Unix : Aide-mémoire

Bart Lamiroy
École Nationale Supérieure des Mines de Nancy

Version 2.1, 2004

1 Introduction

Ce document ne constitue pas un support de cours, mais plutôt un complément permettant de rapidement retrouver des informations de base sur l'utilisation des différents Unix.

Il est important de garder à l'esprit l'un des leitmotiv des concepteurs de Unix : « *Small is beautiful* ». La philosophie du système veut que tout soit concis, petit et rapide. Il en résulte un tas de petites commandes ou petits utilitaires qui sont efficaces pour une seule tâche particulière et qui se combinent à l'infini pour résoudre des problèmes complexes et de tailles considérables.

L'objectif de cet aide-mémoire n'est pas de passer en revue toutes les commandes et utilitaires disponibles, mais de références les plus courantes et les plus utiles. Pour des informations plus détaillés référez vous aux manuels en ligne : `man` et/ou `info`[1] (`$ man intro` étant un bon point de départ).

2 Les shells

L'environnement d'interaction avec le système de chaque utilisateur est défini dans le dernier champ de son entrée dans `/etc/passwd` (ou équivalent) et s'appelle *shell*. Ce *shell* permet de lancer des processus, de naviguer dans l'arborescence des fichiers et d'interagir avec le système. Historiquement il existe deux classes de *shell* : ceux issus du *Bourne shell* (typiquement `sh` et `bash`) et ceux issus du `C`-*shell* (typiquement `csh` et `tcsh`). Les deux utilisent une syntaxe légèrement différente.

L'interpréteur du *shell* permet de saisir des commandes au clavier et de les exécuter, mais il dispose également d'un langage de programmation succinct, permettant des structures de contrôle tels **if-then-else**, **for** ou **while** et la manipulation de variables.

[1] La commande `man` est toujours présente et très standardisée dans sa présentation d'information. `info`, en revanche, est moins généralisée. Elle permet une navigation plus souple dans les informations via des hyperliens.

2.1 L'interpréteur

L'interpréteur affiche un « invite de commande » (*prompt*) et attend que l'utilisateur entre une commande. Dans ce document, le prompt sera représenté comme le signe `$`, souligné pour le distinguer de l'opérateur d'accès aux variables.

Une commande prend toujours la forme suivante :

```
$ cmd -options paramètres
```

Les options sont généralement précédés du signe « - » qui leur est accolé. Dans certains cas, la même option est représentée sous deux formes : une courte (généralement une seule lettre), précédée de « - » et une longue (écrite en toutes lettres), précédée dans ce cas de « – ». Dans les descriptions des commandes (p. ex. pages `man`) on les met souvent entre [ ] pour indiquer que leur présence n'est pas obligatoire. On fait de même pour les paramètres que l'on n'est pas toujours obligé de fournir. On n'écrit pas ces [ ] lorsqu'on veut exécuter la commande.

La commande est toujours exécuté par rapport au répertoire courant du shell. On change de répertoire courant avec la commande `cd`. Tout fichier (et donc potentiellement un fichier exécutable) ou répertoire peut être exprimé en fonction du répertoire courant ou par rapport à la racine. Dans ce dernier cas son nom commence par `/` et contient, depuis la racine, tous les répertoires successifs (séparés par `/`) pour atteindre l'endroit où il est stocké. S'il est exprimé de façon relative au répertoire courant, son nom commence par le premier répertoire contenu dans le répertoire courant, puis contient tous les répertoires successifs (séparés par `/`) pour atteindre l'endroit où il est stocké depuis ce répertoire. Chaque répertoire contient au moins deux sous-répertoires : « `..` », le répertoire parent, et « `.` », le répertoire courant.

Pour avoir accès à un fichier, on doit avoir les droits d'accès appropriés à tous les répertoires que l'on doit parcourir.

> **Note :** il est important de réaliser que la notion de commande est ambiguë. Soit il s'agit d'une commande propre au *shell* qui fait partie de son langage de programmation (p. ex. `cd`, `set`, `if`, ...). Ce sont les commandes dites shell, *shell-commands* ou *built-in commands*. Si l'on fournit un nom de commande qui ne fait pas partie du shell, l'interpréteur va parcourir un ensemble de répertoires du système à la recherche d'un programme (exécutable) qui porte le nom fourni et l'exécutera, une fois trouvé. La différence entre les deux est subtile et relativement transparente pour l'utilisateur.

