S. Lavirotte
Polytech'Nice – Sophia
SI 3
2014-2015

TD séance n° 15
Introduction à la Programmation Shell Unix 1

1 Introduction

1.1 Rappels

Lors des premiers cours d'Environnement Informatique, vous avez appris à utiliser un interprète de commandes (ou *Shell* en anglais) qui permet d'interagir avec le système. L'exécution de commandes vous a permis de consulter ou modifier l'état du système (par exemple, la commande `ls` pour consulter la liste des fichiers dans un répertoire ou la commande `cp` ou `rm` respectivement pour copier ou effacer un fichier).

Une utilisation avancée de ce système de commandes consiste à combiner ces commandes en les exécutant les unes à la suite des autres pour automatiser certains traitements ou encore en combinant l'exécution de plusieurs commandes en redirigeant le résultat d'une commande comme l'entrée à traiter de la commande suivante.

1.2 Script et Langage de script

La programmation *Shell* consiste à combiner des commandes au sein d'un script dans le but d'automatiser certaines tâches qui peuvent être réalisées par l'enchainement de commandes Unix de base.

Les scripts sont des programmes en langages interprétés : ce sont des fichiers écrits dans des langages qui sont convertis en instructions directement exécutables par la machine au moment de son exécution. On appelle ces langages, langages de script.

Un script *Shell* est un fichier exécutable, dont le nom comporte souvent l'extension `.sh` (mais ce n'est pas une obligation).

1.3 Shells

Il existe de nombreux *shell*, sur les systèmes de type Unix, qui se classent en deux grandes familles :

- la famille *C Shell* dont la syntaxe est inspirée du langage C (par exemple `csh`, `tcsh`, ...).
- la famille *Bourne Shell*, dont `sh` fut le premier présent sur les systèmes Unix et à l'origine de tous les autres (par exemple `sh`, `bash`, `ksh`, ...)

Enfin, `zsh` est un shell qui contient les caractéristiques des deux familles précédentes et qui est le plus récent (mais qui date toutefois de 1990). Il comprend donc une syntaxe un peu plus complète et permet l'utilisation de structures de données plus complexes (comme des tableaux par exemple).

Bash, acronyme de « Bourne-Again SHell », est le shell le plus utilisé aujourd'hui (et que nous utiliserons dans ce cours). Son nom est un jeu de mots sur le nom du shell historique d'Unix, le Bourne shell (`sh`). Littéralement, Bourne again signifie « Bourne encore », mais se prononce également presque comme born again, signifiant « né de nouveau » ou encore « réincarné ». Également, to bash signifie « frapper violemment » en anglais.

Chaque shell possède sa propre syntaxe pour les structures de contrôle (faire un test, une boucle, ...). Comme nous allons utiliser le shell bash, nous étudierons donc sa syntaxe qui lui est propre, mais commune en de nombreux points avec les autres shells de la même famille.

1.4 Environnement pour développer en Shell

Pour écrire des programmes shell, pas besoin d'environnement de développement très évolué, un simple éditeur de texte suffit. Vous pourrez dont utiliser `gedit` pour créer vos premiers scripts ou tout autre éditeur de texte équivalent (mais pas de traitement de texte pour créer de tels fichiers, qui ne sont pas adaptés).

TD séance n° 15
Introduction à la Programmation Shell Unix 1

2 Ecriture de vos premiers scripts shell

2.1 Entête d'un script shell

Un script shell est donc un fichier texte qui doit toujours commencer par une ligne contenant le nom du *shell* avec lequel lire la suite du script. Si vous souhaitez développer un script `bash`, votre fichier devra donc débuter par :

```
#!/bin/bash
```

N'oubliez pas que votre fichier doit aussi être exécutable (au sens propriété des fichiers, modifiable avec la commande `chmod`).

Rappelons, que le Bash est principalement une évolution de sh. Ainsi, il est possible, au stade de vos connaissances en programmation shell, d'écrire en première ligne :

```
#!/bin/sh
```

Dans le cas ci-dessus, on fait appel au shell sh et non au Bash. C'est parfois préférable pour que votre script soit standard et qu'il fonctionne ainsi sur toutes les plateformes de type Unix qui parfois, ne disposent pas de Bash (ça reste assez rare tout de même).

Vous modifierez votre entête en fonction de l'endroit où est installé le shell pour lequel vous écrivez le script, si vous choisissez `ksh` ou `zsh` par exemple.

