TP OS n°6

Sommaire

Les objectifs de ce tp sont d'être capable de manipuler des variables dans le shell Linux et de personnaliser son environnement.

Introduction

Environnement de travail

Il vous faut ouvrir une **session** sur votre poste de travail. Vous pouvez utiliser soit le mode console soit l'interface graphique. Dans les deux cas, vous allez travailler **« en ligne de commande »**.

Pour ce TP, il vous faudra préalablement créer un répertoire de travail `tpos6` dans `$HOME/tpos` en tapant la commande suivante :

```
$ mkdir -p $HOME/tpos/tpos6
```

Pour se déplacer dans l'arborescence de travail, il faut taper la commande suivante :

```
$ cd $HOME/tpos/tpos6
```

Rappels : le shell

Shell (coquille qui entoure le noyau) est un **interpréteur de commandes** qu'on utilise pour lancer des commandes ou programmer une suite de commandes (script). L'utilisateur dialogue avec le *shell*, qui dialogue avec le noyau, qui à son tour dialogue avec le matériel. Originellement, le *shell* était utilisé sous Unix puis, il s'est répandu avec différentes versions, la forme la plus simple est `sh`.

Les versions les plus connues :
- `sh` : *shell Bourne*
- `ksh` : *korn shell*
- `csh` : *shell* syntaxe du C
- `tcsh` : `csh` amélioré
- **`bash` : *Bourne Again Shell***
- `zsh` : le petit dernier
- `fish` : un *shell* pour les débutants

Lister les shells disponibles :

```
$ cat /etc/shells
/bin/bash
/bin/csh
/bin/dash
/bin/sh
/bin/tcsh
/usr/bin/fish
```

Afficher son shell de connexion :

```
$ echo $SHELL
```


Environnement d'exécution

Un environnement d'exécution (ou contexte d'exécution) désigne un état du système défini par des ressources allouées à un processus et un ensemble de valeurs associées à des **variables d'environnement** auxquelles ce processus a accès.

Dans le monde Unix, on considère que chaque processus est lancé dans un environnement d'exécution distinct. Lors de la création d'un nouveau processus, un nouvel environnement est créé par copie de l'environnement d'exécution du processus parent (clonage).

Ceci permet de garantir qu'un processus ne pourra pas interférer avec les informations disponibles pour un autre processus, puisqu'ils tournent dans des environnements différents : un cloisonnement des ressources est ainsi mis en oeuvre.

Pour permettre à des processus de communiquer, on peut toutefois utiliser différents mécanismes comme les *sockets* ou les tubes (*pipes*).

Dans le cas où l'on veut que plusieurs tâches distinctes aient un accès direct aux mêmes ressources, on découpe un processus en threads, qui partageront alors le même environnement d'exécution.

Quand un programme est invoqué, il reçoit un tableau de chaînes que l'on appelle environnement. Il s'agit d'une liste de paires de la forme `nom=valeur`.

Afficher l'environnement :

```
$ env
$ printenv
```

Fichiers de configuration

Les fichiers de configuration du *shell* permettent de personnaliser l'environnement soit :
– pour tous les utilisateurs (les fichiers sont stockés dans `/etc` et sont de la responsabilité de l'administrateur *root*) :

```
/etc/profile : exécuté automatiquement lors d'un shell de connexion (à chaque login), quel
    que soit le shell
/etc/bashrc : chargé automatiquement lors d'un shell interactif bash
/etc/profile.d/ : contient un ensemble de fichiers de personnalisation spécifique
```

– seulement pour sa session (les fichiers sont stockés dans le répertoire `$HOME` de chaque utilisateur, lequel peut les modifier à sa convenance) :

```
$HOME/.profile : exécuté automatiquement lors d'un shell de connexion (à chaque login),
   quel que soit le shell
```

Fichiers de configuration spécifiques au *shell* `bash` :

```
$HOME/.bash_profile : exécuté automatiquement lors d'un shell de connexion (à chaque login)
$HOME/.bashrc : chargé automatiquement lors d'un shell interactif
$HOME/.bash_logout : chargé lors de la fermeture du shell (à la déconnexion)
$HOME/.bash_history : fichier texte contenant l'historique des commandes tapées
```

Remarque : pour éviter que l'administrateur ne recopie tous ces fichiers à chaque fois qu'il crée un utilisateur, il existe un squelette dans `/etc/skel/`

Variables d'environnement

Les variables d'environnement sont des **variables** dynamiques utilisées par les différents processus d'un système d'exploitation (Windows, Unix, etc.).