Avant d'exécuter la commande saisie, l'interpréteur du shell effectue une série d'opérations de substitution. Le shell reconnaît les caractères spéciaux suivants, et les interprète comme suit :

; est un séparateur de commandes. Ainsi

```
$ cmd1 arg1 ... argn ; cmd2 arg1 ... argm ; cmd3 ...
```

exécute d'abord `cmd1` avec les arguments spécifiés, ensuite `cmd2`, puis `cmd3`, *etc.*

D'un point de vue de la gestion des processus, on lance un premier processus (`cmd1`). On attend qu'il termine. Puis on lance le deuxième processus (`cmd2`) et ainsi de suite.

`&` est un opérateur de lancement en *tâche de fond*.

```
$ cmd1 arg1 ... argn &
```

crée un processus et le lance, mais n'attend pas qu'il termine avant de rendre la main à l'utilisateur. L'utilisateur reprend immédiatement la main.

De la même façon,

```
$ cmd1 arg1 ... argn & ; cmd2 arg1 ... argm & ; cmd3 ... &
```

exécutera les trois commandes, comme vu au point précédent, mais ils sont exécutés quasiment en parallèle.

Attention : un processus lancé en tâche de fond ne peut pas faire appel à des entrées clavier.

`$` est l'opérateur d'accès à une variable de shell. Le shell permet de définir un ensemble de variables qui peuvent être transmise à tous les processus fils. Par contre, un processus fils ne peut pas transmettre de variables à son processus père. La création et la consultation des variables peut varier en fonction du shell (`sh`, `csh`, `tcsh`, `bash`) utilisé.

`*` est un joker ou *wildcard*. Lorsqu'il fait partie d'une suite de caractères (éventuellement vide), il est replacé par le shell par une autre suite de caractères (éventuellement vide) telle que la suite de caractères résultant de leur concaténation est un fichier existant sur le disque. Et cela autant de fois que possible, de telle façon que le résultat est une liste de fichiers.

`?` est un joker ou *wildcard*. Lorsqu'il fait partie d'une suite de caractères (éventuellement vide), il est replacé par le shell par un et un seul caractère telle que la suite de caractères résultant de leur concaténation est un fichier existant sur le disque. Et cela autant de fois que possible, de telle façon que le résultat est une liste de fichiers.

`[ ]` fait également partie des jokers ou *wildcards*. Il ressemble à `?`, mais permet d'être plus strict sur la substitution et de n'autoriser qu'une intervalle ou une liste de caractères comme candidats à la substitution.

`~` représente la racine du répertoire personnel de l'utilisateur.

`"` est un séparateur de chaînes de caractère. Jusqu'à maintenant nous avons implicitement utilisé le caractère `<espace>` comme séparateur entre la commande, les options et les paramètres. Or il peut être nécessaire de considérer une suite de mots comme paramètre unique et non pas

comme une suite de paramètres. Toute chaîne de caractères délimitée par deux " est donc considérés comme un unique nom (nom de commande, nom de fichier, ou paramètre). À l'intérieur des délimiteurs, les variables d'environnement et les délimiteurs `, sont interprétés par le shell.

' est une variante du délimiteur ". À l'intérieur de deux délimiteurs ' l'interpréteur ne fait aucune substitution.

` est un délimiteur de substitution. En effet on place à l'intérieur une commande dont le résultat (affiché sur la sortie standard) sera utilisé comme paramètre (ou liste de paramètres) à la commande initiale.

! suivi d'un mot recherche dans l'historique des commandes, l'instruction commençant par ce mot dernièrement exécutée, et l'exécute.

2.2 Les variables

Tous les shells permettent de stocker et d'utiliser des variables. Les Bourne shells ne connaissent que des variables de type chaîne de caractères. Les C-shells ont également des variables de type tableau et de type numérique.

On distingue deux classes de variables. Les variables locales, dites variables de shell ou *shell variables* et les variables globales ou variables d'environnement. L'ensemble des variables d'environnement est inclus dans celui des variables de shell. Les variables locales sont propres au shell qui les a définies et ne sont pas transmis à des sous-processus éventuels. Elles sont typiquement utilisées dans des scripts. Les variables d'environnement, elles, sont transmises aux sous-processus éventuels. En aucun cas, un processus fils peut transmettre une variable d'environnement à son parent.

