TP n°4

Architecture des systèmes informatiques
INFO3 - S5

Environnement, script et administration distante

L'objectif de ce TP est de se familiariser avec une utilisation plus avancée du terminal : gestion des variables, des variables d'environnement, des scripts ; ainsi que la manipulation de machines à distance via *ssh*.

Ce TP est largement inspiré des sujets d'administration système de Denis Cousineau (lix) et d'Emmanuel Viennet (l2ti).

1 Le shell

1.1 Variables d'environnement

En plus des usages que vous avez vus jusqu'ici, le shell propose d'autres fonctionnalités (c'est en fait un véritable langage de programmation, comme nous le verrons par la suite). En particulier, il est possible d'utiliser des variables.

Variables du shell Une variable est repérée par un nom appelé identificateur, qui peut être n'importe quelle suite de caractères commençant par une lettre ou le caractère « souligné » `_` et ne contenant que des lettres, des chiffres ou le caractère souligné. On peut assigner une valeur à la variable var avec l'instruction `var=valeur` (sans espace avant ni après le signe '='). Cette valeur peut être une chaîne de caractères ou un entier par exemple. On accède à la valeur associée à une variable en faisant précéder son identificateur du symbole `$`, comme par exemple dans la commande `echo $var` qui affiche la valeur de la variable `var`.

Question 1. Vérifiez que la variable `SHELL` a déjà une valeur lorsque vous lancez votre terminal. Quelle est cette valeur ?

Question 2. Déclarez une variable `NOM` contenant votre prénom et votre nom. Comment résoudre le problème de l'espace entre le prénom et le nom ? Affichez la valeur de `NOM` pour vérifier que l'affectation est correcte.

Question 3. La commande `set` utilisée sans argument affiche la liste de toutes les variables dont la valeur est instanciée dans le shell courant. Vérifiez que la variable précédemment définie apparaît bien dans cette liste.

Question 4. Écrivez une ligne de commande qui affiche `Bonjour, prénom nom !` (remplacez `prénom nom` par votre prénom et votre nom) en utilisant la variable `NOM`.

Question 5. Depuis le terminal courant, ouvrez un nouveau shell (via la commande `bash`), puis affichez la valeur de la variable `NOM` (quand vous avez terminé, tapez exit pour revenir au shell précédent). Faites de même dans un nouveau terminal lancé depuis l'interface graphique du bureau. Qu'en déduisez-vous sur la portée des variables du shell ?

Variables d'environnement Dans certains cas, on souhaite que la valeur d'une variable soit accessible également aux processus fils du shell courant. De telles variables sont appelées variables d'environnement.

QUESTION 1. Affichez la liste des variables d'environnement grâce à la commande `env`. Que remarquez-vous par rapport à la sortie de la commande set ?

QUESTION 2. On transforme une variable `var` en variable d'environnement grâce à la commande `export var`. Transformez `NOM` en variable d'environnement et répétez la question 5. Qu'en déduisez-vous sur la portée des variables d'environnement ?

QUESTION 3. La commande `cd` permet de changer de répertoire courant. Utilisée sans argument, elle fixe le répertoire courant au répertoire personnel de l'utilisateur. Ce comportement est en fait déterminé par la valeur d'une certaine variable d'environnement. Repérez cette variable dans la liste des variables d'environnement, et faites en sorte que la commande cd renvoie par défaut vers le répertoire racine.

Variable `PATH` Certaines variables d'environnement on une signification très importante dans le système d'exploitation. C'est le cas notamment de la variable `PATH` (que vous retrouverez dans la quasi totalité des systèmes d'exploitation).

QUESTION 1. Quelle est la valeur de la variable d'environnement `PATH` ? Le caractère `:` sert de séparateur entre les différents chemins contenus dans la valeur de `PATH`.

QUESTION 2. Créez (ou copiez) dans votre répertoire de base un exécutable et essayez de l'exécuter en tapant uniquement son nom (sans répertoire devant) dans le shell. Que se passe-t-il ? Pourquoi ?

QUESTION 3. Ajoutez maintenant à votre variable `PATH` le chemin de votre répertoire de base. Vous devrez pour cela affecter à la variable `PATH` son ancienne valeur à laquelle vous ajouterez la chaîne de caractères `:/chemin/vers/répertoire/de/base/` et essayez d'exécuter votre fichier. Qu'en déduisez-vous sur l'utilité de la variable `PATH` ?

