

UNIX MAGIC



UNITECH SOFTWARE INC. RESTON, VIRGINIA
(703) 860-3000

Table des matières

1	Rappels	3
1.1	Terminologie	3
1.2	Les shells	4
1.3	L'invite de commandes	5
1.4	Les fichiers	6
1.5	Les descripteurs de fichier	8
2	Bases de bash	11
2.1	La ligne de commandes	11
2.2	Le shell interactif	12
2.2.1	REPL	12
2.2.2	Le répertoire courant d'exécution	14
2.2.3	Environnement et héritage	15
2.3	Les variables	17
2.3.1	Déclaration et affectation	17
2.3.2	Accès au contenu d'une variable	18
2.3.3	Variables d'environnement et portée	20
2.3.4	Variables d'environnement globales	21
2.4	Quoting et échappement	24
2.5	Les commandes	27
2.5.1	Commandes internes et externes	27
2.5.2	Affichage, retour et statut de sortie	27
3	Les bases du scripting	29
3.1	Contrôle du flux d'exécution	29
3.1.1	Les conditions	29
3.1.2	Les boucles	29
3.2	Gestions des arguments	29
3.3	Gestion des options	29

4	Notions avancées de bash	31
4.1	Opérateur de contrôle	31
4.2	Sous-shells	31
4.3	Redirection de flux	31
4.4	Expansion de variables	31
5	Scripting avancé	33
5.1	Fonctions	33
5.2	Gestion des erreurs	33
	Bibliographie	35

Ce document vise à couvrir les bases du langage de scripting `bash`, l'un des langages les plus répandus sur les systèmes d'exploitation de type UNIX et Linux. Les thématiques abordées seront les suivantes. Dans un premier temps un certain nombre de définitions et de rappels théoriques seront présentés. Des concepts relatifs aux systèmes d'exploitation Linux et UNIX seront également décrits, ces derniers permettant une meilleure compréhension du fonctionnement du langage étudié.

Dans une seconde section, nous présenterons les principes et la syntaxe de base de `bash`. Cette section sera abordée sous l'angle de l'invite de commande¹ et donc dans le cadre d'une utilisation interactive du langage.

La troisième section s'intéressera à la création de scripts et couvrira les notions qui lui sont intimement liées telles que la portée des variables ou les arguments.

Les quatrième et cinquième sections viseront à introduire des concepts plus avancés de la programmation `bash` tels que les opérateurs de contrôle, la gestion des flux ou encore la déclaration et l'utilisation de fonctions.

Pour finir, il est important de noter que, même si un certain nombre de commandes est introduit, notamment à des fins d'exemples, ce document n'a pas pour vocation à recenser les commandes utiles. Une cheat sheet vous a été fournie à cet effet.

1. Une définition de l'invite de commande est donnée à la sous-section 1.3.

1 Rappels

Ce chapitre vise à poser les bases théoriques et sémantiques du document. Nous effectuerons un point sur le vocabulaire utilisé, introduiront les définitions des concepts utilisés mais aussi présenterons rapidement certains concepts relatifs aux OS Linux et UNIX. Si la lecture de cette section n'est pas nécessaire pour suivre le document, cette dernière donnera les clés permettant de mieux comprendre le fonctionnement du langage étudié.

1.1 Terminologie

Nous détaillons ici les différentes terminologie qui seront utilisées tout au long du document.

Console Historiquement, une console est un périphérique physique permettant à un opérateur d'interagir avec le système de traitement de l'information.

Terminal Historiquement, un terminal est un type de console communiquant avec le système de traitement de l'information via un réseau.

Émulateur de terminal (terminal virtuel) Logiciel permettant de reproduire le fonctionnement d'une console physique. Il s'agit de logiciels tels que `gnome-terminal`, `xterm`, `konsole`, ... Par abus de langage ces derniers sont également appelés terminaux ou consoles remplaçant le sens précédemment mentionné. Au cours de ce document nous utiliserons indistinctement les termes *émulateurs de terminal*, *terminal virtuel*, *terminal*, *console virtuelle* et *console*¹.

TTY Abréviation de Teletype. Teletype était une des marques les plus répandues de téléscripteurs. Un téléscripteur étant un périphérique pouvant envoyer ou recevoir de l'information via communication électrique ou électromagnétique (ondes radio). Si un certain nombre de téléscripteurs sont des consoles (permettant la communication avec un système de traitement de l'information), tous ne le sont pas. En effet, les téléscripteurs apparaissent dans les années 1880 avec un usage destiné à la télégraphie. Aujourd'hui, un TTY désigne un type particulier de terminal, connecté directement à l'entrée standard de l'OS et accessible via les combinaisons de touches de type `<Ctrl> + <Alt> + <Fx>`, où `x` prend des valeurs allant de 1 à 12 en fonction des distributions et des configurations.

1. Il est à noter que dans un certain nombre de sources, la console est définie comme étant ce que nous appellerons ici un tty.

Interface en ligne de commandes (CLI) Interface Homme Machine dans laquelle les communications entre l'utilisateur et le processus mettant à disposition cette interface se font en mode texte.

Interface système (shell) Logiciel offrant une interface en ligne de commandes avec le système d'exploitation. Lorsqu'un utilisateur se connecte via un TTY, le système d'exploitation démarre pour ce dernier un shell. La terminaison de ce processus entraîne la déconnexion de l'utilisateur. Par abus de langage, cette interface est également appelée *ligne de commandes*. Ce terme sera également utilisé en ce sens dans ce document.

Nous attirons l'attention du lecteur sur le fait que, ce document portant uniquement sur bash, l'utilisation du terme *shell* englobera uniquement les interfaces systèmes des systèmes UNIX et Linux. À ce titre, le contenu présenté dans les sections suivantes (et plus particulièrement la section 1.2) n'est pas nécessairement applicable à cmd et dans une plus forte mesure PowerShe`ll`².

1.2 Les shells

Le *Thompson shell*, premier shell UNIX, a été développé par Ken Thompson³ en 1971 [1]. D'autres shells ont existé préalablement notamment pour des systèmes d'exploitation tels que *Multics*. Très rudimentaire, ce dernier a très rapidement été refondu pour obtenir le *Programmer Workbench shell* (*PWB* ou *Mashey Shell*), écrit et maintenu par John Mashey [1]. L'objectif de ce dernier étant d'offrir une interface de programmation plus poussée pour faciliter l'automatisation des tâches. C'est, par exemple, le premier shell à intégrer, en interne, des structures de contrôle de flux telles que la syntaxe **if then else** end i f ou les boucles **while**.

Ces shells ne sont plus utilisés de nos jours, mais serviront de base pour la création du *Bourne shell* (*sh*). Développé par Stephen Bourne et publié en 1979 avec la septième mouture UNIX, ce dernier s'inspire du langage de programmation `Algol 68`[2]. Il est le premier shell à être pensé pour être un langage de scripting à part entière. Il introduit également des concepts que nous détaillerons dans des sections ultérieures tels que la notion d'environnement d'exécution ou la portée des variables. Ce shell est présent nativement dans la quasi totalité des distributions Linux et UNIX au chemin `/bin/sh`⁴.

Un grand nombre de shells très largement utilisés aujourd'hui s'inspirent du *Bourne Shell*, malgré des différences syntaxiques très marquées pour certains comme `csh` ou `tcsh`, voire sont directement dérivés de ce dernier et assurent une rétrocompatibilité complète comme `ksh`, `zsh` ou encore `bash`.

Même si ce document est écrit en rapport au *Bourne again shell* (`bash`), la quasi totalité des syntaxes présentées sont compatibles `sh`, ce choix trouvant sa justification dans les trois points

2. À l'inverse, il est important de noter que ce qui est présenté n'est pas nécessairement faux non plus pour ces shells.

