UNIVERSITÉ Grenoble Alpes

Grenoble INP ensimag

Unix et les variables d'environnement

—

Ensimag 1A

Grégory Mounié

2018-2019

1 L'environnement d'un processus

Le but du TP est de vous faire comprendre la gestion de l'état de l'environnement d'un processus dans un système. Cet environnement est stocké par chaque processus dans sa mémoire. Cet environnement contient, par exemple, la langue dans laquelle il faut afficher les informations (français, anglais, etc.). Il est hérité du processus qui l'a créé. Le plus souvent il est donc hérité du *shell*, l'interpréteur de commande, qui a lancé le processus.

Dans ce TP nous allons manipuler l'environnement d'un shell, et donc celui de tous les processus qu'il lancera. Nous verrons aussi comment propager et rendre pérennes ces modifications. Certaines modifications sont particulièrement délicates (cf. section 4.2 et 4.3), mais vous aurez besoin de les faire pour certains projets (comme le projet C en fin d'année).

Dans la suite de ce TP, nous allons nous promener au travers de multiples commandes. Le but n'est pas de détailler les commandes elles-mêmes (pour cela : **RTFM**) mais la compréhension globale de ce qu'elles font. Donc n'hésitez pas à poser des questions à votre encadrant de TP.

1.1 Sh, Bash, Zsh, Ksh, Csh, Tcsh et quelques autres

Sous UNIX, il existe plusieurs shells couramment utilisés. Ils appartiennent soit à la famille des Bourne shells, soit à celle des C-shells [1]. A l'intérieur d'une même famille, ils ont des syntaxes légèrement différentes. Dans ce TP, nous utiliserons la syntaxe du shell par défaut sous Linux, `bash`.

Si vous utilisez un autre shell, il faudra adapter votre syntaxe.

1. Prononcez si-shell, et non sey-chelles, ...

1.2 Pour aller plus loin

Pour ceux ayant déjà de bonnes connaissances sur les variables d'environnement, nous vous proposons quelques exercices supplémentaires moins guidés sur le sujet (indiqués "Variantes bonus" dans le sujet).

2 Les variables d'environnement

Chaque programme est placé dans un environnement qui lui est propre mais qui est, par défaut, une copie de celui du programme qui l'a lancé. Souvent, c'est donc une copie de l'environnement du shell de commande `bash` executé dans le terminal.

Cet environnement est composé de *variables*. Pour démarrer, nous allons taper dans un shell la commande :

```
env
```

pour voir la liste des variables d'environnement de votre shell. Essayez de repérer quelques variables intéressantes : la langue d'affichage (français ou anglais), le nom de la machine, votre propre nom d'utilisateur.

Pour être très précis, `env` est un programme normal comme `firefox`. Ce que vous voyez, ce sont donc les variables dont `env` a hérité du shell.

2.1 Variante bonus : affichage en python et en C

Un script Python peut afficher ses variables d'environnement en affichant le contenu de la variable `environ` du module `os`.

Un programme C trouvera ses variables d'environnement dans la variable globale `char **environ;` (cf. `man 7 environ`), ou comme troisième paramètre de la fonction `main`, dont le prototype est le suivant :

```
int main(int argc, char *argv[], char *envp[]);
```

Cette extension est documentée dans l'annexe J.5.1 du standard.

2.2 Filtrage et sélection

Comme ces variables sont nombreuses, il est parfois utile de les filtrer. Par exemple, pour voir les variables contenant dans leur nom la chaine de caractère `LANG`, il suffit de taper :

```
env | grep LANG
```

On peut désormais exploiter l'information contenue dans ces variables. Pour utiliser ou afficher le contenu d'une variable, on y fait référence en ajoutant un `$` devant son nom, par exemple :

```
echo "$LANG"
```

Affichez la valeur des variables `HOME`, `PWD` et `OLDPWD`. Changez maintenant le répertoire courant du shell, en utilisant par exemple la commande :

```
cd /usr
```

Quelle est la nouvelle valeur des variables `HOME`, `PWD` et `OLDPWD` ? Suivez leur évolution au fur et à mesure que vous vous déplacez dans l'arborescence du système de fichiers.

Le shell permet de retourner dans le répertoire précédent avec :

```
cd -
```

À vous avis, comment cela est-il implémenté ?

