Administration Linux

Clément Weill

105 Shells et Scripts Shell

105.1 Personnalisation et utilisation de l'environnement du shell
105.2 Personnalisation ou écriture de scripts simples

105 Shells et Scripts Shell

105.1.a Personnalisation et utilisation de l'environnement du shell

Introduction

- **Bash (Bourne Again SHell)**
  - Bash est un interpréteur en ligne de commande de type script. C'est le shell Unix du projet GNU
  - Une fois lancé, la première chose que Bash (et les autres interpréteurs aussi) est d'exécuter un ensemble de scripts de démarrage. Ces scripts peuvent être globaux ou propre à un utilisateur
  - C’est là que sont définis les variables d'environnement, les alias et les fonctions

Différent Types de Shell

- Shells Interactif / Non-Interactif
  - L'utilisateur fournit des inputs avec le clavier, et le shell des outputs en envoyant des messages dans le terminal
- Shell Login / Non-login
  - L'utilisateur fournit un identifiant et un mot de passe

Exemple:

- *Shell Interactif Login :* Quand un utilisateur se connecte au système
- *Shell Interactif Non-Login :* Tous les terminaux ouvert après s'être connecter
- *Shell Non-Interactif Login :* `/bin/bash --login <script>` OU `<commande> | shh <user>@<server>`
- *Shell Non-Interactif Non-login :* Tâches répétitives d'administration ou de maintenance, comme celle que l'on trouve dans les cronjobs

Ouvrir un Terminal

- **Deux types de shell**
  - `pts` (pseudo terminal slave) quand il est ouvert depuis un émulateur terminal dans GUI, comme *gnome-terminal* ou *konsole*
  - `tty` (teletypewritter) quand il est lancé depuis une console système

Exemple:

`Ctrl + Alt +F1-F6` ouvre une console avec un shell interactif de login

`Ctrl + Alt +F7` ramène au bureau principal

Lancé un Shell avec `bash`

Apres s'être connecté, on peut lancer un nouveau shell en utilisant la commande `bash` qui sera une processus enfant du shell actuel

- `bash -l` ou `bash --login`
  - invoke un shell login
- `bash -i`
  - invoke un shell interactif
- `bash --noprofile`
  - avec un shell login, ignore les fichiers de paramètres globaux `/etc/profile` et ceux de l'utilisateur `~/.bash_profile`, `~/.bash_login` et `~/.profile`

Lancé un Shell avec `bash`

- `bash --norc`
  - avec un shell interactif, ignore les fichiers de paramètres globaux `/etc/bash.bashrc` et ceux de l'utilisateur `~/.bashrc`
- `bash --rcfile <file>`
  - avec un shell interactif, prend le `<file>` comme fichier de paramètres et ignore les fichiers de paramètres globaux `/etc/bash.bashrc` et ceux de l'utilisateur `~/.bashrc`

Lancé un Shell avec su et sudo

- su
  - su - user2 , su -l user2 ou su --login user2 démarre un shell login interactif en tant que user2
  - su user2 démarre un shell non-login en tant que user2
  - su - root ou su - démarre un shell interactif login en tant que root
  - su root ou su démarre un shell interactif non-login en tant que root

Lancé un Shell avec su et sudo

- sudo

Exécute une commande comme un autre utilisateur, souvent root et donc il est nécessaire que l'utilisateur qui l'utilise soit dans le fichier sudoers

Pour ajouter un utilisateur au sudoers, il faut être root et éxecuter la commande suivante :

```
usermod -aG sudo user2
```

Quelques exemples:

- sudo -u user2 -s demarre un shell interactif non-login en tant que user2
- sudo -i démarre un shell interactif login en tant que root
- sudo -s ou sudo -u root -s démarre un shell non-login en tant que root

Quels Shell avons nous?

Pour savoir dans quel shell nous évoluons, il suffit de taper `echo $0`

- Interactif Login
  - `-bash ou -su`
- Interactif Non-Login
  - `/bash ou /bin/bash`
- Non-Interactif Non-login (scripts)
  - `<nomDuScript>`

Pour savoir combien de `bash` shell tourne sur une machine, on peut utiliser

```
ps aux | grep bash
```

Les Fichiers de Paramètrage

Shell Interactif Login

- **Niveau Global**
  - /etc/profile
    - A travers une série de if, ce fichier attribue un certains nombre de variable tel que PATH et PS1 ainsi que le fichier /etc/bash.bashrc
  - /etc/profile.d/*
    - Contient les scripts qui sont exécuté par /etc/profile