3. Aussi connu pour avoir développé, avec Dennis Ritchie, le langage C.

4. Il arrive que cet exécutable soit un lien symbolique vers un shell totalement compatible.

suivants:

1. la compatibilité avec `sh` assure une très grande portabilité (notamment avec UNIX et ses dérivés⁵),
2. le temps de cours ne permet pas de couvrir de manière satisfaisante l'ensemble des syntaxes et les subtilités d'interprétation entre les syntaxes `sh` et `bash`,
3. ce cours se veut être une introduction au scripting `bash`, il ne nécessite donc pas (à de rares exceptions) d'avoir recours aux fonctionnalités avancées de introduites par `bash`.

1.3 L'invite de commandes

Également appelé *prompt*, l'invite de commandes est le texte affiché par le shell préfixant la frappe utilisateur. Un exemple d'invite de commandes est donnée à la Figure 1.1. Sur cette dernière, l'invite de commandes est le texte: `guillaume@saturn:~$`.

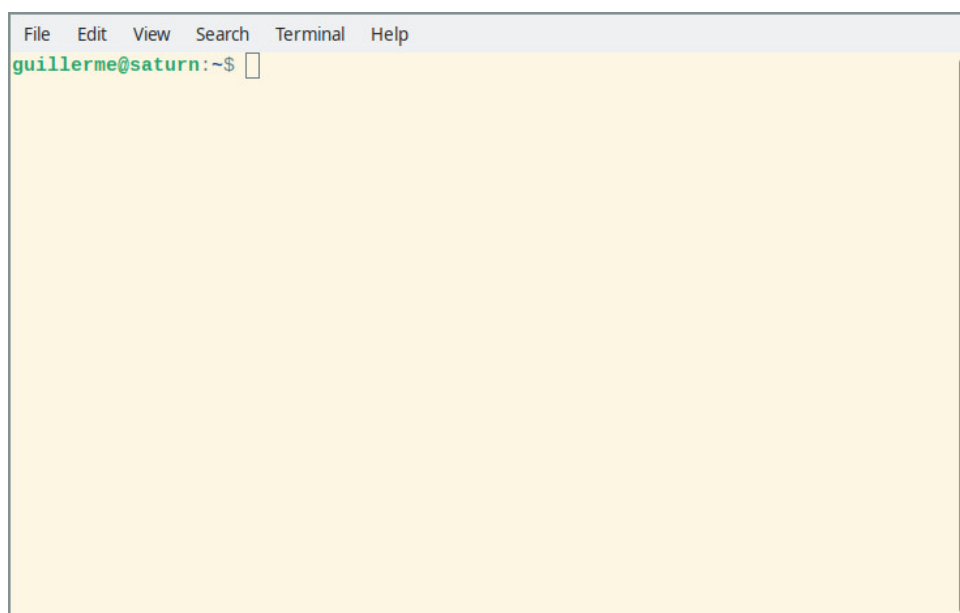


FIGURE 1.1 – Capture d'écran du terminal `gnome-terminal` exécutant `bash`

Cette dernière donne un certain nombre d'informations utiles. Ainsi on retrouve:

- `guillaume`: l'utilisateur exécutant le shell courant,
- `saturn`: le nom de la machine sur laquelle est exécuté le shell courant,

5. Incluant également MacOS.

-
- `~`: le répertoire courant ⁶. `~` est un alias pour le répertoire personnel de l'utilisateur, ici il est équivalent à `/home/guill erme`.
 - `$`: est le symbole représentant le fait que l'utilisateur connecté est un utilisateur normal, en opposition au super utilisateur pour lequel le symbole sera `#`.

À noter que votre prompt pourrait être totalement différent, ce dernier est configurable au travers de la variable `PS1` ⁷. Son contenu varie donc d'un shell à l'autre, d'une distribution à l'autre, ...

Dans la suite du document, l'invite de commande ne sera pas reprise, seuls les caractères `$` et `#` seront placés en début de ligne afin d'indiquer qu'il s'agit de commandes exécutées dans une session interactive de bash et de déterminer les permissions avec lesquelles les commandes sont exécutées (simple utilisateur ou super utilisateur).

1.4 Les fichiers

Comme évoqué en Section 1.2, les shells permettent d'interagir avec le système d'exploitation. Il est donc nécessaire de comprendre certains concepts pour bien comprendre le fonctionnement de bash. Parmi ces concepts se trouve la notion de fichier.

Un fichier peut être vu comme une couche d'abstraction de l'information permettant à l'utilisateur de ne pas avoir à se préoccuper du support sous-jacent [3]. De manière plus intuitive, un fichier est une interface permettant d'accéder et/ou de manipuler de l'information indépendamment de sa représentation physique.

Lorsque vous copiez par exemple un morceau de musique depuis une clé USB vers un disque dur, la représentation en mémoire de ces deux copies est possiblement différente (dépendant du système de fichier) mais également la manière de lire et d'écrire l'information est différente, les supports physiques étant différents. Cependant l'information décrite est la même pour les deux copies, les deux fichiers sont alors identiques.

Il découle de cette abstraction que toute information (et de manière générale toute source d'information) peut être représentée par un fichier. Ainsi, un disque dur peut être vu comme un fichier, un répertoire peut être vu comme un fichier et même l'affichage généré par un processus peut être vu comme un fichier. Il s'agit là du parti pris des systèmes des familles UNIX et Linux. Ainsi sous Linux, tout est fichier ⁸.

Attentions cependant, même s'ils présentent la même interface et même s'ils peuvent être manipulés et/ou lus de manière similaire, tous les fichiers ne sont pas équivalents. Il existe en effet différents types de fichiers:

6. La notion de répertoire courant est détaillée en Section 2.2.

7. Et ce depuis la sortie de sh.

8. Et tout ce qui n'est pas fichier est processus.

les fichiers spéciaux de blocs (*block special files*) sont des fichiers représentant des périphériques et/ou espaces mémoires (clé USB, RAM, mémoire partagée). Notez qu'ils représentent le périphérique en lui-même et non son contenu, ainsi la lecture ou l'écriture dans un de ces fichiers permet une manipulation directe de la mémoire.

les fichiers spéciaux de caractère (*character special files*) ces fichiers représentent des périphériques d'entrée/sortie tels que des imprimantes, des modems ou plus généralement des périphériques acceptant ou générant des flux de caractères.

les tubes (*pipes*) les tubes sont des fichiers⁹ permettant à deux processus de communiquer. Ainsi lorsque deux processus, disons p_1 et p_2 , communiquent par pipe, ils établissent chacun de leur côté la connexion au même pipe. Lorsque p_1 veut envoyer de l'information à p_2 , il écrit cette dernière dans le pipe, comme il écrirait dans un fichier, à l'inverse p_2 pourra accéder à cette information en lisant dans le pipe, comme il lirait dans un fichier.

les répertoires les répertoires sont utilisés principalement pour ordonner les fichiers sur le disque. Il s'agit de fichiers dont le contenu est une liste d'autres fichiers à savoir les fichiers qu'ils *contiennent*. Ils sont conceptuellement équivalents aux dossiers sous Windows. Un répertoire peut contenir n'importe quel type de fichier, en ce compris d'autres répertoires, on parle alors de sous-répertoire. Le répertoire unique contenant directement ou indirectement (*i.e.* via des sous-répertoires) l'ensemble des fichiers du disque est appelé répertoire **racine** et est noté `/`. La hiérarchie globale induite par la relation de contenance est appelée **arborescence de fichiers**. L'identification d'un fichier dans cette hiérarchie se fait en énumérant les répertoires contenant ce dernier, et ce en partant depuis la racine. On parle alors de **chemin absolu** du fichier. Ainsi un fichier `file` contenu dans un répertoire `dir1`¹⁰ lui-même contenu dans le répertoire racine aura pour chemin relatif `dir1/file`.

les liens les liens sont des fichiers contenant une référence vers un autre fichier de l'arborescence. Ils peuvent être *symboliques* ou *physiques*¹¹ et se rapprochent (spécialement pour les liens symboliques) des raccourcis Windows.

les fichiers réguliers un fichier est dit régulier s'il n'entre dans aucune des catégories précédentes.