Si d'ailleurs le déplacement dans les différents répertoires vous est pénible, c'est que vous ne connaissez pas `dirs`, `pushd` et `popd`.

2.3 Variante bonus : lire HOME et PWD en Python et en C

Dans tout programme qui s'exécute, il est possible de lire ces variables.

Un script python manipulera la variable `environ` du module `os`.

Un programme C lira la variable en utilisant la fonction `getenv()`.

3 Manipulation des variables d'un shell et exportation

3.1 Manipulation sans exportation

Pour créer ou modifier une variable `TOTO` dans un shell, utiliser la syntaxe suivante en veillant bien à ne pas mettre d'espace avant ou après le symbole `=` :

```
TOTO=truc
echo "$TOTO" # cela affiche 'truc'
```

En revanche, la variable TOTO n'est pas héritée dans les programmes lancés par le shell. On peut vérifier que rien ne s'affiche en lançant :

```
env | grep TOTO # cela n'affiche rien
```

Pourtant, on peut vérifier que la variable est toujours bien définie :

```
echo "$TOTO" # cela affiche 'truc'
```

On peut aussi le vérifier en lançant un autre shell dans un autre terminal. Attention à bien le lancer depuis le shell dans lequel la variable a été définie.

```
terminology
```

et dans ce nouveau terminal, en faisant

```
echo "$TOTO" # cela n'affiche rien, sinon, relire et recommencer
```

3.2 Exportation

Pour que la variable soit héritée, il faut explicitement demander son *exportation*.

Pour cela, si vous êtes dans le shell qui a déjà défini la variable TOTO, il suffit de faire :

```
export TOTO
```

sinon, il faut définir puis exporter (syntaxe portable)

```
TOTO=truc
export TOTO
```

ou faire les deux en même temps (syntaxe ne fonctionnant pas en `sh` de base, genre celui qui est lancé au boot de la machine).

```
export TOTO=truc
```

Pour vérifier, il suffit de faire

```
env | grep TOTO # cela affiche 'truc'
```

On peut aussi le vérifier en lançant un autre shell dans un autre terminal. Attention à bien le lancer depuis le shell dans lequel la variable a été fixée.

```
terminology
```

et dans ce nouveau terminal, en faisant

```
echo "$TOTO" # cela affiche 'truc', sinon, relire et recommencer
```

Lancer un terminal (terminology) mais cette fois en utilisant la souris (menu applications, ou bien en cliquant sur l'icône de terminology apparaissant après avoir mis la souris dans le coin en haut à gauche). Ce terminal n'étant pas démarré par le shell exportant TOTO, rien ne s'affiche :

```
echo "$TOTO" # cela n'affiche rien, sinon, relire et recommencer
```

3.3 Variante bonus : exportation temporaire

Il est possible de modifier et exporter la variable uniquement pour un programme particulier et ses descendants.

La modification de la variable est faite au lancement de la commande. Pour lire le manuel de `bash` en français

```
LANG=fr_FR.UTF-8 man bash
```

Il est aussi possible de grouper les commandes dans un sous-shell avec des parenthèses. La portée de l'exportation sera alors limitée aux commandes du sous-groupe.

```
(export TOTO=truc; env | grep TOTO) # Devrait afficher 'TOTO=truc'
echo "$TOTO" # cela n'affiche rien, sinon, relire et recommencer
```

3.4 Variante bonus : HOME et PWD en Python et en C

Dans tout programme qui s'exécute, il est possible de lire et de modifier ces variables.

Un programme python manipulera la variable `environ` du module `os`.

Un programme C manipulera la variable en utilisant les fonctions `getenv()` et `setenv()`.

Essayez de changer la variable PWD de vos programmes.

4 Les variables d'environnement essentielles

Il existe des variables importantes car elles servent au système pour choisir quel sera le programme qui sera vraiment exécuté et les bibliothèques de fonctions qu'il utilisera.

Ces variables, vous les modifierez plusieurs fois au cours de votre scolarité à l'Ensimag, notamment pour vos gros projets de 1A, de 2A et de 3A. Ces modifications seront sporadiques, mais essentielles lors de la mise en place des projets. Il vaut mieux comprendre ce qu'elles font car de mauvaises manipulations vous embêteront « un tout petit peu » (impossibilité de se connecter ou de lancer un programme particulier, plantage de programmes qui fonctionnaient avant, etc).