QUESTION 4. Comme vous avez dû le remarquer dans l'exercice précédent, les valeurs des variables d'environnement sont accessibles à tous les processus fils du shell courant. Comment faire pour que le chemin de votre répertoire de base soit inclus dans la valeur de la variable `PATH`, dès qu'on lance un nouveau terminal ?

QUESTION 5. Pourquoi le répertoire `./` n'est pas présent dans la variable `PATH` ?

Valeur de retour d'une commande Chaque commande transmet au programme appelant un code, appelée valeur de retour *(exit status)* qui stipule la manière dont son exécution s'est déroulée. Par convention du shell *bash*, la valeur de retour est toujours 0 si la commande s'est déroulée correctement, sans erreur et différente de 0 en cas d'erreur. Une variable système spéciale `$?` contient toujours la valeur de retour de la précédente commande. Pour tester les valeurs de retour, exécutez la séquence suivante :

```
$ ll ~
$ echo $?     --> 0
$ ll /root
$ echo $?     --> 1, si l'utilisateur n'est pas root
```

Enchainement conditionnels des commandes Les séparateurs `&&` et `||` sur la ligne de commande sont des séparateurs qui jouent les rôles d'opérateurs conditionnels, en ce sens que la $2^{ème}$ commande sera exécutée en fonction du code de retour de la $1^{ère}$ commande.

Dans `commande1 && commande2`, `commande2` ne sera exécutée que si le code de retour de `commande1` est 0 (exécution correcte).

Dans `commande1 || commande2`, `commande2` ne sera exécutée que si le code de retour de `commande1` est différent de 0 (exécution erronée).

QUESTION 1. Trouver leur signification des exemples ci-dessous

1. `cd ~/tmp || mkdir $HOME/tmp`
2. `cd /root && echo "root rentre chez lui !"`
3. `[ -f /usr/sbin/inetd ] || exit 0` (extrait de `/etc/rc.d/inet.d/inetd`)

1.2 Les expansions

Quand un processus shell reçoit une ligne de commande, il pré-traite cette chaîne de caractères avant de l'exécuter. Tout d'abord, il découpe cette chaîne de caractères, selon le séparateur défini dans la variable `IFS` (par défaut, c'est le caractère espace). Ainsi la ligne

```
cat fic1 fic2 fic3
```

est découpée en quatre chaînes de caractères

`cat` `fic1` `fic2` `fic3`

Ce qui permet au processus shell de différencier la commande des arguments dans la chaîne de caractères qu'on lui fournit. Mais ce n'est pas tout, il existe d'autres caractères spéciaux qui permettent de demander au shell de faire d'autres pré-traitements sur la chaîne de caractères, avant de l'exécuter (on dit que le shell procède à l'expansion de cette chaîne de caractères). Nous avons déjà vu, dans la partie précédente que lorsqu'on fournit la chaîne de caractères `$id` au shell, le shell va la remplacer par la valeur de la variable dont l'identifiant est `id`. Le tableau 1 récapitule les différents types d'expansions en bash.

Syntaxe	Expression	Expansion	Explication
{ }	`ba{ba,bu}`	`baba` `babu`	énumération
\	`\"\'\$\{\ \A B`	`"'${ A` `B`	échappement
$	`$HOME`	`/home/user`	valeur de variable
${...}	`${HOME}`	`/home/user`	valeur de variable
$(...)	`$(which true)`	`/bin/true`	sortie d'une commande
`...`	`` `which true` ``	`/bin/true`	sortie d'une commande
$((...))	`$((12 + 4 * 2))`	`20`	valeur d'une expression
"..."	`" a $HOME $((1+1)) ' "`	`a /home/user 2 '`	$ expansés
'...'	`' a $HOME $((1+1)) " '`	`a $HOME $((1+1)) "`	$ non-expansés

TABLE 1 – Tableau récapitulatif des principales expansions.

D'autres expressions sont expansées selon le contenu du répertoire courant. Par exemple, dans un répertoire contenant des fichiers nommés `abx`, `abd`, `abcde`, `adx` et `bdx`, on obtiendra :

Caractères	Expression	Expansion	Explication
*	`ab*`	`abx` `abd` `abcde`	joker chaîne de caractères
?	`ab?`	`abx` `abd`	joker un caractère
[...]	`a[abcd]x`	`abx` `adx`	liste de possibilités
{ , , }	`a{a,ab,cd}x`	`aax` `aabx` `acdx`	énumération

QUESTION 1. Que signifient les expressions suivantes : `$*`, `$@`, `$#`, `$0`, ..., `$9` ?