Un exemple d'arborescence est donnée à la Figure 1.2. Sur cet exemple on peut identifier:

- trois fichiers spéciaux de blocs: `/dev/sda`, `/dev/sda1` et `/dev/sda2`. Ces fichiers identifient respectivement le disque dur (`sda`), la première partition sur ce dernier (`sda1`) ainsi que la seconde partition du même disque (`sda2`),
- deux fichiers spéciaux de caractères: `/dev/lp0` et `/dev/video0` identifiant respectivement

9. Ce sont en réalité des pseudo-fichiers [3]: ils offrent les mêmes interfaces de lecture et d'écriture qu'un fichier, mais ne permettent pas d'accès à des adresses arbitraires par exemple.

10. Par convention un répertoire est noté avec le suffixe `/`, qui est le caractère servant de séparateur dans l'écriture des chemins.

11. La différence entre les deux sortant du cadre du cours.

-
- une imprimante (sortie) et une webcam (entrée) ¹²,
- deux fichiers réguliers: `/home/guillaume/git/bashrc` et `/home/guillaume/git/.git/config`,
 - un lien: `/home/guillaume/.bashrc` pointant sur le fichier `git/bashrc`. Notons ici l'absence du `/` en début de chemin, indiquant un **chemin relatif**. Un chemin relatif est exprimé en fonction d'une référence dans l'arborescence. Ici le lien se trouve dans le répertoire `/home/guillaume/`, ce dernier est utilisé comme référence. Un chemin absolu est alors obtenu par concaténation du chemin absolu du répertoire de référence (`/home/guillaume/`) et du chemin relatif du fichier (`git/bashrc`) donnant ici `/home/guillaume/git/bashrc`.
 - ainsi qu'un certain nombre de répertoires,

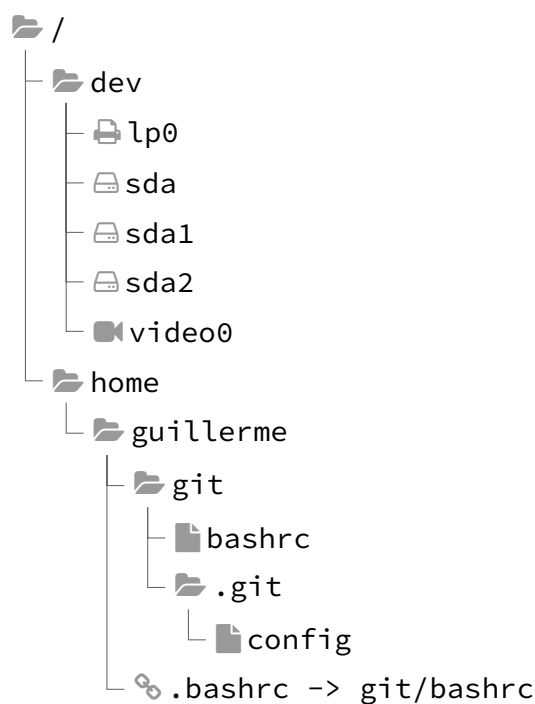


FIGURE 1.2 – Exemple d'arborescence de fichiers

1.5 Les descripteurs de fichier

Avant de pouvoir lire ou écrire dans un fichier, il est nécessaire d'ouvrir ce dernier. C'est lors de cette opération que les permissions sur ce dernier sont vérifiées. Si les permissions de l'utilisateur sont

12. Il est intéressant de remarquer que tous les fichiers spéciaux sont contenus dans le répertoire `/dev/`, ce dernier a en effet pour vocation de contenir les fichiers spéciaux relatifs aux périphériques branchés à l'ordinateur. Notez cependant qu'il n'est pas le seul répertoire à contenir des fichiers spéciaux.

suffisantes, le système retourne un entier identifiant l'accès. Cet identifiant est appelé un *descripteur de fichier*. Notons bien que cet identifiant identifie la “connexion” de l'utilisateur au fichier et pas uniquement le fichier en lui même¹³. Ainsi, deux utilisateurs ouvrant le même fichier manipuleront deux descripteurs de fichier différents.

Rappelons que, sous Linux et UNIX, tout (ou presque) est fichier. Ainsi, les entrées et sorties d'un processus sont également des fichiers. Les fichiers virtuels identifiant ces dernières sont ouverts au démarrage du système et donnent donc lieu à des descripteurs de fichiers. Ces derniers sont partagés par l'ensemble des utilisateurs du système et sont au nombre de trois:

1. `0`: identifie l'entrée standard (aussi appelée `stdin`) du processus. L'entrée standard est le flux utilisé par le processus pour lire les données. Il existe différentes manières d'alimenter ce dernier, telles que les pipes et les fichiers (abordés à la Section 4.3), ou simplement le clavier.
2. `1`: identifie la sortie standard (aussi appelée `stdout`) du processus. La sortie standard est le flux utilisé, par défaut, par le processus pour écrire des données. La gestion de ce flux sera abordé à la Section 4.3. Dans la cadre de ce cours, la sortie standard sera généralement l'affichage textuel du terminal dans lequel est exécuté le script.
3. `2`: identifie la sortie d'erreur (aussi appelée `stderr`) du processus. Cette sortie fonctionne comme la sortie standard à ceci près qu'elle a pour vocation à être la destination des messages d'erreur. Le flux sous-jacent est indépendant du flux `stdout` et peut donc être manipulé indépendamment de ce dernier. Lors de l'exécution d'un script dans un terminal, la sortie d'erreur correspond, tout comme la sortie standard, à l'affichage textuel du terminal.

13. Il s'agit là d'une vue simplifiée des descripteurs de fichiers, mais la gestion de ces derniers par le système sort du cadre de ce cours.

2 Bases de bash

2.1 La ligne de commandes

Une ligne de commande peut être définie comme une suite d'un ou plusieurs mots séparés par des tabulations ou des espaces [4]. Le premier mot de la ligne est alors appelé **commande**, les autres mots, s'ils existent sont alors des **paramètres** (ou **arguments**) de cette commande.

Considérons la ligne de commande suivante:

```
1 $ ls -l --all -w 100 --block-size=M /var/log
```

Dans cet exemple, `ls` est la commande tandis que `-l`, `--all`, `-w 100`, `--block-size=M` et `/var/log` sont des paramètres de la commande `ls`. Cependant tous ces paramètres ne sont pas interprétés de la même manière, en effet les paramètres ayant pour préfixe `-` sont appelés **options**.

Les **options** sont un type de paramètres visant à modifier le comportement d'une commande. Certaines de ces options attendent leur propre argument, on parle alors d'**argument d'option** ou de **paramètre d'option**. Ces options et leur interprétation sont définies par la commande en question et leur documentation est accessible via la page de manuel de la commande.

Pour notre exemple:

- `ls` est la **commande** permettant de lister les fichiers présents dans un répertoire passé en paramètre ou si ce dernier n'existe pas dans le répertoire courant ¹.
- `-l` est une option *courte* n'attendant pas d'argument activant l'affichage au format long (affiche un certain nombre d'informations supplémentaires pour chaque fichier). Une option dite *courte* est une option dont le nom est composé d'un unique caractère et préfixé par `-`.
- `--all` est une option dite *longue* sans argument permettant d'afficher les fichiers cachés (*i.e* les fichiers dont le nom est préfixé par un `.`). Une option longue est généralement ² préfixée par

1. cf Section 2.2 pour une définition du répertoire courant.

2. C'est en tout cas la convention de nommage définie par le projet GNU (<https://www.gnu.org/gnu/gnu.html>) et qui prédomine sur Linux. Il est cependant à noter qu'un certain nombre de commandes, notamment celles directement héritées des systèmes UNIX (et plus particulièrement BSD) et IBM, ne respectent pas cette convention. Ainsi des commandes telles que `find` définissent des options longues préfixées par un unique `-`. Il est donc important de se référer à la page de manuel.

un double tiret `--` et dont le nom est composé d'un mot ou de plusieurs mots séparés par des tirets. Cette option longue est équivalente à l'option *courte* `-a`.