	Bourne shell	**C-shell**
Variables de shell		
Affectation	`variable=valeur`	`set variable=valeur`
Afficher toutes les variables valides	`set`	`set`
Détruire une variable	`unset variable`	`unset variable`
Variables d'environnement		
Affectation	`variable=valeur` puis `export variable`	`setenv variable valeur`
Afficher toutes les variables d'environnement valides	`env`	`env` ou `printenv`
Détruire une variable	`unset variable`	`unsetenv variable`

Variables de type tableau et de type numérique en C-shell

En C-shell on dispose également de tableaux et de variables numériques. On utilise les tableaux comme suit (avec extension évidente vers les variables d'environnement). Les tableaux sont indicés à partir de 1.

```
set tableau=(a b c)
set tableau[2]=d
echo $tableau
echo $tableau[1-2]
echo $tableau[1]
echo $#tableau
```

Les variables numériques, par contre, ne peuvent pas faire partie de l'espace des variables d'environnement.

```
@ a=3
@ b=4
echo "$a $b"
@ a+=$b
@ b*=3
echo "$a $b"
```

2.3 Les entrées/sorties

Le shell connaît trois canaux de communication : l'*entrée standard*, affecté par défaut au clavier, la *sortie standard* et la sortie *erreur standard*, affectés par défaut au terminal d'affichage.

En général, la philosophie Unix veut que, lorsqu'une commande demande à lire dans un fichier dont le nom est passé en paramètre, l'appel de la commande sans ce paramètre fait lire la commande dans l'*entrée standard*. Le résultat est, en général écrit dans la *sortie standard*, et d'éventuels messages de contrôle dans l'*erreur standard*.

Les trois canaux peuvent être redirigés à souhait par l'utilisateur à l'aide des opérateurs `< > << >> |` . Les C-shells admettent également `>&` tandis que les Bourne shells permettent de faire une redirection un peu plus spécifique en utilisant des descripteurs de fichiers précis (*cf.* `man sh`).

`>` redirige la sortie standard dans le fichier dont le nom suit l'opérateur en écrasant son contenu éventuel.

`>>` redirige la sortie standard dans le fichier dont le nom suit l'opérateur en écrivant à la suite du contenu déjà existant.

`<` redirige l'entrée standard à partir du fichier dont le nom suit l'opérateur.

`>&` redirige la sortie standard et l'erreur standard dans le fichier dont le nom suit l'opérateur en écrasant son contenu éventuel. (les autres opérateurs se déclinent de la même façon en y post-fixant le `&`.)

`|` (*pipe*) attache la sortie standard de la commande à gauche de l'opérateur à l'entrée standard de la commande à droite de l'opérateur.

`$<` permet de lire des données sur l'entrée standard et de les affecter à une variable. `bash` permet également de le faire (de façon un peu plus élaborée) mais la commande interne `read`.

Pour éviter que des redirections intempestives détruisent des fichiers sensibles, il existe la variable `noclobber`. Si elle est mise, les opérateurs décrits ici vérifient si le fichier de redirection de sortie existe déjà. Si la variable n'est pas mise, le shell écrase les fichiers existants.

2.4 Les scripts et la programmation

Le shell n'est pas seulement un environnement interactif de commande évolué. En effet, le shell lui même peut être vu comme une commande shell (un shell peut donc lancer des sous-shells) et respecte les règles de redirection des flots d'entrée et de sortie. Au lieu de taper les commandes au clavier, on peut les stocker dans un fichier, puis rediriger l'entrée standard du shell depuis ce fichier. *Idem* pour les sorties. On appelle de tels fichiers *scripts*.

Les *scripts* servent principalement à regrouper des ensembles de commandes qui seraient fastidieuses à refaire systématiquement, et s'apparentent par conséquent à des programmes. De ce fait, les différents shells offrent un ensemble de structures de contrôle comme tout langage de programmation. On se contentera ici de donner la syntaxe des structures les plus classiques. Le lecteur intéressé se référera au `man` pour de plus amples informations ou d'autres structures, raccourcis et options.

2.4.1 Les structures de contrôle

Comme tout langage de programmation le shell permet d'introduire des boucles ou des branchements conditionnels. Pour ce qui suit, il est néanmoins important de souligner une différence fondamentale entre les shells de type Bourne et les `C`-shells. Les Bourne shells ne connaissent que la notion de commande. Les conditionnelles sont alors exécutées suivant l'état de retour d'une commande (*exit status*). Les `C`-shells connaissent la notion d'*expression* dont la valeur sert à l'exécution des conditionnelles. La commande `test` permet d'avoir des semblants d'expressions dans les Bourne shells.

Dans tous les shells, une commande a une valeur de retour (variable réservée `$status`)[2]. La valeur de retour vaut `0` si l'exécution s'est effectuée sans problème, et prend une autre valeur selon l'erreur produite lors de l'exécution. La valeur d'une suite d'instructions est la valeur de retour de la dernière instruction.