Pour cela, créez le fichier suivant :

```
echo '$#='"$#"
echo '$*='"$*"
echo '$@='"$@"
for i in 0 1 2 3 4 5 6 7 8 9; do
    echo -n \$$i=
    eval "echo \$$i"
done
```

Commentez.

QUESTION 2. Utilisez la commande `echo` pour écrire exactement les lignes suivantes dans un fichier

1. `A B`
2. `L'ensemble {1, 2, 3} est inclus dans N`
3. `Elle s'écria : "Ciel mon mari !"`
4. `$HOME = chemin` où `chemin` est le chemin de votre répertoire de login d'après le bash
5. `19 * 216 = produit` où `produit` est la valeur du produit calculée par le bash

2 Les scripts

2.1 Principe

Pour exécuter plusieurs commandes séquentiellement, Il est possible de créer un fichier regroupant une série de commande. Ce type de fichier s'appelle un script. Il va être lu par un interpréteur de commande (dans notre cas, *bash*) qui décodera et exécutera chaque instruction contenue dans le fichier.

Il existe plusieurs interpréteur de commandes et afin de permettre une exécution directe du script sans avoir besoin de connaitre le bon interpréteur à utiliser, il est permis de spécifier sur la première ligne du script sous Unix le chemin de l'interpréteur à utiliser en le précédant par les caractères `#!`. Par exemple le fichier suivant sera interprété par l'interpréteur *bash*.

```
#!/bin/bash
if [ "${1##*.}" = "tar" ]
then
    echo $1 est une archive tar
else
    echo $1 n\?est pas une archive tar
fi
```

Il existe en *bash* la possibilité d'utiliser des structures de contrôle et des boucles.

Exécution conditionnelle L'instruction `if` permet d'exécuter des instructions si une condition est vraie. Sa syntaxe est la suivante :

```
if [ condition ]
then
    action
fi
```

`action` est une suite de commandes quelconques. L'indentation n'est pas obligatoire mais très fortement recommandée pour la lisibilité du code. On peut aussi utiliser la forme complète :

```
if [ condition ]
then
    action1
else
    action2
fi
```

Opérateurs de comparaison Le shell étant souvent utilisé pour manipuler des fichiers, il offre plusieurs opérateurs permettant de vérifier diverses conditions sur ceux-ci : existence, dates, droits. D'autres opérateurs permettent de tester des valeurs, chaînes ou numériques. Le tableau ci-dessous donne un aperçu des principaux opérateurs :

Opérateur	Description	Exemple
	Opérateurs sur des fichiers	
-e filename	vrai si filename existe	`[ -e /etc/shadow ]`
-d filename	vrai si filename est un répertoire	`[ -d /tmp/trash ]`
-f filename	vrai si filename est un fichier ordinaire	`[ -f /tmp/glop ]`
-L filename	vrai si filename est un lien symbolique	`[ -L /home ]`
-r filename	vrai si filename est lisible (r)	`[ -r /boot/vmlinuz ]`
-w filename	vrai si filename est modifiable (w)	`[ -w /var/log ]`
-x filename	vrai si filename est exécutable (x)	`[ -x /sbin/halt ]`
file1 -nt file2	vrai si file1 plus récent que file2	`[ /tmp/foo -nt /tmp/bar ]`
file1 -ot file2	vrai si file1 plus ancien que file2	`[ /tmp/foo -ot /tmp/bar ]`
	Opérateurs sur les chaînes	
-z chaine	vrai si la chaine est vide	`[ -z "$VAR" ]`
-n chaine	vrai si la chaine est non vide	`[ -n "$VAR" ]`
chaine1 = chaine2	vrai si les deux chaînes sont égales	`[ "$VAR" = "totoro" ]`
chaine1 != chaine2	vrai si les deux chaînes sont différentes	`[ "$VAR" != "tonari" ]`
	Opérateurs de comparaison numérique	
num1 -eq num2	égalité	`[ $nombre -eq 27 ]`
num1 -ne num2	inégalité	`[ $nombre -ne 27 ]`
num1 -lt num2	inférieur (<)	`[ $nombre -lt 27 ]`
num1 -le num2	inférieur ou égal (<=)	`[ $nombre -le 27 ]`
num1 -gt num2	supérieur (>)	`[ $nombre -gt 27 ]`
num1 -ge num2	supérieur ou égal (>=)	`[ $nombre -ge 27 ]`