- `-w` est une option (dite *courte* car préfixée par `-` et étant identifiée par une unique lettre) permettant de définir la longueur maximum de la ligne lors de l'affichage de `\s`. Cette option attend un paramètre d'option.
- `100` est le paramètre de l'option `-w` indiquant qu'une ligne ne peut pas excéder `100` caractères.
- `--block-size=M` est un paramètre composé d'une option longue `--block-size` et d'un paramètre d'option `M`. Pour les options longues, les paramètres d'options sont généralement³ définis avec un `=`. L'option indique ici que la taille devra être calculée et affichée avec le méga octet comme unité de mesure (la valeur étant arrondie à l'unité supérieure).

Par abus de langage, dans la suite de ce document, nous utiliserons de manière indifférenciée les termes *ligne de commande* et *commande* lorsque le contexte n'est pas ambigu.

2.2 Le shell interactif

2.2.1 REPL

Lors du lancement d'un émulateur de terminal (ou lors d'une connexion à un `tty`), une invite de commandes est affichée à l'utilisateur. Cette invite de commandes indique l'utilisation interactive de `bash`. Ces sessions interactives se caractérisent par:

- l'écriture de lignes de commandes dans le terminal, chacune terminée par un retour à ligne⁴,
- l'évaluation par le shell de la ligne de commande fournie,
- un nouvel affichage de l'invite de commande, indiquant à l'utilisateur la fin de l'évaluation de la ligne précédente.

À ce stade, nous considérons l'évaluation d'une ligne de commande comme la réalisation des étapes suivantes:

1. lecture de la ligne de commandes,
2. exécution de la commande avec les bons paramètres,
3. affichage de la sortie standard et de la sortie d'erreur dans le terminal.

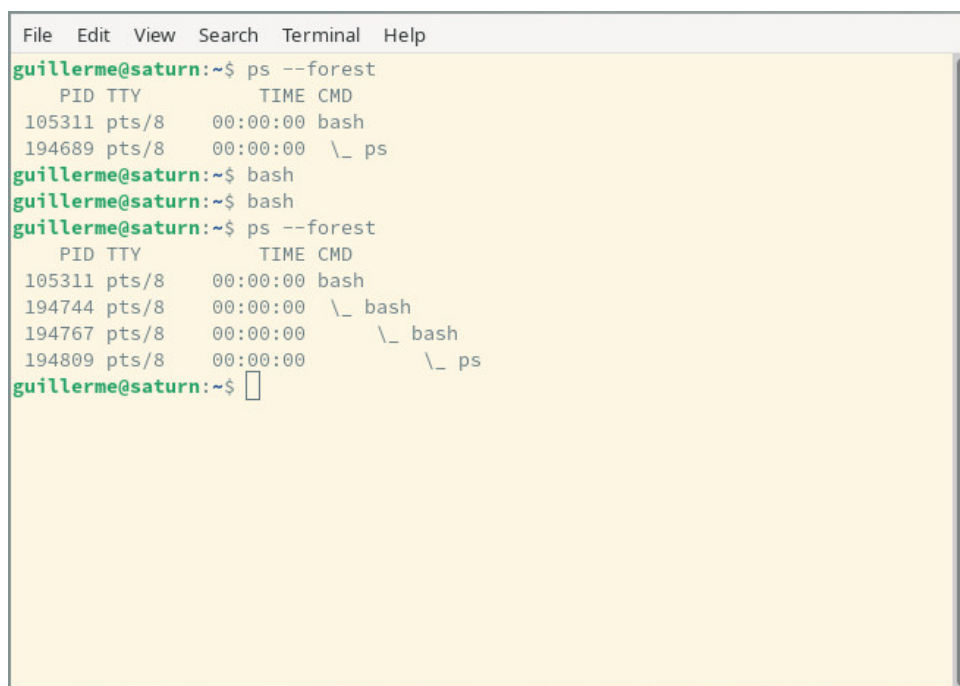
Ce type de fonctionnement est caractéristique d'un REPL (Read-Eval-Print Loop). Le concept de REPL a été introduit pour LISP en 1964 par Peter Deutsch [5], il définit la séquence suivante:

3. C'est en tout cas la convention de nommage définie par le projet GNU (<https://www.gnu.org/gnu/gnu.html>) et qui prédomine sur Linux. Il est cependant à noter qu'un certain nombre de commandes, notamment celles directement héritées des systèmes UNIX (et plus particulièrement BSD) et IBM, ne respectent pas cette convention. Ainsi des commandes telles que `find` définissent des options longues préfixées par un unique `-`. Il est donc important de se référer à la page de manuel.

4. On suppose ici un retour à la ligne non échappé. La définition des échappements de caractères spéciaux est donnée à la Section 2.4.

-
1. lecture de l'entrée (Read),
 2. exécution de la commande (Eval),
 3. affichage du résultat (Print),
 4. retour à l'action 1 (Loop).

Il est intéressant de noter que l'exécution de la commande (étape 2) crée généralement un nouveau processus⁵, ce processus est alors un processus **fil**s du shell ayant invoqué la commande. Ces liens de filiation sont mis en évidence sur la Figure 2.1. Sur cet exemple, nous utilisons la commande `ps` permettant de lister les processus actifs. Sans options, cette commande affiche uniquement les processus actifs de l'utilisateur courant ayant été lancé depuis le même terminal que celui dans lequel s'exécute la commande. L'option `--forest` permet de changer le format de la sortie de la commande.



```
File Edit View Search Terminal Help
guillerme@saturn:~$ ps --forest
  PID TTY          TIME CMD
 105311 pts/8        00:00:00 bash
 194689 pts/8        00:00:00 \_ ps
guillerme@saturn:~$ bash
guillerme@saturn:~$ bash
guillerme@saturn:~$ ps --forest
  PID TTY          TIME CMD
 105311 pts/8        00:00:00 bash
 194744 pts/8        00:00:00 \_ bash
 194767 pts/8        00:00:00 \_ bash
 194809 pts/8        00:00:00 \_ ps
guillerme@saturn:~$
```

FIGURE 2.1 – Illustration des relations de filiation entre les processus

Ici le premier appel à `ps` met en évidence un processus `ps` ayant pour PID 194689 et pour processus parent, le processus `bash` de PID 105311. La seconde commande exécute un processus `bash` depuis le `bash` courant et la troisième un processus `bash` dans le processus `bash` dans le processus `bash`. Ce jeu de poupées russes est mis en évidence lors du second appel à la commande `ps`.

Nous pouvons également souligner que les processus `bash` s'*imbriquent* les uns dans les autres. En

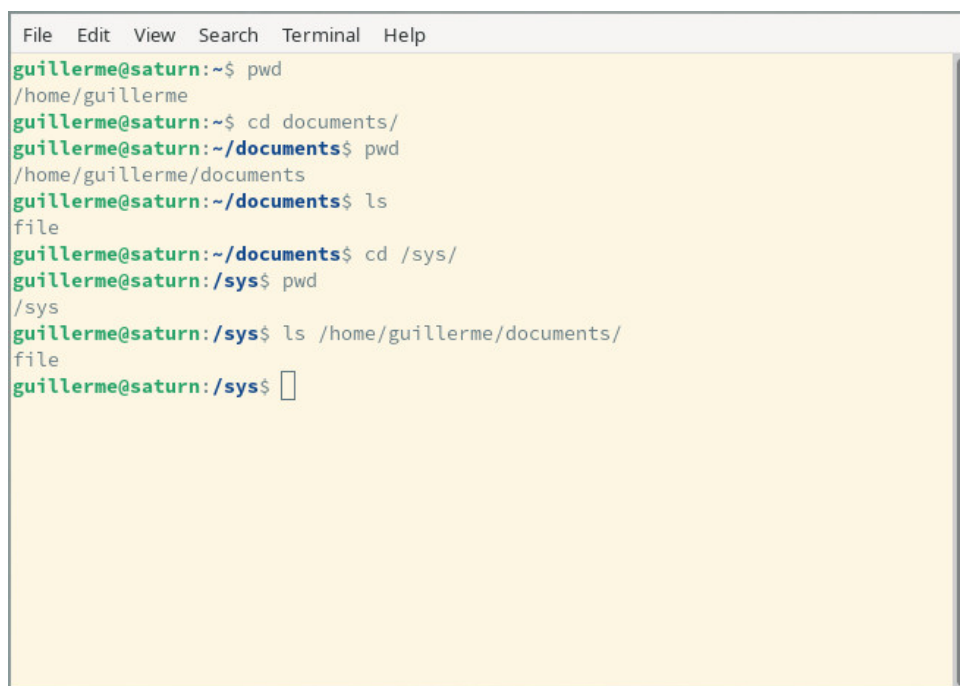
5. Ce n'est pas le cas lors de l'invocation de fonctions (cf Section 5.1).