Les Fichiers de Paramètrage

Shell Interactif Login

Niveau Local

- `~/.bash_profile`
  - Configure l'environnement utilisateur. Peut aussi etre utiliser pour sourcer `~/.bash_login` et `~/.profile`
- `~/.bash_login`
  - Est utilisé si il n'y a pas de fichier `~/.bash_profile`
- `~/.profile`
  - N'est appelé quand l'absence des deux précédents. Si un shell bash est utilisé, source `~/.bashrc` si il existe. Ajoute `~/bin` au `PATH`
- `~/.bash_logout`
  - S'il existe, est utilisé pour le nettoyage à la fermeture du shell

Les Fichiers de Paramètrage

Shell Interactif Login

- **Exemple**
  - echo 'echo Hello from /etc/profile' >> /etc/profile
  - echo 'echo Hello from ~/.profile' >> ~/.profile
  - Puis utiliser ssh pour lancer un shell interactif login

Les Fichiers de Paramètrage

Shell Interactif Non-Login

- **Niveau Global**
  - `/etc/bash.bashrc`
    - Vérifie que le bash est bien en mode interactif, vérifie la taille de la fenêtre après chaque commande et change les valeurs de `LINES` et `COLUMNS` si besoin. Il définit aussi quelques variables.
- **Niveau Local**
  - `~/.bashrc`
    - Similaires au fichier précédent mais au niveau utilisateur. Il définit aussi des variables historique ainsi que de sourcer `~/.bash_aliases`. Il est aussi utilisé pour définir des alias et des fonctions spécifiques à un utilisateur.
    - Il est aussi implémenté si `bash` détecte que `<stdin>` est une connexion réseau.

Les Fichiers de Paramètrage

Shell Interactif Non-Login

- **Exemple**
  - echo 'echo Hello from /etc/bash.bashrc' >> /etc/bash.bashrc
  - echo 'echo Hello from ~/.bashrc' >> ~/.bashrc
  - Puis utiliser bash pour lancer un shell interactif non-login

Attention : au vu de l'ordre d'exécution des fichiers, les fichiers locaux ont le précédent sur les fichiers globaux.

Les Fichiers de Paramètrage

Shell Non-Interactif Non-Login (Scripts)

Les scripts ne lisent aucun des fichiers précédemment mentionné, mais cherche dans l'environnement la variable `BASH_ENV`.

Ils l’utilisent comme nom de fichier de paramétrage pour lire et exécuter les commandes.

Les Fichiers de Paramètrage

En pratique, `/etc/profile` et `~/.profile` vérifie que `/etc/bash.bashrc` et `~/.bashrc` on bien été exécuté après un login réussi.

Pour s'en assurer, on peut exécuter la commande `su - <user>`

On devrait voir `/etc/profile` sourcer `/etc/bash.bashrc`

puis `/etc/profile` qui fini et a bien été exécuté entièrement

Ensuite `~/.profile` devrait sourcer `~/.bashrc`

enfin `~/.profile` qui fini et a bien été exécuté entièrement

Sourcer un Fichier

L’opérateur . (point) est traditionnellement trouvé dans les fichiers de paramétrage. On peut aussi utiliser `source`

Exemple :

rajoutons un aliase dans `~/.bashrc` avec la commande suivante

```
echo “alias hi=’echo We salute you.’” >> ~/.bashrc
```

puis sourcons le fichier nous même avec

`. ~/.bashrc` ou `source ~/.bashrc`

Les Origines des Fichiers de Paramétrage

`SKEL` est la variable qui pointe vers le répertoire `/etc/skel`.

C'est le template qui sert à la création d'un nouvel utilisateur. Il contient l'ensemble des fichiers qui seront hérité par le nouvel utilisateur.

La variable `SKEL` est stocké dans `/etc/adduser.conf` qui est le fichier de configuration pour `adduser`, dont les fichiers de paramétrage.

On peut donc y ajouter ou modifier des fichiers qui seront présent pour tous les futures utilisateurs.

Exercices

105 Shells et Scripts Shell

105.1.b Personnalisation et utilisation de l'environnement du shell

Variables

Une variable peut être définit comme un nom contenant une valeur.

Assigner une variable, c'est en la définir la valeur.
Référencer une variable, c'est accéder à la valeur.

La syntaxe pour assigner une variable est :

```
<nomDeLaVariable>=<valeurDeLaVariable>
```

Pour référencer une variable, il faut utiliser l'opérateur `$`

```
echo $<nomDeLaVariable>
```

Les seuls caractères autorisés dans le nom d'une variable sont `a-z`, `A-Z`, `0-9` et `_` (underscore) et les noms ne peuvent pas commencer par un chiffre.

Variables

Une variable peut contenir l'ensemble des caractères alphanumérique `a-z`, `A-Z`, `0-9`, ainsi que la plupart des caractères spéciaux.

Elles peuvent aussi contenir des espaces si on l'encapsule avec des guillemets. De même si elle contiennent des symboles de redirection (`<`,`>`) ou de pipe (`|`).