	Windows	Unix/Linux
Affichage de la liste des variables d'environnement	La commande **set** sans aucun paramètre permet de lister la plupart des variables d'environnement. Les variables **%ERRORLEVEL%, %DATE%, %TIME%, %CD%, %RANDOM%** ne sont pas affichées par la commande **set**.	la commande **env**. l'instruction du *shell* **set**.
Modification d'une variable d'environnement	Pour modifier une variable d'environnement : **> set path=%path %;c:\WINDOWS\System32\wbem** Pour modifier une variable d'environnement d'une manière permanente il faut : -Modifier l'**autoexec.bat** avec la commande **set** sous Windows 98. -Ajouter ou modifier la clé de registre **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Environment** ou **HKEY_CURENT_USER\Environment** **sous windows 2000/xp/2003** Il est aussi possible de modifier les variables d'environnement en utilisant le **panneau de configuration**.	Avec le *shell* bash : **$ export PATH=$PATH:/opt/toto/bin** Cette modification n'est que temporaire (pour la session en cours). Pour modifier la variable **$PATH** de manière permanente, il faut ajouter la ligne suivante dans le fichier **~/.bashrc** : **$ export PATH=$PATH:/opt/toto/bin**
Visualisation du contenu d'une variable d'environnement	**> echo %path%**	**$ echo $PATH** **$ printenv PATH**

Décomposition de l'affichage avec ls -l

Elles servent à communiquer des informations sur le contexte d'exécution aux processus.

Le *shell* permet de manipuler l'environnement de plusieurs façons. Au démarrage, le *shell* analyse son propre environnement, et crée une variable pour chaque nom trouvé, en le marquant comme exportable vers les processus fils. Les commandes exécutées héritent de cet environnement. Les commandes `export` et `declare -x` permettent d'ajouter ou de supprimer des paramètres ou des fonctions de l'environnement.

Sous Unix et Linux, les *shells* utilisent deux types de variables, utilisant la même syntaxe :
– des variables locales au *shell*,
– des variables d'environnement transmises aux applications lancées par le *shell*.

Remarque : ces variables n'existant que pour la session en cours, il faut donc des fichiers pour les conserver et les retrouver à chaque démarrage.

Variables

Une variable est un espace de stockage pour un résultat.

Une variable est un symbole (habituellement un nom qui sert d'identifiant) qui renvoie à une position de mémoire (adresse) dont le contenu peut prendre successivement différentes valeurs pendant l'exécution d'un programme.

De manière générale, on aura six caractéristiques :
- son nom : c'est-à-dire sous quel nom est déclaré la variable ;
- son type : c'est la convention d'interprétation de la séquence de bits qui constitue la variable. Le type de la variable spécifie aussi la longueur de cette séquence (8 bits, 32 bits, 64 bits) ;
- sa valeur : c'est la séquence de bit elle même. Certaines variables sont déclarées constantes (ou en lecture seule) et sont donc protégées contre l'écriture ;
- son adresse : c'est l'endroit dans la mémoire ou elle est stockée ;
- sa visibilité : c'est un ensemble de règles qui fixe qui peut utiliser la variable ;
- sa durée de vie : c'est la portion de code dans laquelle la variable existe. Il ne faut pas confondre la durée de vie d'une variable locale et sa visibilité.

Cependant les possibilités d'une variable sont intimement liées au langage de programmation auquel on fait référence. Toutefois on peut trouver des langages qui simplifient ces caractéristiques. Par exemple, une grande partie des langages scripts ne possèdent pas un typage fort. On parle de **typage fort** lorsque le langage impose que les variables soient déclarées et utilisées dans un type (exemples : C ou C++).