effet, exécuté de la sorte, le shell est exécuté en mode interactif, le parent ne redevenant actif que lorsque le processus fils se termine.

Depuis sh, l'évaluation d'une ligne de commandes n'est pas réalisée en dehors de tout contexte. Ce contexte, appelé **environnement**, définit tout un ensemble de paramètres influant sur l'évaluation de la ligne de commandes. Parmi ces paramètres se trouve le **répertoire courant d'exécution (current working directory)**.

2.2.2 Le répertoire courant d'exécution

Le **répertoire courant d'exécution** est le répertoire dans lequel est évaluée la ligne de commande et donc dans lequel s'exécute la commande tapée. Il sert notamment de référence lors de l'évaluation des chemins relatifs. Lors du lancement du terminal (ou de la connexion à un t ty), le shell lancé a, par défaut, pour répertoire courant d'exécution, le répertoire personnel de l'utilisateur⁶. Ce dernier peut être obtenu à l'aide de la commande **pwd** et modifié à l'aide de la commande **cd**.



```
File Edit View Search Terminal Help
guillerme@saturn:~$ pwd
/home/guillerme
guillerme@saturn:~$ cd documents/
guillerme@saturn:~/documents$ pwd
/home/guillerme/documents
guillerme@saturn:~/documents$ ls
file
guillerme@saturn:~/documents$ cd /sys/
guillerme@saturn:/sys$ pwd
/sys
guillerme@saturn:/sys$ ls /home/guillerme/documents/
file
guillerme@saturn:/sys$
```

FIGURE 2.2 – Illustration de la notion de répertoire courant

La Figure 2.2 illustre la notion de répertoire courant au travers de l'exécution de différentes commandes:

6. Par défaut le répertoire courant de l'utilisateur user est /home/user/.

-
1. L'utilisateur demande l'exécution de la commande `pwd`. L'utilisateur venant de lancer le terminal, son répertoire personnel est, par défaut, utilisé pour initialiser le répertoire courant. L'invite de commande nous indique que l'utilisateur en question est `guillaume`, son répertoire personnel devrait être `/home/guillaume/`. L'affichage de la commande `pwd` sur la sortie standard (`/home/guillaume`) est donc cohérent.
 2. L'utilisateur modifie le répertoire courant à l'aide de la commande `cd`, le paramètre passé à cette dernière indiquant le nouveau répertoire courant souhaité. Ici le paramètre est `documents/`, le chemin n'étant pas absolu⁷ il est interprété comme relatif par rapport au répertoire courant. Une fois la commande évaluée le nouveau répertoire courant devrait être `/home/guillaume/documents/`.
 3. Un nouvel appel à `pwd` permet de valider le changement de répertoire courant. On remarque également que le changement de répertoire courant est visible dans l'invite de commande.
 4. La commande `ls` est invoquée. Sans paramètre, cette dernière liste les fichiers présents dans le répertoire courant. On peut donc voir que le répertoire `/home/guillaume/documents/` contient le fichier `file`. Son chemin absolu est `/home/guillaume/documents/file`.
 5. Un nouvel appel à la commande `cd` avec un chemin absolu nous permet de changer la valeur du répertoire courant d'exécution à `/sys/`.
 6. Le changement est validé par l'invite de commande et la commande `pwd`.
 7. Le dernier appel à `ls` est cette fois-ci réalisé avec un paramètre contenant un chemin absolu, n'étant donc pas influencé par le répertoire courant d'exécution.

Notons trois notations de répertoires intéressants:

1. `~`: est un alias du chemin absolu du répertoire personnel de l'utilisateur. Il apparaît notamment dans les invites de commande de la Figure 2.2. Pour cette session, `~` a pour valeur `/home/guillaume/` et `~/documents/` est équivalent à `/home/guillaume/documents`.
2. `..`: identifie le répertoire courant.
3. `...`: identifie le répertoire parent du répertoire courant. Ainsi lorsque le répertoire courant a pour valeur `/home/guillaume/`, `..` identifie le répertoire `/home/`. Notez cependant que tout chemin commençant par `..` est un chemin relatif.

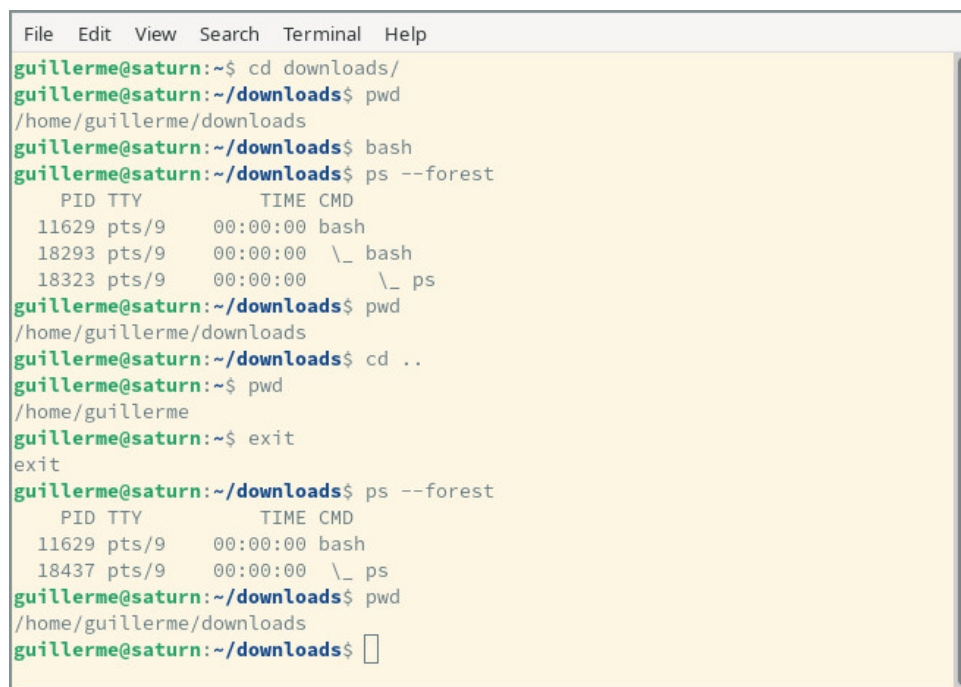
2.2.3 Environnement et héritage

Une question intéressante à se poser est la suivante:

Dans l'exemple précédent, si `ls` ou `pwd` sont des processus à part entière créés par l'instance courante de `bash`, comment déterminent-ils le répertoire courant de ce dernier?

