

Sommaire

GNU/Linux	3
Présentation	3
Notion de système d'exploitation	3
Philosophie UNIX	4
« L'univers a 40 ans »	4
Des programmes qui effectuent une seule chose et qui le font bien	4
Le silence est d'or	4
Des programmes qui collaborent	4
Des programmes pour gérer des flux de texte	4
Citations	4
Conclusion	5
Manipuler sous Linux	5
L'interface homme-machine (IHM)	5
Conventions	5
Conseils	5
Structure d'une commande	6
Différents types de commande	6
Obtenir de l'aide	7
Environnement de travail	7
Shell Bash	8
Historique des commandes	9
Groupement de commandes	10
Une liste de commandes de base	11
Manipuler des fichiers	11
Système de fichiers	11
Chemin d'accès	12
Structure de l'arborescence Unix/Linux	13
Les Fichiers	13
Les Fichiers « texte »	15

L'encodage des caractères	17
Créer un répertoire (dossier) et se déplacer dans l'arborescence	18
Créer un fichier texte	19
Afficher le contenu d'un fichier texte	20
Examiner le contenu d'un fichier texte	21
Modifier le contenu d'un fichier texte	21
Éditer un fichier texte (vim)	22
Manipuler des fichiers et des répertoires	23
Contrôler l'accès à vos fichiers	24
Caractères spéciaux	27
Automatiser des tâches	30
Objectifs	30
Les shell scripts	31
Les variables	32
Les substitutions de variables	34
Les substitutions de commandes	34
L'évaluation arithmétique	35
Les variables internes du shell	35
La gestion des options	36
Les commentaires	36
L'affichage sur la sortie standard	37
La saisie de données	37
Les tests et conditions	37
Les structures conditionnelles	38
La structure if-then-else	38
Les choix multiples case et select	39
Les contrôles itératifs (les boucles for, while et until)	40
La boucle for	40
La boucle while	41
La boucle until	42
Les fonctions	43
Annexe 1 : Une liste de commandes de base	44
Annexe 2 : L'arborescence Unix/Linux	46

GNU/Linux

Présentation

Linux est le nom couramment donné à tout **système d'exploitation** (*operating system*) libre fonctionnant avec le noyau Linux. C'est une implémentation libre du système **UNIX** respectant les spécifications **POSIX** (normes techniques de l'IEEE).

Remarque : un système d'exploitation est une couche logicielle (software) qui permet et coordonne l'utilisation du matériel (hardware) entre les différents programmes d'application.

GNU/Linux est le nom parfois donné à un système d'exploitation associant des éléments essentiels (*shell*, compilateurs, bibliothèques C, commandes, etc ...) du projet **GNU** (*GNU's Not UNIX*) et d'un noyau (*kernel*) Linux. C'est une terminologie créée par le projet Debian et reprise notamment par **Richard Stallman**, à l'origine du projet de travail collaboratif GNU et de la licence libre **GPL** (*General Public Licence*).

Par exemple : Android est un système basé sur Linux mais pas sur GNU.

Le noyau Linux a été initialement écrit par **Linus Torvalds**, un étudiant finlandais au début des années 90. Depuis, des centaines de développeurs et des entreprises de toutes tailles participent au projet, dont Linus Torvalds est toujours le coordinateur.

Le système avec les applications est le plus souvent distribué sous la forme de **distributions Linux** comme Slackware, Debian, Red Hat, Mandriva ou **Ubuntu** ...

La différence essentielle de Linux par rapport à d'autres systèmes d'exploitation concurrents (comme Mac OS, Microsoft Windows et Solaris) est d'être un système d'exploitation libre, apportant quatre libertés aux utilisateurs, définies par la licence GNU GPL, les rendant indépendants de tout éditeur et encourageant l'entraide et le partage :

- « utiliser le logiciel sans restriction »
- « étudier le logiciel »
- « modifier pour l'adapter à ses besoins »
- « redistribuer sous certaines conditions précises »

Remarque : Un logiciel libre n'est pas nécessairement gratuit, et inversement un logiciel gratuit n'est pas forcément libre.

Notion de système d'exploitation

De manière générale, un système d'exploitation :

- permet l'exploitation des périphériques matériels dont il coordonne et optimise l'utilisation ;
- propose aux logiciels applicatifs des interfaces de programmation standardisées qui simplifient l'utilisation des matériels et des services qu'il offre ;
- coordonne l'utilisation du ou des processeur(s), et accorde un certain temps pour l'exécution de chaque processus (multi-tâche) ;
- gère l'espace mémoire pour les besoins des programmes ;
- organise le contenu des disques durs ou d'autres mémoires de masse en fichiers et répertoires ;
- fournit les interfaces homme-machine des différents programmes ;
- réalise enfin différentes fonctions visant à assurer la fiabilité (tolérance aux pannes, isolation des fautes) et la sécurité informatique (traçabilité, confidentialité, intégrité et disponibilité).

Philosophie UNIX

« L'univers a 40 ans »

UNIX a marqué à jamais l'histoire de l'informatique et continue à le faire, ceci pour une raison très simple : derrière cette famille de systèmes, il y a une idée ou plutôt un ensemble d'idées et de préceptes. Derrière UNIX, il y a une philosophie qui sert de ligne de conduite et de fil d'Ariane. Comprendre cette philosophie et la respecter le mieux possible assure une stabilité et une pérennité sans précédent.

Résumer la philosophie d'UNIX n'est pas chose évidente. Il s'agit d'un ensemble de principes. Nombreux sont ceux qui ont essayé de les résumer ou les lister (taper « philosophie UNIX » ou « *less is more* » dans un moteur de recherche).

Des programmes qui effectuent une seule chose et qui le font bien

Voilà la base de toutes choses dans le monde UNIX (« dans le monde » tout court peut-être également).

Le silence est d'or

En d'autres termes, lorsqu'un programme n'a rien à dire, il doit garder le silence. Ce n'est que lorsqu'il y a un problème qu'un outil doit devenir bavard et signaler explicitement une erreur. Un programme qui fait ce qu'on lui demande n'affiche rien, ne signale rien. C'est le cas de la plupart des outils de base en ligne de commande.

Des programmes qui collaborent

Si tous les programmes ne font, chacun, qu'une chose et qu'ils la font bien, ceci implique qu'ils doivent alors fonctionner de concert pour pouvoir achever des tâches plus importantes. Ces « briques » doivent alors collaborer les unes avec les autres du mieux possible. Le but est de former un système complet où la somme des parties est supérieure à l'ensemble. Lorsqu'on dispose d'un ensemble de briques fiables, il est possible de construire un mur solide.

Des programmes pour gérer des flux de texte

Les flux de texte représentent une interface universelle (la seule?). La notion de flux de texte est véritablement caractéristique des UNIX.

Citations

« Unix est convivial. Cependant Unix ne précise pas vraiment avec qui. » Steven King

« Unix ne dit jamais 's'il vous plaît'. » Rob Pike

« Unix est simple. Il faut juste être un génie pour comprendre sa simplicité. » Denis Ritchie

« Unix n'a pas été conçu pour empêcher ses utilisateurs de commettre des actes stupides, car cela les empêcherait aussi des actes ingénieux. » Doug Gwyn

Conclusion

Si je devais répondre à la question « Qu'est-ce qu'un UNIX ? », je répondrais par ce type de commande (pleine de magie et d'intelligence) :

```
$ history | grep -v " h" | sed 's/[ \t]*$//' | sort -k 2 -r | uniq -f 1 | sort -n
```

[Extrait d'un article de Denis Bodor dans GNU/Linux Magazine HS n°46]

Manipuler sous Linux

L'interface homme-machine (IHM)

L'interface homme-machine (**IHM**) permet à un utilisateur de dialoguer avec la machine.

On distingue deux types d'IHM :

- **GUI** (*Graphical User Interface*) ou « interface utilisateur graphique » : les parties les plus typiques de ce type d'environnement sont le pointeur de souris, les fenêtres, le bureau, les icônes, les boutons, les menus, les barres de défilement, ... Les systèmes d'exploitation grand public (Windows, MacOS, GNU/Linux, etc.) sont pourvus d'une interface graphique qui, dans un soucis d'ergonomie, se veut conviviale, simple d'utilisation et accessible au plus grand nombre pour l'usage d'un ordinateur personnel.
- **CLI** (*Command Line Interface*) ou « interface en ligne de commande » est encore utilisée en raison de sa puissance, de sa grande rapidité, son uniformité, sa stabilité et du peu de ressources nécessaires à son fonctionnement. Le système d'exploitation permet cette possibilité par l'intermédiaire d'un interpréteur de commandes (le *shell*). Beaucoup de serveurs ne s'administrent qu'en ligne de commande.

Conventions

Tous les exemples d'exécution des commandes sont précédés d'une **invite** utilisateur ou **prompt** spécifique au niveau des droits utilisateurs nécessaires sur le système :




- toute commande précédée de l'invite **\$** ne nécessite aucun privilège particulier et peut être utilisée au niveau utilisateur simple ;
- toute commande précédée de l'invite **#** nécessite les privilèges du super-utilisateur (*root*).

Évidemment, il ne faudra jamais taper l'invite (**\$** ou **#**) lorsque vous testerez par vous même les commandes indiquées.













Conseils

Travailler toujours en mode « plein écran ».

N'utilisez pas la souris (ou très peu). Il existe beaucoup de raccourcis clavier et de touches « magiques » :

- la touche **tabulation**  (la plus utile) permet la complétion en ligne de commande. Le *shell* effectue la complétion en considérant successivement le texte comme une variable (s'il commence par **\$**), un nom d'utilisateur (s'il commence par **~**), un nom d'hôte (s'il commence par **@**), ou une commande (y compris les alias et les fonctions). Si rien ne fonctionne, il essaye la complétion en nom de fichier.
- Les touches flèches  et  servent à parcourir l'historique des commandes déjà saisies.

Utiliser plusieurs sessions *shell* (ou onglets ou fenêtres) en parallèle. Par exemple, vous en utiliserez une pour saisir vos commandes et l'autre pour consulter les indispensables pages de manuel. Pour basculer de l'une à l'autre :

- en mode console :  +  (où x est un chiffre identifiant le terminal)
- en mode graphique, avec 2 onglets :  +  ou  +  ,  +  ou  + 
- en mode graphique, avec 2 fenêtres :  + 

Structure d'une commande

Une **commande** Unix est un ensemble de mots séparés par des **espaces**. Les caractères espace et tabulation sont interprétés comme des séparateurs par le *shell* (voir la variable *IFS*). La syntaxe d'une commande est la suivante :

```
$ commande [options] <parametres>
```

Le premier mot est le nom de la commande. Les autres mots sont des paramètres (ou arguments) de la commande. Certains mots sont des options qui changent le comportement de la commande. Les 2 crochets « [» et «] » indiquent que les options ne sont pas obligatoires. Il ne faut pas taper ces crochets sur la ligne de commande.

Avant, une option était introduite par le signe « - » suivi d'une seule lettre. Le standard actuel GNU pour les options est d'utiliser « -- » suivi du nom de l'option pour des raisons de clarté et de portabilité. L'ordre des options n'a pas souvent d'importance :

```
$ ls --all -l --si      ou      $ ls -l --si --all
$ ls -l $HOME/tmp
```

Différents types de commande

Il existe plusieurs type de commandes :

- les **commandes internes** (au *shell*) : comme *history*, *test*, ...
- les **commandes externes** (donc des programmes) : comme *ls*, *mkdir*, ...
- les **alias** (voir plus loin) : comme *ll*, ...