[2] C'est la raison pour laquelle, en `C`, la commande `main` retourne systématiquement un `int`, et que la commande système `exit( )` prend un paramètre.

Dans les Bourne shells, une condition est vraie si la liste de commandes qui la constitue retourne `0`. Dans les `C`-shells, à l'instar du langage `C`, c'est une approche inverse qui prévaut : une expression valant `0` est considérée fausse ; toute expression valant autre chose est considérée vraie. Une commande placée entre { } est une expression qui vaut `1` si la valeur de retour de la commande vaut `0` et qui vaut `0` sinon.

<table>
<tr><th></th><th>Bourne shell</th><th>C-shell</th></tr>
<tr><td>if-then-else</td><td>

```
if list ; then
   list
else
   list
fi
```

</td><td>

```
if (expression) then
   list
else
   list
endif
```

</td></tr>
<tr><td>while-do</td><td>

```
while list ; do
   list
done
```

</td><td>

```
while (expression)
   list
end
```

</td></tr>
<tr><td>for</td><td>

```
for nom in liste ; do
   list ;
done
```

</td><td>

```
foreach nom (liste)
   list
end
```

</td></tr>
<tr><td></td><td colspan="2">La variable <code>$nom</code> prend successivement toutes les valeurs énumérées dans la liste <code>liste</code>. Et la suite des commandes <code>list</code> est exécutée pour chaque valeur. La liste <code>liste</code> peut contenir les jokers connus par le shell, et sera interprétée par rapport au répertoire courant.</td></tr>
<tr><th colspan="2">Exemple Bourne shell</th><th>Exemple C-shell</th></tr>
<tr><td colspan="2">

```
echo "Quel est votre login ?"
read nom
if [ "$nom" = `whoami` ] ;
then
   echo "Bonjour $nom"
else
   echo "Faux"
fi
```

</td><td>

```
echo "Quel est votre login ?"
set nom=$<
if ($nom == `whoami`) then
   echo "Bonjour $nom"
else
   echo "Faux !"
endif
```

</td></tr>
</table>

2.4.2 Lancement d'un script

Un fichier script peut s'exécuter de deux façons :

1. Soit en le passant explicitement comme flot d'entrée (avec une redirection) ou comme paramètre à une commande qui lance un shell (`/bin/sh`, p. ex., ou `/bin/tcsh`).
2. Soit en rendant le fichier script exécutable (*cf.* `chmod`) et en l'appelant comme s'il s'agissait d'une commande classique. Dans ce cas trois cas de figures sont possibles :
   (a) Soit la première ligne du script est vide. Dans ce cas, ce sera le Bourne shell (`/bin/sh`) qui interprétera le script.
   (b) Soit la première ligne du script commence par `#!`. Dans ce cas, ce sera la commande qui suit le `#!`, séparé par un blanc, qui interprétera le script. Si aucune commande suit le `#!`, c'est le shell par défaut qui s'en chargera.
   (c) Soit la première ligne ne rentre dans aucune des deux catégories précédentes, au quel cas le shell par défaut interprétera le script.

Attention, les commandes dans les scripts sont interprétés comme les commandes dans le mode interactif, ce qui veut dire que c'est le <retour-chariot> en fin de commande qui cause son exécution. Une erreur courante est d'oublier ce retour-chariot pour la dernière commande du script.

2.5 Configuration et initialisation

Lorsqu'un shell s'exécute, il va systématiquement voir s'il existe un (ou plusieurs) fichier(s) de configuration. Ces fichiers de configuration sont des scripts qui s'exécutent avant de lire le flot d'entrée attaché au shell (le clavier en mode interactif, un autre script en mode d'exécution de scripts). Ils permettent principalement de mettre des variables d'environnement propres à l'installation locale de la machine ou de définir des commandes personnelles pour chaque utilisateur. Voici l'ordre d'exécution des différents fichiers de configuration.

	Bourne shells		C-shells	
	`sh`	`bash`	`csh`	`tcsh`
1			`/etc/csh.cshrc`	
2*	`/etc/profile`		`/etc/csh.login`	
3		`~/.bashrc`	`~/.cshrc`	`~/.tcshrc` ou à défaut `~/.cshrc`
4*	`~/.profile`	`~/.bash_profile` ou à défaut `~/.bash_login` ou à défaut `~/.profile`	`~/.login`	

* *pour des login-shells seulement. (cf.* `man` *des différents shells pour la définition d'un login-shell.)*

Notons que pour les login-shells il existe également un script de déconnexion (`~/.bash_logout` ou `~/.logout`).