De manière générale, la plupart des langages de scripts admettent :
- qu'une variable puisse changer de type au cours de son existence. On parle de **typage faible**.
- que ce ne sont pas les variables qui ont un type, mais les valeurs. On parle de **typage dynamique**.

Dans un shell Unix/Linux, une variable existe dès qu'on lui attribue une valeur (par défaut le type sera une chaînes de caractères). Une chaîne vide est une valeur valide. Une fois qu'une variable existe, elle ne peut être détruite qu'en utilisant la commande interne `unset`.

Une variable peut recevoir une valeur par une affectation de la forme :

```
nom=[valeur]
```

Si aucune valeur n'est indiquée, la variable reçoit une chaîne vide.

Le caractère '`$`' permet d'introduire le remplacement des variables. Le nom de la variable peut être encadré par des accolades, afin d'éviter que les caractères suivants ne soient considérés comme appartenant au nom de la variable.

Manipulations

Étape n°1 : les variables

Créer une variable :

```
$ chaine=bonjour
```

Afficher une variable :

```
$ echo $chaine
$ echo ${chaine}
$ echo $HOME
```

Manipuler des variables :

```
$ chaine=
$ echo $chaine
$ chaine="hello world"
$ echo $chaine
$ chaine='hello world'
$ echo $chaine
$ chaine=2+2
$ echo $chaine

$ unset chaine
$ echo $chaine
```

Remarque : Il n'existe pas de typage fort dans le shell et, par défaut, TOUT EST UNE CHAINE de caractères.

Manipuler des chaînes de caractères :

```
$ nom=tv
$ echo $nom
$ chaine="hello $nom"
$ echo $chaine
$ chaine='hello $nom'
$ echo $chaine
$ chaine="hello \$nom"
$ echo $chaine
```

Remarque : Il existe une différence d'interprétation par le shell des chaînes de caractères. Les protections (quoting) permettent de modifier le comportement de l'interpréteur. Il y a trois mécanismes de protection : le caractère d'échappement, les apostrophes (quote) et les guillemets (double-quote).

- *Le caractère \ (backslash) représente le caractère d'échappement. Il préserve la valeur littérale du caractère qui le suit, à l'exception du <retour-chariot>.*
- *Encadrer des caractères entre des apostrophes simples (quote) préserve la valeur littérale de chacun des caractères. Une apostrophe ne peut pas être placée entre deux apostrophes, même si elle est précédée d'un backslash.*
- *Encadrer des caractères entre des guillemets (double-quote) préserve la valeur littérale de chacun des caractères sauf $, ‘, et \. Les caractères $ et ‘ conservent leurs significations spéciales, même entre guillemets. Le backslash ne conserve sa signification que lorsqu'il est suivi par $, ‘, ", \, ou <fin-de-ligne>. Un guillemet peut être protégé entre deux guillemets, à condition de le faire précéder par un backslash.*

Par défaut, les variables sont **locales** au *shell*.

```
$ chaine="hello world"
$ echo $chaine
hello world

$ bash
$ echo $chaine

$ exit
```

Les variables n'existent donc que pour le *shell* courant. Si on veut les rendre accessible aux autres *shells*, il faut les exporter avec la commande `export` (pour `bash`), ou utiliser la commande `setenv` (`csh`).

```
// Exporter des variables :
$ export chaine

$ bash
$ echo $chaine
hello world

$ unset chaine
$ echo $chaine

$ exit

$ echo $chaine
hello world
```

Remarque : le deuxième shell (le fils) a hérité des variables et de l'environnement du premier shell (son père). On constate que ces variables ont été dupliquées.

