DM - LIFPCA Programmation Concurrente

—

Fondamentaux du shell

Sylvain Brandel, Yves Caniou, Guillaume Damiand, Meriem Ghali,
Laurent Lefèvre, Thibaut Modrzyk, Grégoire Pichon, Alec Sadler,
Florence Zara, Jerry Lacmou Zeutouo

Printemps 2024

Pour réaliser ce qui suit, vous pouvez : utiliser une machine sous Unix (Linux, MacOS, en installation de base ou en VM), utiliser une VM sur la plate-forme cloud du Département Informatique, ou vous connecter sur `https:shell.univ-lyon1.fr` par exemple.
Dans la suite, l'énoncé suppose que vous êtes connecté en tant que l'utilisateur `etudiant` dans un shell `bash` – ce qui est le cas par défaut en utilisant une VM ASR7 d'Openstack.

I Configuration du shell bash

Avant de commencer, un peu de vocabulaire :

— Un *shell* est un programme qui vous permet de lancer d'autres programmes. Dans ce cours on considère les shells Unix textuels, donc le rôle se résume à exécuter, en boucle, les étapes :

1. Afficher *l'invite de commande* (« *prompt* »)
2. Lire une ligne de commande au clavier
3. Exécuter la commande

— `bash` est le nom du shell Unix le plus utilisé aujourd'hui. D'autres shells sont `zsh` (qui a plus de fonctionnalités), `ksh` qui est un ancêtre de `bash`, `tcsh`, *etc.*

— Un *terminal* est un dispositif permettant d'accéder à un ordinateur. Historiquement, un terminal a d'abord été un ensemble clavier + écran (voire machine à écrire). Dans ce document, nous utilisons le mot « terminal » pour « terminal virtuel », c'est à dire une fenêtre graphique dans laquelle l'ordinateur lit et affiche du texte. En général, on ouvre un terminal pour exécuter un shell dedans (c'est fait par défaut).

— `xterm`, `gnome-terminal`, `konsole`, `eterm` ou encore `rxvt` sont des noms de terminaux classiques sous Linux. Vous pouvez également accéder à un shell en vous connectant en mode console (ctrl-alt-F1 par exemple sur la plupart des distributions, mais les ordinateurs des salles de TP de l'université sont configurés pour que ça soit le serveur graphique : il faut donc aller sur ctrl-alt-F2 (et suivant) – taper aft-F1 pour revenir à l'affichage graphique –. Le passage en mode console peut s'avérer pratique quand l'ordinateur répond moins et que l'on souhaite stoppper ou tuer des applications, voire simplement savoir ce qu'il se passe sur la machine).

— Il existe des systèmes d'invite de commande [1] pour avoir un shell avec certaines fonctionnalités/présentations configurées par défaut (certains ne peuvent être utilisés que pour un shell donné, donc à vérifier avant son installation).

Parler de *terminal*, c'est parler de *tty*, abréviation de *teletype* (nous aurons l'occasion d'avoir un aperçu en regardant le résultat de la commande `w` par exemple). Pour avoir la configuration actuelle, ou définir une configuration (ajouter un raccourci clavier par exemple), on peut utiliser la commande `stty` : ainsi, si dans le résultat de `stty --all` on voit `Werase = ^W`, on sait que taper les touches `Ctrl` et `W` permettent d'effacer le mot précédent le curseur.

I.1 Les variables d'environnement

Q.I.1) - En tant qu'utilisateur `etudiant`, exécutez les commandes :

```
TOTO=tutu
echo "$TOTO"
```

La première ligne crée une variable du shell nommée `TOTO` et de valeur tutu. La seconde affiche la valeur de la variable `TOTO` (donc la chaîne de caractères « tutu »).

Q.I.2) - Pour vous convaincre que les espaces sont significatifs en shell, essayez aussi :

```
TOTO = tutu
```

Ça ne marche pas (tentative d'exécuter la commande `TOTO` avec les arguments `=` et `tutu`).

Q.I.3) - Lancez un nouveau shell avec la commande `bash`. Vous aurez peut-être l'impression que rien ne s'est passé, en réalité :

— Le premier shell a créé un nouveau processus (primitive `fork()`)

— Le nouveau processus exécute la commande `bash` (primitive `exec()`)

— Le premier shell attend que le second termine (primitive `waitpid()`)

Les commandes que vous entrez maintenant sont donc exécutées par le nouveau shell.