Les commandes externes (donc des exécutables) sont généralement stockées dans un répertoire de nom *bin*. Il existe des exécutables dans :

- le répertoire */sbin* : les commandes pour *root* (l'administrateur)
- le répertoire */bin* : des commandes et des *shells*
- le répertoire */usr/bin* : le répertoire de base des programmes

Remarque : comme le système ne connaît pas les endroits où vous placez vos programmes, il faudra lui indiquer dans la variable d'environnement \$PATH.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
$ type echo
echo est une primitive du shell
```

```
$ type strings
strings est /usr/bin/strings
```

```
$ type ll
ll est un alias vers « ls -half »
```







Il existe plusieurs modes d'exécution :

```
cmd    // exécute la commande cmd
cmd &  // exécute la commande cmd en tâche de fond (elle se détache alors du terminal)
! cmd  // inverse le code retour de la commande cmd (il y a un espace entre ! et cmd)
(cmd)  // exécute la commande cmd dans un sous-shell
```

Obtenir de l'aide

Pour obtenir la **page de manuel** sur une commande, il faut taper par exemple :

```
$ man cat
```

On utilise les flèches pour se déplacer, la barre « espace »  pour avancer d'une page et la touche  (*back*) pour reculer. La touche  (*quit*) permet de quitter. Vous pouvez faire une recherche en tapant `/motif` puis, vous pouvez vous déplacer sur les occurrences de motif en utilisant les touches  (*next*, en avant) et  (en arrière). La touche « Echap »  permet d'annuler la recherche.

La commande **man** donne accès aux pages de manuel qui sont réparties selon des sections comme suit :

- section 1 : commandes normales
- section 2 : appels systèmes
- section 3 : fonctions de programmation C
- section 4 : périphériques et pilotes de périphériques
- section 5 : format de fichiers
- section 6 : jeux
- section 7 : divers
- section 8 : administration du système

Par exemple, vous obtiendrez deux pages de manuel différentes :

```
$ man 1 mkdir
$ man 2 mkdir
```

Pour rechercher les pages faisant référence à un mot-clé ("mot-clé" peut être un mot simple ou le nom d'une commande), on utilise la commande :

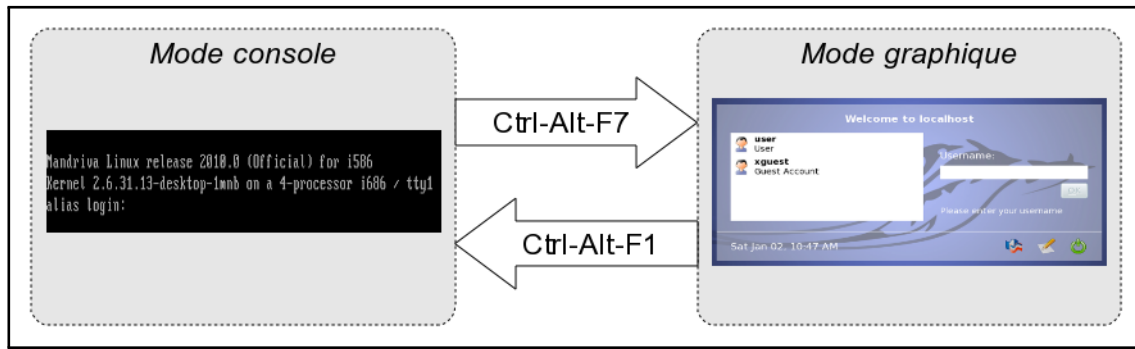
```
$ apropos commande/mot-clé
```

Pour obtenir l'aide sur une commande, il faut taper par exemple :

```
$ cat --help
$ help echo
```

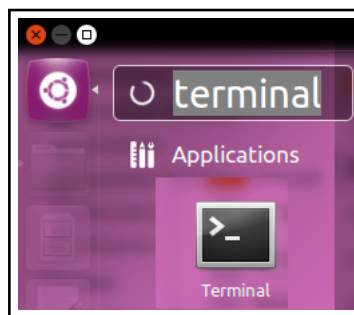
Environnement de travail

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



Ouvrir une session sur un système Mandriva

Remarque : Une « session » est l'ensemble des actions effectuées par l'utilisateur d'un système informatique, entre le moment où il se connecte à celui-ci et le moment où il s'en déconnecte.



Ouvrir une console sur un système Ubuntu à partir de l'interface graphique



On peut maintenant travailler « en ligne de commande » à partir de l'interface graphique

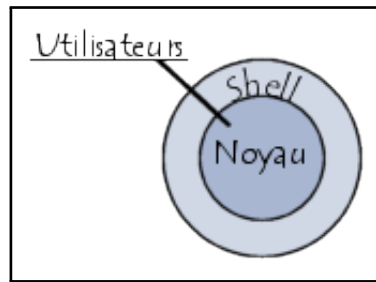
Shell Bash

Bash (*Bourne-again shell*) est le *shell* du projet GNU. Bash est un logiciel libre publié sous GNU GPL. Il est l'interprète par défaut sur de nombreux Unix libres, notamment sur les systèmes GNU/Linux. C'est aussi le *shell* par défaut de Mac OS X et il a été porté sous Windows par le projet Cygwin.

Aujourd'hui **bash** est le *shell* le plus répandu, bien qu'il existe beaucoup d'autres interpréteurs de commandes, comme **sh**, **ksh**, **csh**, **tcsh**, **zsh**, **ash**, ...

Un *shell* Unix, aussi nommé **interface en ligne de commande** Unix, est un *shell* destiné au système d'exploitation Unix et de type Unix. L'utilisateur lance des commandes sous forme d'une entrée texte exécutée ensuite par le *shell*. Celui-ci est utilisable en conjonction avec un terminal (souvent virtuel).

Dans les différents systèmes d'exploitation Microsoft Windows, le programme analogue est `command.com` ou `cmd.exe`.



Le shell (coquille) est une interface permettant d'accéder au noyau (kernel) d'un système d'exploitation

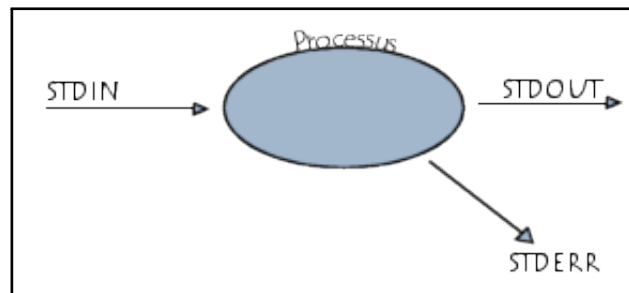
Tout processus Unix/Linux démarre avec 3 **flux** déjà ouverts :

- un pour l'**entrée des données** (canal 0)
- un pour la **sortie des données** (canal 1)
- un pour les **messages d'erreur** (canal 2)

Remarque : un processus (identifié par un PID) est un programme en cours d'exécution.

Par défaut, ces flux sont :

- 0 : le **clavier** (*stdin* : *standard input*)
- 1 : l'**écran** (*stdout* : *standard output*)
- 2 : `/dev/null` (*stderr* : *standard error*)



Il est possible de rediriger ces flux vers des **fichiers** (en utilisant les opérateurs `<`, `>`, `<<` et `>>`) ou vers des **processus** en utilisant un tube (*pipe*). Un **tube** (`|`) est un canal entre deux processus (redirection de la sortie d'un processus vers l'entrée d'un autre processus).

Historique des commandes

Le *shell* permet de rappeler les commandes précédemment exécutées. Pour cela, vous pouvez utiliser les touches flèches  et .

// Visualiser l'ensemble de l'historique :

```
$ history
```

// ou

```
$ history | more
```

// Rechercher une commande : (voir aussi Ctrl + r)

```
$ history | grep commandeRecherchée
```

```
// Rappeler une commande et l'exécuter :
$ !ls      : rappelle la dernière commande commençant par ls
$ !100     : rappelle la commande n°100
$ !!      : rappelle la dernière commande
$ !10:p    : rappelle la commande n°10 et l'affiche (aucune exécution)

// Formes syntaxiques :
// !$ : correspond au dernier argument de la dernière commande
// !* : représente tous les arguments de la dernière commande sauf le premier

// Effacer l'historique
$ history -c
```

Vous pouvez également rechercher une commande précédemment tapée via le raccourci **Ctrl** + **r**. Tapez les premières lettres de la commande recherchée, et la recherche se met à jour au fur et à mesure.

Vous pouvez alors appuyer à nouveau sur **Ctrl** + **r** afin de sélectionner un résultat plus ancien. Enfin, tapez **Enter** pour valider, ou **Ctrl** + **g** pour annuler.

L'aide de la commande interne `history` se trouve dans :

```
$ help history

$ man bash
// Pour rechercher dans l'aide faire : /history
// puis on se déplace avec n (en avant) ou N (en arrière)

// Ou :
$ man bash | colcrt | egrep -A 5 history

// Les options -A (After) -B (Before) -C (autour) -n (numéro de ligne) de la commande egrep
```

Grouper de commandes

Il est possible de grouper plusieurs commandes :

```
cmd1 ; cmd2 // exécution séquentielle de cmd1 puis cmd2
cmd1 | cmd2 // tube (pipe) entre cmd1 et cmd2
cmd1 && cmd2 // si cmd1 retourne VRAI alors cmd2 sera exécuté
cmd1 || cmd2 // si cmd1 retourne FAUX alors cmd2 sera exécuté

// Le groupement || est notamment adapté à l'envoi conditionné de messages d'erreurs :
$ rm fff || echo "Houston, on a un problème !"
$ ls || echo "Houston, on a un problème !"