4.1 Organisation des programmes Unix

Sur un système Unix, tous les programmes ne sont pas rangés dans le même répertoire. L'organisation des répertoires change selon les systèmes, mais il y a quelques conventions générales :

- `/bin` contient les programmes essentiels au système : `/bin/ls`, `/bin/bash` ...
- `/usr/bin` contient les programmes moins critiques installés par ailleurs sur le système : `firefox`, `git` ...
- `/sbin` et `/usr/sbin` contiennent les programmes destinés à l'administrateur du système : `reboot`, `wireshark`, `fdisk` ...
- `/usr/local/bin` contient les programmes installés « localement » sur le système, par exemple à l'Ensimag : `vlc`, `blender` ... Souvent, il s'agit de logiciels qui ne sont pas installables par le système de gestion de paquets de l'OS, et qu'il a donc fallu compiler et installer manuellement.

Prenez le temps d'explorer ces différents répertoires sur votre machine Ensimag. Vous remarquerez que les développeurs du système d'exploitation (CentOS 7) ont fait le choix de ne pas séparer `/bin` et `/usr/bin`, via l'utilisation d'un lien symbolique.

On peut utiliser la commande `which` pour savoir où se trouve un programme donné :

```
which firefox   # affiche /usr/bin/firefox : firefox est dans /usr/bin/
which wireshark # affiche /usr/sbin/wireshark
which vlc       # affiche /usr/local/bin/wireshark
```

Attention, si vous avez défini des `alias`, c'est à dire des raccourcis comme par exemple :

```
alias ls='ls --color=auto'
```

alors `which` ne les voit pas ! Cependant, sur les machines de l'Ensimag, une petite astuce permet à `which` de s'en sortir quand même :

```
alias ls='ls --color=auto'
/usr/bin/which ls  # affiche /usr/bin/ls
which ls           # affiche l'alias ainsi que /usr/bin/ls
```

Comment pouvez-vous expliquer la différence de comportement des deux versions de `which` ? Avec un alias bien sûr !

```
which which
cat /etc/profile.d/which2.sh
```

Le dossier `/etc/profile.d/` contient des fichiers qui sont lus au lancement du shell, et qui peuvent contenir des définitions d'alias, des exports ou modifications de variable d'environnement...

On peut aussi utiliser la commande `type` pour obtenir de l'information sur un alias ou une commande intégrée au shell :

```
type ls     # ls est un alias vers « ls --color=auto »
type which
type echo   # echo est une commande intégrée au shell, pas un programme
```

4.2 La variable `PATH`

La variable `PATH` définit la liste des répertoires dans lesquels chercher un programme à exécuter. Le contenu de cette variable est une liste de répertoires séparés par `":"`

```
echo "$PATH"
```

Sur les machines de l'Ensimag, cette liste est longue. De nombreux logiciels sont installés "à la main" pour satisfaire les besoins pédagogiques. Il devrait s'afficher quelque chose de l'ordre de :

```
/opt/use/bin:/usr/local/texlive/2016/bin/x86_64-linux:/opt/rust/bin:/us\
r/lib64/qt-3.3/bin:/usr/local/bin-python3:/opt/oracle:/opt/opam/system/\
bin:/usr/java/jdk1.8.0_92/bin:/usr/lib64/openmpi/bin:/opt/emacs-26.1/bi\
n:/usr/local/bin:/usr/bin:/usr/local/sbin:/usr/sbin:/opt/avispa-1.1:/op\
t/exercism:/opt/mongodb/bin:/opt/simgrid/bin
```

Nous allons maintenant tout casser dans un nouveau terminal

```
terminology # ou bien "gnome-terminal" si vous préférez
```

Dans ce nouveau terminal, nous allons supprimer la variable `PATH` avec la commande `unset` :

```
unset PATH
ls # ne fonctionne pas
vlc # ne fonctionne pas
/usr/bin/ls # fonctionne en indiquant explicitement où est ls
```

Après avoir supprimé `PATH`, même les commandes les plus courantes ne fonctionnent plus sans indiquer explicitement où elles sont. Il ne reste en fait que quelques *commandes internes* comme `cd; echo; pwd; dirs; pushd; popd` dont on peut obtenir la liste en tapant

```
help
```

Toujours dans ce terminal, nous allons recréer une variable `PATH` :

```
export PATH=/usr/bin
ls # fonctionne à nouveau
vlc # ne fonctionne toujours pas
```

Une opération courante est l'ajout au `PATH` de répertoires supplémentaires. Pour cela, on évalue la valeur courante de `PATH`, on ajoute les répertoires et on réaffecte `PATH` :

```
export PATH="$PATH":/usr/local/bin
vlc # fonctionne à nouveau
```

Si plusieurs programmes ont le même nom mais sont placés dans des répertoires différents (par exemple `/usr/bin/` et `/usr/local/bin/`), c'est le premier répertoire présent dans `PATH` qui est choisi.