7. Pour rappel, un chemin absolu commence par le répertoire racine `/`.

L'exemple dépeint par la Figure 2.3 vise à apporter un début de réponse à cette question ⁸.



```
File Edit View Search Terminal Help
guillerme@saturn:~$ cd downloads/
guillerme@saturn:~/downloads$ pwd
/home/guillerme/downloads
guillerme@saturn:~/downloads$ bash
guillerme@saturn:~/downloads$ ps --forest
  PID TTY          TIME CMD
 11629 pts/9        00:00:00 bash
 18293 pts/9        00:00:00 \_ bash
 18323 pts/9        00:00:00 \_ ps
guillerme@saturn:~/downloads$ pwd
/home/guillerme/downloads
guillerme@saturn:~/downloads$ cd ..
guillerme@saturn:~$ pwd
/home/guillerme
guillerme@saturn:~$ exit
exit
guillerme@saturn:~/downloads$ ps --forest
  PID TTY          TIME CMD
 11629 pts/9        00:00:00 bash
 18437 pts/9        00:00:00 \_ ps
guillerme@saturn:~/downloads$ pwd
/home/guillerme/downloads
guillerme@saturn:~/downloads$
```

FIGURE 2.3 – Mise en évidence du principe d'héritage d'environnement entre processus.

Dans cet exemple, l'utilisateur `guillerme` commence par changer le répertoire courant (commande `cd` puis confirmation avec la commande `pwd`). Par la suite un second processus `bash` est lancé (commande `bash`), ce dernier étant le fils du processus `bash` initial, comme l'indique la commande `ps --forest`. Cependant malgré la création du nouveau processus, la commande `pwd` indique toujours un répertoire courant égal à `/home/guillerme/downloads/`. Le répertoire courant, tout comme une partie de l'environnement du nouveau processus `bash` a été **hérité** du processus père.

L'utilisateur change le répertoire courant du processus `bash` fils en cours d'exécution puis termine ce dernier avec la commande `exit`. Le processus fils étant maintenant terminé, le processus père reprend la main comme le montre la commande `ps --forest`. Cependant la modification apportée au répertoire courant du fils ne s'est pas répercuté au répertoire courant du père. En effet, lors de la création du processus, l'environnement du processus appelant est en partie copié, la copie servant d'environnement au nouveau processus. Ce dernier travaille donc sur une instance distincte de celle du père et n'est donc pas en mesure de modifier cette dernière.

8. Les mécanismes concrets permettant ce partage d'information seront abordés dans les Sections 2.3 et 2.3.3.

2.3 Les variables

S'il est intéressant de savoir que l'environnement d'un processus est hérité de son père, il est également intéressant de savoir les mécanismes impliqués dans cet héritage. Un grand nombre des paramètres d'environnement sont contenues dans des variables spéciales appelées **variables d'environnement globales**. Avant d'aborder en détail ces dernières (*cf* Section 2.3.3), intéressons-nous à la notion de variable.

Le concept de variable en bash est similaire à celui dans d'autres languages. On peut le définir comme un nom auquel est associée une valeur^[4] ou de manière similaire comme un espace mémoire nommé permettant de stocker le résultat d'une opération^[6].

En bash, le nom d'une variable est également appelé **identifiant** de la variable.

2.3.1 Déclaration et affectation

Il existe différente manière de déclarer une variable. La plus courante et la plus simple est de réaliser une *affectation initiale*, i.e. de lier un mot arbitraire, le nom de la variable, à une valeur à l'aide d'un caractère =. Ainsi, les lignes de commande

```
1 $ my_var="Une valeur"
2 $ my_empty_var=
```

définissent deux variables:

1. la première nommée `my_var` contenant la chaîne de caractères `"Une valeur"`,
2. la seconde nommée `my_empty_var` avec un contenu vide.

La seconde méthode repose sur l'utilisation de la fonction interne `declare`. Les commandes précédentes sont équivalentes aux commandes suivantes:

```
1 $ declare my_var="Une valeur"
2 $ declare my_empty_var
```

Il est à noter que la syntaxe de l'affectation (et de bash de manière plus générale) est très stricte sur la présence (ou non) des espaces⁹. Ainsi, la syntaxe de l'affectation se fait **sans** espace autour du signe =.

Il est également à noter que les noms de variables doivent respecter quelques règles pour être valides:

9. Ceci est principalement dû au fait que l'espace est un délimiteur en bash. Pour rappel il aide à séparer la commande de ses paramètres ainsi que les paramètres entre eux (*cf* Section 1.3).

-
- ils doivent uniquement contenir des caractères alphanumériques ou underscores,
 - ils doivent commencer par un caractère alphabétique ou un underscore.

2.3.2 Accès au contenu d'une variable

Il est possible d'accéder à la valeur stockée dans le contenu d'une variable à l'aide de l'opérateur `$` ¹⁰.
Considérons les lignes de commande suivantes:

```
1 $ my_var="Une valeur"
2 $ my_empty_var=
3 $ echo $my_var
4   Une valeur
5 $ echo my_var
6   my_var
7 $ echo $my_empty_var
8
9 $ echo $unknown_variables
10
```

Les deux premières commandes permettent de déclarer deux variables (`my_var` et `my_empty_var`) et d'affecter la valeur `"Une valeur"` à `my_var`. La troisième ligne utilise la commande `echo`, cette dernière permet d'afficher sur la sortie standard l'ensemble des paramètres qui lui sont fournis. L'affichage de la commande est donné ligne 4. On peut voir que le contenu de la variable `my_var` est affiché par la commande `echo`.

En effet, le nom de la variable ayant été préfixé par `$`, la commande `echo` accède donc à son contenu. On peut d'ailleurs voir avec la ligne de commande donnée ligne 5 que si le nom de la variable n'est pas préfixé par un dollar, le nom sera interprété comme une simple chaîne de caractères par `echo`.

Il est important de noter que la substitution de la variable par sa valeur est faite par bash **avant** l'exécution de la commande `echo`. Ainsi, lors de la saisie de la commande:

```
1 $ echo $my_var
```

bash substitue dans un premier temps la variable par sa valeur:

10. Cette syntaxe a été reprise sur un certain nombre de langages. On peut par exemple citer php, powershell, R, ...

```
1 $ echo Une valeur
```

puis exécute cette dernière (donnant lieu à l’affichage Une valeur).

Enfin les deux dernières lignes de commandes permettent de mettre en avant le fait qu’en bash une variable non définie et une variable contenant une chaîne de caractères vides, se comportent, à de très rares détails près ¹¹, de manière identique.

Enfin il est à noter que lors d’un accès au contenu d’une variable, le nom de cette dernière peut être entouré par des accolades. Ainsi, la syntaxe `$my_var` est équivalente à la syntaxe `${my_var}`. Une telle syntaxe est extrêmement utile pour délimiter le nom de la variable lorsque cette dernière est utilisé, par exemple, dans une concaténation. L’exemple suivant illustre un scénario de ce type:

```
1 $ user="guillaume"
2 $ mkdir /home/$user/$user_new_dir/
3 mkdir: cannot create directory '/home/guillaume//': File exists
```

Ici la variable `user` est déclarée et initialisée avec la valeur `guillaume`. Dans un second temps la commande `mkdir` (permettant de créer un répertoire dont le chemin est passé en paramètre) est appelée. Comme vu précédemment, la substitution des variables est effectuée avant l’évaluation de la commande. Lors de cette étape, bash identifie deux variables candidates à substitution:

1. `$user`: le `/` suivant le nom n’étant pas un caractère autorisé pour un identifiant, il marque la fin de ce dernier. La valeur de la variable est `guillaume`.
2. `$user_new_dir`: ici `_` (et ce qui suit) est un caractère valide pour un identifiant, bash utilise le plus long identifiant valide. Or `user_new_dir` est une variable non déclarée, elle fonctionne donc comme une variable vide.

Il en résulte la substitution suivante: `mkdir /home/guillaume//`. Le fichier existant déjà, la commande `mkdir` affiche donc une erreur appropriée.

Pour pouvoir créer le répertoire `/home/guillaume/guillaume_new_dir/`, il faudra utiliser la syntaxe suivante:

```
1 $ user="guillaume"
2 $ mkdir /home/$user/${user}_new_dir/
```

Ici les accolades permettent de délimiter l’identifiant de la variable.