Étape n°2 : substitutions de variables

Les substitutions de paramètres (ou expressions de variables) sont décrites ci-dessous :

```
$nom La valeur de la variable.
${nom} Idem.
${#nom} Le nombre de caractères de la variable.
${nom:-mot} Mot si nom est nulle ou renvoie la variable.
${nom:=mot} Affecte mot à la variable si elle est nulle et renvoie la variable.
${nom:?mot} Affiche mot et réalise un exit si la variable est non définie.
${nom:+mot} Mot si non nulle.
${nom#modèle} Supprime le petit modèle à gauche.
${nom##modèle} Supprime le grand modèle à gauche.
${nom%modèle} Supprime le petit modèle à droite.
${nom%%modèle} Supprime le grand modèle à droite.
```


Les formes les plus utilisées sont :

```
// Obtenir la longueur d'une variable :
$ PASSWORD="secret"
$ echo ${#PASSWORD}

// Fixer la valeur par défaut d'une variable nulle :
$ echo ${nom_utilisateur:=`whoami`}

// Utiliser des tableaux :
$ tableau[1]=tv
$ echo ${tableau[1]}

//Supprimer une partie d'une variable :
$ fichier=texte.htm
$ echo ${fichier%htm}html
$ echo ${fichier##*.}
```

Étape n°3 : substitutions de commandes

La substitution de commandes permet de remplacer le nom d'une commande par son résultat. Il en existe deux formes :

```
$(commande) ou `commande`
```

Cette syntaxe effectue la substitution en exécutant la commande et en la remplaçant par sa sortie standard, dont les derniers sauts de lignes sont supprimés.

On utilise très souvent la substitution de commandes pour récupérer le résultat d'une commande dans une variable :

```
// Déterminer le nombre de fichiers présents dans le répertoire courant :
$ NB=$(ls | wc -l)
$ echo $NB

// Récupérer le chemin d'accès :
$ CHEMIN=`dirname /un/long/chemin/vers/toto.txt`
$ echo $CHEMIN
/un/long/chemin/vers

// Récupérer le nom du fichier :
$ NOM_FICHIER=`basename /un/long/chemin/vers/toto.txt`
$ echo $NOM_FICHIER
toto.txt

// Afficher le chemin absolu de son répertoire personnel :
$ getent passwd | grep $(whoami) | cut -d: -f6
$ echo $HOME
```


Étape n°4 : évaluation arithmétique

L'évaluation arithmétique permet de remplacer une expression par le résultat de son évaluation. Le format d'évaluation arithmétique est :

```
$((expression))
```

Par exemple, pour calculer la somme de deux entiers :

```
$ somme=$((2+2))
$ echo $somme
4
```

Pour manipuler des expressions arithmétiques, on pourra utiliser aussi :

```
=> la commande expr :
$ expr 2 + 3

=> la commande interne let :
$ let res=2+3
$ echo $res

=> l'expansion arithmétique ((...)) (syntaxe style C) :
$ (( res = 2 + 2 ))
$ echo $res

=> la calculatrice bc (notamment pour des calculs complexes ou sur des réels)
$ echo "scale=2; 2500/1000" | bc -lq
2.50
$ VAL=1.3
$ echo "scale=2; ${VAL}+2.5" | bc -lq
3.8
```

Étape n°5 : les variables internes du shell

Ces variables sont très utilisées dans la programmation des scripts :

```
$0      : Nom du script ou de la commande
$1,$2, ... : Paramètres du shell ou du script
$*      : Tous les paramètres
$@      : Idem (mais "$@" eq. à "$1" "$2"...)
$#      : Nombre de paramètres
$-      : Options du shell
$?      : Code retour de la dernière commande
$$      : PID du shell
$!      : PID du dernier processus shell lancé en arrière-plan
$_      : Le dernier argument de la commande précédente. Cette variable est également mise
   dans l'environnement de chaque commande exécutée et elle contient le chemin complet de
   la commande.
```

Comme n'importe quel programme, il est possible de passer des paramètres (arguments) à un script. Les arguments sont séparés par un espace (ou une tabulation) et récupérés dans les variables internes `$0`, `$1`, `$2` etc ... (voir le `man bash` et la commande `shift`).


```
// Afficher quelques variables internes :
#!/bin/bash
echo "Ce sript se nomme : $0"
echo "Il a reçu $# paramètre(s)"
echo "Les paramètres sont : $@"
echo
echo "Le PID du shell est $$"

$ chmod +x variablesInternes.sh

$ ./variablesInternes.sh
Ce sript se nomme : ./variablesInternes.sh
Il a reçu 0 paramètre(s)
Les paramètres sont :

Le PID du shell est 8807

$ ./variablesInternes.sh le petit chat est mort
Ce sript se nomme : ./variablesInternes.sh
Il a reçu 5 paramètre(s)
Les paramètres sont : le petit chat est mort

Le PID du shell est 8078
```

Remarque : un script commence par une ligne dite "shebang" qui contient les caractères "#!" (les deux premiers caractères du fichier) suivi par l'exécutable de l'interpréteur, de préférence avec son chemin complet.