Q.I.4) - Exécutez `echo "$TOTO"`. La commande devrait afficher une chaîne vide. Pourquoi ?

La variable `TOTO` est définie dans le premier shell, mais elle n'est pas héritée par le second (techniquement, l'appel à `exec()` réinitialise les variables).

Q.I.5) - Quittez le second shell (`exit`), et vérifiez que `echo "$TOTO"` affiche bien `tutu` maintenant que vous êtes revenu au premier shell.

Q.I.6) - Exécutez la commande `export TOTO` pour transformer `TOTO` en variable d'environnement. Une variable d'environnement est transmise aux processus fils (contrairement à ce qui vient de se passer quand nous avons affiché la valeur de `TOTO` dans le deuxième shell ci-dessus). Trouvez un moyen de vérifier que c'est bien le cas.

1. Un article assez fourni a été publié en novembre 2023 : `https://linuxfr.org/news/comparaison-critique-de-systemes-d-invite-de-commande`

Il suffit de refaire la même manipulation que ci-dessus :

```
bash
echo "$TOTO"
```

Cette fois-ci la valeur tutu devrait être affichée.

Q.I.7) - La commande `env` permet d'afficher les variables d'environnement du shell courant. Vérifiez que vous retrouvez bien la définition de `TOTO`.

En résumé :

- `VARIABLE=valeur` : affectation de valeur à la `VARIABLE`.
- `echo "$VARIABLE"` : Affichage de la valeur de la variable.
- `export VARIABLE` : faire de `VARIABLE` une variable d'environnement.
- `export VARIABLE=valeur` : raccourcis pour `VARIABLE=valeur; export VARIABLE`.

Q.I.8) - Rappel de MUNIX en L1 : quelle est la différence entre l'utilisation de `$TOTO`, `${TOTO}` et `"$TOTO"` ?

La notation avec `{` et `}` doit être utilisée dans le cas où un nom de variable est le préfixe d'un autre : un exemple simple est celui où l'on récupère les arguments donnés en paramètre d'une commande (`argv[]` en C) qui sont respectivement `$1 $2 $3 ... $10` *etc.* Ainsi utiliser `${10}` permet de s'assurer que ce n'est pas la valeur de `$1` qui sera utilisée concaténée au caractère '0' (`$0` est le nom du programme exécuté).

I.2 La variable d'environnement `PATH`

La variable d'environnement `PATH` définit les répertoires dans lesquels le système va chercher les exécutables quand on lance une commande. C'est une liste de répertoires séparés par des « deux points » (`:`).

Q.I.9) - Affichez le contenu de cette variable. Vérifiez que les répertoires classiques `/bin/`, `/usr/bin/` s'y trouvent.

Q.I.10) - Exécutez la commande `PATH="$HOME"/test1:"$PATH":"$HOME"/test2` puis affichez le nouveau contenu de la variable `PATH`.

Q.I.11) - Créez les répertoires `"$HOME"/test1` et `"$HOME"/test2`.

Q.I.12) - Créez un fichier `"$HOME"/test1/less` contenant :

```
#!/bin/sh
echo "Je suis test1"
```

Q.I.13) - Créez un fichier `"$HOME"/test2/less` contenant :

```
#!/bin/sh
echo "Je suis test2"
```

Q.I.14) - Lancez la commande `command -v less` pour voir quelle commande sera exécutée quand vous entrerez `less` en ligne de commande. Vérifiez que la commande se comporte comme prévu. Pour l'instant, les deux fichiers que nous avons créés ne sont pas considérés comme des exécutables car les fichiers n'ont pas le droit `x` (*eXecutable*).

`command -v less` va chercher dans le `PATH` le premier répertoire qui contient un exécutable `less`, qui doit être `$HOME/test1`. Il existe d'autres commandes faisant la même chose (`which` et `type`). `command -v` est celle qui est dans la norme POSIX.

Q.I.15) - Lancez la commande `chmod +x "$HOME"/test1/less "$HOME"/test2/less`. Ré-essayez les commandes `command -v less` et `less`. Si vous ne voyez pas de changement par rapport à la question précédente, exécutez la commande `hash -r`. Que remarquez-vous ?

Cette fois-ci, les fichiers sont exécutables. La commande `less` sera cherchée dans `"$HOME"/test1/less` puis dans les répertoires du système dont `/usr/bin` où se trouve la commande `less` par défaut, et enfin dans `"$HOME"/test2/less`. Comme la commande est trouvée dans `"$HOME"/test1/less`, la recherche s'arrête et notre premier script est exécuté. Vous devez donc voir :

```
$ command -v less
/home/etudiant/test1/less
$ less
Je suis test1
```

Q.I.16) - Fermez votre shell et ouvrez-en un nouveau. Ré-essayez les commandes `command -v less` et `less`. Que remarquez-vous ?