// Le groupement && est notamment adapté à l'exécution d'un programme (cmd2) conditionné par
// la bonne exécution d'un autre programme (cmd1) :
$ ls *.txt && rm -f *.txt
$ ls *.log && rm -f *.log

// la commande test de réaliser de nombreux tests et de retourner le résultat du test sous
// forme d'un code retour ($?) :
$ touch test.log # crée un fichier vide
```

```
$ test -s test.log || echo "le fichier est vide"
le fichier est vide

$ test -e test.log && echo "le fichier existe"
le fichier existe

$ test -x test.log && echo "le fichier est executable"

$ help test
```

Tous les processus se terminant renvoie un **code de retour** au *shell*. Ce code de retour est accessible par la variable **\$?** et traduit (le plus souvent) l'état de l'exécution du programme. On utilise un programme pour remplir une tâche (processus) et celui-ci nous donne un rapport booléen par le code retour : VRAI (la tâche a été accomplie avec succès) et FAUX (la tâche a rencontré une erreur). Au minimum sous Unix/Linux, le code de retour sera 0 (ok) ou 1 (erreur), mais dans le cas d'une autre valeur numérique, il pourra aussi traduire un type d'erreur :

```
$ ls ; echo $?
0

$ rm zzz* ; echo $?
1

$ ls zzz ; echo $?
2
```

Une liste de commandes de base

Voir l'Annexe n°1 page 44.

Manipuler des fichiers

Système de fichiers

Un système de fichiers (*filesystem*) est une structure de données permettant de stocker les informations et de les organiser dans des fichiers sur ce que l'on appelle des mémoires secondaires ou de stockage (disque dur, disquette, CD-ROM, clé USB, etc.). Il faut faire une opération de **formatage** pour créer et initialiser un système de fichiers sur une partition. Une partition ne peut contenir qu'un seul système de fichiers.

*Remarque : Il faut préalablement partitionner son disque (avec **fdisk** par exemple) avant de pouvoir installer un système de fichiers.*

Une telle gestion des fichiers permet de traiter, de conserver des quantités importantes de données ainsi que de les partager entre plusieurs programmes informatiques. Il offre à l'utilisateur une vue abstraite sur ses données et permet de les localiser à partir d'un **chemin d'accès**.

Remarque : Il existe d'autres façons d'organiser les données, par exemple les bases de données.

Pour l'utilisateur, un système de fichiers est vu comme une **arborescence** : les fichiers sont regroupés dans des répertoires (concept utilisé par la plupart des systèmes d'exploitation). Ces répertoires contiennent soit des fichiers, soit d'autres répertoires. Il y a donc un répertoire racine et des sous-répertoires. Une telle organisation génère une hiérarchie de répertoires et de fichiers organisés en **arbre**.

Il existe de très nombreux systèmes de fichiers différents : FAT, NTFS, HFS, ext2, ext3, UFS, reiserfs, ISO 9660, etc.

Chemin d'accès

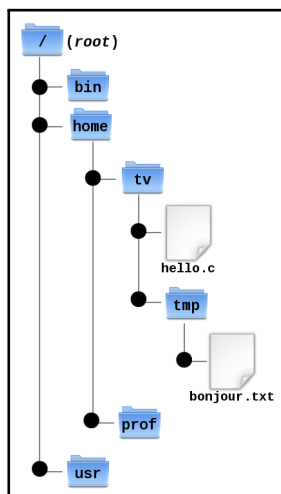
Le chemin d'accès d'un fichier ou d'un répertoire est une chaîne de caractères décrivant la position de ce fichier ou répertoire dans le système de fichiers. Chemins d'accès selon le système d'exploitation :

OS	Répertoire racine	Séparateur de répertoire
Système de type Unix/Linux	/	/
DOS et ses dérivés (OS/2 et Microsoft Windows)	< lettre du lecteur >: \	\
Classic Mac OS	< nom du disque >:	:

Remarque : les systèmes Unix/Linux disposent d'une arborescence unique.

On distingue deux types de chemins d'accès :

- le **chemin absolu** dont la référence est la **racine**. Sous UNIX/Linux, un chemin absolu commence toujours par /.
- le **chemin relatif** dont la référence est le **répertoire courant** (.), le **répertoire parent** (..) ou le répertoire personnel (~).



Quel est le chemin d'accès à "hello.c" ?

→ Avec un chemin d'accès absolu : `/home/tv/hello.c`

→ Avec un chemin d'accès relatif : tout dépend de l'endroit où on exécute la commande, c'est à dire le répertoire de travail (ou répertoire courant). Pour cela, on peut utiliser deux références connus du système d'exploitation : le répertoire courant (noté `.`) ou le répertoire parent (noté `..`) :

- Supposons que le répertoire courant est `prof`, on pourra désigner `hello.c` par `../tv/hello.c`
- Supposons que le répertoire courant est `tv`, on pourra désigner `hello.c` par `./hello.c`

Quel est le chemin d'accès à "bonjour.txt" ?

→ Avec un chemin d'accès absolu : `/home/tv/tmp/bonjour.txt`

→ Avec un chemin d'accès relatif : tout dépend de l'endroit où on exécute la commande, c'est à dire le répertoire de travail (ou répertoire courant). Pour cela, on peut utiliser deux références connus du système d'exploitation : le répertoire courant (noté `.`) ou le répertoire parent (noté `..`) :

- Supposons que le répertoire courant est `prof`, on pourra désigner `bonjour.txt` par `../tv/tmp/bonjour.txt`
- Supposons que le répertoire courant est `tv`, on pourra désigner `bonjour.txt` par `./tmp/bonjour.txt`

Structure de l'arborescence Unix/Linux

Voir l'Annexe n°2 page 46.

Les Fichiers

Un fichier est une suite d'octets portant un nom et conservé dans une mémoire.

Le contenu du fichier peut représenter n'importe quelle donnée binaire : un programme, une image, un texte, etc.

Les fichiers sont classés dans des groupes appelés répertoires, chaque répertoire peut contenir d'autres répertoires, formant ainsi une organisation arborescente appelée **système de fichiers**.

Les fichiers sont la plupart du temps conservés (stockés) sur des mémoires de masse tels que les disques durs mais il existe aussi des systèmes de fichiers en RAM (**ramfs** par exemple).

Dans un système d'exploitation multiutilisateurs, les programmes qui manipulent le système de fichier effectuent des contrôles d'accès (notion de droits).

Quelques caractéristiques de base des fichiers :

- Le nommage et ses restrictions (nombre de caractères, caractères autorisés)
- Le chemin d'accès est une "formule" qui sert à indiquer l'emplacement où se trouve un fichier dans l'arborescence du système de fichier. La syntaxe diffère d'un système d'exploitation à l'autre.
- La taille du fichier indique la quantité d'informations conservée (exprimée en octets) en sachant que la taille physique (réellement occupée) est légèrement supérieure à la taille du fichier en raison de l'utilisation de blocs d'allocation de taille fixe.
- L'extension est un suffixe (précédé d'un point `.`) ajouté au nom du fichier pour indiquer la nature de son contenu. L'usage des extensions est une pratique généralisée sur les systèmes d'exploitation Windows et une pratique courante sur les systèmes d'exploitation Unix.
- Les données descriptives : la date de création et de modification, le propriétaire du fichier ainsi que les droits d'accès ...

Chaque fichier est vu par le système de fichiers de plusieurs façons :

- un descripteur de fichier (souvent un entier unique) permettant de l'identifier ;
- une entrée dans un répertoire permettant de le situer et de le nommer ;
- des métadonnées sur le fichier permettant de le définir et de le décrire ;
- un ou plusieurs blocs (selon sa taille) permettant d'accéder aux données du fichier (son contenu).

Métadonnées : des données servant à définir ou décrire d'autres données

Le terme **inode** désigne le descripteur d'un fichier sous UNIX/Linux. Les inodes (contraction de « *index* » et « *node* », en français : nœud d'index) sont des structures de données contenant des informations concernant les fichiers stockés dans certains systèmes de fichiers (notamment de type Linux/Unix).

À chaque fichier correspond un numéro d'inode (*inumber*) dans le système de fichiers dans lequel il réside, unique au périphérique sur lequel il est situé. Un inode occupera 128 ou 256 octets (taille définie à la création du système de fichiers suivant la version).

Les métadonnées les plus courantes sous UNIX sont :

- les droits d'accès en lecture, écriture et exécution selon l'utilisateur, le groupe, ou les autres ;
- les dates de dernier accès, de modification des métadonnées (inode), de modification des données (block) ;
- les identifiants du propriétaire et groupe propriétaire du fichier ;
- la taille du fichier ;
- le nombre d'autres inodes (liens) pointant vers le fichier ;
- le nombre et numéros de blocs utilisés par le fichier ;
- le type de fichier : fichier simple, lien symbolique, répertoire, périphérique, etc.

Remarque : par défaut, un bloc a une taille de 4096 octets (4 KiO).

```
// Crée un fichier vide
```

```
$ touch fichier
```

```
// Affiche le numéro d'inode (-i)
```

```
$ ls -il fichier
```

```
655480 -rw-rw-r-- 1 tv tv 11 sept. 5 12:14 fichier
```

```
// Écrit dans un fichier
```

```
$ echo "helloworld" >> fichier
```

```
// Vide le tampon (force l'écriture dans le FS)
```

```
$ sync
```

```
// Affiche les informations contenues dans un inode
```

```
$ stat fichier
```

```
Fichier : «fichier»
```

```
  Taille : 11          Blocs : 8          Blocs d'E/S : 4096 fichier
```

```
Périphérique : 812h/2066d   In%ud : 655480   Liens : 1
```

```
Accès : (0664/-rw-rw-r--) UID : ( 1026/   tv)   GID : (65536/   tv)
```

```
Accès : 2015-09-05 12:13:05.615190874 +0200
```

```
Modif. : 2015-09-05 12:14:08.019191386 +0200
```

```
Changt : 2015-09-05 12:14:08.019191386 +0200
```

```
Créé :
```

```
// Affiche les informations complètes contenues dans un inode
```

```
$ echo "stat <655480>" | sudo debugfs /dev/sdb2
```

```
Inode: 655480 Type: regular  Mode: 0664  Flags: 0x80000
```

```
Generation: 1177868735 Version: 0x00000000:00000001
```

```
User: 1026  Group: 65536 Size: 11
```

```
File ACL: 0  Directory ACL: 0
```

```
Links: 1  Blockcount: 8
```

```
Fragment: Address: 0  Number: 0  Size: 0
```

```
  ctime: 0x55eac070:04935968 -- Sat Sep 5 12:14:08 2015
```

```
  atime: 0x55eac031:92ac4568 -- Sat Sep 5 12:13:05 2015
```

```
  mtime: 0x55eac070:04935968 -- Sat Sep 5 12:14:08 2015
```

```
  crtime: 0x55eac031:92ac4568 -- Sat Sep 5 12:13:05 2015
```

```
Size of extra inode fields: 28
```

```
EXTENTS:
```

(0):2656872

```
// Affiche (en hexa et en ASCII) les données contenues dans un bloc
sudo dd if=/dev/sdb2 bs=4096 skip=2656872 count=1 | hexdump -C
1+0 enregistrements lus
1+0 enregistrements écrits
00000000 68 65 6c 6c 6f 77 6f 72 6c 64 0a 00 00 00 00 |helloworld.....|
4096 octets (4,1 kB) copiés00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
|.....|
*
, 0,0136829 s, 299 kB/s
```

```
// Efface un fichier
$ rm fichier
```

```
// Vide le tampon (force l'écriture dans le FS)
$ sync
```

// Les données du fichier ne sont pas vraiment effacées ! Vérifions :

```
$ echo "stat <655480>" | sudo debugfs /dev/sdb2
Inode: 655480 Type: regular Mode: 0664 Flags: 0x80000
Generation: 1177868735 Version: 0x00000000:00000001
User: 1026 Group: 0 Size: 0
File ACL: 0 Directory ACL: 0
Links: 0 Blockcount: 0
Fragment: Address: 0 Number: 0 Size: 0
 ctime: 0x55eac21a:e0e00c3c -- Sat Sep 5 12:21:14 2015
 atime: 0x55eac031:92ac4568 -- Sat Sep 5 12:13:05 2015
 mtime: 0x55eac21a:e0e00c3c -- Sat Sep 5 12:21:14 2015
 crtime: 0x55eac031:92ac4568 -- Sat Sep 5 12:13:05 2015
 dtime: 0x55eac21a -- Sat Sep 5 12:21:14 2015
Size of extra inode fields: 28
EXTENTS:
```

// L'inode existe toujours mais des métadonnées ont été nettoyées (notamment le numéro de bloc occupé qui est maintenant déclaré libre)

// Mais les données sont bien toujours là !