Attention les guillemets simples et doubles ne sont pas toujours interchangeable.

Les guillemets simples prennent la valeur *littérale*, quand les guillemets doubles autorisent les substitution de variable.

Variables

Exemple :

```
$ lizard=uromastyx
$ animal='My $lizard'
$ echo $animal
My $lizard
$ animal="My $lizard"
$ echo $animal
My uromastyx
```

Variables

Une variable qui contient des espaces au début, ou plusieurs à la suite, doit être référencé avec des doubles guillemets, pour éviter *field splitting* et *pathname expansion.*

Exemple :

```
$ lizard="    genus    |    uromastyx"
$ echo $lizard
genus | uromastyx
$ echo "$lizard"
    genus    |    uromastyx
```

On peut utiliser un contre-slash (\) pour échapper à l'interprétation d'un caractère spécial.

Variables Shell ou Local

Une variable shell ou local n'existe que dans le shell où elle a été créée. La convention veut qu'elles soit écrite en lettres minuscules.

On peut utiliser `readonly` pour rendre une variable immuable.

On peut aussi utiliser `readonly -p` pour lister l'ensemble des variables immuable de la session en cours.

la commande `set` permet de lister toutes les variables et fonctions de la session en cours. Essayez `set | less`.

Pour libere une variable, on peut utiliser la commande `unset` avec le nom de la variable sans le symbol `$`

Variables Globale ou d'Environnement

Une variable globale ou d'environnement existe dans la session en cours, ainsi que toutes les processus qui en sont issue. La convention veut qu'elles soit écrite en lettres majuscules.

Pour transformer une variable locale en une variable global, on peut utiliser `export`.

On peut aussi utiliser `export` ou `export -p` pour lister l'ensemble des variables globale de la session en cours.

la commande `env` et `printenv` permet de lister toutes les variables de l'environnement.

Variables Globale ou d'Environnement

la commande `env` peut aussi permettre de modifier un environnement au moment de l'exécution.

Pour lancer une nouvel session bash avec un environnement le plus vide possible, on peut utiliser l'option `-i` :

```
env -i bash
```

On a vu précédemment que les scripts utilisent la variable `BASH_ENV` pour définir l'environnement.

On peut aussi l'utiliser pour lancer un script dans un environnement particulier.

```
env BASH_ENV=<some_startup_script> test.sh
```

Variables d'Environnement

- DISPLAY

  en relation avec le serveur X
  - le hostname (son absence signifie localhost)
  - deux point pour séparer
  - un nombre (souvent 0 et fait référence à la sortie écran)

- HISTCONTROL

  contrôle ce qui est sauvé dans le HISTFILE avec trois valeurs possible
  - ignorespace les commandes commencant par un espace ne sont pas sauvé
  - ignoredups les commandes identiques successives ne sont pas sauvés
  - ignoreboth les deux précédents

- HISTSIZE

  contrôle le nombre de commande enregistrées en mémoire pour la session en cours

Variables d'Environnement

- `HISTFILESIZE`
  contrôle le nombre de commande enregistrées dans `HISTFILE` au début et à la fin de la session
- `HISTFILE`
  nom du fichier où sont sauvé les commandes tapées.
- `HOME`
  stocke le chemin absolu du répertoire de l'utilisateur actuel, équivalent à `~`
- `HOSTNAME`
  stocke le nom TCP/IP de la machine qui héberge la session
- `HOSTTYPE`
  stocke l'architecture du processeur de la machine
- `LANG`
  stocke la langue local du système

Variables d'Environnement

- LD_LIBRARY_PATH
  stocke l'ensemble des répertoires, séparés par deux points, où sont partagé les librairies partagées
- MAIL
  stock le fichier que bash utilise pour chercher les mails
- MAILCHECK
  stocke une variable numérique qui indique la fréquence en seconde où bash vérifie les mails
- PATH
  stocke la liste des répertoires où bash cherche les fichiers exécutables, séparé par deux points.

Variables d'Environnement

- `PS1`
  stocke la valeur de l'invite bash
- `PS2`
  normalement fixé à `>` et utilisé pour la continuation de l'invite pour les commandes multilignes
- `PS3`
  utilisé comme l'invite de commande pour `select`
- `PS4`
  normalement fixé à `+` et utilisé pour le débugging
- `SHELL`
  stock le chemin absolu du shell en cours
- `USER`
  stock le nom de l'utilisateur courant

Exercices

105 Shells et Scripts Shell

105.1.c Personnalisation et utilisation de l'environnement du shell

Création d'Alias

Un alias est nom de substitution pour une autre commande. Pour définir un alias, la syntaxe est la suivante:

```
alias alias_name=command(s)
```

Exemple :

```
alias ls='ls --color=auto'
```

```
alias git_info='which git;git --version'
```

La commande `alias` fournit une liste des alias disponible sur le système.