11. Cf. section ?? pour plus de détails sur les points de différences.

2.3.3 Variables d'environnement et portée

Comme énoncé en début de section, les paramètres de l'environnement d'exécution sont partagés du processus père au processus fils par l'intermédiaire de variables d'un certain type: les **variables d'environnement globales**. Bash définit trois types de variables [7]:

les variables d'environnement globales Aussi appelées simplement *variables d'environnement*, sont des variables partagées par le processus bash et l'ensemble de ses processus fils.

les variables d'environnement locales Aussi appelées simplement *variables* ou *variables de shell* sont des variables internes au processus bash en cours.

les variables locales Sont des variables internes à des fonctions. Ce type sera couvert plus en détail dans la Section 5.1.

2.3.3.1 Variables d'environnement locales

Ces variables sont celles ayant été manipulées jusqu'à présent dans les exemples présentés. Leur déclaration peut se faire à l'aide du mot clé **declare** seul ou d'une simple affectation. Le contenu de ces variables est seulement accessible depuis le processus bash courant. Ce comportement est mis en évidence dans la suite de lignes de commandes suivante:

```
1 $ bash
2 $ ps --forest
3     PID TTY          TIME CMD
4     24400 pts/6        00:00:00 bash
5     71732 pts/6        00:00:00 \_ bash
6     71761 pts/6        00:00:00 \_ ps
7 $ my_var="Une valeur"
8 $ echo "$my_var"
9 Une valeur
10 $ bash
11 $ echo "$my_var"
12
13 $ exit
14 $ exit
15 $ ps --forest
16     PID TTY          TIME CMD
17     24400 pts/6        00:00:00 bash
18     74405 pts/6        00:00:00 \_ ps
```

```
19 $ echo "$my_var"
```

```
20
```

Dans cet exemple, la première commande permet de lancer un sous-shell interactif, l'imbrication des shells est mise en évidence par la commande `ps --forest`. Dans ce sous-shell, une variable `my_var`, contenant la valeur `"Une valeur"`, est créée à l'aide d'une syntaxe d'affectation. L'invocation de la commande `echo` et l'affichage qui en résulte mettent en avant le fait que le contenu de la variable est effectivement accessible.

Un nouveau sous-shell est ensuite créé, dans lequel la variable `my_var` ne contient aucune valeur. Les appels successifs à `exit` permettent de terminer les deux sous-shells précédemment créés. L'invocation de la commande `ps` et son affichage permettent de mettre en évidence le shell dans lequel sont exécutés les commandes `ps` et `echo`. Pour finir, l'invocation de cette dernière montre que le contenu de la variable `my_var`, tel que définit dans le sous-shell, n'est pas accessible au shell parent.

2.3.4 Variables d'environnement globales

Contrairement aux variables d'environnement locales, ces variables se transmettent des processus parents vers les processus enfants. C'est via ce type de variables que sont transmis les valeurs relatives à l'environnement d'exécution.

Un certain nombre d'entre elles sont définies au lancement de `bash` et permettent de définir le fonctionnement de ce dernier. Parmi celles-ci, il peut être intéressant de citer:

- `$PATH` contenant l'ensemble des répertoires dans lesquels les exécutables sont recherchés.
- `$PS1` définissant le texte affiché dans l'invite de commandes.
- `$HOME` contenant le chemin absolu du répertoire personnel de l'utilisateur courant.
- `$PWD` contenant le répertoire courant d'exécution.
- `$UID` l'identifiant de l'utilisateur courant.

L'ensemble des variables d'environnement globales définies, ainsi que leur valeur, peut être obtenu à l'aide de la commande `env`.

Il est également possible de déclarer de nouvelles variables d'environnement à l'aide de la commande `export` ou de l'option `-x` de la commande `declare`.

```
1 $ declare -x my_env_var="Bad value"
2 $ my_env_var="Correct value"
3 $ export another_env_var="Correct value again"
4 $ echo "$my_env_var"
```

```

5 Correct value
6 $ echo "$another_env_var"
7 Correct value again
8 $ bash
9 $ ps --forest
10      PID TTY          TIME CMD
11      24400 pts/6        00:00:00 bash
12      81626 pts/6        00:00:00  \_ bash
13      81683 pts/6        00:00:00    \_ ps
14 $ echo "$my_env_var"
15 Correct value
16 $ echo "$another_env_var"
17 Correct value again
18 $ my_env_var="Changed value"
19 $ echo "$my_env_var"
20 Changed value
21 $ bash
22 $ ps --forest
23      PID TTY          TIME CMD
24      24400 pts/6        00:00:00 bash
25      81626 pts/6        00:00:00  \_ bash
26      82065 pts/6        00:00:00    \_ bash
27      82112 pts/6        00:00:00      \_ ps
28 $ echo "$my_env_var"
29 Changed value
30 $ echo "$another_env_var"
31 Correct value again
32 $ exit
33 $ export $child_env_var="Coming from son"
34 $ exit
35 $ echo $my_env_var
36 Correct value
37 $ echo "$child_env_var"
38

```

L'exemple précédent permet de mettre en évidence les comportements suivants:

- les valeurs des variables déclarées à l'aide de `declare -x` ou avec le mot clé `export` sont

-
- accessible dans les processus fils du shell dans lequel elles ont été créées,
 - la modification de la valeur d'une variable d'environnement globale est propagée aux processus fils du shell dans lequel a effectué la modification mais ne se propage pas au processus père,
 - la valeur d'une variable d'environnement globale créée dans un shell fils n'est pas accessible au shell père.

Pour finir, il est également possible de définir des variables d'environnement globales dont la valeur sera accessible uniquement aux processus fils et pas au processus en cours. L'exemple suivant présente cette syntaxe et met en évidence le comportement attendu.

```
1 $ echo $tmp_env_var
2
3 $ ps --forest
4     PID TTY          TIME CMD
5     24400 pts/6        00:00:00 bash
6     134541 pts/6        00:00:00 \_ ps
7 $ echo $tmp_env_var
8
9 $ tmp_env_var="variable temporaire" bash
10 $ ps --forest
11     PID TTY          TIME CMD
12     24400 pts/6        00:00:00 bash
13     134766 pts/6        00:00:00 \_ bash
14     134810 pts/6        00:00:00 \_ ps
15 $ echo $tmp_env_var
16     variable temporaire
17 $ exit
18 $ echo $tmp_env_var
19
```

La ligne de commande de la ligne 9 permet d'invoquer un processus bash pour lequel une variable d'environnement globale `tmp_env_var` est créée et contient la valeur `"variable temporaire"`. Cette variable est bel et bien accessible dans le processus fils, mais n'existe plus une fois de retour dans le processus père. À noter que si la variable `tmp_env_var` avait existé avant la ligne de commande de la ligne 9, la valeur contenue dans cette dernière n'aurait pas été modifiée par cette ligne de commande.

Dans la suite de ce document, lorsque le contexte sera non ambigu, nous utiliserons l'appellation *variable d'environnement* pour désigner une *variable d'environnement globale* et le terme *variable* pour désigner une *variable d'environnement locale*.

2.4 Quoting et échappement

Bash définit un ensemble de caractères spéciaux qu'il est important de connaître et de savoir manipuler afin de se prémunir de tout comportement non désiré. Ces caractères ont une interprétation fortement liée aux mécanismes d'interprétations du shell et plus particulièrement aux différents types de quoting¹².

Parmi les caractères spéciaux déjà rencontrés, citons par exemple `$` qui, lorsqu'il préfixe une chaîne de caractères donnée, permet d'accéder à la valeur contenue dans la variable ayant pour identifiant cette même chaîne de caractères.