Vous pouvez fermer le terminal.

4.3 Variante bonus : `LD_LIBRARY_PATH`

La variable `LD_LIBRARY_PATH` est complémentaire de `PATH` mais pour la liste des bibliothèques chargées dynamiquement par un exécutable.

Vous pouvez voir la liste des bibliothèques de fonctions utilisées par un logiciel (ici : ls, vim, gvim, emacs, et atom) lorsqu'il démarre en faisant

```
ldd /bin/ls
ldd /usr/bin/vim
ldd /usr/bin/gvim
ldd /usr/bin/emacs
ldd /usr/share/atom/atom
```

5 Manipulation de quelques variables utiles, mais cosmétiques

Dans cette partie, nous allons insister sur le fait que chaque programme est placé dans un environnement propre qui lui indique comment se comporter.

Cette partie est volontairement moins guidée.

5.1 `TZ`, la Time Zone

Sous UNIX, le temps du système en interne est le *temps universel coordonné* (UTC pour "Coordinated Universal Time"). La variable TZ définie le fuseau horaire du shell.

Question 1 *Quelle heure est-il à Tahiti ?*

indice : Vous pouvez demander la syntaxe de la valeur de la variable TZ à la commande `tzselect`. Tahiti est dans l'Archipel de la Société, en Polynésie Française, dans l'océan Pacifique.

Pour afficher l'heure, vous pouvez utiliser simplement le programme `date` dans un terminal, mais il est plus pratique d'avoir l'heure en continu avec `xclock -digital`, ou une autre horloge `man -s 1 -k clock` (avec un affichage 24h)

5.2 Le prompt de votre shell

Lorsque votre shell attend une de vos commandes, il affiche un prompt. Par exemple :

```
[monlogin@ensipcXXX ~]$
```

Le contenu du prompt est défini par la variable `PS1`. Elle peut contenir des caractères spéciaux évalués à chaque apparition de prompt comme $\backslash h$ pour le nom de la machine, ou $\backslash W$ le répertoire courant.

Question 2 *Ajouter l'heure ($\backslash t$) à votre prompt courant et exporter la valeur de `PS1`.*

Vous pouvez lancer `terminology` depuis le shell où vous avez changé `PS1` pour vérifier si la variable est bien exportée.

Pour indiquer que le shell attend quelque chose de la part de l'utilisateur, une autre variable, PS2, est positionnée. Pour plus d'informations, cherchez `invites` dans le man.

5.3 Variante bonus : la variable `DISPLAY`

Lorsqu'une application veut afficher une fenêtre graphique, elle l'envoie vers l'écran désigné par la variable `DISPLAY`.

Dans un shell, vous pouvez afficher la valeur de la variable

```
echo "$DISPLAY" # à l'ensimag, :0
```

le plus souvent, l'écran numéro 0, de la machine courante.

Si vous vous connectez par ssh au PC de votre voisin, en autorisant l'affichage distant

```
ssh -X ensipc_du_voisin
```

si vous affichez la valeur de la variable `DISPLAY`

```
echo "$DISPLAY" # probablement :10
```

elle indiquera probablement l'écran numéro 10, ou un numéro proche suivant celui qui était disponible au moment de la connexion ssh.

Vous pouvez changer, dans votre connexion ssh, l'écran de destination et essayer d'afficher un programme sur l'écran de votre camarade :

```
DISPLAY=:0 xclock
```

Heureusement pour la sécurité du système, ça ne fonctionne pas par défaut ! Votre camarade doit explicitement vous autoriser à afficher des fenêtres, en utilisant la commande suivante :

```
xhost +local:
```

Il fera ensuite

```
xhost -local:
```

pour révoquer l'accès, car cette autorisation permet également d'accéder à tout ce que vous tapez au clavier.