La commande `unalias` permet de supprimer un alias.

On peut échapper à un alias avec un contre-slash (`\`) , ce qui est utile pour les alias avec le même nom qu'une commande existante.

Evaluations des guillemets

Avec des guillemets simples, l'expansion est dynamique :

```
$ alias where?='echo $PWD'
$ where?
/home/user2
$ cd Music
$ where?
/home/user2/Music
```

Evaluations des guillemets

Mais avec des guillemets doubles, l’expansion est statique:

```
$ alias where?=”echo $PWD”
$ where?
/home/user2
$ cd Music
$ where?
/home/user2
```

Persistance des Alias

Tout comme les variables, pour rendre un alias persistent, il faut l'ajouter à un fichier de paramétrage.
Un bon candidat est `~/.bashrc`, où l'on peut déjà trouver des alias commenter prêt à être utilisé.

Essayez : `grep alias .bashrc`

On peut aussi les stocker dans `~/.bash_aliases` qui est sourcé par `~/.bashrc`

Création de Fonctions

En comparaison des alias, les fonctions sont plus programmable et flexible, et fonctionne bien avec des boucles, de conditions ou des structures de contrôle du flux.

Il existe deux syntaxe pour créer des fonctions:

- Le mot-clé `function`

```
function function_name {command #1;…;command#n}
```

- L'operateur `()`

```
function_name() {command #1;…;command#n}
```

Création de Fonctions

Tout comme les variables et la alias, pour rendre une fonction persistante, il faut l'ajouter dans un des fichiers de paramétrage tel que `/etc/bash.bashrc` (global) ou `~/.bashrc` (local).

Attention : pour qu'une fonction ou un alias soit pris en compte, il faut sourcer le fichier en question, ou redémarrer la machine.

Fonctions Spécial de Bash

Bash vient avec un ensemble de fonction pré-établie, qui ne peuvent que être référencé, mais pas assigné.

- `$?`

  Cette variable renvoi le résultat de la commande précédente, `0` si c'est un succès, et plus en cas d'erreur.

- `$$`

  renvoi le PID du shell

- `$!`

  renvoi le PID du dernier job en background

- `$0` jusqu'à `$9`

  renvoi les paramètres passés à la fonction ou l'alias, `$0` étant le nom du script ou du shell

Fonctions Spécial de Bash

- `$#`

  renvoie le nombre d'arguments passés à la commande

- `$@`,`$*`

  renvoi les arguments passés à la commande

- `$_`

  renvoi le dernier paramètre utilisé, ou le nom du script précédent

Variables dans une Fonction

On peut évidemment utiliser des variables dans une fonction.
Exemple :

Dans un fichier vide appelé `funed`, mettons la fonction suivante:

```
editors() {
editor=vim
echo "The text editor of $USER is : $editor."
}
editors
```

Il faut sourcer le fichier pour pouvoir invoquer la fonction

```
. funed
```

Variables dans une Fonction

Exemple :

Dans un fichier vide appelé funed, mettons la fonction suivante:

```
editors() {
editor=vim
echo "The text editor of $USER is : $editor."
echo “Bash is not a $1 shell”
}
editors turtle
```

Il faut sourcer le fichier pour pouvoir invoquer la fonction

```
. funed
editors tortoise
```

Fonctions dans un Script

Transformons `funed` en un script shell (`funed.sh`):

```
#!/bin/bash

editors() {

editor=emacs

echo "The text editor of $USER is: $editor."

echo "Bash is not a $1 shell."

}

editors turtle
```

Fonctions dans un Script

Il suffit de deux lignes seulement:

- la première ligne définit le programme qui va interpréter le script, c'est le *shebang.* Ici, `bash`.
- la dernière ligne qui sert à invoquer la fonction

Il suffit maintenant de rendre le script exécutable:

```
chmod +x funed.sh
```

puis de l'exécuter

```
./funed.sh
```

Fonction dans un Alias

Exemple :

```
alias great_editor='gr8_ed() { echo $1 is a great text editor; unset -f gr8_ed; }; gr8_ed'
```

D’abord la fonction elle-même : `gr8_ed() { echo $1 is a great text editor; unset -f gr8_ed; }`

La dernière commande de la fonction, `unset -f gr8_ed`, supprime la fonction pour ne pas qu’elle reste dans le bash après l’appelle de l’alias

Finalement, on invoque la fonction à la fin de l’alias : `gr8_ed`

Exercices

105 Shells et Scripts Shell

105.2.a Personnalisation ou écriture de scripts simples

Introduction

- **Scripts**
  - Fichiers textes qui se comportent comme un programme, c’est l'interpréteur qui lis et exécute les commandes.
  - Automatisent la plupart des tâches de maintenance du système.
  - Utile pour les tâches répétitives, comme renommer une quantité importantes de fichier, collecté ou analyser des données, etc.
  - Étant des fichiers textes, ils peuvent être crée ou modifié avec des éditeur de texte.