2.2 Un langage de programmation

2.2.1 Commandes externes

Une commande externe est une commande dont le programme est dans un fichier externe au shell. Parmi l'ensemble des commandes externes que l'on peut trouver dans un système, nous utiliserons principalement les commandes Unix que nous avons utilisées jusqu'à présent (ex : `ls`, `mkdir`, `cp`, `rm`) et les autres scripts shell que vous aurez pu déjà écrire.

2.2.2 Commandes internes

Une commande interne est une commande 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. Les commandes internes courantes de la plupart des shells sont : `cd`, `echo`, `pwd`, `for`, `if`, `test`, ...

Les commandes internes peuvent être décomposées en deux catégories :

- les commandes simples (comme `cd`, `echo`, ...)
- les commandes composées de plusieurs éléments : `case` ... `esac`, `if` ... `fi`, `for` ... `done`, `while` ... `done`, `until` ... `done`, etc. que nous étudierons un peu plus tard dans le cours.

Nous allons détailler au cours de ces séances consacrées à la programmation shell comment utiliser ces commandes et comment réaliser des commandes plus complexes à l'aide de celles-ci.

2.3 Les commandes et les structures de base pour un langage

2.3.1 Les commentaires

Toute ligne qui commence par le caractère # est considéré comme un commentaire (ce qui est écrit n'est pas interprété par le shell) jusqu'à la fin de la ligne. Pour que le caractère # soit reconnu en tant que début de commentaire, il ne doit pas être inséré à l'intérieur d'un mot ou terminer un mot.

TD séance n° 15
Introduction à la Programmation Shell Unix 1

Attention à la première ligne de votre script qui commence par `#!` et non seulement par # et qui est commentaire spécifique qui indique le nom du shell à utiliser pour interpréter les commandes.

2.3.2 echo

La commande de base pour afficher un message est la commande `echo` :

```
$ echo Bonjour    tout le                    monde
Bonjour tout le monde
```

Dans cet exemple, la commande `echo` est appelée avec 4 arguments. L'affichage ne conserve alors pas les espaces. Pour conserver ceux-ci, on utilisera les guillemets `" "`.

```
$ echo "Bonjour    tout le                    monde"
Bonjour    tout le                   monde
```

On dit que les espaces ont été protégés de l'interprétation du shell. L'option `-n` à la commande `echo` permet de ne pas faire le retour à la ligne à la fin de l'affichage.

2.3.3 Les variables et l'affectation de valeur

Une variable est identifiée par un nom, c'est-à-dire une suite de lettres, de chiffres ou du caractère souligné et 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, my_var ; ...)
- les variables prédéfinies du shell (ex : `PS1`, `PATH`, `REPLY`, `IFS`, `USER`, `HOSTNAME`...)
- les variables prédéfinies de commandes unix (ex : `TERM`).

En général, mais c'est une convention et pas une obligation, 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 =
- la commande interne `read`.

2.3.3.1 Affectation directe

Il est impératif que le nom de la variable, le symbole = et la valeur à affecter ne forment qu'une seule chaîne de caractères. En d'autres termes, il ne peut y avoir d'espaces entre le nom, le symbole = et la valeur que l'on affecte à la variable.

```
x=23       y="Bonjour"
```

2.3.3.2 Affectation par lecture

La commande `read` permet de récupérer une saisie faite par l'utilisateur au clavier (on verra plus tard que cela permet de faire plus que cela).

```
$ read -p "Entrez votre prenom : " prenom
Entrez votre prenom : <l'utilisateur tape son prénom>
```

L'option –p de `read` affiche une chaîne d'appel avant d'effectuer la lecture.

2.3.3.3 Utiliser une variable

Pour récupérer le contenu d'une variable on préfixe son nom par le symbole `$`

```
echo $prenom
<affiche le prénom saisi par l'utilisateur lors de la commande précédente>
```

Mais certaines fois, il peut y avoir une ambigüité entre un nom de variable et un message que l'on veut concaténer : dans ce cas, on ajoute des accolades autour du nom de la variable.

Par exemple, si on a les variables `x` et `x1` qui sont initialisées aux valeurs suivantes :

```
$ x=bon
$ x1=jour
$ echo $x$x1
bonjour
$ echo ${x}1
bon1
```

2.3.4 Récupérer l'exécution d'une commande dans une variable

Si vous utilisez la commande `date` sous Unix, elle vous imprime sur la console la date à laquelle vous venez d'exécuter cette commande (donc la date et l'heure du moment de l'exécution). Il peut être intéressant de stocker cette valeur dans une variable. Il faut alors utiliser la syntaxe suivante avec le symbole ` (que l'on obtient sur un clavier français avec la combinaison de touche `Alt-Gr` et la touche `7`).

```
$ date
<affiche la date courante>
$ d=`date`
<met le résultat de la commande date dans la variable d>
```

Comparez le contenu de la variable d et si vous le comparez au résultat obtenu à l'exécution précédente.