Au niveau du *shell*, une commande qui se termine avec un code de retour (variable interne `$?`) nul est considérée comme réussie. Le zéro indique le succès. Un code de retour non-nul indique un échec.

```
$ ls
$ echo $?
0

$ type fff
bash: type: fff : non trouvé
$ echo $?
1

$ ls fff
ls: impossible d'accéder à fff: Aucun fichier ou dossier de ce type
$ echo $?
2
```

Quand une commande se termine à cause d'un signal fatal, bash utilise la valeur 128+signal comme code de retour.

```
$ find / -name "*a*" -print
...
Ctrl-c

$ echo $?
130
```



```
$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT     4) SIGILL      ...
```

Remarque : La commande find a reçu le SIGnal SIGINT (130-128 = 2) émis par Ctrl-c. Ce signal, quand il n'est pas ignoré, INTerrompt l'exécution de la commande qui l'a reçu.

Si une commande n'est pas trouvée, le processus fils créé pour l'exécuter renvoie la valeur 127.

```
$ fff
bash: fff : commande introuvable
$ echo $?
127
```

Si la commande est trouvée mais pas exécutable, la valeur renvoyée est 126.

```
$ ./essai
bash: ./essai: Permission non accordée
$ echo $?
126
```

Étape n°6 : les variables d'environnement

La commande `env` (commande externe) sans aucun paramètre permet de lister les variables dites d'environnement. Dans les shell usuels d'Unix/linux, l'instruction `set` (commande interne du *shell*) permet d'afficher à la fois les variables d'environnement et les autres variables.

```
// Lister les variables d'environnement :
$ env

// Afficher les variables d'environnement et les autres variables :
$ set
```

Remarque : sous Windows, la commande set sans aucun paramètre permet de lister la plupart des variables d'environnement. Les variables %ERRORLEVEL%, %DATE%, %TIME%, %CD%, %RANDOM% ne sont pas affichées par la commande set.

Afficher l'environnement :

```
// Affiche page par page
$ printenv | more

// Afficher l'aide sur les variables :
$ help variables | more

// Utiliser la complétion sur les variables :
$ $<Tab>
```

La variable <PATH> contient la liste des répertoire(s) dans lesquels vont être recherchés les fichiers exécutables. On retrouve cette variable sous Windows (`%PATH%`) et sous Unix et Linux (`$PATH`). Le séparateur utilisée dans la variable `PATH` est les deux-points (:) sous Unix/Linux et le point virgule (;) sous Windows.

Si un exécutable n'est pas placé dans l'un de ces répertoires, il sera nécessaire d'indiquer le chemin exact (absolu ou relatif) chaque fois qu'on l'appellera sous la ligne de commande (inutile dans le cas d'une

interface graphique si l'on clique directement sur le fichier dans le gestionnaire mais utile pour un icône sur le bureau par exemple).

Remarque : Le piège du répertoire courant Sous Unix, pour exécuter un fichier qui est dans le répertoire courant, on est en général obligé de préfixer la commande par « `./` », ce qui permet d'indiquer que le fichier est dans le répertoire courant. Cette particularité étonne les utilisateurs qui ont l'habitude de Windows (ou de l'ancien MS-DOS) où on peut appeler directement un programme qui est dans le répertoire courant. Cependant, il s'agit là d'une mesure de sécurité : à titre d'exemple, si un intrus malveillant parvient à placer un programme néfaste nommé `ls` dans le répertoire courant, ce programme sera exécuté dès que vous souhaiterez lister le répertoire (au lieu de la commande `ls` se trouvant normalement dans le répertoire `/bin/`, qui lui se trouve dans le `PATH`, mais qui n'est modifiable que par l'administrateur du système).