Attention aussi au logiciel que vous envoyez sur l'écran du voisin : c'est votre programme avec vos droits et votre `HOME`. Même un logiciel d'affichage d'image peut avoir un menu pour effacer vos fichiers.

5.4 Variante hyper-bonus : comportement de la ligne de commande

La ligne de commande de `bash` est très configurable grâce à un support de l'édition nommé `readline`. Vous pouvez lire la partie du manuel de `bash` concernant `readline` et configurer à l'envie vos déplacements et raccourcis clavier [2]. Néanmoins, il existe deux modes « privilégiés » configurables par défaut : `emacs` (bouh !) et `vi` (alleluia !). L'utilisation du mode `vi` grâce à :

```
set -o vi
```

sera la preuve que vous êtes un(e) vrai(e) informaticien(ne) !

6 Configurer son environnement : `.bashrc` et `.profile`

Toutes les modifications précédentes ne fonctionnent que temporairement, dans un shell et ses descendants.

A chaque fois qu'un shell `bash` démarre, il lit et exécute les commandes du fichier `~/.bashrc`. Il est donc possible d'y mettre les alias et les initialisations et exportation de variables que vous désirez rendre pérennes.

Question 3 *Modifiez la variable `PS1` dans votre fichier `~/.bashrc` et vérifiez que lorsque vous démarrez un nouveau terminal, votre prompt reflète bien la variable telle que vous l'avez positionnée.*

Attention, si vous cassez quelque chose dans `~/.bashrc`, il ne sera même plus possible d'ouvrir un terminal, et il est possible que votre session graphique refuse également de démarrer... Assurez-vous donc bien que tout fonctionne avant de fermer votre terminal courant !

2. Pour mémoire, pour rechercher `xxxx` dans un man, il suffit de taper `/xxxx` dans le terminal.

7 Récupérer une erreur de configuration

Plusieurs méthodes existent mais la plus simple sur les machine de l'ensimag est d'utiliser ssh pour lancer directement un éditeur, sans lancer votre shell.

```
ssh -X -t monlogin@ensipcXXX emacs .bashrc
```

Vous pouvez éditer votre `~/.bashrc` en utilisant ssh et l'éditeur texte de votre choix (nano, vim, etc.).

Question 4 *Modifiez à nouveau votre prompt (cf. la section PROMPTING de `man bash`).*

8 Variante bonus : Pour aller plus loin

Il existe de nombreuses façons d'améliorer (beaucoup) le prompt de votre shell. C'est particulièrement vrai pour les utilisateurs de `Zsh` (voir la partie *PROMPT THEMES* de `man zshcontrib`).

Il existe aussi deux logiciels dédiés aux prompts :

1. powerline (qui fonctionne pour vim, bash, zsh, etc.)
2. liquidprompt (pour bash et zsh)

La configuration du prompt est particulièrement utile lorsque vous utilisez `git` pour connaître l'état de vos sources, pensez-y !

Vous pouvez les essayer et les rendre permanent dans votre `~/bashrc`.

9 Pour conclure. . .

Bien sûr, il y a bien plus dans les variables d'environnement que ce que nous venons de voir. Elles contrôlent des aspects subtils de l'exécution comme la façon de lire ou d'afficher des nombres, les dates, les numéros de téléphone, l'accès à l'agent ssh, à DBUS, le programme shell courant, etc.

Les principaux points fondamentaux à retenir :

Héritage : un programme en exécution hérite de l'environnement du programme qui l'a créé ;

TOTO=truc : affecte une variable dans un shell, mais sans l'exporter ;

export TOTO=truc : affecte et exporte une variable d'environnement afin qu'elle soit hérité par les programmes lancés par le shell ;

echo "$TOTO" : affiche la variable TOTO, mais ne permet pas de savoir si elle fait partie de l'environnement ;

PATH : la variable qui contrôle les répertoires où chercher les programmes exécutables ;

LD_LIBRARY_PATH : la variable qui contrôle les répertoires où chercher les bibliothèques de fonction des programmes exécutables ;

`env` **:** permet d'afficher sur le terminal le contenu de l'environnement ;

`~/.bashrc` **:** le fichier lu à chaque démarrage d'un shell bash et donc le fichier où définir les variables d'environnement que l'on souhaite utiliser de manière pérenne.