```
$ sudo dd if=/dev/sdb2 bs=4096 skip=2656872 count=1 | hexdump -C
00000000 68 65 6c 6c 6f 77 6f 72 6c 64 0a 00 00 00 00 |helloworld.....|
1+0 enregistrements lus
1+0 enregistrements écrits
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
4096 octets (4,1 kB) copiés*
, 4,1084e-05 s, 99,7 MB/s
00001000
```

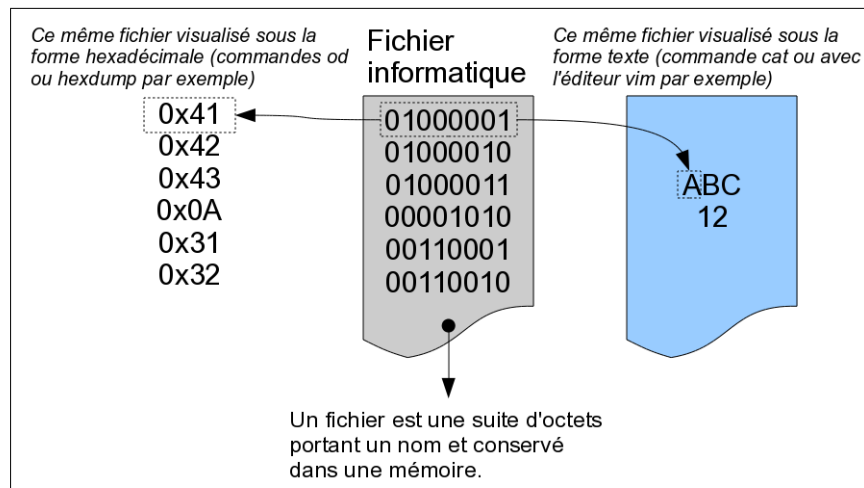
Les Fichiers « texte »

On distingue en général deux types de fichiers : **texte** et **binaire**.

Remarque : "Un fichier binaire est un fichier informatique qui n'est pas assimilable à un fichier texte." (source wikipedia). Donc, tout ce qui n'est pas un fichier texte est un fichier binaire.

Les fichiers texte ont un contenu pouvant être interprété directement comme du texte (une suite de bits représentant un caractère), la plupart du temps en codage **ASCII** (*American Standard Code for Information Interchange*).

Remarque : L'ASCII est la norme de codage de caractères en informatique la plus ancienne et la plus connue. Avec l'avènement de la mondialisation des systèmes d'information, son usage se restreint progressivement à des domaines très techniques.



Un fichier texte au “microscope”

Remarque : Comment sera interprété en ASCII l'octet 0x0A ? La réponse (et bien plus) est accessible dans le manuel en ligne de commande en faisant `man ascii`.

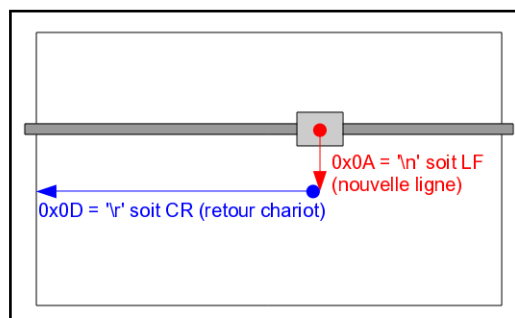
On utilise généralement un **éditeur de texte** (vi, **vim**, emacs, nano, kwrite, kate, gedit, geany, Notepad, Notepad++, UltraEdit, ...) pour manipuler ce type de fichiers.

Remarque : L'éditeur de texte est le programme le plus important et le plus utilisé par un informaticien dans l'exercice de son métier (administration, programmation). Il ne faut pas confondre éditeur de texte et traitement de texte.

Quelques exemples de fichiers textes : code source d'un programme, fichiers de configuration, etc .

Autres termes : fichier texte ou fichier texte brut ou fichier texte simple ou fichier ASCII.

En fait, les fichiers texte n'ont pas de structure car ce ne sont qu'une suite d'octets encodant des caractères. Par contre, la notion de “fin de ligne” est ambiguë. Historiquement, cela provient des premiers terminaux qui nécessitaient deux actions pour un “saut de ligne” :



Principe simplifié du saut de ligne sur un télétype (TTY)

Dans un fichier texte, la fin d'une ligne est représentée par un caractère de contrôle (ou une paire). Plusieurs conventions coexistent :

- sous les systèmes Unix/Linux, la fin de ligne est indiquée par une nouvelle ligne (LF, *Line Feed*, 1 octet) ;
- sous les machines Apple II et Mac OS jusqu'à la version 9, la fin de ligne est indiquée par un retour chariot (CR, *Carriage Return*, 1 octet) ;
- sous les systèmes CP/M, MS-DOS, OS/2 ou Microsoft Windows, la fin de ligne est indiquée par un retour chariot suivi d'une nouvelle ligne (CRLF, 2 octets).

Remarque : CRLF a été aussi adopté comme la fin de ligne standard pour les communications réseau (protocoles "Internet" comme HTTP, FTP, ...).

L'encodage des caractères

Les éditeurs de texte peuvent créer des fichiers texte avec l'encodage de caractères de leur choix. Un codage de caractères définit une manière de représenter les caractères (lettres, chiffres, symboles) dans un système informatique.

Le premier codage largement répandu fut l'ASCII. Pour des raisons historiques (les grandes sociétés associées pour mettre au point l'ASCII étaient américaines) et techniques (7 bits disponibles seulement pour coder un caractère), ce codage ne prenait en compte que 2^7 soit 128 caractères. De ce fait, l'ASCII ne comporte pas les caractères accentués, les cédilles, etc. utilisés par des langues comme le français. Ceci devint vite inadapté et un certain nombre de méthodes furent utilisées pour l'étendre.

L'ISO a donc défini de nouvelles normes, ISO 8859-1, ISO 8859-2, etc. jusqu'à ISO 8859-15. Ces jeux de caractères permettent de coder la plupart des langues occidentales. Le français utilise le plus souvent ISO 8859-1, aussi nommé `latin1`, ou ISO 8859-15 (`latin9`), qui a l'avantage de contenir des caractères (ligatures) comme le « œ » ou le symbole « € ».

Il est indispensable pour l'échange d'information de connaître le codage utilisé. Ne pas le savoir peut rendre un document difficilement lisible (remplacement des lettres accentuées par d'autres suites de caractères, ...).

Le besoin de supporter de multiples écritures demandait un nombre nettement plus élevé de caractères supportés et nécessitait une approche systématique du codage de caractère utilisé. Le codage Unicode a pour ambition d'être un surensemble de tous les autres, et est souvent représenté en UTF-8 ou en UTF-16.

L'UTF-8, spécifié dans le RFC 3629, est le plus commun pour les applications Unix/Linux et Internet. L'UTF-16 est utilisé par Java et Windows.

La norme internationale ISO/CEI 10646 définit l'*Universal Character Set* (UCS) comme un jeu de caractères universel (représenter sans ambiguïté tous les signes écrits de toutes les langues humaines connues). Ce standard est le fondement d'Unicode. Environ 10 000 caractères (symboles, lettres, nombres, idéogrammes, logogrammes) sont recensés dans l'UCS.

Remarque : L'ASCII (jeu standard sur 7 bits) n'est pas modifié par UTF-8, et les gens utilisant uniquement l'ASCII ne remarqueront aucun changement : ni dans le codage, ni dans les tailles de fichiers.

Il est conseillé de consulter les pages de manuel suivantes :

```
$ man ascii
$ man iso_8859-1    (et man iso_8859-15)
$ man utf-8        (et man unicode)
$ man charsets
```

Quel est l'encodage utilisé par un fichier ?

```
$ file bonjour.txt
bonjour.txt: UTF-8 Unicode text
```

Il existe plusieurs commandes sous Linux qui permettent de convertir des fichiers texte d'un encodage vers un autre : `iconv`, `recode`, etc ...

Comment convertir un fichier texte qui est en UTF-8 en **ISO8859-1** (latin1) ?

```
$ iconv -f UTF-8 -t ISO8859-1 bonjour.txt -o bonjour_latin1.txt
```

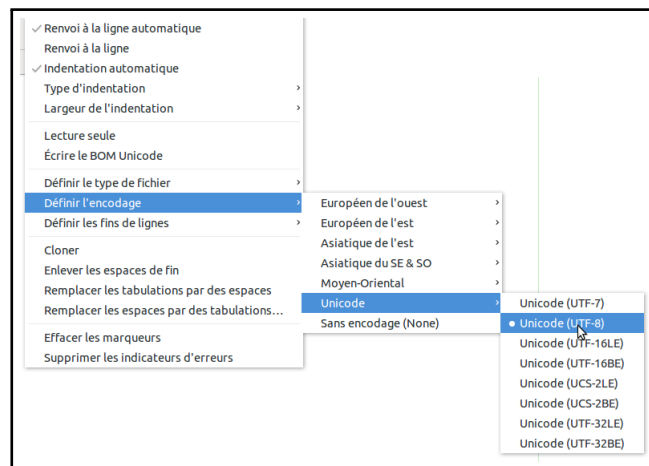
La commande `iconv -l` permet de lister l'ensemble des jeux codes connus et supportés.

La situation concernant l'encodage des caractères n'est pas encore stable. Les problèmes se régleront probablement avec l'uniformisation Unicode. Mais, il y a des risques dans le cas d'échange entre systèmes hétérogènes. Cela concerne notamment :

- l'utilisation des flux de texte dans les programmes
- les échanges sur internet
- les noms de fichiers et de répertoires

Par précaution (et le technicien informatique est prudent !), il est donc conseillé de ne jamais utiliser de caractères étendus ou spéciaux (comme l'espace) dans les noms de fichiers et de répertoires, de privilégier l'encodage Unicode et d'être cohérent avec les fichiers qui permettent de déclarer l'encodage utilisé (cas des fichiers **html** et **xml** par exemple).

Exemple : choix de l'encodage avec l'éditeur de texte geany



Créer un répertoire (dossier) et se déplacer dans l'arborescence

La commande `mkdir` permet de créer un nouveau répertoire et la commande `cd` de se déplacer à l'intérieur de celui-ci :

```
$ mkdir tmp
```

```
$ ls -l
drwxrwxr-x 2 tv tv 4096 sept. 2 18:27 tmp
```

```
$ cd tmp
```

```
$ ls -al
drwxrwxr-x 2 tv tv 4096 sept. 2 18:27 .
```

```
drwxrwxr-x 3 tv tv 4096 sept. 2 18:27 ..  
  
$ cd ..  
  