Si malgré tout l'utilisateur souhaite retrouver cette « ergonomie » de Microsoft, il faut qu'il rajoute le chemin « `./` » dans son `PATH` :

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

La valeur de base de `$PATH`, pour tous les *login*, est défini dans le fichier `/etc/profile`, et personnalisable dans le fichier `$HOME/.bash_profile` pour le *shell* `bash`.

Afficher le contenu de la variable `PATH` en ligne de commande :

```
$ printenv PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games:/usr/lib/qt4/bin

$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games:/usr/lib/qt4/bin
```

Il existe de nombreuses autres variables d'environnement.

On peut citer par exemple :
– `SECONDS` : contient la durée d'activité du *shell* courant en secondes.
– `RANDOM` : contient une valeur numérique aléatoire

Afficher la durée d'activité du *shell* courant en heures/minutes/secondes :

```
$ echo $SECONDS
$ echo $[$SECONDS/3600] h $[($SECONDS%3600)/60] mn $[$SECONDS%60] s
```

Remarque : `bash` ne connaît que les entiers et donc que les divisions euclidiennes, `/` permet d'en obtenir le quotient et `%` d'en obtenir le reste (ou modulo).

```
$ bash
$ echo $[$SECONDS/3600] h $[($SECONDS%3600)/60] mn $[$SECONDS%60] s
0 h 0 mn 6 s
$ exit
```

Remarque : chaque fois que vous exécutez un shell, sa variable `SECONDS` est initialisée à 0.

Jouer à pile ou face :

```
$ echo $RANDOM
$ test $[RANDOM%2] -eq 1 && echo "Pile" || echo "Face"
```


Les résultats du remplacement des variables, de la substitution de commandes, et de l'évaluation arithmétique, qui ne se trouvent pas entre guillemets sont analysés par le *shell* afin d'appliquer le découpage des mots.

L'interpréteur considère chaque caractère de la variable `IFS` comme un délimiteur, et redécoupe le résultat des transformations précédentes en fonction de ceux-ci. Si la valeur du paramètre `IFS` est exactement <espace><tabulation><retour-chariot>, (la valeur par défaut), alors toute séquence de caractères `IFS` sert à délimiter les mots.

```
$ set | grep IFS | head -1
IFS=$' \t\n'
```

Remarque : si un espace est utilisé dans un nom de fichier, il faudra soit le protéger par des doubles quotes (" ") soit préfixer l'espace par un back slash (\ ). Par précaution, le technicien informatique préfèrera ne jamais utiliser l'espace dans les noms de fichier ou de répertoire.

Étape n°7 : les fichiers de configuration

Rappel : Les variables n'existent que pour la durée d'une session. Il faut donc des fichiers pour les conserver et les retrouver à chaque démarrage.

```
// Ajouter un alias dans le fichier $HOME/.bashrc :
$ echo "alias dir='ls -la | more'" >> $HOME/.bashrc
```

Pour vérifier, il vous faut lancer un nouveau `Terminal` ou ouvrir une session avec `Ctrl-Alt-F1`, puis taper `dir` ou `alias`.

Questions de révision

L'idée de base des questions de révision est de vous donner une chance de voir si vous avez identifié et compris les points clés de ce TP.

Question 1. Quel est le symbole préfixant une variable dans un *shell* Unix/Linux ?

Question 2. Que contient par défaut une variable dans un *shell* Unix/Linux ?

Question 3. Quel est le rôle de la variable d'environnement `PATH` sous Unix/Linux ? A-t-elle le même rôle sous Windows ?

Question 4. Quel est le rôle du fichier `$HOME/.bash_profile` ?

Question 5. Est-il possible de faire des calculs arithmétiques sous un shell Unix/Linux ?

Travail demandé

Exercice 1 : la variable d'environnement PS1

Question 6. Quel est le rôle de la variable d'environnement PS1 ?

Question 7. Quelle est sa valeur ? Décomposer et commenter.