Une autre façon de l'écrire est d'utiliser la notation `$(cmd)` qui est équivalente à la notation `` `cmd` ``.

2.4 Paramètres d'un script

Pour récupérer les paramètres d'un script shell (les arguments qui ont été passés sur la ligne de commande lors de l'appel du script), on va utiliser des variables spéciales qui contiennent ces valeurs :

- `$0` : contient le nom complet de la commande shell qui s'exécute
- `$1`, `$2`, `$3`, … : contiennent respectivement, le 1er, 2ème et 3ème argument passé à la commande
- `$*` : contient l'ensemble des arguments qui ont été passés à la commande
- `$#` contient le nombre d'arguments

2.5 Code de retour d'un script shell

Le code de retour d'un programme shell est le code de retour de la dernière commande qu'il a exécutée. Mais vous pouvez avoir besoin dans votre script de gérer des erreurs (paramètres non valides, erreurs dans l'exécution du script, …). Par défaut, si un script s'exécute correctement (ou une commande Unix), le code retourné est `0`.

Il est parfois nécessaire de positionner explicitement le code de retour d'un programme shell (que vous écrivez) avant qu'il ne se termine : on utilise alors la commande interne `exit`. Elle provoque l'arrêt du programme shell avec un code de retour égal à n (le paramètre spécifié en argument).

```
#!/bin/bash
echo "Bonjour"
exit 1
```

La valeur de retour du script est alors `1`. Pour consulter cette valeur, il faut utiliser une autre variable spéciale : `$?`. Après exécution du script précédent, cela donnera :

```
$ echo $?
1
```

Attention, si vous utilisez exit n dans votre interpréteur de commande, cela aura pour effet de l'arrêter en renvoyant la valeur n spécifiée.

TD séance n° 15
Introduction à la Programmation Shell Unix 1

3 Exercices

Pour ce premier TD, nous allons nous contenter de faire des petits exercices simples pour apprendre à maitriser les rudiments de la programmation shell (la syntaxe de base). Lors des prochaines séances, nous commencerons à combiner ces différentes commandes et structures de programmation pour faire des choses un peu plus utiles.

Exercice n°1:

Comme dans tout nouveau langage de programmation, vous allez faire votre premier script shell `hello.sh` en faisant le très classique programme « Hello World !». L'exécution de votre programme devra afficher ce message à l'écran.

Exercice n°2:

Ecrire un programme shell qui s'appelle `deuxfois.sh` et qui affiche le message "Entrez un mot : ", lit le mot saisi par l'utilisateur puis affiche ce mot deux fois sur la même ligne.

Exercice n°3:

Faite un script qui se nomme `copie.sh` et qui affiche le nom du programme, le nombre d'arguments, l'ensemble des arguments qui ont été passés à ce script ainsi que les 2 paramètres sources et destination. Enfin, le programme devra réaliser la copie du fichier source spécifié à la bonne destination.

```
$ ./copie.sh /etc/passwd X
Nom du programme : ./copie.sh
Nb d'arguments : 2
Tous les arguments : /etc/passwd X
Source : /etc/passwd
Destination : X
```

Exercice n°4:

Ecrire un programme shell `cp2fois.sh` prenant trois arguments : le premier désigne le nom du fichier dont on veut copier le contenu et les deuxième et troisième arguments sont les noms des fichiers vers lesquels on veut faire la copie du premier. Aucun cas d'erreur ne sera à prendre en compte.

Exercice n°5:

Faites un script `mon_onzieme.sh` qui prend douze paramètres et qui affiche la valeur du onzième.

```
mon_onzieme un deux trois quatre cinq six sept huit neuf dix onze douze
```

Exercice n°6:

Ecrivez le script `welcome1.sh` qui affiche un message de bienvenue :

```
$ ./welcome1.sh
Bonjour zorro, bienvenu sur hal9000, nous sommes le 07/01/2013 et il est 10:17
```

Ici, *zorro* est le nom de l'utilisateur qui exécute ce programme, *hal9000* est le nom de la machine sur laquelle on l'exécute. Ces informations sont contenues dans les variables USER et HOSTNAME.
Les commandes nécessaires sont : echo, date (rappelez-vous que la lecture de la page de manuel des commandes n'est pas optionnelle ! RTFM !)
Mais où trouver la page de manuelle des commandes internes ?