Partie IV – Écriture de scripts shell Linux

1. Premiers scripts shell

1.1 Écriture et exécution d'un script shell

Création

Un script shell est un fichier qui contient une suite de commandes Unix que l'utilisateur veut exécuter plusieurs fois. Voici un exemple tout simple de script dont le nom est `base`.

```
#! /bin/bash
echo Bonjour !
pwd
ls
```

Les commandes du script sont exécutées de manière séquentielle, ligne après ligne.

La première ligne du fichier `#! /bin/bash` (appelée shebang) désigne l'interpréteur de commandes qui doit être utilisé pour exécuter le script (ici c'est `bash` qui sera utilisé). Les caractères **#!** doivent obligatoirement se trouver dans les colonnes 1 et 2. L'espace entre `!` et `/bin/bash`, le chemin absolu de l'interpréteur de commandes, est facultatif.

Exécution

Ce script peut être exécuté de 2 façons.

(1) Avec la commande suivante, la permission de lecture est suffisante :

```
sh base
```

(2) Avec la commande suivante, le script est considéré comme une commande. Il faut donc placer la permission d'exécution sur le fichier :

```
chmod u+x base
base
```

Les commandes sont recherchées dans les répertoires de la variable `PATH`. Par conséquent, si on veut exécuter notre script `base` avec la commande précédente, il faut ajouter le répertoire qui le contient dans `PATH`, par exemple le répertoire courant dans la commande :

```
PATH=$PATH:.
```

Si on ne veut pas modifier `PATH`, il faut préciser l'emplacement du script en absolu ou en relatif lors de son lancement :

```
# en relatif : à partir du répertoire personnel
~/home/jdupont/scripts/base

# en relatif : à partir du répertoire courant
./base
```

1.2 Paramètres

Au lancement d'un script shell Unix, un certain nombre de variables sont initialisées par le shell. Ces variables

sont accessibles en lecture seulement.

`$#` : est le nombre de paramètres reçus par le script

`$0` : est le nom du script

`$1,$2,...$n` : contiennent les valeurs des paramètres 1 à `n`

`n` est généralement limité à 9. seuls `ksh` et `bash` autorisent un nombre plus grand de paramètres, auquel cas les accolades sont obligatoires pour accéder à leur contenu (par exemple `${10}`, `${11}`, etc.)

`$*` : est la liste des paramètres au format "`$1 $2 $3` ..." (séparés par des espaces)

`$@` : est la liste des paramètres au format "`$1`" "`$2`" "`$3`" ...

`$?` : le code d'erreur de la dernière commande exécutée. Toute commande Unix retourne un code d'erreur compris entre 0 et 255 :

- `0` représente la valeur VRAI signifiant le succès de la commande en question
- `> 0` tout retour supérieur à 0 représente la valeur FAUX signifiant l'échec de la commande

`exit` : la commande qui permet de retourner le code d'erreur d'un script shell, qui est lui aussi une commande (une valeur entre 0 et 255). Cette commande met fin au script.

1.3 Fonctions

Une fonction regroupe un certain nombre de commandes qui sont exécutées plus d'une fois dans un script. Les deux syntaxes suivantes sont équivalentes : les parenthèses ou le mot-clé `function` indiquent au shell que `fonct` est une fonction.

```
fonct() {
      commandes
}
function fonct {
      commandes
}
```

L'accès aux paramètres d'une fonction se fait de la même manière que pour le script lui-même (seul `$0` garde sa valeur initiale, à savoir le nom du script) :

```
fonct() {
      for param; do echo $param; done
}
function fonct {
      echo $0", "$1", "$2", "$3
}
```

`return` : la commande qui permet de retourner le code d'erreur d'une fonction dans un script (une valeur entre 0 et 255). Cette commande met fin à l'exécution de la fonction.

```
fonct() {
      if [ -f $1 ]
      then
            echo "le fichier existe"
            return 0
      else
            return 1
```

```
      fi
}
fonct nom_fichier
```

Exercice 1 (7)

1. Exécutez le script `param.sh` avec les commandes suivantes et observez la différence entre `$*` et `$@`.

   a. `./param.sh bonjour le monde`

   b. `./param.sh bonjour "le monde"`

2. Après avoir exécuté un script dans le shell, comment pouvez-vous afficher son code retour (la valeur renvoyée par un `exit`).

3. Que fait la commande suivante :

```
[ $# -eq 5 ] || { echo "Erreur: paramètres incomplets"; exit 1; }
```

4. Dans un script, la boucle suivante affiche la liste des paramètres. Qu'en déduisez-vous quant à la liste parcourue implicitement ?

```
for param; do echo $param; done
```

   Écrivez une boucle qui fait exactement la même chose mais avec une liste explicite.

5. Écrivez un script qui affiche les `n` premiers nombres de la suite de Fibonacci.

   Votre script contiendra une fonction `affiche_fibo` dont le contenu est :

   - si `n` est égal à 0 : afficher un message d'erreur et arrêter l'exécution du script en renvoyant le code `2`

   - sinon : afficher la suite de Fibonacci en calculant la valeur d'un élément comme la somme de ses deux successeurs : `f=$(($f1+$f2))`.

   Le script devra être lancé avec la commande suivante (où `n` est un nombre quelconque entre 10 et 100) :

```
./fibo.sh n
```

6. Modifiez le script précédent de manière à afficher les nombres de Fibonacci un nombre par ligne dans un fichier `fibo.txt` qui est le deuxième paramètre du script. Le script devra d'abord vérifier si le fichier `fibo.txt` existe déjà : si oui, il faudra le supprimer en début de script.

   Le script sera lancé avec la commande suivante :

```
./fibo.sh n fibo.txt
```

7. Créez un script `sauvegarde_c.sh` qui fait une sauvegarde de tous les fichiers se terminant par `.c` dans votre répertoire personnel. Chaque fichier `fich.c` devra être copié dans un nouveau fichier dont le nom est `fich.c.<extension>` où `extension` sera fourni par l'utilisateur comme premier argument du script. Le script sera par exemple lancé avec la commande suivante si vous voulez utiliser l'extension `.copie` (le fichier `fich.c` sera alors copié dans un fichier `fich.c.copie`) :

```
./sauvegarde_c.sh copie
```

8. Écrivez un script qui prend en argument un nom de fichier. Le fichier en question contient un nombre entier par ligne. Votre script contiendra :

1. Un test permettant de vérifier que le fichier existe bien, sinon afficher un message d'erreur et faire un retour erreur.

2. Une fonction `addition` qui calcule et affiche la somme des lignes du fichier

3. Une fonction `nlignes` nombre lignes qui compte et affiche le nombre de lignes dans le fichier

4. Une fonction `minmax` qui calcule et affiche le minimum et le maximum des valeurs du fichier