Question 8. Modifier cette variable pour obtenir ceci :

```
[nom d'utilisateur@nom machine - heure au format hh:mm:ss - chemin courant]$
```

Question 9. Maintenant, lancer un nouveau shell dans votre shell courant (taper `bash`). Qu'en est-il du *prompt* ? Visualiser le contenu de la variable `PS1` ? Puis sortir de ce *shell* en tapant `exit`. Et visualiser denouveau la variable `PS1`.

Question 10. Que faudrait-il faire pour garder la variable `PS1` modifiée pour chaque session ?

Exercice 2 : les variables associées aux répertoires

La variable $HOME contient le chemin absolu vers le répertoire personnel de l'utilisateur connecté. La variable $PWD contient le chemin absolu vers le répertoire courant (permet de savoir où on est dans l'arborescence). La variable $OLDPWD contient le chemin absolu vers le répertoire courant précédent (permet de savoir d'où on vient). La variable $CDPATH contient la liste des chemins d'accès utilisés par la commande cd lors d'un changement de répertoire en relatif. Il est ainsi possible, en mode relatif, d'accéder à un répertoire dont la racine est dans la variable `CDPATH`.

Changer de répertoire en relatif avec CDPATH :

```
$ pwd
/tmp
$ CDPATH=/usr
$ cd bin
/usr/bin
$ cd toto
bash: cd: toto: Aucun fichier ou dossier de ce type
```

Question 11. Ajouter le répertoire `$HOME/tpos` dans la variable `CDPATH` ? Montrez son utilisation.

Pour rappel, il existe des commandes externes (situées dans des répertoires renseignés dans la variable `PATH` ou n'importe où dans l'arborescence du système), des commandes internes au *shell* et des *alias*.

Question 12. Déterminer l'ordre d'exécution d'une commande par le *shell* `bash` ? Décrire en détails sa démarche. Par exemple : que se passe-t-il si une commande possédait le même nom pour ces trois possibilités ?

Exercice 3 : les variables

Question 13. Créer une variable `MESSAGE` et l'initialiser avec une chaîne de caractère. Afficher cette variable.

Question 14. Afficher ensuite le nombre de caractères qu'elle contient comme ceci : `"La variable $MESSAGE a une longueur de _ caractères"`

Question 15. Créer une variable `CALCUL` qui contiendra "2+2". Que faut-il faire pour afficher 4 quand on manipule cette variable ?

Question 16. En examinant le contenu de sa variable PATH, en déduire où placer ses propres programmes et commandes personnelles ?

```
// par exemple pour l'utilisateur tv, cela donne :
$ printenv PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin/:/usr/games:/home/tv/bin
```

Question 17. Que fait cette commande ? Expliquez.

```
$ test $[RANDOM%2] -eq 1 && echo "Pile" || echo "Face"
```

Question 18. Que fait cette commande ? Expliquez.

```
$ N=$(ls -R -F -1 $HOME/*/ | grep ':' | sed '/^$/d' | wc -l)
```

Exercice 3 : personnalisation de l'environnement

Par défaut, l'affichage de l'historique est le suivant :

```
$ history
   40  ls
   41  pwd
   42  history
```

Pour ajouter un horodatage (*timestamp*) de chaque commande de l'historique, il faut utiliser cette variable d'environnement :

```
$ HISTTIMEFORMAT='%F %T '
```

Et maintenant on obtient l'affichage suivant :

```
$ history
   40  2010-08-02 15:44:41 ls
   41  2010-08-02 15:44:42 pwd
   42  2010-08-02 15:44:49 history
```

Question 19. Indiquer la procédure pour appliquer cette modification personnelle de votre environnement de manière permanente.

L'administrateur *root* désire appliquer, pour tous les utilisateurs, les règles suivantes dans la gestion de l'historique :

- *The default value of the name of the file in which command history is saved is `~/.bash_history` : not change*
- *The maximum number of lines contained in the history file : 100*
- *The number of commands to remember in the command history : 100*
- *Eliminate duplicates across the whole history : set the HISTCONTROL to erasedups*

Question 20. Indiquer la procédure (quelle(s) variable(s) et quel(s) fichier(s)) que devra réaliser l'administrateur.