Nous donnons ci-dessous une liste de quelques un de ces caractères spéciaux à connaître:

- `$` comme précédemment évoqué permet, lorsqu'il préfixe une chaîne de caractère, l'accès à la valeur contenue dans la variable ayant pour identifiant ladite chaîne de caractères.
- espace** sert de délimiteur entre la commande et ses différents paramètres
- retour à la ligne** déclenche l'interprétation de la ligne de commande que ce caractère termine
- `\` est le caractère d'échappement, utilisé comme préfixe d'un caractère spécial, `\` permet d'empêcher l'interprétation du caractère. Ainsi `echo \$a` affichera `$a`.
- `;` permet de marquer la fin d'une commande. Une ligne de commandes peut contenir plusieurs commandes séparées par `;`¹³. Lors de l'exécution de la ligne de commandes, les commandes la composant seront exécutées de manière séquentielle.
- `"` permet, lorsqu'il délimite de part et d'autre une chaîne de caractères, d'échapper l'ensemble des caractères spéciaux contenus dans la chaîne à l'exception des caractères `$`, `\` et `"`. L'action d'entourer une chaîne de caractère avec deux de ces symboles est appelée **double quoting** ou simplement **quoting**.
- `'` permet, lorsqu'il délimite de part et d'autre une chaîne de caractères, d'échapper l'ensemble des caractères spéciaux contenus dans la chaîne y compris les caractères `$`, `\` et `"`. L'action d'entourer une chaîne de caractère avec deux de ces symboles est appelée **single quoting**.

Il est important de noter que le type de quoting peut grandement changer le sens d'une ligne de commandes. Les exemples ci-dessous donnent quelques illustrations des effets du quoting.

```
1 $ echo Une super mise en page
2 Une super mise en page
```

12. Ce terme, qui désigne l'action de mettre une chaîne de caractères entre guillemets simples (`' '`) ou guillemets doubles (`" "`), n'a pas d'équivalent en français. Aussi pour des questions de lisibilité, nous utiliserons cet anglicisme tout au long du document ainsi que le verbe *quoting*. De plus pour éviter toute confusion avec les guillemets typographiques («»), nous utiliserons les anglicismes *simple quote*, respectivement *double quote*, pour désigner le caractère `"`, respectivement `"`.

13. On parle alors de **liste de commandes**.

```
3 $ echo "Une  super  mise en page"
4   Une  super  mise en page
```

Ce premier exemple permet de mettre en évidence le rôle de délimiteur de l'espace lorsqu'il est non échappé. Lors de l'évaluation de la ligne de commande, les étapes suivantes sont réalisées:

1. la ligne est découpée en fonction des différents séparateurs (ici seuls des espaces sont présents),
2. `echo` est identifié comme la commande, la liste des paramètres est construite:
["Une", "", "", "super", "", "", "mise", "en", "page"].
3. Les paramètres vides sont supprimés: ["Une", "super", "mise", "en", "page"].
4. La commande est exécutée avec la liste des paramètres identifiés. On obtient donc un affichage dans lequel la mise en page n'est pas maintenue.

```
1 $ my_var="Une  super  mise en page"
2 $ echo $my_var
3   Une super mise en page
4 $ echo "$my_var"
5   Une  super  mise en page
6 $ echo '$my_var'
7   $my_var
```

Dans ce second exemple, la chaîne précédente, cette fois ci quotée, est sauvegardée dans une variable. Lors du premier appel à `echo`, on peut remarquer que la mise en page n'est pas maintenue, malgré la présence de double quotes lors de la déclaration de la variable. En effet ces derniers sont interprétés lors de l'affectation et ne sont pas stockés dans la variable. Il est donc nécessaire de double quoter la variable lors de l'appel à `echo`. À noter que les quotes (doubles ou simples) sont nécessaires pour affecter à une variable une valeur contenant des espaces, sans quoi bash tente d'interpréter la ligne de commande comme une commande exécutée avec une variable d'environnement temporaire (cf. Section 2.3.4). L'exemple suivant utilise cette syntaxe pour illustrer l'intérêt que peuvent avoir les simples quotes.

```
1 $ env_var="Variable temporaire" \
2   bash -c 'ps --forest; echo "Var: $env_var"'
3     PID TTY          TIME CMD
4     24400 pts/6        00:00:00 bash
5     142917 pts/6        00:00:00 \_ bash
```

```
6 142918 pts/6    00:00:00    \_ ps
7  Var: Variable temporaire
```

Notons dans un premier temps, l'échappement du retour à la ligne à l'aide du caractère `\`. Grâce à ce dernier, le retour à la ligne n'est pas interprété comme une injonction à interpréter la ligne.

Procédons par étape pour comprendre les mécanismes en action. Premièrement notons que l'option `-c` de `bash` permet de lancer un processus fils `bash` qui exécutera la ligne de commande passée en paramètre de l'option puis se terminera. De plus, une variable d'environnement temporaire `env_var` a été définie pour ce processus `bash`, cette dernière contenant la valeur `"Variable temporaire"`.

En d'autres termes, le processus `bash` créé exécute la commande contenue dans la chaîne `'ps --forest; echo "Var: $env_var"'`, ce dans un environnement contenant une variable `env_var`. Lors de son évaluation dans le processus père, la chaîne de caractère précédente voit l'ensemble de ses caractères spéciaux échappés grâce à l'utilisation de simple quotes. Il en résulte que le processus fils exécute la ligne de commande: `ps --forest; echo "Var: $env_var"`:

1. la ligne est découpée en deux lignes de commandes au niveau du `;`,
2. la première commande invoque `ps` et génère un affichage montrant les différents shells imbriqués,
3. la seconde commande invoque `echo`. Puisque les simples quotes ont été interprétés par le shell père, ces derniers ne sont plus présents pour échapper les caractères spéciaux qui sont alors interprétés, affichant donc le contenu de la variable `env_var`.

Si une mauvaise gestion des caractères spéciaux dans les exemples précédents n'a que peu de conséquences, l'exemple suivant met en évidence un cas plus problématique. Ce cas pourrait l'être d'autant plus avec l'utilisation de commandes telles que `rm` ou `shred`.

```
1 $ dir="un nouveau repertoire"
2 $ mkdir $dir
3 $ ls -l
4 total 12
5 drwxr-xr-x  nouveau
6 drwxr-xr-x  repertoire
7 drwxr-xr-x  un
8 $ mkdir "$dir"
9 $ ls -l
10 total 16
11 drwxr-xr-x  nouveau
```

```
12  drwxr-xr-x    repertoire
13  drwxr-xr-x    un
14  drwxr-xr-x    'un nouveau repertoire'
```

2.5 Les commandes

2.5.1 Commandes internes et externes

2.5.2 Affichage, retour et statut de sortie

3 Les bases du scripting

3.1 Contrôle du flux d'exécution

3.1.1 Les conditions

3.1.2 Les boucles

3.2 Gestions des arguments

3.3 Gestion des options

4 Notions avancées de bash

4.1 Opérateur de contrôle

4.2 Sous-shells

4.3 Redirection de flux

4.4 Expansion de variables

5 Scripting avancé

5.1 Fonctions

5.2 Gestion des erreurs

Bibliographie

1. Mashey JR. Using a command language as a high-level programming language. Proceedings of the 2nd international conference on software engineering. Citeseer; 1976. pp. 169–176.
2. McIlroy MD. A research UNIX reader: Annotated excerpts from the programmer's manual, 1971-1986. AT; T Bell Laboratories. Computing Science; 1987.
3. Tanenbaum A., Boschung HT. Modern operating systems. 2018.
4. Newham C., Rosenblatt B. Learning the bash Shell. 3rd ed. Beijing ; Sesbastopol, [Calif.]: O'Reilly; 2005.
5. Deutsch LP., Edmund C. The LISP implementation for the PDP-1 computer. Berkeley; 1964.
6. Pelisse R. Les mécanismes d'interprétation du shell. GNU/Linux Pratique. Éditions Diamond; 2017; Hors Série 39: 36–41.
7. Blum R., Bresnahan C. Linux command line and shell scripting bible. 4th edition. Indianapolis: John Wiley; Sons; 2020.