La variable `PATH` que nous avions positionnée n'est pas persistante d'un shell à l'autre. Notre nouveau shell a donc repris la valeur par défaut qui n'inclut pas les répertoires `"$HOME"/test*/less`.

Q.I.17) - De la même manière, créez la commande `"$HOME"/test1/which` (exécutable) en configurant `PATH`. Exécutez la commande `which which`. Que voyez-vous ?

À priori, le résultat continuera d'être consistant malgré la configuration que vous avez apportée : `bash` contient des commandes `built-in` dont `which`. Lire la manpage de `which` vous apprendra que pour utiliser le binaire `/usr/bin/which`, il faut définir une fonction de même nom (et nous verrons plus loin une autre syntaxe que celle utilisée dans la page man).

I.3 Les fichiers de configuration : `.bashrc`, `.bash_profile`

Les manipulations que nous avons faites jusqu'ici étaient temporaires : leur effet était limité au shell courant. La plupart du temps, quand on modifie une variable d'environnement (comme `PATH`), on souhaite que la modification soit persistante et que tous les shells ouverts dans le futur aient la nouvelle configuration automatiquement. Pour cela, nous allons entrer

les commandes comme `PATH=...` non pas dans la ligne de commande interactive, mais dans des scripts utilisés à chaque démarrage du shell. Pour bash, ces fichiers sont `~/.bash_profile`, `~/.bash_login`, `~/.profile` et `~/.bashrc`. Nous nous limiterons à :

— `~/.bashrc` : au démarrage du shell

— `~/.bash_profile` : au démarrage des shells « login » (c'est à dire quand un shell est ouvert suite à une entrée de nom d'utilisateur et mot de passe, par exemple suite à une connexion SSH). Sur certains systèmes (comme Mac OS X), ouvrir un terminal lance un shell « login ».

En pratique, la plupart des éléments de configurations doivent être les mêmes pour les deux types de shell, donc nous allons nous arranger pour que `~/.bash_profile` inclue automatiquement `~/.bashrc`.

Les manipulations ci-dessous sont potentiellement dangereuses donc faites-les sur une VM (ou dans un compte utilisateur créé pour l'occasion). Si vous voulez travailler sur votre machine personnelle, assurez-vous que vous savez ce que vous faites à chaque étape.

Q.I.18) - Connectez-vous sur votre VM et faites toutes les manipulations ci-dessous sur ce compte.

Q.I.19) - Pour repartir à zéro, supprimez les fichiers `.bashrc` et `.bash_profile` s'ils existent (ATTENTION : à ne pas faire sur votre compte habituel Lyon 1 ou sur votre compte principal de votre machine locale ! Éventuellement, considérez un `mv` plutôt qu'un `rm` – votre `PATH` n'étant plus défini correctement par défaut, il faudra utiliser `/bin/mv` pour revenir à l'état initial).

Q.I.20) - Créez le fichier `.bashrc` avec le contenu :

```
echo "chargement de .bashrc"
```

Q.I.21) - Créez le fichier `.bash_profile` avec le contenu :

```
echo "chargement de .bash_profile"
```

Q.I.22) - Ouvrez une nouvelle connexion. Quel fichier est chargé ?

Q.I.23) - Depuis un shell existant, relancez un shell avec la commande `bash`. Quel fichier est chargé ?

Q.I.24) - Ajoutez les lignes suivantes au fichier `.bash_profile` :

```
if [ -f ~/.bashrc ]; then
    . ~/.bashrc
fi
```

La signification de ces lignes est : « si le fichier `~/.bashrc` existe, alors le charger ». C'est ce qui nous permettra d'écrire la configuration de l'utilisateur dans `~/.bashrc`, qui sera chargée quoi qu'il arrive.

Q.I.25) - Une proposition de configuration commune à tous les utilisateurs est disponible dans `/etc/profile`. Ajoutez les lignes suivantes en début de `~/.bashrc` pour en bénéficier :

```
if [ -f /etc/profile ]; then
    . /etc/profile
fi
```

Q.I.26) - Refaites les manipulations ci-dessus pour ouvrir des shells.