3 Quelques commandes

Dans cette partie on énumère quelques unes des commandes les plus courantes. L'énumération ni la description se veulent exhaustives, mais tentent juste de donner les fonctionnalités principales. Les commandes sont regroupées par domaine d'application.

3.1 Le système de fichiers

`cd` Invocation : `cd` *`répertoire`*

change le répertoire courant du shell. L'argument est le nouveau répertoire, soit décrit de façon absolue depuis la racine `/`, soit relatif par rapport au répertoire actuel en utilisant éventuellement les répertoires spéciaux `..` (répertoire parent) et `.` (répertoire courant).

`cd -` retourne au répertoire courant précédent.

`cd` sans argument retourne au répertoire racine de l'utilisateur.

`chmod` Invocation : `chmod` *`droit_accès fichier`*

change les droits d'accès à un fichier ou répertoire. Les droits possibles à un fichier (ou répertoire) sont principalement divisés en trois groupes (`u` – *user* pour le propriétaire, `g` – *group* pour le groupe d'appartenance du fichier, `o` – *others* pour les autres) chacun peut avoir trois types d'accès différents : `r` (*read*, droit de lecture d'un fichier ou droit de voir le contenu d'un répertoire – `ls`), `w` (*write*, droit de modifier le contenu d'un fichier ou droit de créer des fichiers dans un répertoire),

`x` (*execute*, droit d'exécuter un fichier ou droit de manipuler le contenu d'un répertoire – `cd` dans le répertoire, `rm` et `mv` de son contenu).

`cp` Invocation : `cp` ***`source destination`***
crée une copie d'un fichier ou d'une liste de fichiers à un autre endroit. Si ***`source`*** est un nom de fichier, ***`destination`*** peut être soit un répertoire, soit un autre nom de fichier. Si ***`source`*** est une liste de noms de fichiers, ***`destination`*** doit être un répertoire. `cp` permet de copier des arborescences complètes si l'option `-R` est utilisée.

`ln` Invocation : `ln` ***`source destination`***
crée un lien nommé ***`destination`*** vers ***`source`***. Si ***`destination`*** est un répertoire, ou s'il se trouve sur un filesystem différent, seuls les liens de type *soft* sont autorisés. Par défaut la commande crée un lien de type *hard*, qui consiste à partager le même *i-node* par différents répertoires. L'option `-s` permet de créer des liens de type *soft* qui consiste en une redirection vers un fichier ou répertoire, et qui est interprété à chaque accès, et qui peut ne pas aboutir si la ***`source`*** a été déplacée.

`locate` Invocation : `locate` ***`nom`***
localise sur le disque tout fichier comportant dans son chemin d'accès la chaîne de caractères ***`nom`***. Nécessite que l'administrateur système ait activé la base de données qui indexe tous le fichiers.

`ls` Invocation : `ls` ***`liste`***
sans argument, affiche le contenu du répertoire courant. Si l'argument est une liste de fichier et répertoires, affiche la liste des fichiers et le contenu des répertoires. Une grande quantité d'options permet de trier la liste par date d'accès, par taille ou encore de parcourir une hiérarchie de façon récursive.

`mkdir` Invocation : `mkdir` ***`liste`***
crée un nombre de répertoires vides (dans l'ordre spécifié) les noms de répertoires peuvent être en absolu par rapport à la racine ou en relatif par rapport au répertoire courant.

`mv` Invocation : `mv` ***`source destination`***
« déplace » un fichier ou un répertoire (avec tout son contenu) ***`source`*** à un nouvelle ***`destination`***. Il est important de noter qu'à l'issue d'un `mv` le *i-node* du fichier ou répertoire n'a pas changé, ce qui est important dans le cas de liens *hard* qui pointeraient vers lui. (en fait, `mv` est le résultat combiné d'un `ln` et d'un `rm`)

`pwd` Invocation : `pwd`
affiche le répertoire courant.

`rm` Invocation : `rm` ***`liste`***
« efface » la ***`liste`*** des fichiers. Seulement si l'option `-R` (effacement récursif) est utilisée la liste peut également contenir des répertoires. En

réalité, le fichier n'est pas vraiment effacé, mais *unlinked*. Seulement s'il n'existe plus de liens *hard* sur le fichier il est effectivement effacé.

`rmdir` Invocation : `rmdir` *`liste`*
efface la *`liste`* des répertoires. On ne peut effacer un répertoire que s'il est vide.

`which` Invocation : `which` *`commande`*
donne le chemin d'accès vers la commande.