Structure et Exécution des Scripts

- Une séquence ordonnée de commandes
  - Comment l'interpréteur lis les commandes peut varier, et l'interpréteur par défaut est indiqué dans la première ligne du fichier, juste après le ***shebang*** `#!`
  - Toutes les autres lignes qui commence par `#` ou les lignes vides sont ignorées.

Exemple avec `script.sh`**:**

```
#!/bin/bash

#Un script simple

echo -n “Bonjour depuis le script, il est : “

date +%H:%M
```

Structure et Exécution des Scripts

On peut lancer le fichier précédent avec Bash en utilisant :

```
$ bash script.sh
```

```
Bonjour depuis le script, il est 9:26
```

Remarque, le suffix `.sh` n'est pas nécessaire, mais il est utile pour lister et chercher les scripts existant.

De plus, Bash appelle la commande après `#!` pour interpréter le fichier. Par exemple, cela fonctionne pour *Python* (`#!/usr/bin/python`), *Perl* (`#!/usr/bin/perl`), ou *awk* (`#!/usr/bin/awk`).

Structure et Exécution des Scripts

Si d’autres utilisateurs du système vont utiliser le script, il est important de vérifier qu’il peut être lu de tous. Par exemple `chmod o+r script.sh`, ou `chmod +x script.sh`.

Si le bit d'exécution est activé, alors le script peut être lancé avec `./script.sh`.

Attention : Un script qui opère des actions restreintes peut avoir ses permissions SUID activées, pour qu’un utilisateur lambda puisse l'exécuter. Il est alors vitale de vérifier que personne n’a les droits d'écrire sur le fichier.

Structure et Exécution des Scripts

Dans un script shell, il y a une équivalence entre `\n` (newline) et `;`.

Quand un script est exécuté, il est lancé dans un nouveau processus Bash, appelé un ***sub-shell***. Ca évite de modifier l'environnement de la session en cours, comme les variables, les fonctions, ou les alias.

Si on souhaite exécuter un script dans la session en cours, il faut le sourcer avec `source sript.sh` ou `. script.sh`.

Structure et Exécution des Scripts

Comme avec l'exécution de n’importe quelle commande, l’invite de commande n’est disponible qu'après la fin de l'exécution du script, et le code de sortie est disponible dans `$?`.

Pour que le shell se ferme à la fin de l'exécution du script, on peut utiliser `exec` qui remplace le code de sortie du shell en cours par celui de l’argument.

Variables

Les variables dans un script shell se comporte de la même manière que dans une session interactive. Par convention, elles sont nommé en caractères majuscules.

Les scripts Bash ont accès à un ensemble de variables spécials appelé *paramètres*. (`$*`,`$@`,`$#`,`$0`,`$!`,`$$`,`$?`)

Les paramètres de position sont les arguments passé au lancement du script, `$1` étant le premier, etc. Si la position est plus grande que 9, il faut utiliser des accolades, tel que `${11}`.

Variables

Les variables ordinaires sont souvent utilisé pour stocker des valeurs insérés manuellement, ou des résultats généré par d'autres commandes.

La commande `read` permet de demander de l'input de l'utilisateur pendant l'exécution.

Exemple:

```
echo "Continuer  ? (oui/non)"

read REPONSE
```

Si aucune variable n'est fourni, la réponse est stocké dans `REPLY`.

Variables

Il est aussi possible de passer plusieurs variables simultanément avec `read`.

Exemple:

```
echo “Veuillez entrer vos nom et prénom :”

read NOM PRENOM
```

Chaque espace sert de séparateur, et si le nombre d’arguments donnée est trop important, l'excédent est stocké dans la dernière variable.
En utilisant l’option `-p` avec `read`, on peut aussi afficher un message à l’utilisateur,ce qui rend `echo` redondant.

Exemple:

```
read -p “Veuillez entrer vos nom et prénom :” NOM PRENOM
```

Variables

Les scripts utilisé dans les tâches système on souvent besoin d'information donnée par d'autres programmes. La *notation backtick* permet de stocker le résultats d'une commande dans une variable.

Exemple:

```
OS=`uname -o`
```

De manière similaire, on peut utiliser `$( )`:

```
OS=$(uname -o)
```

Variables

La longueur d'une variable, c'est à dire le nombre de caractères contenu dedans, peut être récupéré en utilisant `#{VARIABLE}`.