Q.I.27) - Ajoutez la ligne suivante au fichier `~/.bashrc` :

```
PATH="$HOME"/bin:"$PATH"
```

Q.I.28) - Créez le répertoire `~/.local/bin` (pensez à `mkdir -p`!) et placez-y un ou plusieurs fichiers exécutables (par exemple un script, comme nous l'avons fait avant pour `~/test*/less`).

Q.I.29) - Exécutez ces exécutables en entrant simplement leur nom. Vous aurez probablement besoin de relancer un shell pour que la modification du `PATH` soit prise en compte (ou de sourcer `~/.bashrc`).

Petite astuce pour relancer un shell : `exec bash`.

Q.I.30) - Supprimez les lignes `echo "..."` des deux fichiers : ces lignes étaient là pour nous aider à comprendre mais c'est une très mauvaise idée de produire du texte sur la sortie standard dans un de ces fichiers.

Q.I.31) - Question subsidiaire mais importante : quelle est la différence entre sourcer (`source script`) et exécuter (`./script`) ?

Sourcer permet de lire le contenu et d'appliquer les configurations au terminal en cours d'exécution. Exécuter va créer un nouvel environnement (`fork()`) et y exécuter le script : les modifications apportées à l'environnement par le script seront donc perdues lorsque le script aura terminé et le nouvel environnement été quitté.
On note qu'on peut sourcer un fichier qui n'a pas de droit d'exécution ; par contre, pour l'exécuter, il faut soit le lui mettre, soit demander explicitement à exécuter le script.

I.4 Un peu de confort et de couleurs !

La variable `PS1` contient une chaîne de caractères qui est affichée comme invite de commande (le *prompt*, que vous avez lue comme `$` dans les exemples de commandes à taper. Sa notation devient `#` lorsque les commandes sont tapées en tant qu'administrateur). Il est très utile d'avoir un prompt bien configuré : il peut l'être en y apportant de la couleur, des effet d'inversion vidéo, mais les codes couleurs peuvent également être utilisés de façon dynamique en fonction des branches `git` dans lesquelles vous évoluez par exemple ! Une bonne configuration permet également d'avoir un prompt « intelligent », comme nous allons le voir après. Configurer le prompt en rouge pour le compte administrateur permet par exemple de prendre un instant de réflexion avant de taper des commandes dangereuses pour le système (comme `hdparm` ou `dd`).

Q.I.32) - Essayez par exemple :

```
PS1="Bonjour maitre, que dois-je faire ? "
```

Q.I.33) - Essayez d'autres valeurs pour `$PS1` comme :

```
PS1="\h:\w\\\$ "
PS1="\[\e[31m\]rouge\[\e[m\] \[\e[32m\]vert\[\e[m\] "
PS1="\[\033]0;\u@\h:\w\007\]\[\033[01;32m\]\u@\h\[\033[01;34m\] \w \$\[\033[00m\] "
```

Q.I.34) - Vous pourrez plus tard générer un joli prompt personnalisé avec un outil comme `http://ezprompt.net/` si vous le souhaitez, et le déployer où vous le voulez.

Q.I.35) - Comment rendre une telle modification persistante sur votre compte, ou vos comptes (dont celui de Lyon 1) ?

La complétion dite « intelligente » permet, lorsqu'on appuie sur la touche tabulation (TAB) pendant qu'on entre une commande (donc après avoir tapé le nom de la commande !), de fournir une complétion dépendante de la commande et du contexte. Par exemple, `ls` *[TAB]* (si besoin, répéter *[TAB]* une deuxième fois) proposera des noms de fichiers, alors que `ls~--`*[TAB]* proposera les options de la commande `ls`. Pour l'activer il suffit d'inclure le fichier `/etc/bash_completion` dans un fichier d'initialisation du shell. C'est souvent le cas par défaut, mais si ce n'est pas le cas il suffit d'ajouter des lignes au `~/.bashrc`, généralement :

```
if [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
fi
```

Q.I.36) - Mettez en place également la complétion "intelligente". Vérifiez avec l'exemple ci-dessus (`ls`) qu'elle fonctionne.

Il faut s'assurer que le package `bash-completion` est installé, et faire `source /etc/bash_completion`. Attention la façon de procéder peut être différente sur d'autres distributions que Debian et Ubuntu (par exemple Gentoo..)

Q.I.37) - Avec `git` *[TAB]*, regardez la liste des commandes git disponibles. Comment entrer la commande `git commit --amend` en appuyant seulement sur 13 touches (pour 18 caractères) ?