3.2 Les variables d'environnement

Il existe un nombre de variables d'environnement utiles (ou de commandes d'environnement) dont on énumère ici quelques uns :

`autofill` cette variable, lorsqu'elle est mise permet la complétion automatique des commandes ou de leurs arguments avec la touche <Tab>. Cette complétion peut être configurée (`man complete` pour les détails) en fonction des commandes utilisés.

`autolist` cette variable, lorsqu'elle est mise affiche les complétions possibles des commandes ou de leurs arguments avec la touche <Tab>. Cette complétion peut être configurée (`man complete` pour les détails) en fonction des commandes utilisés.

`DISPLAY` est utilisé par les serveurs `X11`. Elle indique sur quel écran l'affichage doit avoir lieu.

La variable est de la forme `hostname :displaynumber.screennumber`. Dans la plupart des cas, la partie `displaynumber.screennumber` prend la forme `0.0`. Cette variable permet d'exécuter une commande `X11` sur une machine, mais d'afficher le résultat sur une autre (si celle-ci autorise l'affichage).

`HOME` est le répertoire de login (*homedir*) de l'utilisateur propriétaire du shell, tel qu'il est défini dans `/etc/passwd`.

`LD_LIBRARY_PATH` contient la liste des répertoires à parcourir (dans l'ordre indiqué) pour chercher des bibliothèques dynamiques (*dynamically linked libraries* ou *DLL*), nécessaires à un certain nombre d'applications (les bibliothèques dynamiques sont de la forme `lib*.so`).

Note : suite à un nombre de failles de sécurité que cette variable induit, elle est de plus en plus remplacée par un autre méchanisme qui permet de stocker le chemin des bibliothèques dynamiques dans `/etc/ld.so.conf`.

`MANPATH` contient la liste des répertoires à parcourir pour chercher des pages `man` lors de l'exécution de la commande `man`.

`noclobber` variable déterminant l'autorisation de détruire des fichiers existants avec les opérateurs de redirection.

`path` **ou** `PATH` contient la liste des répertoires à parcourir (dans l'ordre indiqué) pour chercher une commande qui ne fait pas partie du langage interne du shell. Il est considéré comme un trou de sécurité d'avoir le répertoire « `.` » dans son `path`.

`umask` commande permettant de configurer le shell courant afin d'attribuer par défaut un certain type de droit d'accès à chaque fichier ou répertoire créé (*cf.* `chmod`). La notation utilisée dans `umask` est le complément binaire de celle utilisée par `chmod` : on indique les drapeaux `rwx` à ne pas activer par défaut.

`USER` nom de login de l'utilisateur courant.

3.3 Les processus

`bg` Invocation : `bg` *`[job]`*

relance un processus arrêté (avec Ctrl-Z ou avec un signal `STOP`) en tâche de fond. Le résultat est comme si on avait lancé le processus avec `&`.

`fg` Invocation : `fg` *`[job]`*

relance un processus arrêté (avec Ctrl-Z ou avec un signal `STOP`) en tâche principale. Le résultat est comme si on avait lancé le processus sans `&`.

`kill` Invocation : `kill -`*`signal pid`*

envoie le *`signal`* au processus *`pid`*. Voici la définition de quelques signaux définies dans la norme POSIX.1 :

Signal	Value	Action	Comment
SIGHUP	1	A	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	A	Interrupt from keyboard
SIGQUIT	3	A	Quit from keyboard
SIGILL	4	A	Illegal Instruction
SIGABRT	6	C	Abort signal from `abort()`
SIGFPE	8	C	Floating point exception
SIGKILL	9	AEF	Kill signal
SIGSEGV	11	C	Invalid memory reference
SIGPIPE	13	A	Broken pipe : write to pipe with no readers
SIGALRM	14	A	Timer signal from `alarm()`
SIGTERM	15	A	Termination signal
SIGUSR1	30,10,16	A	User-defined signal 1
SIGUSR2	31,12,17	A	User-defined signal 2
SIGCHLD	20,17,18	B	Child stopped or terminated
SIGCONT	19,18,25		Continue if stopped
SIGSTOP	17,19,23	DEF	Stop process

Les lettres dans la colonne « Action » ont la signification suivante, sachant que pour la majorité des signaux un programme peut intercepter ou ignorer des signaux et modifier le comportement par défaut :

A Default action is to terminate the process.
B Default action is to ignore the signal.
C Default action is to dump core.
D Default action is to stop the process.
E Signal cannot be caught.
F Signal cannot be ignored.