$ ll  
drwxrwxr-x 3 tv tv 4,0K sept. 2 18:27 ./  
drwx----- 13 tv tv 4,0K sept. 2 18:27 ../  
drwxrwxr-x 2 tv tv 4,0K sept. 2 18:27 tmp/  
  
$ alias  
alias ll='ls -halF'  
...
```

*Remarque : La commande **ls** permet de lister le contenu d'un répertoire. **ll** est un alias sur la commande **'ls -halF'**.*

Le nom de répertoire `".."` indique, où que vous soyez, le répertoire qui se trouve immédiatement au dessus. On l'appelle le **répertoire parent**. Un autre nom de répertoire particulier est `"."` : c'est le répertoire dans lequel vous êtes actuellement. On l'appelle le **répertoire courant**. Ils sont très utilisés pour créer des chemins relatifs dans l'arborescence.

Si vous voulez connaître le chemin absolu où vous vous trouvez, vous pouvez utiliser la commande **pwd** :

```
$ pwd  
/home/tv/tmp
```

Remarque : un chemin absolu est toujours référencé par rapport à la racine de votre arborescence et commence donc toujours par un slash `/`.

Créer un fichier texte

– vide :

```
$ touch vide  
  
$ ls -l vide  
-rw-r--r-- 1 tv tv 0 2010-07-17 15:56 vide
```

```
$ file vide  
vide: empty
```

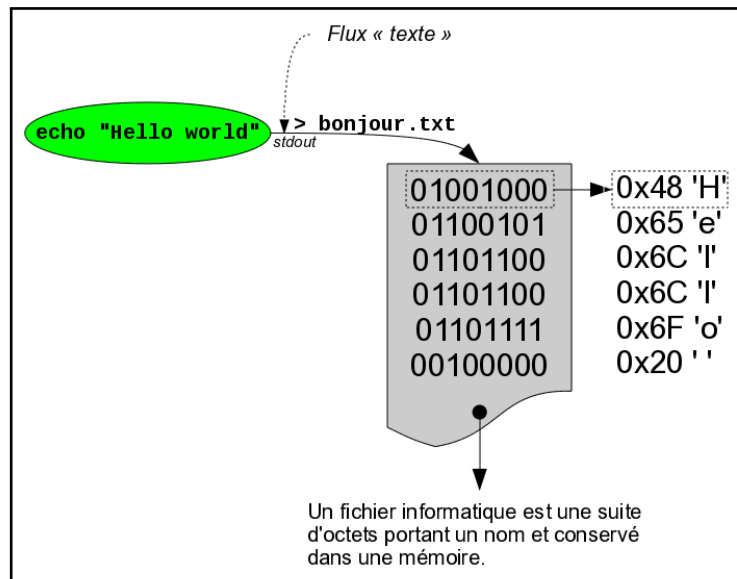
– avec un contenu :

```
$ echo "Hello world" > bonjour.txt  
$ ls -l bonjour.txt  
-rw-r--r-- 1 tv tv 12 2010-07-17 15:55 bonjour.txt  
  
$ file bonjour.txt  
bonjour.txt: ASCII text
```

Remarque : Les redirections d'entrées/sorties

*Par défaut, les commandes récupèrent les données tapées par l'utilisateur au clavier (**stdin**). Le résultat de leur exécution s'affiche à l'écran (**stdout**). En cas d'erreur à l'exécution, les messages d'erreur apparaissent aussi à l'écran (**stderr**). Il est possible d'indiquer à l'interpréteur de commandes de rediriger ces flux d'E/S vers (ou depuis) un fichier. Par exemple : **> sortie** signifie que les données*

générées par la commande seront écrites dans le fichier de nom *sortie* plutôt qu'à l'écran. Si le fichier *sortie* existait déjà, son ancien contenu est effacé, sinon ce fichier est créé au lancement de la commande.



Utilisation d'une redirection de *stdout* (>) vers un fichier

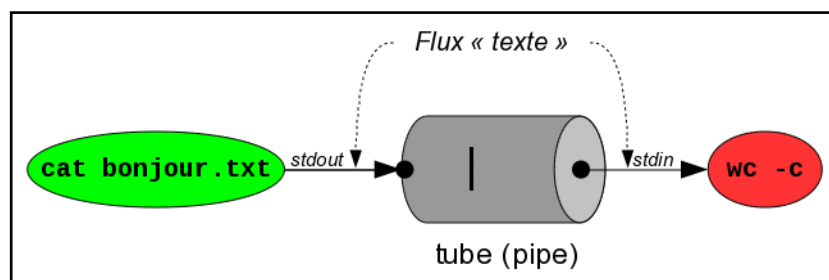
Afficher le contenu d'un fichier texte

Il existe de nombreuses possibilités pour afficher le contenu d'un fichier texte. En voici quelques-unes :

```
$ cat bonjour.txt
$ cat -n bonjour.txt ; nl bonjour.txt
$ strings bonjour.txt
$ more bonjour.txt
$ less bonjour.txt
```

Sous Unix/Linux, il est possible de “relier” des commandes :

```
$ cat bonjour.txt | wc -c
```



Utilisation d'un tube (pipe) en ligne de commande

*Remarque : Le shell Unix dispose d'un mécanisme appelé **tube** (ou pipe). Ce mécanisme permet de chaîner des processus (commandes en cours d'exécution) de sorte que la sortie d'un processus (*stdout*) alimente directement l'entrée (*stdin*) du suivant. Le symbole utilisé pour créer des tubes dans les shells Unix est la barre verticale |, appelée communément pipe. Le pipe est très utilisé sur Unix pour associer plusieurs commandes dont on enchaîne les traitements. C'est un mécanisme de communication inter-processus (IPC).*

Une dernière qui illustre bien la philosophie UNIX/Linux :

```
$ while read ligne ; do echo "contenu : $ligne"; done < bonjour.txt
```

Remarque : Les redirections d'entrées/sorties

*Ici l'utilisation de < permet de rediriger le flux d'E/S depuis un fichier (**bonjour.txt**).*

Examiner le contenu d'un fichier texte

Un fichier texte contient fondamentalement une suite de bits. La particularité d'un fichier texte est que l'ensemble du fichier respecte un codage de caractères standard. Il existe de nombreux standards de codage de caractères, ce qui peut rendre problématique la compatibilité des fichiers texte.

La norme ASCII (*American Standard Code for Information Interchange*) est la norme de codage de caractères en informatique la plus connue et la plus largement compatible. L'ASCII définit 128 caractères numérotés de 0 à 127 et codés en binaire de 0000000 à 1111111. Sept bits suffisent donc pour représenter un caractère codé en ASCII. Toutefois, les ordinateurs travaillant (presque) tous sur huit bits (un octet), chaque caractère d'un texte en ASCII est stocké dans un octet dont le 8e bit est 0. Les caractères 0 à 31 et le 127 ne sont pas affichables. Ils correspondent à des caractères (commandes) de contrôle de terminal informatique.

Pour en savoir plus :

- `man ascii`
- fr.wikipedia.org/wiki/Ascii

Pour afficher le contenu brut d'un fichier (texte ou binaire), on utilisera soit la commande `od` soit la commande `hexdump` :

```
$ od -ca -t x1 bonjour.txt
$ hexdump -C bonjour.txt
```

Modifier le contenu d'un fichier texte

Le système d'exploitation ne permet que de très simples modifications d'un fichier : on peut soit modifier un (ou plusieurs) octet soit ajouter des octets en fin de fichier.

Vous pouvez modifier 'w' en 'W' :

```
$ hexedit bonjour.txt
$ cat bonjour.txt
```

Remarque : il est impossible en utilisant les services de l'OS de supprimer ou d'insérer du texte dans un fichier (sauf à la fin). Ce sont des opérations bien trop complexes car elles nécessiteraient un décalage d'un ensemble d'octets dans le fichier. Pour réaliser cela, il faut soit utiliser un éditeur de texte soit écrire soi-même un programme équivalent.

Ou on peut ajouter du texte à la fin du fichier :




```
$ date +"le %A %d %B %Y à %T" >> bonjour.txt
$ echo "by $USER" >> bonjour.txt
$ cat bonjour.txt
```

Remarque : Les redirections d'entrées/sorties

» **sortie** semblable à la redirection > sauf que si le fichier **sortie** existait déjà, son ancien contenu est conservé et les nouvelles données sont copiées à la suite.

Éditer un fichier texte (vim)

vi est l'éditeur de texte standard d'Unix et il a été l'éditeur favori de nombreux *hackers* jusqu'à l'arrivée d'**Emacs** en 1984. Tout système se conformant aux spécifications Unix intègre **vi** et il est donc encore largement utilisé par les utilisateurs (surtout les administrateurs et programmeurs) des différentes variantes d'Unix.

La version incluse actuellement dans les **Linux** est le plus souvent **vim** (*vi improved*), un clone de **vi** qui comporte quelques différences avec celui-ci. **vi/vim** comprend trois modes de fonctionnement : le mode normal, le mode **commande** et le mode **insertion**. Après le lancement de **vi/vim**, c'est le mode normal qui est actif. Pour passer en mode insertion (de texte évidemment) il faut appuyer sur la touche  ou . On sait que l'on est en mode insertion par l'affichage de **INSERT** en bas de la fenêtre. Pour sortir de ce mode, il faut appuyer sur la touche  et l'affichage de **INSERT** en bas de la fenêtre disparaît. Pour passer en mode commande, il faut taper **':'**.

Quelques commandes intéressantes :

```
:q! : sortie sans sauvegarde
:wq : sortie avec sauvegarde
:x   : sortie avec sauvegarde
$    : se déplacer sur le dernier caractère de la ligne
ZZ   : sortie avec sauvegarde
:w   : sauvegarde sans sortie
Ctrl f : afficher la page suivante
Ctrl b : afficher la page précédente
Ctrl d : afficher la demi-page suivante
Ctrl u : afficher la demi-page précédente
e      : se déplacer à la fin du mot
b      : se déplacer au début du mot
w      : se déplacer au début du mot suivant
H      : se déplacer en haut de l'écran
L      : se déplacer en bas de l'écran
M      : se déplacer au milieu de l'écran
z.     : décaler l'affichage avec la ligne courante au centre
z(return) : décaler l'affichage avec la ligne courante en haut
z-     : décaler l'affichage pour que la ligne courante
:num_ligne : se déplacer à la ligne num_ligne
G (ou :$) : aller à la fin du fichier
u       : annulation de la dernière modification
dd      : suppression de la ligne courante
2dd     : suppression des deux lignes suivantes
D       : suppression de la fin de la ligne à partir du curseur
:3,7 d  : suppression des lignes 3 à 7
:3,7 t 10 : copie des lignes 3 à 7 après la ligne 10
:3,7 m 10 : transfert des lignes 3 à 7 après la ligne 10
yy      : mémorisation de la ligne courante (copier)
3yy     : mémorisation des 3 lignes suivantes (copier)
p       : copie ce qui a été mémorisé après le curseur
P       : copie ce qui a été mémorisé avant le curseur
:set nu : affichage des numéros de ligne
/mot    : recherche le mot mot (on se déplace avec n ou N ou *)
```

*Remarque : Il existe en réalité une quantité astronomiques de commandes dans **vi**, et en particulier dans **vim**, et chaque personne utilise, en général, qu'une petite partie d'entre elles en fonction de ses habitudes (et souvent, pas les mêmes que vous...).*

Manipuler des fichiers et des répertoires

Se déplacer dans l'arborescence :

```
$ cd $HOME/tmp
```

Créer le répertoire `temp` :

```
$ mkdir temp
```

Se déplacer dans le répertoire `temp` :

```
$ cd temp
```

Copier le fichier `/etc/passwd` dans le répertoire courant (désigné par un `.`) :

```
$ cp /etc/passwd .
```

Lister le contenu du répertoire :

```
$ ls
```

*Remarque : le fichier **passwd** contient la liste des utilisateurs de la machine (sans les mots de passe) et le répertoire **/etc** contient l'ensemble des fichiers de configuration de la machine (ce sont tous des fichiers textes ASCII)*

Faire une copie de sauvegarde d'un fichier :

```
$ sudo cp -a /etc/passwd ../passwd.bak
```

```
$ ls -l ..
```

Copier un fichier :

```
$ cp ./passwd ./utilisateurs
```

Renommer un fichier :