Boucle `for` Comme dans d'autre langages, la boucle `for` permet d'exécuter une suite d'instructions avec une variable parcourant une suite de valeurs. Exemple :

```
for x in un deux trois quatre
do
    echo x= $x
done
```

affichera :

```
x= un
x= deux
x= trois
x= quatre
```

On utilise fréquemment la boucle `for` pour énumérer des noms de fichiers, comme dans cet exemple :

```
for fichier in /etc/rc*
do
    if [ -d "$fichier" ]
    then
        echo "$fichier (repertoire)"
    else
        echo "$fichier"
    fi
done
```

Ou encore, pour traiter les arguments passés sur la ligne de commande :

```
#!/bin/bash
for arg in $*
do
    echo $arg
done
```

Instruction `case` L'instruction case permet de choisir une suite d'instruction suivant la valeur d'une expression :

```
case "$x" in
go)
    echo "demarrage"
;;
stop)
    echo "arret"
;;
*)
    echo "valeur invalide de x ($x)??
esac
```

Noter les deux `;` pour signaler la fin de chaque séquence d'instructions.

Définition de fonctions Il est souvent utile de définir des fonctions. La syntaxe est simple :

```
mafonction() {
    echo "appel de mafonction..."
}
mafonction
mafonction
```

qui donne :

```
appel de mafonction...
appel de mafonction...
```

Voici un exemple de fonction plus intéressant :

```
tarview() {
    echo -n "Affichage du contenu de l'archive $1 "
    case "${1##*.}" in
    tar)
        echo "(tar compresse)"
        tar tvf $1
```

```
    ;;
    tgz)
        echo "(tar compresse gzip)"
        tar tzvf $1
    ;;
    bz2)
        echo "(tar compresse bzip2)"
        cat $1 | bzip2 -d | tar tvf -
    ;;
    *)
        echo "Erreur, ce n'est pas une archive"
    ;;
    esac
}
```

Plusieurs points sont à noter :

1. `echo -n` permet d'éviter le passage à la ligne
2. La fonction s'appelle avec un argument (`$1`) : `tarview toto.tar`

2.2 Exercices

QUESTION 1. Proposez un fichier de commandes permettant de tester au plus fin un fichier : son existence, sa nature, les accès autorisés (lecture, écriture, exécution), ...

QUESTION 2. Expliquez, commenter, éventuellement corriger et compléter, le fichier de commandes ci-dessous

```
echo *
if [$# -ne 3]; then
    echo "message"
else
    case $1 in
    a*) if  [! -d "$1"]; then
            echo "message"
        else
            for i in s i l r  $3; do
                if  [-f  $1/$2.$i]; then
                    echo  $1/$2.$i message
                else
                    echo  $1/$2.$i message
                fi
        fi ;;
    *)  fic=/tmp/$$
        echo -n $1 > $fic
        taille=‘cat $  fic  | wc -c‘
        while [ $taille -lt $2]; do
            taille=$(expr $taille + 1)
            cat $3 >> $fic
        ls -l $fic
        cat $fic
        rm $fic;;
fi
```

2.3 Configuration et personnalisation du shell : `.bashrc`

Le fichier `$HOME/.bashrc` contient vos variables, alias et environnement personnel. Il est exécuté après `/etc/bash.bashrc`. Consulter `man sh` à la section `INVOCATION`.

Analysez les fichiers `/etc/profile`, `/etc/bash.bashrc`, `$HOME/.bash_history`

Modifier votre fichier `.bashrc` de manière à y introduire vos personnalisations : variables, alias, ajustement de umask, limites limit ...

3 Administration à distance

3.1 ssh

Le protocole *SSH* (pour Secure Shell) est le remplaçant de *rsh* (remote shell) qui correspond grossomodo à *telnet*. *SSH* permet bien plus de choses que *telnet*, notamment de de transférer des fichiers de façon sécurisée (fiable et cryptée) via les protocoles *scp* et *sftp*.

SSH existe en deux versions majeures 1 et 2 qui sont incompatibles. La version 2 est la plus sécurisée et il est conseillé de l'utiliser à chaque fois que cela est possible. L'utilitaire client *ssh* existe par défaut sous de nombreuses distributions de Linux et est assez bien documenté.