`ps` Invocation : `ps`

affiche les processus présents sur la machine à l'instant t du lancement de la commande. Les options que l'on passe à `ps` varient beaucoup de version à version. Comme pour la commande `ls` elles concernent principalement le format et le détail d'affichage.

`source` Invocation : `source` *`fichier`*

exécute le *`fichier`* en tant que script dans le shell courant. Normalement l'exécution d'un script passe par le lancement d'un sous-shell qui se charge de l'exécution. L'utilisation de la commande `source` permet alors de modifier des variables du shell courant par l'intermédiaire d'un script, ce qui ne serait pas possible sinon puisque le sous-shell ne peut pas passer de variables à son parent.

`top` Invocation : `top`

affiche en continue (avec une fréquence de mise à jour variable) les processus présents sur la machine.

3.4 Manipulation de données

`awk` Invocation : `awk` *`programme fichier`*

est un langage de programmation pour la recherche et le traitement de motifs comme défini dans le POSIX 1003.2 *Command Language And Utilities Standard*. Il permet d'écrire des programmes de recherche de motifs à l'aide d'expressions régulières et de faire un traitement sommaire sur les champs trouvés. Le langage étant trop complexe pour cet aide-mémoire, le lecteur devra se référer aux `man`pages, ou mieux encore, aux ouvrages disponibles dans le commerce.

`cat` Invocation : `cat` *`fichier`*

écrit le contenu du *`fichier`* dans la sortie standard. Sans paramètre, écrit le contenu de l'entrée standard dans la sortie standard.

`cut` Invocation : `cut` *`fichiers`*

écrit dans la sortie standard une copie du contenu des fichiers dont certaines parties de chaque ligne ont été enlevées. Sans paramètre, lit l'entrée standard. Cette fonction est particulièrement utile pour des fichiers qui sont organisés en colonnes et dont on ne veut sélectionner qu'un certain nombre de colonnes. Par exemple :

```
cut -f 1,4-6 -d : /etc/passwd
```

affiche seulement les colonnes 1 et 4 à 6 du fichier `/etc/passwd` dont les séparateurs de colonne sont des `:`.

`find` Invocation : `find` *`chemin expression`*

parcourt récursivement le *`chemin`* et affiche les fichiers qui vérifient l'*`expression`*. Par exemple :

```
find . -type l -name "*.tex" -print
```

affiche le nom de tous les fichiers de type lien symbolique et dont le nom termine par `.tex` qui se trouvent dans l'arborescence de répertoires du répertoire courant.

Ou encore,

```
find . -type f -name "*.tex" -exec head -1 {} \;
```

affiche la première ligne de tous les vrais fichiers dont le nom termine par `.tex` qui se trouvent dans l'arborescence de répertoires du répertoire courant.

`grep` Invocation : `grep` *`chaîne fichiers`*

écrit dans la sortie standard les lignes des *`fichiers`* passés en paramètre qui contiennent la chaîne de caractères *`chaîne`*. Sans paramètre

fichiers, écrit les lignes de l'entrée standard qui contiennent *chaîne*. Il existe une version de `grep` qui fait des recherches basées sur des expressions régulières, plutôt que sur des occurrences de chaînes de caractères. Elle s'appelle `egrep`.

`head` Invocation : `head` *fichiers*

écrit dans la sortie standard les premières lignes des *fichiers* passés en paramètre. Sans paramètre, lit l'entrée standard.

`more` Invocation : `more` *fichiers*

écrit le contenu du *fichier* dans la sortie standard en arrêtant l'affichage à chaque page pour faciliter la lecture. Sans paramètre, écrit le contenu de l'entrée standard dans la sortie standard. Si la sortie standard est redirigée, fonctionne comme `cat`.

`paste` Invocation : `paste` *fichiers*

concatène des *fichiers* ligne par ligne. Cet outil est pratique dans le cas où l'on veut créer un fichier à plusieurs colonnes et dont les résultats proviennent de différents traitements.

`sed` Invocation : `sed` *scripts fichier*

est un éditeur de flots de données (*stream **ed**itor*) permettant de rechercher, à l'aide d'expressions régulières, des motifs dans un flot d'entrée (ou le *fichier* passé en paramètre) et de les réécrire, transformés, dans un flot de sortie.