Exemple:

```
$ OS=$(uname -o)
$ echo $OS
GNU/Linux
$ echo ${#OS}
9
```

Variables

Bash permet aussi l'utilisation de tableau unidimensionnel. Chaque éléments du tableau a un index numérique qui permet de lire ou d'écrire sur l'élément correspondant. Il faut utiliser la commande `declare`.

Exemple:

```
declare -a SIZES
```

On peut aussi implicitement déclaré un tableau en utilisant des parenthèses et en remplissant.

```
SIZES=( 1048576 1073741824 )
```

Variables

Pour récupérer les valeurs stocké, il faut utiliser des accolades et des crochets. Le premier élément est `${SIZES[0]}`, le deuxième `${SIZES[1]}`, etc.

```
$ echo ${SIZES[1]}
```

```
1073741824
```

Pour changer un élément du tableau, il suffit d'utiliser les crochets.

```
SIZES[0]=1048576
```

La longueur d'un tableau est retourné avec soit `@` ou `#` comme index :

```
$ echo ${#SIZES[@]}
```

```
2
```

Variables

On peut aussi déclarer un tableau en utilisant la sortie d'une commande, en utilisant la substitution de commande.

Dans l'exemple suivant, on crée un tableau contenant l'ensemble des fichiers supporté par le système :

```
$ FS=( $(cut -f 2 < /proc/filesystems) )
```

Par défaut, les éléments séparé par un ***espace***,une ***tabulation***, ou un ***retour à la ligne*** deviennent des éléments distincts du tableau.

Les délimiteurs de Bash sont définie dans `$IFS` (*Input Field Separator*). En changeant cette variable d'environnement, on change le délimiteur.

Expressions Arithmétiques

Bash donne une manière simple d'opérer des opérations arithmétiques avec la commande `expr`.

Exemple:

```
$ SUM=`expr $VAL1 + $VAL2`
```

Elle peut aussi être remplacé par `$(( ))`, ce qui donnerait :

```
$ SUM=$(( $VAL1 + $VAL2 ))
```

Pour les puissances, il faut utiliser l'opérateur double astérisques.

Donc `SIZES=( 1048576 1073741824)` aurait pu aussi être déclaré comme `SIZES=( $((1024**2)) $((1024**3)) )`.

Expressions Arithmétiques

La substitution de commande peut aussi être utilisé dans les expressions arithmétiques.

Par exemple, le fichier `/proc/meminfo` contient des informations détaillées sur la mémoire du système, et en autres le nombre de bytes libres dans la RAM.

```
$ FREE=$(( 1000 * `sed -nre '2s/[^[:digit:]]//gp' < /proc/meminfo` ))
```

Dans cette exemple, `sed` est utilisé pour analyser le contenu du fichier. La deuxième ligne de `/proc/meminfo` contient la mémoire libre en milliers de bytes.

Exécution Conditionnelle

En séparant des commandes avec `&&`, la commande de droites n'est exécuté que si celle de gauche est sortie sans erreur, c'est à dire avec un code de sortie de `0`.

Exemple:

```
COMMANDE A && COMMANDE B && COMMANDE C
```

C'est exactement l'inverse avec `||`, c'est à dire que si la commande à un code de sortie différent de `0`.

La meilleure façon de créer une exécution conditionnelle est d'utiliser la commande `if`, qui exécute une ou plusieurs commandes seulement si la commande qui lui ai donné en argument retourne `0`.

Exécution Conditionnelle

La commande `test` peut être utilisé pour vérifier un ensemble de critère, et est très souvent utilisé en conjonction avec `if`.

Exemple:

```
if test -x /bin/bash ; then
    echo "Confirmed: /bin/bash is executable."
fi
```

On peut aussi remplacer la commande `test` par des crochets:

```
if [ -x /bin/bash ] ; then
    echo "Confirmed: /bin/bash is executable."
fi
```

Exécution Conditionnelle

L'instruction `else` est une option de la structure `if`, et permet de définir une séquence de commandes à exécuter si la condition est fausse.

Exemple:

```
if [ -x /bin/bash ] ; then
    echo "Confirmed: /bin/bash is executable."
else
    echo "No, /bin/bash is not executable."
fi
```

Attention, la structures `if` doit se terminer avec un `fi`.

Sortie de Script

Il est important d'informer l'utilisateur de la progression du scripts, et des potentiels problèmes.
La commande `echo` est souvent utilisé à cette effet. Avec l'option `-e`, la commande `echo` peut afficher des caractère spéciaux en utilisant des séquences échappées.

- `\n` pour insérer un retour à la ligne (*newline*)
- `\t` pour insérer une tabulation (*tab*)

Exemple:

```
echo -e "Operating system:\t$OS\nUnallocated RAM:\t$(( $FREE / 1024**2 )) MB"
```

Sortie de Script

La commande `printf` permet d'avoir plus de control que `echo`.

