le nombre de lignes trouvées obéissant au critère
-l	donne seulement le nom des fichiers où le critère a été trouvé
-v	donne les lignes où le critère **n'a pas** été trouvé
-i	ne pas tenir compte de la casse
-n	pour n'afficher que les numéros des lignes trouvées
-w	pour imposer que le motif corresponde à un mot entier d'une ligne

La commande sed

- La commande sed permet d'éditer du texte de manière non-intéractive.
- Elle permet d'exécuter un certain nombre de commandes sur du texte en entrée standard ou dans un fichier.
- Il n'y a pas d'altération du texte d'origine, les modifications sont affichées sur la sortie standard.

La commande sed

- Syntaxe

```
sed [-n] [-e commande] [-f fichier de commandes] [fichier]
```


`-n` écrit seulement les lignes spécifiées (par l'option /p) sur la sortie standard

`-e` permet de spécifier les commandes à appliquer sur le fichier. Cette option est utile lorsque vous appliquez plusieurs commandes. Afin d'eviter que le shell interprette certains caracteres, il faut mieux encadrer la commande avec des ' ou des " .

`-f` lecture des commandes à partir d'un fichier.

La commande sed

- Fonctionnement
  - Pour chaque ligne du fichier d'entrée, la commande est exécutée. La ligne, modifiée ou non, est affichée sur la sortie standard.
- Syntaxe d'une commande
  - "liste_des_adresses commande"
  - Avec la liste_des_adresses :

rien	Toutes les lignes
Num	La ligne num (dernière ligne : $)
Num1,Num2	Les lignes entre num1 et num2

La commande sed

- La substitution (commande **s**)
  - Syntaxe
    `num1,num2s/RE/remplacement/flags`
  - Entre les lignes num1 et num2 : remplace les chaînes caractérisées par l'expression régulière RE par la chaîne de remplacement
  - Flags :
    - g : toutes les occurences sur une même ligne
    - p : affiche la ligne traîtée (pratique avec –n)
    - w fichier : écrit la sortie dans le fichier spécifié
  - Exemple :
    ```
    sed "s/[Cc]omputer/COMPUTER/g" fichier
    ```

Utiliser sed

- Instructions de recherche et remplacement entre guillemets!
- Adresses sed
  - sed 's/dog/cat/g' pets
  - sed '1,50s/dog/cat/g' pets
  - sed '/digby/,/duncan/s/dog/cat/g' pets
- Multiples instructions sed
  - sed -e 's/dog/cat/' -e 's/hi/lo/' pets
  - sed -f myedits pets

La commande sed

- Les tampons
  - Ils permettent d'isoler des sous-chaînes afin de pouvoir les réutiliser.
  - Isoler une chaîne : \(\)
  - Réutiliser une chaîne : \1, \2, …
  - Il est possible d'isoler des sous-chaînes récursivement
- Exercice : que fait cette commande ?
- date +'%d/%m/%Y' | sed -e 's/\([0-9]*\)\/\([0-9]*\)\/\([0-9]*\)/\2-\1-\3/'

```
"s/\([0-9][0-9]*\)/**\1**/"
```

Elle encadre les chiffres/nombres trouvés sur une ligne par les caractères **

```
Computer=599 euros -> Computer=**599** euros
```

La commande sed

- La suppression (commande **d**)
  - Efface les lignes (au niveau de la sortie, le fichier d'origine n'est pas modifié)
  - Exemple :
    `sed "1,10d" fichier` : sortie du fichier à partir de la onzième ligne

La commande sed

- Autres commandes
  - a,i : permettent d'insérer du texte dans le résultat
  - q : quitte
  - = : écrit les numéros de ligne

- Exercice : Afficher le résultat numérique contenu dans cette chaîne sous la forme "resultat = xxx"
  - chaîne de départ : `edtf fta 158k frd`

```
sed -e 's/[^0-9]*\([0-9][0-9]*\).*/resultat=\1/'
```

La commande awk (overview)

- La commande awk permet d'effectuer un certain nombre d'action sur un fichier.
- La syntaxe s'inspire du C.
- Un programme awk est une suite de commandes de la forme :

  `motif {action}`

  Le motif permet de déterminer sur quel enregistrement doit s'effectuer l'action
- Exemple :

  `awk 'length($0)>75 {print}' fichier`

  Imprime les lignes de plus de 75 caractères

This would print the usernames of all users with UID > 1000:

getent passwd | awk -F: '$3 > 1000 {print $1}'
And this would just print found if at least one such is found:

getent passwd | awk -F: '$3 > 1000 {print "found"; exit}'