`git com[TAB]--am[TAB]`

Q.I.38) - Exécutez sur des machines différentes `ifconfig` : est-il possible d'utiliser la complétion automatique pour compléter la commande (par exemple `ifc`*[TAB]* ? Pourquoi ?

Oui, mais elle ne fonctionne que si la commande est exécutable **et** dans `$PATH`. En local, il y aura une erreur car `ifconfig` n'est à priori pas dans `$PATH`, c-à-d `/sbin/` (ou `/usr/sbin/` sur certaines distributions) n'est pas dans le `PATH` de l'utilisateur par défaut.

Petite remarque : on utilise `ifconfig` ici pour l'exercice, car la commande est dépréciée (obsolète) et on lui préférera l'utilisation de la commande `ip` (Voir `https://fr.wikipedia.org/wiki/Ifconfig`).

I.5 Les alias

Les alias permettent de gagner un peu de temps en définissant des raccourcis pour les commandes qu'on utilise le plus souvent.

Q.I.39) - Exécutez et commentez pour chaque **ligne** suivante :

```
alias
l
```

```
alias l="ls -l --color=auto"
l
alias
```

Q.I.40) - Peut-on appeler une commande définie par un alias dans un script ?

Pour le savoir, il suffit de configurer un alias et de créer un script qui appelle cet alias, par exemple :

```
$ alias yo='echo Happy birthday'
$ echo '#!/bin/sh
yo
' > monscript.sh
$ chmod +x monscript.sh && ./monscript.sh
```

On voit que `yo` est considérée comme une commande introuvable : elle n'existe pas dans le `PATH`, l'alias n'est pas reconnu, même si défini de façon persistante.

I.6 Les fonctions

On peut aussi utiliser des fonctions, par exemple la suivante qui permet d'effacer automatiquement les fichiers postfixés par `~` qu'`emacs` crée comme backups avant l'édition des fichiers voulus.

```
function rmc {
    files=`ls --color=never .*~ *~ 2> /dev/null`
    if [ "x${files}" != "x" ] ; then
        for i in ${files}; do
            echo rm ${i}
            rm "${i}"
        done
    fi
}
```

II Aller plus loin

II.1 Un peu plus loin avec des scripts

Des commandes supplémentaires peuvent être pratiques dans les shells, par exemple `grep`, `cut`, `sed`, `tr`, mais aussi `bc` ou `dc` pour faire des calculs, ou encore `awk` qui permet de facilement de faire des `cut` et des opérations mathématiques (`bc` ou `dc`). Ainsi pour calculer une moyenne d'exécution d'un programme qui retourne son temps d'exécution, on peut par exemple utiliser :

```
tmp="scale=6 ; (0"
idxmax=3
for j in $(seq 1 $idxmax); do
    tmp="$tmp+$(./monprogramme)"
done
moy_duration=$(echo "$tmp)/$idxmax" | bc | awk '{printf "%f", $0}')
```

```
echo $moy_duration
```

On utilise ici `bc` pour faire des calculs : on établit dans son script qu'on souhaite des nombres codés sur 6 digits (0.000001 ou 999999) puis l'opération à effectuer. `awk` est ensuite utilisée car elle permet de convertir automatiquement la valeur .004 en 0.004.

II.2 Une commande less dopée aux stéroïdes

Q.II.1) - Aller plus loin :

— `lesspipe -h` et `https://www-zeuthen.desy.de/~friebel/unix/less/README`
Exécutez `less filename` où filename est un fichier C, une archive ou `/bin/bash`.
Puis : `export LESSCOLOR=yes; export LESS="-R -M --shift 5"; LESSOPEN="| lesspipe %s"`
et recommencez. Commentaire ?

Attention, il y a plusieurs variantes de `lesspipe`, celle décrit sur le README pointé gère la coloration syntaxique paramétrée par `$LESSCOLOR` mais pas celui fourni par défaut par Debian.

— Exécutez `man man` puis

```
man() {
env LESS_TERMCAP_mb=$'\E[01;31m' \
    LESS_TERMCAP_md=$'\E[01;38;5;74m' \
    LESS_TERMCAP_me=$'\E[0m' LESS_TERMCAP_se=$'\E[0m' \
    LESS_TERMCAP_so=$'\E[38;5;246m' \
    LESS_TERMCAP_ue=$'\E[0m' \
    LESS_TERMCAP_us=$'\E[04;38;5;146m' \
    man "$@"
}
man man
```