TP6 - Linux: Bash avancé

Travail préliminaire : locales

Les aspects liés aux habitudes culturelles de l'utilisateur s'appellent les **locales**. Ils concernent la langue mais aussi les façons de noter les valeurs numériques, dates, etc. De façon à avoir les pages de manuel en français, il faut installer le pack de langue correspondant et choisir la "locale" française.

Tapez la commande **`dpkg-reconfigure locales`**, aller cocher (touche ESP) **`fr_FR.UTF8`**, et valider (touche Entrée). Valider ensuite la deuxième page en vérifiant que c'est bien cette locale qui est sélectionnée.

Redémarrer la machine avec **`shutdown -r now`** (l'option -r indique "redémarrage"), relancer une console, et vérifier en tapant **`man useradd`**.

1 Substitution de commande

Dans les TP précédents, vous avez vu comment on assigne une valeur à une variable et comment on utilise la valeur de la variable.

Mais comment faire si on veut affecter à une variable le **résultat** d'une commande ? Par exemple, le filtre **`| wc -l`** placé derrière une commande permet d'avoir le nombre de lignes produit par la commande. Comment sauvegarder cette valeur dans une variable ?

La solution consiste à utiliser la **substitution de commande**, avec la syntaxe **`var=$(commande)`**. Tapez la ligne suivante dans votre "home" :

```
NB_FICHIERS=$(ls -l | wc -l)
```

puis afficher la variable.

Travail à effectuer Ecrire un script **`tp61.sh`** qui affiche le nombre de fichiers d'un dossier donné en argument (la commande ci-dessus est erronée, il faut soustraire 1 pour avoir la bonne valeur). Afficher un message d'erreur et sortir du script si l'utilisateur ne donne pas d'argument.

2 Lecture de fichier de données

Dans le TP5, vous avez vu la boucle "for" qui permet d'itérer sur un ensemble de fichiers :

```
for file in *.abc ...
```

En réalité, cette énumération se fait sur les chaînes de caractères. Essayer la commande suivante, en interactif :

```
for file in aaa bbb ccc; do echo $file; done
```

On va ainsi pouvoir utiliser cette forme pour parcourir le contenu d'un fichier contenant un mot sur chaque ligne, en remplacant dans la forme ci-dessus les différents mots par ceux contenus dans un fichier.

Travail à effectuer

Q2.1. Créer avec **`nano`** un fichier **`liste1.txt`** qui contient sur chaque ligne un identifiant d'utilisateur à créer :

```
dupont
durand
duval
```

Q2.2. En utilisant cette dernière syntaxe de "for" et la substitution de commande, écrire un script **`tp62.sh`** qui va lire le fichier **`liste1.txt`** et qui va créer les utilisateurs, **de façon non-interactive**. Il faudra utiliser la commande **`cat`** poour lister le fichier et pouvoir récuperer son contenu.
Pour créer les utilisateurs, il faudra utiliser la commande **`adduser`** avec les options `--disabled-password -gecos ""`. Vous devrez également affecter tous ces utilisateurs au groupe **`users`**.

Q2.3. Ecrire un autre script **`tp62_supp.sh`** qui va lui supprimer les utilisateurs, au lieu de les créer.

Boucle "tant que" En réalité, cette façon de procéder est limitée : le fichier ne peut contenir que 1 mot par ligne, on ne peut pas donner d'autres informations dans le fichier. On utilise plutôt avec la commande **`while`**, en effectuant une lecture de la ligne complète, tant qu'on n'est pas arrivé à la fin du fichier. Les différents champs de texte pourront être placés automatiquement dans des variables différentes.

```
while read ligne
do
   commande
done < fichier
```

Q2.4. Essayer la ligne suivante (en interactif), et vérifier que vous la comprenez :

```
while read ligne; do echo $ligne; done < liste1.txt
```

Q2.5. Copier le fichier **liste1.txt** en **liste2.txt**. Donner la commande correspondante :

Q2.6. Editer ce fichier **liste2.txt** avec **nano** et le compléter en mettant en 2e position un mot de passe.

```
dupont a3bGthR
durand aa58tR
duval rJKo9(P
```

Q2.7. Ecrire un script **tp63.sh** qui va lire ce fichier, et contenant le code suivant. Vérifier le fonctionnement.

```
while read v_nom v_passwd
do
   echo nom=$v_nom passwd=$v_passwd
done < fichier
```

Q2.8. Remplacer la commande echo par la commande qui va créer l'utilisateur, comme précedemment. Pour lui assigner le password, on utilisera la commande **chpasswd**, de la façon suivante :

```
echo username:new_password | chpasswd
```

Changement de séparateur Le problème ici, c'est qu'on ne peut pas avoir de caractère "espace". Si ce n'est pas important jusqu'ici (espace interdit dans les passwd et dans les identifiants), d'autres situations en auront besoin. Or, l'espace est le séparateur de champ par défaut.

La solution consiste àspécifier le caractère séparateur dans le fichier avec la variable **IFS** ("*Internal Field Separator*").

Tout d'abord, vous allez remplacer les espaces du fichier **liste2.txt** par des ";", avec un outil dédié au traitement de fichiers de données de type "texte" : **sed** (*Stream EDitor*). Il permet (entre autre) de faire du remplacement d'une chaîne par une autre, dans toutes les lignes du fichier.

La syntaxe d'utilisation est la suivante : `sed s/X/Y fichier`
Ceci va rechercher le 1er caractère 'X' dans le fichier et le remplacer par Y. Pour faire un remplacement global, il faut ajouter "/g" :

```
sed s/X/Y/g fichier
```

Q2.9. Donner la commande pour remplacer tous les caractères "espace" par des ";" et rediriger la sortie vers un fichier **liste3.txt**. Attention, ces deux caractères devront être entre "".

Q2.10. Dans le code donné à la question **??**, modifier le "while" comme suit et vérifier le fonctionnement :

```
while IFS=";" read v_nom v_passwd
```

3 Fonctions

Comme pour tout langage de programmation, il est pertinent de décomposer les différentes tâches en sous-tâches via le concept de **fonction**. On peut aussi passer des arguments aux fonctions, via les variables automatiques **$1**, **$2**, ...

Q3.1. Tapez le code suivant dans un fichier **tp64.sh** et vérifiez que vous en comprenez le fonctionnement :

```
function mafonction {
        echo fonction: arg=$1
}

mafonction aaa
mafonction bbbb
```