```
$ mv ./utilisateurs ./listeUtilisateurs.txt
```

Visualiser le contenu d'un fichier texte ASCII :

```
$ more listeUtilisateurs.txt
```

```
$ cat listeUtilisateurs.txt
```

```
$ less listeUtilisateurs.txt
```

Rechercher un fichier dans son répertoire personnel :

```
$ find $HOME -name listeUtilisateurs.txt -print
```

```
$ find $HOME -name *.txt -print
```

```
$ find $HOME -name *.txt -exec ls -l {} \;
```

Remarque : l'étoile `` est un caractère joker qui a la particularité de remplacer n'importe quel caractère autant de fois que nécessaire*

Effacer un fichier :

```
$ rm listeUtilisateurs.txt
```

Remarque : l'option `-f` force la suppression (sans demander de confirmation) et celui-ci a été supprimé de manière définitive !

Copier un répertoire :

```
$ cd ..
```

```
$ cp -r ./temp ./temp1
```

Renommer un répertoire :

```
$ mv ./temp1 ./temp2
```

Déplacer un répertoire :

```
$ mv ./temp2 ./temp
```

Déplacer un fichier :

```
$ mv ~/tmp/passwd.bak $HOME
```

Effacer un répertoire :

```
$ cd ..
```

```
$ rm -rf $HOME/tmp/temp
```

Remarque : Le répertoire (et tout son contenu avec l'option `-r`) a été supprimé définitivement !

Effacer un fichier :

```
$ rm $HOME/passwd.bak
```

Contrôler l'accès à vos fichiers

Sous UNIX, il existe deux types de sécurité pour les fichiers et répertoires : les droits et permissions UNIX, disponibles sur tous les UNIX et les ACL (*Access Control List*), plus complets.

Il est primordial de connaître la sécurité UNIX standard, dont le fonctionnement est très simple, car elle suffit le plus souvent.

Pour afficher les permissions, il faut utiliser la commande `ls` avec l'option `-l` :

```
$ ls -l /kernel*
-r-xr-xr-x 1 root wheel 1926444 Jul 13 11:15 /kernel
-rwx---r-x 1 root wheel 5606172 Jul 13 11:59 /kernel.GENERIC
```

Le premier caractère (ici `'-'`) correspond au type de fichier :

- `'-'` pour un fichier normal ;
- `'d'` pour un répertoire,
- `'l'` pour un lien symbolique ;
- `'s'` pour une *socket* ;
- `'c'` pour un fichier spécial de type “périphérique caractère” ;
- `'b'` pour un fichier spécial de type “périphérique bloc” ;

*Remarque : sous Unix, **TOUT EST FICHIER**. Ce principe offre une interface générique pour manipuler n'importe quelle ressource (cf. les appels `open`, `read`, `write` et `close`).*

`rw-rw-rw-` correspond aux droits, de, respectivement : l'utilisateur propriétaire (`rw-`), le groupe propriétaire (`-rw-`) et "les autres" (`-rw-`). Les fichiers, dans cet exemple, appartiennent à l'utilisateur `root` et au groupe `wheel`.

Il y a trois types de permissions :

- **r** : accès en lecture (*read*)
- **w** : accès en écriture (*write*)
- **x** : possibilité d'exécution pour un fichier ou de "traversée" pour un répertoire

Remarque : il faut distinguer les permissions qui s'appliquent aux fichiers et aux répertoire. Par exemple : pour modifier le contenu d'un fichier (cad "écrire dedans"), il vous faut le droit `w` sur ce fichier. Par contre, pour créer, supprimer ou renommer un fichier, il vous faudra le droit `w` sur le répertoire dans lequel vous voulez faire l'opération.

Chacune de ces permissions peuvent être attribuée à :

- **u** : *user*, l'utilisateur
- **g** : *group*, le groupe
- **o** : *other*, les autres
- **a** : *all*, tout le monde

*Remarque : attention, la vérification des droits d'accès se fait dans l'ordre **ugo**. Dès qu'une concordance est trouvée, elle s'applique !*

En plus de ces droits de base, il existe aussi des **droits spéciaux** pour les **fichiers** :

- le droit **s** (dans le bloc **u**) : utilise l'UID (identifiant) du propriétaire (*Set-UID* ou *SUID*) lors de l'exécution du fichier à la place l'UID de l'utilisateur
- le droit **s** (dans le bloc **g**) : utilise l'ID (identifiant) du groupe propriétaire (*Set-GID* ou *SGID*) lors de l'exécution du fichier
- le droit **t** (dans le bloc **o**) : pour la conservation du code en mémoire lors de l'arrêt de l'exécution

C'est grâce au bit *SUID* que `sudo` permet d'exécuter des commandes en "root" :

```
$ ls -l /usr/bin/sudo
-rwsr-xr-x 2 root root 70K mars 12 17:35 /usr/bin/sudo
```

*Attention : attribuer le droit **s** (Set-User-ID) abusivement peut entraîner de sérieuses failles de sécurité (par exemple ne jamais le faire pour le programme `cat` par exemple sinon n'importe qui pourra visualiser TOUS les fichiers du système!).*

Des droits spéciaux s'appliquent aussi pour les **répertoires** :

- le droit **s** (dans le bloc **g**) : (*SGID bit*) lorsqu'un répertoire sera créé, il le sera avec le GID du répertoire parent et non avec celui du propriétaire qui le crée (modification du fonctionnement par défaut et permet un travail collaboratif)
- le droit **t** (dans le bloc **o**) : (*sticky bit*) seul le propriétaire d'un fichier pourra le supprimer (restriction du droit `w` pour tous)

La commande `chmod` permet de changer les permissions en utilisant un **mode littéral** :

```
$ ls -l .Xdefaults
-rw----- 1 calimero promo00 61 Aug

$ chmod g+rx .Xdefaults
$ ls -l .Xdefaults
```

```
-rw-r-x--- 1 calimero promo00 61 Aug

$ chmod a-x .Xdefaults
$ ls -l .Xdefaults
-rw-r----- 1 calimero promo00 61 Aug

$ chmod u=rx .Xdefaults
$ ls -l .Xdefaults
-r-xr----- 1 calimero promo00 61 Aug
```

Vous pouvez aussi utiliser le **mode octal** pour changer les permissions. Les valeur possibles sont :

- 0 → --- : aucun droit
- 1 → --x : exécution
- 2 → -w- : écriture
- 3 → -wx : écriture + exécution
- 4 → r-- : lecture
- 5 → r-x : lecture + exécution
- 6 → rw- : lecture + écriture
- 7 → rwx : lecture + écriture + exécution

Remarque : $r (2^2 = 4) + w (2^1 = 2) + x (2^0 = 1) = 7$, soit par exemple $(644 \rightarrow rw - r - -r - -)$

Par exemple :

```
$ ls -l .Xdefaults
-r-xr----- 1 calimero promo00 61 Aug 1 13:29 .Xdefaults

$ chmod 750 .Xdefaults
$ ls -l .Xdefaults
-rwxr-x--- 1 calimero promo00 61 Aug 1 13:29 .Xdefaults
```

*Remarque : Pour la valeur du mode, on peut fournir 3 ou 4 chiffres (le premier chiffre étant facultatif). Le premier chiffre (facultatif) correspond au droit **s** ou **t**, le deuxième chiffre à **u**, le troisième à **g** et le quatrième à **o**.*

Lorsqu'un nouveau fichier est créé, on distingue deux situations particulières :

- la création d'un fichier : quels sont les droits par défaut ?
- la copie d'un fichier : quels sont les droits du fichier copié ?

Les droits par défaut d'un nouveau fichier sont définis par rapport à un **masque** des droits défini pour chaque utilisateur avec la commande **umask**. La commande **umask** permet donc d'afficher ou de modifier le masque de création de fichier de l'utilisateur.

```
$ help umask
```

```
// En octal :
```

```
$ umask
0022
```

```
// En littéral :
```

```
$ umask -S
u=rwx,g=rx,o=rx
```

```
// On indique donc :  
// - avec un bit '0' les droits que l'on autorise et  
// - avec un bit '1' les droits que l'on interdit  
  
// Exemple : 2 -> 010 soit r-x (le droit w sera bloqué par le masque)
```

Un fichier est toujours créé par un programme : une commande (`touch`, `cat`, `cp`, ...), un éditeur (`vim`, `geany`, ...), un compilateur (`gcc`), ou tout autre application (`nautilus`, `syslog`, ...).

Exemple :

- a) Le programme utilisé définit les droits qu'il désire pour le fichier à créer, par exemple :
- `rw-rw-rw-` (666) pour des fichiers réguliers (non exécutable)
 - `rw-rwxrwx` (777) pour des fichiers exécutables

- b) Puis le système applique le masque défini par `umask` pour créer les droits du fichier :

droits d'origine (666)	r	w	-	r	w	-	r	w	-
masque (022)	0	0	0	0	1	0	0	1	0
opération droits d'origine & ~masque	↓	↓	↓	↓	X	↓	↓	X	↓
droits du fichier nouvellement créé	r	w	-	r	-	-	r	-	-

Soit l'opération suivante : $666 \& \sim 022 = 644 = rw-r-r-$

Lors de la copie d'un fichier, c'est le même principe qui est appliqué en utilisant cette fois les **droits du fichier source**. Il existe des options (`-p`, `-a`, ...) qui modifient ce comportement et permettent de préserver les propriétés du fichier source.

Remarque : par contre si le fichier destination existe (écrasement), le masque n'est pas utilisé et à la place on utilise les droits du fichier destination : droits fichier source & droits fichier destination

Les commandes **chown** et **chgrp** permettent de changer, respectivement, l'utilisateur propriétaire et le groupe.

Caractères spéciaux

Les caractères spéciaux ou génériques (*wildcard characters*) permettent de désigner un ensemble d'objet et notamment un ensemble de noms de fichiers (le caractère `*` étant le plus connu et le plus utilisé).

Ils peuvent aussi désigner un ensemble de chaînes de caractères. On parle alors d'**expressions rationnelles** (ou **expressions régulières**) qui s'appliquent aux commandes d'édition (`vi`, `sed`, ...) ou à des filtres (`grep`, `egrep`, `awk`, ...).

Une **expression rationnelle** (ou **expression régulière**) est une chaîne de caractères que l'on appelle parfois un motif et qui décrit un ensemble de chaînes de caractères possibles selon une syntaxe précise. Elles sont notamment aujourd'hui utilisées dans l'édition et le contrôle de texte.

En savoir plus : `$ man 7 regex`

Les caractères associés aux noms de fichier sont interprétés par le *shell* avant le lancement de la commande :

* désigne toutes les chaînes de caractères (y compris la chaîne vide)
? désigne un caractère quelconque
[...] désigne un caractère quelconque appartenant à la liste
[!...] désigne une liste de caractères à exclure
{...,...} désigne une liste de caractères (une chaîne)

Exemples :