La commande `printf` utilise le premier argument pour formater la sortie, avec des variables qui seront remplacé par les arguments correspondants dans leurs ordres d'apparitions.

Exemple:

```
printf "Operating system:\t%s\nUnallocated RAM:\t%d MB\n" $OS $(( $FREE / 1024**2 ))
```

Avec `%s` servant pour les variables texte (*string*) et `%d` pour les nombres (*int*).

De plus `printf` ne rajoute pas de retour à la ligne en fin de texte, c'est pourquoi il faut rajouter `\n`.

Sortie de Script

Avec printf, les variables sont placé en dehors du texte, ce qui permet de stocker le texte dans une variable séparé :

```
MSG='Operating system:\t%s\nUnallocated RAM:\t%d MB\n'

printf "$MSG" $OS $(( $FREE / 1024**2 ))
```

Ce qui permet de d'afficher différent format de message selon les besoin de l'utilisateur.

Exercices

105 Shells et Scripts Shell

105.2.b Personnalisation ou écriture de scripts simples

Tests Étendus

Bash est un langage script conçu principalement pour travailler sur des fichiers. C'est pourquoi la commande `test` a autant d'options pour évaluer les propriétés des objets du système de fichiers (*File System*).

Essayez: `man test`

Il est recommandé de placer la variable testé entre guillemets, pour éviter de causer une erreur de syntax avec `test` si elle est vide.

Attention, dans une comparaison alphabétique, la langue du système peut avoir un impacte. Il est donc conseillé de mettre la variable d'environnement `LANG` à `C`, soit `LANG=C`, avant de faire la comparaison. Cela conservera la langue du système pour les messages, mais ne devrait être utilisé que dans un script.

Tests Étendus

Une autre construction conditionnelle, `case`, peut être vu comme une variation de `if`.

La commande `case` exécute une list de commandes si un élément spécifié se trouve dans une list d'éléments séparé par des barres verticales `|` (*pipes*) et terminé par `)`.

Exemple:

```
#!/bin/bash

DISTRO=$1

echo -n "Distribution $DISTRO uses "

case "$DISTRO" in
```

Tests Étendus

```
        debian | ubuntu | mint)
        echo -n "the DEB"
    ;;
        centos | fedora | opensuse )
        echo -n "the RPM"
    ;;
        *)
        echo -n "an unknown"
    ;;
esac
echo " package format."
```

Tests Étendus

Chaque liste de patterns et de commandes associées doivent se terminer par ;;, ;&, ou ;;&.

L'asterisk correspond uniquement si aucun des patterns précédent a correspondu.

L'esac permet de terminer la construction du case, tout comme avec la construction du if.

Bash a une option appelé nocasematch qui permet d'activer ou non la sensibilité à la casse (e.g. Majuscule ou minuscule) pour case et d'autres commandes conditionnelle.

Tests Étendus

La commande `shopt` qui permet de changer les valeurs des paramètres contrôlant le comportement du shell.

`shopt -s` permet d'activer l'option passé en paramètres, et `shopt -u` de la désactiver.

Exemple:

`shopt -s nocasematch` permet de rendre la construction de `case` insensible à la casse.

Les options modifier par `shopt` n'affecte que la session en cours, et donc dans un script, que le *sub-shell* dans lequel elle est exécutée.

Construction de Boucle

Bash a trois instructions pour construire des boucles: for, until, et while.

L'instruction for parcourt une liste d'éléments, le plus souvent séparé par des espaces (voir $IFS), et exécute un même ensemble de commandes pour chacun des éléments.

A chaque itération, l'instruction for assigne l'élément courant à une variable, qui peut être utilisé dans les commandes.

La syntaxe du for est la suivante:

```
for VARNAME in LIST
do
    COMMANDS
done
```

Construction de Boucle

```
#!/bin/bash
for NUM in 1 1 2 3 5 8 13
do
        echo -n "$NUM is "
        if [ $(( $NUM % 2 )) -ne 0 ]
        then
                echo "odd."
        else
                echo "even."
        fi
done
```

Construction de Boucle

Bash permet aussi une construction alternative du for, avec la notation double parenthèses.

Cette construction est similaire à la syntax C, et est particulièrement utile pour travailler sur un tableau.

Construction de Boucle

```
#!/bin/bash
SEQ=( 1 1 2 3 5 8 13 )
for (( IDX = 0; IDX < ${#SEQ[*]}; IDX++ ))
do
        echo -n "${SEQ[$IDX]} is "
        if [ $(( ${SEQ[$IDX]} % 2 )) -ne 0 ]
        then
                echo "odd."
        else
                echo "even."
        fi
done
```

Construction de Boucle

De manière similaire, l’instruction `until` exécute une séquence de commandes jusqu'à ce que la commande test se termine avec `0`.