Par exemple :

```
sed -e s/toto/titi/
```

transformera, pour chaque ligne de l'entrée standard, la première occurrence de la chaîne « toto » en « titi » et écrira le résultat dans la sortie standard.

```
sed -e s/toto/titi/g
```

fait la même chose mais pour toutes les occurrences, et non seulement la première.

```
sed -e 's/t[o,a,e]t./titi/g'
```

fera la même chose mais transformera en titi tous les mots de quatre lettres dont la première et la troisième lettre sont « t », dont la seconde lettre est soit « o », soit « a » ou soit « e ».

```
sed -e s/'t[o,a,e]t\(.\)'/'tit\1'/g
```

transformera tous les mots de quatre lettres dont la première et la troisième lettre sont « t », dont la seconde lettre est soit « o », soit « a » ou soit « e » en un mot commençant par « tit » et ayant la dernière lettre non modifiée.

`tail` Invocation : `tail` *`fichiers`*

écrit dans la sortie standard les dernières lignes des *`fichiers`* passés en paramètre. Outre les options permettant de spécifier le nombre de lignes à afficher, l'option `-f` permet d'afficher les dernières lignes au fur et à mesure que le fichier se remplit. Sans paramètre, lit l'entrée standard.

`tar` Invocation : `tar` *`[options] liste de fichiers`*

crée une *archive* contenant tous les fichiers spécifiés en paramètre (ou toute l'arborescence si on spécifie un répertoire). La commande permet aussi d'extraire des fichiers d'une archive.Par exemple :

`tar cvf archive.tar` *`f1 f2 f3`*

va créer un fichier (l'archive) `archive.tar` dans le répertoire courant, et qui contiendra les fichiers *`f1`*, *`f2`* et *`f3`*.

`tar xvf archive.tar` *`f1 f2`*

Permet d'extraire *`f1`* et *`f2`* de l'archive crée.

`xargs` Invocation : `xargs` *`commande`*

Lit sur l'entrée standard une liste d'arguments, et lance, pour chaque argument la commande spécifiée avec celui-ci.

4 Les expressions régulières

Bien que ne faisant pas partie du système Unix, à proprement parler, les expressions régulières y tiennent une place importante (*cf.* les commandes `awk`, `egrep`, `sed` ou encore le langage `perl`).

Une expression régulière (en simplifiant honteusement les choses) est une description d'un ensemble d'expressions, vérifiant un nombre de critères. Dans le shell, par exemple, `*.java` est une expression régulière désignant tous les fichiers dans le répertoire courant dont le nom termine par `.java`.

Les expressions régulières sont l'outil de base pour les tâches de recherche de motifs (*pattern matching*) et leur transformation. Ils se basent sur un certain nombre d'opérateurs dont on énumère ici les plus courants. Il faut noter que ces opérateurs sont en conflit avec les opérateurs de substitution du shell. Il est donc nécessaire (si on les utilise sur la ligne de commande) de veiller à utiliser les opérateurs shell de non-interprétation (`" ' \`) afin que le shell n'interprète pas ces opérateurs avant.

`.` désigne un caractère quelconque

`?` désigne la sous-expression précédente (si elle existe) ou alors le caractère précédent, répété un nombre quelconque de fois, mais au moins une fois.

`*` désigne la sous-expression précédente (si elle existe) ou alors le caractère précédent, répété un nombre quelconque de fois, éventuellement zéro fois.

`[ ]` désigne un caractère parmi ceux entre les `[ ]`. On peut également mettre des intervalles entre les `[ ]`. p. ex.`[a-z]` désigne une minuscule quelconque.

`( )` délimite une sous-expression.

`\` évite l'interprétation du caractère suivant. Ainsi `\*` désigne le caractère `*` plutôt que l'opérateur de répétition.

`\t` désigne le caractère de tabulation.

`\n` désigne le caractère de retour à la ligne.

`^` désigne le début d'une ligne.

`$` désigne la fin d'une ligne.

`\b` désigne la limite d'un mot (début ou fin ; *word boundary*). Ainsi, l'expression régulière « `\bX?\b` », désigne les mots formés d'un nombre quelconque de `X`, mais ne désigne pas les parties de mots contenant une suite de `X` parmi d'autres lettres.

`\1, \2, ... \9` désignent, le cas échéant, les résultats d'identification des sous-expressions précédent le lieu où le $\backslash n$ est utilisé. Par exemple, l'expression « `(.)t\1` » désigne les palindrômes à trois lettres ayant comme lettre du milieu « `t` ». Les sous-expressions sont numérotées de 1 à 9 de gauche à droite.

Le langage de référence utilisant les expressions régulières est `perl`. C'est un langage de script un plus évolué que les shells qui est beaucoup utilisé pour des utilitaires nécessitant la recherche et la réécriture de données dans des fichiers texte.

Ce document a été réalisé sous XEmacs avec LaTeX.
Version du 12 mars 2004.