```
$ touch abc.s codage codage.c fichier.txt texte
```

```
$ ls *  
abc.s codage codage.c fichier.txt texte
```

```
$ ls *.*  
abc.s codage.c fichier.txt
```

```
$ ls *.*  
abc.s codage.c
```

```
$ ls ??
```

```
$ ls f*  
fichier.txt
```

```
$ ls a?c*  
abc.s
```

```
$ ls *[ac]*  
abc.s codage codage.c fichier.txt
```

```
$ ls [^a]*  
codage codage.c fichier.txt texte
```

```
$ ls *.*??  
fichier.txt
```

```
$ ls *{abc,cod}*  
abc.s codage codage.c
```

```
$ ls [a-c]*  
abc.s codage codage.c
```

```
$ ls *.{c,txt}  
codage.c fichier.txt
```

Les caractères associés aux **expressions régulières** :

. désigne un caractère
* remplace zéro fois ou n fois le caractère qui le précède
\+ remplace 1 fois ou n fois le caractère qui le précède
\? remplace 0 zéro fois ou 1 fois le caractère qui le précède
\b désigne la chaîne vide (en début ou en fin de ligne)
[...] désigne un caractère quelconque appartenant à la liste

~ désigne le début de la ligne
\$ désigne la fin de la ligne
[~...] désigne une liste de caractères à exclure
\{m\} désigne un nombre exact m d'occurrences d'un caractère
\{m,\} désigne un nombre minimum m d'occurrences d'un caractère
\{m,n\} désigne un nombre d'occurrences d'un caractère compris entre un min m et un max n
\(...\.) désigne une chaîne de caractère ou une expression régulière
\\ désigne une alternative
\<mot\> délimitation d'un mot

Il est possible d'annuler l'interprétation d'un caractère spécial ou de contrôle de trois manières en utilisant des caractères de protection :

\ : l'antislash annule la signification du caractère suivant
'...' : les simples quotes annulent tous les caractères
"..." : les doubles quotes annulent tous les caractères sauf ' , \ et \$

grep, egrep, fgrep permettent d'afficher les lignes correspondant à un motif donné. C'est l'une des commandes les plus utilisées (notamment dans des tubes) pour des recherches dans du texte.

grep peut utiliser des classes de caractères prédéfinies comme :

[[:digit:]] (chiffres), [[:lower:]] (minuscules), [[:print:]] (affichables), [[:punct:]] (ponctuation), [[:space:]] (espace), [[:upper:]] (majuscules), et [[:xdigit:]] (chiffres hexadécimaux).

Par exemple, [[[:alnum:]]] correspond à [0-9A-Za-z].

sed est un éditeur ligne non interactif. Il reçoit du texte en entrée, que ce soit à partir de `stdin` ou d'un fichier, réalise certaines opérations sur les lignes spécifiées de l'entrée, une ligne à la fois, puis sort le résultat vers `stdout` ou vers un fichier. A l'intérieur d'un script *shell*, sed est habituellement un des différents outils composant un tube. De toutes les opérations de la boîte à outils sed, on utilise principalement : printing (affichage vers `stdout`), deletion (suppression) et substitution (substitution).

Quelques exemples avec sed :

```
1d      : supprime la première ligne de l'entrée.
/^$/d   : supprime toutes les lignes vides.
/Linux/p : affiche seulement les lignes contenant Linux
s/Windows/Linux/ : substitue Linux à chaque première instance de Windows
s/Windows/Linux/g : substitue Linux à chaque instance de Windows
s/ *$//   : supprime tous les espaces à la fin de toutes les lignes.
s/00*/0/g : compresse toutes les séquences consécutives de zéros en un seul zéro.
/Windows/d : supprime toutes les lignes contenant Windows.
s/Windows//g : supprime toutes les instances de Windows, en laissant le reste de la ligne intact.
```

awk est un langage d'examen et de traitement de motifs. awk possède un langage de manipulation de texte plein de fonctionnalités avec une syntaxe proche du C. awk casse chaque ligne d'entrée en champs. Par défaut, un champ est une chaîne de caractères consécutifs délimités par des espaces (bien qu'il existe des options pour changer le délimiteur). awk analyse et opère sur chaque champ, ce qui le rend idéal pour gérer des fichiers texte structurés, particulièrement des tableaux, des données organisées en ensembles cohérents, tels que des lignes et des colonnes.

```
// Taille des partitions montés :
$ df | sed 1d | awk '{print $1 " = " $2}'
/dev/sda5 = 12G
```

```
/dev/sda7 = 34G
/dev/sda1 = 100M
/dev/sda2 = 49G
/dev/sda4 = 51G
```

// Espace disponible sur les partitions montés :

```
$ df | sed 1d | awk '{print $1 " = " $4}'
/dev/sda5 = 912M
...
```

Afficher toutes les lignes contenant au moins une majuscule :

```
$ getent passwd | grep '[A-Z]'
$ getent passwd | grep '[:upper:]'
```

Afficher toutes les lignes commençant par la lettre a :

```
$ getent passwd | grep '^[a]'
```

Afficher toutes les lignes contenant `bash` :

```
$ getent passwd | grep bash
```

Remplacer le *shell* `bash` par le *shell* `csch` pour tous les utilisateurs :

```
$ getent passwd | grep bash | sed 's/bash/csh/g'
```

Afficher toutes les lignes contenant au moins une occurrence de `bash` ou `sh` :

```
$ getent passwd | grep '\(bash|sh\)\'
```

Exploiter un motif :

```
$ echo "thierry.vaira@orange.fr" | sed 's/\(.*\)@\(.*\)\/nom:\1 domain:\2/'
nom:thierry.vaira domain:orange.fr
```

```
$ ifconfig | sed -e "s/^ *//g" | grep -Eo "([0-9]{1,3}\.){3}[0-9]{1,3}"
192.168.52.2
192.168.52.255
255.255.255.0
127.0.0.1
255.0.0.0
```

Automatiser des tâches

Objectifs

Les administrateurs (ou parfois les programmeurs) ont souvent le besoin d'automatiser des traitements (surveillance, sauvegarde, maintenance, installation, ...). Pour cela, ils peuvent :

- saisir des commandes ou faire des groupement de commandes à partir d'un *shell* : fréquent
- **créer des scripts** (avec le *shell* ou avec un autre langage de script comme Perl, Python, PHP, etc ...) : très fréquent

L'automatisation des tâches est un domaine dans lequel les scripts *shell* prennent toute leur importance. Il peut s'agir de préparer des travaux que l'on voudra exécuter ultérieurement (voir **at**, **batch**) grâce à un système de programmation horaire (voir **crontab**) mais on utilise aussi les scripts pour simplifier l'utilisation de logiciels complexes ou écrire de véritables petites applications.

Les scripts sont aussi très utiles lorsqu'il s'agit de faire coopérer plusieurs utilitaires système pour réaliser une tâche complète. On peut ainsi écrire un script qui parcourt le disque à la recherche des fichiers modifiés depuis moins d'une semaine, en stocke le nom dans un fichier, puis prépare une archive **tar** contenant les données modifiées, les sauvegarde sur une bande, puis envoie un e-mail à l'administrateur pour rendre compte de son action, etc ... Ce genre de programme est couramment employé pour automatiser les tâches administratives répétitives.

Les *shells* proposent un véritable langage de programmation, comprenant toutes les structures de contrôle, les tests et les opérateurs arithmétiques nécessaires pour réaliser de petites applications. En revanche, il faut être conscient que les *shells* n'offrent qu'une bibliothèque de routines internes très limitée. Nous ferons alors fréquemment appel à des utilitaires système externes.

En résumé, les scripts sont utilisés surtout pour :

- automatiser les tâches administratives répétitives
- simplifier l'utilisation de logiciels complexes
- mémoriser la syntaxe de commandes à options nombreuses et utilisées rarement
- réaliser des traitements simples

Les shell scripts

Les scripts sont des suites de commandes exécutées par un *shell*. Les commandes internes d'un *shell* forment ainsi un véritable langage de programmation, souvent appelé langage de commandes.

Ces scripts sont surtout utilisés par les administrateurs et les développeurs car ils permettent de créer très rapidement des programmes, des nouvelles commandes ou des "moulinettes" (utilitaires).

Un *shell script* est :

- un fichier texte ASCII (on utilisera un éditeur de texte)
- exécutable par l'OS (le droit **x** sous Unix/Linux)
- interprété par un *shell* (**/bin/bash** par exemple sous Linux)

*Remarque : on a l'habitude de mettre l'extension **.sh** au fichier script.*

Exemple d'un *shell script* :

```
// Édition d'un script
$ vim script.sh
#!/bin/bash
echo "je suis un script"

// Ajout du droit d'exécution
$ chmod ugo+x script.sh

// Vérification
$ ls -l script.sh
-rwxr-xr-x 1 tv tv 38 2010-08-05 11:22 script.sh

// Exécution du script
$ ./script.sh
je suis un script
```

Remarque : un script se doit de commencer par une ligne "shebang" qui contient les caractères '#!' (les deux premiers caractères du fichier) suivi par l'exécutable de l'interpréteur. Cette première ligne permet d'indiquer le chemin (de préférence en absolu) de l'interpréteur utilisé pour exécuter ce script sinon le shell par défaut sera utilisé.

Avec un *shell script*, il est possible :

- d'exécuter et/ou grouper des commandes
- d'utiliser des variables prédéfinies (\$?, ...), des variables d'environnement (\$HOME, \$USER, ...)
- de gérer ses propres variables
- de faire des traitements conditionnels (**if**, **case**) et/ou itératifs (**while**, **for**)

Il existe plusieurs manières d'exécuter un script :

- le rendre exécutable :

```
$ chmod +x monscript
$ ./monscript
```

- passer son nom en paramètre d'un *shell* :

```
$ sh monscript
```

- utiliser une fonction de lancement de commande du *shell* :

```
$ . monscript
$ source monscript
```

Les variables

Une variable est un espace de stockage pour un résultat.

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

De manière générale, la plupart des langages de scripts admettent :

- qu'une variable puisse changer de type au cours de son existence. On parle de **typage faible**.
- que ce ne sont pas les variables qui ont un type, mais les valeurs. On parle de **typage dynamique**.

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

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

```
nom=[valeur]
```

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

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

Créer une variable :

```
$ chaine=bonjour
```

Afficher une variable :

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


Manipuler des variables :

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

$ unset chaine
$ echo $chaine
```

Manipuler des chaînes de caractères :

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

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

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

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

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

Elles sont généralement définies en MAJUSCULES :

\$PATH sous Unix/Linux

%PATH% sous Windows

Les substitutions de variables

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

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

Les formes les plus utilisées sont :

```
// Obtenir la longueur d'une variable :  
$ PASSWORD="secret"  
$ echo ${#PASSWORD}  
  
// Fixer la valeur par défaut d'une variable nulle :  
$ echo ${nom_utilisateur:=‘whoami’}  
  
// Utiliser des tableaux :  
$ tableau[1]=tv  
$ echo ${tableau[1]}  
  
//Supprimer une partie d'une variable :  
$ fichier=texte.htm  
$ echo ${fichier%.htm}.html  
$ echo ${fichier##*.}
```

Les substitutions de commandes

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

```
$(commande) ou ‘commande’
```

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

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

```
// Déterminer le nombre de fichiers présents dans le répertoire courant :  
$ NB=$(ls | wc -l)  
$ echo $NB  
  
// Récupérer le chemin d'accès :  
$ CHEMIN=‘dirname /un/long/chemin/vers/toto.txt’  
$ echo $CHEMIN  
/un/long/chemin/vers
```

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

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

L'évaluation arithmétique

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

```
$(expression)
```

Par exemple, pour calculer la somme de deux entiers :