En reprenant l’exemple précédent:

```
#!/bin/bash

SEQ=( 1 1 2 3 5 8 13 )

IDX=0

until [ $IDX -eq ${#SEQ[*]} ]

do
```

Construction de Boucle

```
echo -n "${SEQ[$IDX]} is "
if [ $(( ${SEQ[$IDX]} % 2 )) -ne 0 ]
then
        echo "odd."
else
        echo "even."
fi
IDX=$(( $IDX + 1 ))
done
```

Construction de Boucle

L’instruction `until` est souvent plus verbeuse que `for` mais peut est plus adapté à des critères d'arrêts non-numérique.

Attention à bien fournir une condition d'arrêt valable, sinon la boucle peut tourner indéfiniment.

L’instruction `while` est similaire à `until`, mais elle répète les commandes tant que la commande test se termine par `0`.

Dans l’exemple précédent, l’instruction `until [ $IDX -eq ${#SEQ[*]} ]` est equivalent à `while [ $IDX -lt ${#SEQ[*]} ]`.

Un Exemple Plus Élaboré

Imaginons qu'un utilisateur veut périodiquement synchroniser une collection de fichiers et de répertoires dans un autre appareil de stockage, monté à un point arbitraire du système.

D'abord, on va créer une liste de fichiers et de répertoires à synchroniser, depuis un répertoire d'origine que l'on passera en première argument du script vers un répertoire de destination que l'on passera en deuxième argument.

Pour faciliter l'ajout et la suppression d'élément à cette liste, on va la placer dans un fichier séparé, `~/.sync.list`, avec un élément par ligne.

Un Exemple Plus Élaboré

Par exemple:

```
$ cat ~/.sync.list
Documents
Album Photo
.ssh
.config
```

Le fichier `~/sync.list` contient un mélange de fichiers et de répertoires.

C’est une bonne occasion d’utiliser la commande `mapfile`, qui analyse le contenu d’un texte donnée et créé une variable tableau avec, en plaçant chaque ligne dans un éléments du tableau.

Un Exemple Plus Élaboré

Soit sync.sh, notre scripte pour cette opération:

```
#!/bin/bash

set -ef

# Liste des éléments à synchroniser

FILE=~/.sync.list

# Répertoire d’origine

FROM=$1

# Répertoire destination

TO=$2
```

Un Exemple Plus Élaboré

```
# Vérifier que les répertoires sont valides
if [ ! -d "$FROM" -o ! -d "$TO" ]
then
    echo Usage:
    echo "$0 <SOURCEDIR> <DESTDIR>"
    exit 1
fi
```

Un Exemple Plus Élaboré

```
# Create array from file
mapfile -t LIST < $FILE
# Sync items
for (( IDX = 0; IDX < ${#LIST[*]}; IDX++ ))
do
    echo -e "$FROM/${LIST[$IDX]} \u2192 $TO/${LIST[$IDX]}";
    rsync -qa --delete "$FROM/${LIST[$IDX]}" "$TO";
done
```

Un Exemple Plus Élaboré

1. Récupérer et vérifier les paramètres du scripts

   La variable `FILE` est la liste d'éléments à synchroniser depuis `~/.sync.list`. Les variables `TO` et `FROM` sont les répertoires respectivement d'origine et de destination. Comme ils sont fournies par l'utilisateur, ils passent un simple test de validation avec une construction `if`. Si l'un des deux n'est pas un répertoire valide, alors le script renvoie une aide et se termine avec le code `1`.

2. Charger la list des fichiers et des répertoires

   Après que tous les paramètres soient bien définis, un tableau contenant l'ensemble des éléments à copier est créer avec `mapfile -t LIST < FILE`. L'option `-t` permet de retirer les retour à la ligne de chaque entrée avant de l'insérer dans le tableau `LIST`. Le contenu du fichier est lu par redirection de l'input (`<`).

Un Exemple Plus Élaboré

3. Réalisé la copie et en informer l'utilisateur

   Une boucle `for` avec une notation double parenthèses va parcourire le tableau, avec `IDX` la variable qui indique l'index. La commande `echo` informe l'utilisateur que chaque éléments est entrain d'être copié. Le caractère unicode échappé (`\u2192`) pour la flèche vers la droite est présent, donc il faut utiliser l'option `-e`. La commande `rsync` va copier uniquement les fichiers modifiés du répertoires d'origine. Les options `-q` et `-a`, bloque les messages de `rsync`, et préserve les propriétés des fichiers. L'option `--delete` fait que `rsync` supprimera les fichiers dans le répertoire de destination qui ne sont pas dans le répertoire d'origine.

Exercices