3.1.1 Introduction et premiers pas

sshd est un serveur qui permet de communiquer de façon sécurisée, en établissant un canal de communication entre lui et ses clients. Le protocole SSH, utilisé par ce serveur, permet à des utilisateurs d'accéder à une machine distante à travers une communication chiffrée (appelée tunnel). L'établissement d'une liaison passe par deux étapes : authentification des machines afin d'établir un tunnel de communication sécurisé (couche transport) à l'aide d'un couple clé privée/publique puis authentification de l'utilisateur qui souhaite se connecter.

ssh est donc basé sur un mécanisme de clé publique clé privé. Ces deux clés sont mathématiquement reliées ; typiquement la clé publique sert à chiffrer un message et la clé privé sert à le déchiffrer.

Essayez de vous connecter sur la machine de votre voisin à l'aide de la commande *ssh*. S'il s'agit de votre première connexion à la machine, vous devez obtenir un message ressemblant à :

```
The authenticity of host 'myhost (123.245.15.20)' can't be established.
RSA key fingerprint is d7 :69 :a7 :db :89 :71 :ff :13 :54 :ce :b4 :0d :55 :f6 :a9 :04.
Are you sure you want to continue connecting (yes/no)? yes
```

Comme vous ne vous êtes jamais connecté sur la machine `myhost`, vous ne connaissez pas sa clé publique. En répondant `yes`, vous obtenez les messages suivants :

```
Warning : Permanently added 'myhost,123.245.15.20' (RSA) to the list of known hosts.
toto@myhost's password :
```

Vous devez alors saisir le mot de passe qui va être chiffré avec la clé publique de la machine `myhost`. Cette machine va utiliser sa clé privée pour déchiffrer le mot de passe et ainsi va pouvoir vérifier l'identité de l'utilisateur.

ssh se charge de stocker la clé publique afin de vérifier que lors d'une prochaine connexion à la machine, cette clé n'a pas changée.

Authentification par clé Lors de la connexion vers une machine distante, *ssh* demande son mot de passe à l'utilisateur. Pour un administrateur système effectuant cette opération plusieurs fois par jours ou pour une application accédant de manière distante à un système par l'intermédiaire de *ssh*, il existe un moyen de sécuriser la connexion sans prendre le risque de stocker un mot de passe *en clair*. De façon analogue à l'authentification des machines, un utilisateur peut utiliser un couple de clés pour s'authentifier sur un serveur.

Voici la marche à suivre :

1. Création d'un couple de clés : `ssh-keygen -t rsa` (choix par défaut pour la réponse aux 3 questions). La clé privée se trouve dans `~/.ssh/id_rsa` et votre clé publique est dans `~/.ssh/id_rsa.pub`
2. Transfert de votre clé publique sur le serveur :
   `cat ~/.ssh/id_rsa.pub | ssh username@machine "cat - >> ~/.ssh/authorized_keys2"`
   (`authorized_keys2` s'appelle parfois `authorized_keys`). Voir aussi la commande `ssh-copy-id`. (si vous ne pouvez pas restreindre les droits du fichier `~/.ssh/id_rsa.pub`, faites une copie de la clé privée vers `/tmp`)
3. Vous pouvez maintenant vous connecter sur la machine distante sans mot de passe.

Sécurisation de la clé Pour protéger votre clé privée et s'assurer que c'est bien la bonne personne qui utilise la clé, il est possible lors de la création des clés de saisir une *passphrase* qui sera demandée à chaque connexion. Recommencez la procédure de création des clés en utilisant une *passphrase*. On a donc remplacé la saisie du mot de passe par la saisie d'une *passphrase* ! Mais on a quand même amélioré la sécurité. Pourquoi ?

Pour éviter d'avoir à retaper la *passphrase* à chaque connexion, on peut utiliser un agent *ssh* (voir `ssh-agent`) qui va conserver dans un cache votre *passphrase* durant toute la durée de votre session sur la machine cliente. Examiner la documentation d'un agent ssh pour en comprendre le fonctionnement.

3.1.2 Transferts de fichiers, affichage X distant

Le protocole *ssh* permet aussi d'exporter un affichage X11 vers votre machine cliente. Examinez les options nécessaire à cette manipulation et essayer de transférer l'affichage.

Les commandes *scp* et *sftp* permettent de transférer des fichiers par l'intermédiaire de *ssh*. Testez-les et essayez également de transférer un fichier entre deux ordinateurs distants tout en étant sur un troisième.