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

Pour manipuler des expressions arithmétiques, on pourra utiliser aussi :

```
// la commande expr :
$ expr 2 + 3

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

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

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

Les variables internes du shell

Ces variables sont très utilisées dans la programmation des scripts :

```
$0      : nom du script ou de la commande
$1,$2, ... : paramètres du shell ou du script
$*      : tous les paramètres
$@      : idem (mais "$@" eq. à "$1" "$2"... )
$#      : nombre de paramètres
$-      : options du shell
```

`$?` : code retour de la dernière commande
`$$` : le PID du shell
`$!` : le PID du dernier processus shell lancé en arrière-plan
`$_` : le dernier argument de la commande précédente. Cette variable est également mise dans l'environnement de chaque commande exécutée et elle contient le chemin complet de la commande.

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

// Afficher quelques variables internes :

```
#!/bin/bash
```

```
echo "Ce script se nomme : $0"
```

```
echo "Il a reçu $# paramètre(s)"
```

```
echo "Les paramètres sont : $@"
```

```
echo
```

```
echo "Le PID du shell est $$"
```

```
$ chmod +x variablesInternes.sh
```

```
$ ./variablesInternes.sh
```

```
Ce script se nomme : ./variablesInternes.sh
```

```
Il a reçu 0 paramètre(s)
```

```
Les paramètres sont :
```

```
Le PID du shell est 8807
```

```
$ ./variablesInternes.sh le petit chat est mort
```

```
Ce script se nomme : ./variablesInternes.sh
```

```
Il a reçu 5 paramètre(s)
```

```
Les paramètres sont : le petit chat est mort
```

```
Le PID du shell est 8078
```

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

La gestion des options

Il arrive souvent qu'un script ait besoin de traiter des options passées en arguments (avec « `-` » ou « `--` ») ce qui lui permet d'exécuter des actions différentes. Il existe plusieurs techniques pour traiter des options :

- le faire soi-même avec des `if ...` (déconseillée car solution lourde et complexe)
- utiliser la commande interne `getopts` (`man bash`)
- utiliser la commande externe `getopt` (`man getopt`)

Remarque : `getopt()` existe aussi en langage C (`man 3 getopt`).

Les commentaires

Les commentaires sont introduits par le caractère `'#'`. On peut aussi utiliser l'instruction nulle `':'`.

L'affichage sur la sortie standard

Pour afficher du texte, on pourra utiliser `echo` ou `printf` :

```
#!/bin/bash
```

```
CHAINE="world"
```

```
echo -n "Un message : hello $CHAINE"
```

```
echo -e "\tUn message : hello $CHAINE"
```

```
echo 'hello $CHAINE'
```

```
echo "hello \"$CHAINE"
```

```
printf "Un message : hello %s\n" $CHAINE
```

La saisie de données

Pour réaliser la saisie sur le périphérique d'entrée (généralement le clavier), on utilisera `read` :

```
#!/bin/sh
```

```
# Saisie du nom
```

```
echo -n "Entrez votre nom: "
```

```
read nom
```

```
# Affichage du nom
```

```
echo "Votre nom est $nom."
```

```
# Lecture silencieuse d'une et une seule touche avec un timeout de 5s
```

```
read -s -n 1 -t 5 TOUCHE
```

```
echo $TOUCHE
```

```
exit 0
```

Les tests et conditions

Les tests peuvent être lancés par la commande interne `'test'` qui prend en argument les conditions, et renvoie `'0'` si le test est vrai, et `'1'` sinon.

La forme la plus courante est l'utilisation de crochets (`[` ou `[[` depuis la version 2) qui encadrent le test.

Pour connaître la syntaxe des tests sur les fichiers, les chaînes de caractère, les valeurs et les associations, faire :

```
help test
```

```
help [[
```

Remarque : Chaque élément du test, et les crochets, doivent être bien délimités par au moins un espace. C'est une erreur courante que d'oublier les espaces.

Les structures conditionnelles

La structure if-then-else

Il existe plusieurs formes syntaxiques autorisées :

```
// Cas 1 :
if liste de commandes
then liste de commandes
fi

// Cas 2 :
if liste de commandes ;then liste de commandes ;fi

// Cas 3 :
if liste de commandes
then liste de commandes
else liste de commandes
fi

// Cas 4 :
if liste de commandes
then liste de commandes
elif liste de commandes
then liste de commandes
else liste de commandes
fi
```

Exemples :

```
SI le répertoire n'existe pas
ALORS
    on le crée
FIN
```

```
# shell script: if1.sh <nom_repertoire>
if [ ! -d "$1" ]
then
    mkdir "$1"
fi
```

```
# shell script : if2.sh
```

```
# depuis la version 2, on peut utiliser la syntaxe étendue [[
# [[ est un mot clé, pas une commande
if [[ $# != 1 ]] #ou: if [ $# -ne 1 ]
then echo "Usage: $(basename $0) <nom_fichier>"; exit 1
fi
```

```
# la commande test ou [
if [ -e $1 ]
then echo "Le fichier $1 existe"
else echo "Le fichier $1 n'existe pas"
fi
```

```
# shell script : if3.sh
# on peut utiliser directement des commandes dans le if
if cmp a b &> /dev/null
then echo "Les fichiers a et b sont identiques."
else echo "Les fichiers a et b sont différents."
fi
```

Les choix multiples case et select

Contrôle conditionnel à cas : les comparaisons se font au niveau des chaînes de caractères.

La structure case est alors :

```
case "$variable" in
"motif1" ) action1 ;;
"motif2" ) action2 ;;
"motif3a" | "motif3b" ) action3 ;;
...
* ) action_default ;;
esac
```

Exemples :

```
# shell script : case1.sh
case $1 in
1|3|5|7|9) echo "chiffre impair";;
2|4|6|8)  echo "chiffre pair";;
0) echo "zéro";;
*) echo "c'est un chiffre qu'il me faut !!!";;
esac
```

```
# shell script : case2.sh
echo "Des vacances ?"
read reponse
case $reponse in
[yYoO]) echo "fainéant !";;
[nN])  echo "alors pas de vacances !";;
*) echo "erreur saisie invalide";;
esac
```

La construction select, adoptée du *Korn Shell*, est souvent utilisée pour construire des menus :

```
select nom [ in mot ] ; do liste ; done
```

```
select variable [in liste]
do
commande ...
break
done
```

La liste de mots à la suite de in est développée, créant une liste d'éléments. Le symbole d'accueil PS3 est affiché, et une ligne est lue depuis l'entrée standard.

Si la ligne est constituée d'un nombre correspondant à l'un des mots affichés, la variable nom est remplie avec ce mot. Si la ligne est vide, les mots et le symbole d'accueil sont affichés à nouveau. Si une fin de

fichier (EOF) est lue, la commande se termine. Pour toutes les autres valeurs, la variable `nom` est vidée. La ligne lue est stockée dans la variable `REPLY`.

La liste est exécutée après chaque sélection, jusqu'à ce qu'une commande `break` ou `return` soit atteinte.

Exemple :

```
# Affiche l'invite
PS3='Menu: '
select choix in "création" "modification" "suppression" "quitter"
do
echo "Votre choix est $choix"
break # il faut savoir s'arrêter !
done
```

Les contrôles itératifs (les boucles `for`, `while` et `until`)

Il existe de nombreuses utilisations des boucles `for`, `while` et `until`.

La boucle `for`

```
// Cas 1 : Autant de tours de boucle que d'éléments dans la liste et variable prenant
           successivement chaque valeur
for variable in liste_de_valeurs
do liste de commandes
done

// Cas 2 : Autant de tours de boucle que de fichiers dans le répertoire courant et variable
           prenant successivement chaque nom de fichier (non caché)
for variable in *
do liste de commandes
done

// Cas 3 : Autant de tours de boucle que d'arguments dans $* et variable prenant
           successivement $1 $2 etc...
for variable
do liste de commandes
done
```

Exemples :

```
# shell script : for1a.sh
for i in 1 2 3 4 5 6 7 8 9 10
do
echo "\$i=$i"
done

# shell script : for2.sh
for fic in *
do
if [[ -f $fic ]]
then echo "$fic est un fichier"
elif [[ -d $fic ]]
then echo "$fic est un répertoire"
```



```
fi
done

# shell script : for1b.sh
# Syntaxe standard améliorée en utilisant la commande seq
START=1
LIMITE=10
SEQUENCE=$(seq -s ' ' $START $LIMITE)
for i in $SEQUENCE
do
    echo -n "$i "
done

# Idem (style C)
# Double parenthèses, et "LIMITE" sans le "$".
for ((i=START; i <= LIMITE; i++))
do
    echo -n "$i "
done

# Encore mieux : la virgule chaîne les opérations.
for ((i=START, j=LIMITE; i <= LIMITE; i++, j--))
do
    echo -n "$i-$j "
done

# shell script : for3.sh
# Autres boucles for simples
for PLANETE in Mercure Vénus Terre Mars Jupiter Saturne Uranus Neptune Pluton
do
    echo $PLANETE
done

# Possibilité de substitution de commande
EXT="sh"
for ligne in $( find . -type f -name ".*$EXT" | sort )
do
    echo "$ligne"
done

# Si la 'liste' est manquante, la boucle opère sur '$@'
for arg
do
    echo -n "$arg "
done
```

La boucle while

```
// Cas 1 : Tant que la condition est vraie.
while condition
do liste de commandes
done
```

```
// Cas 2 : Boucle infinie dans laquelle il faudra prévoir la sortie par exit, return ou break.
```

```
while true      ou while :
do liste de commandes
done
```

Exemples :

```
#!/bin/bash
# shell script : while.sh
# Compter jusqu'à 10 dans une boucle "while"
```

```
LIMITE=10
a=1
while [ "$a" -le $LIMITE ]
do
    echo -n "$a "
    let "a+=1"
done
```

```
# Idem : syntaxe C
((a = 1))      # a=1
while (( a <= LIMITE ))
do
    echo -n "$a "
    ((a += 1)) # let "a+=1"
done
```

La boucle until

```
// Cas 1 : Jusqu'à ce que la condition soit vraie
until liste de commandes
do liste de commandes
done
```

```
// Cas 2 : Boucle sans fin (idem while true)
until false
do liste de commandes
done
```

Exemples :

```
# shell script : until.sh
ctr=0
until (( ctr >= 10 ))
do
    ((ctr+=1))
    echo "tour numéro $ctr"
done
```

*Remarque : Les commandes de contrôle de boucle **break** et continue correspondent exactement à leur contre partie dans d'autres langages de programmation. La commande **break** termine la boucle (en sort), alors que **continue** fait un saut à la prochaine itération de la boucle, oubliant les commandes restantes dans ce cycle particulier de la boucle.*

Les fonctions

Les fonctions permettent l'appel de commandes dans l'environnement courant (partage des variables du script père).

Les propriétés des fonctions sont :

- passage de paramètres possibles (\$0 \$1 \$2 ...);
- variables locales possibles;
- rapidité car la fonction est lue à la déclaration et non à l'exécution;
- retourner une valeur de retour (**return**).

Exemple :

```
#!/bin/bash
# shell script : rootCheck.sh
CLEARSCR=clear

function root_check()
{
    # seul root peut executer ce script :)
    id | grep "uid=0(root)" > /dev/null 2>&1
    if [ $? != "0" ]
    then return 1
    else return 0
    fi
}

function say_hello()
{
    if [[ $1 == "true" ]]
    then
        $CLEARSCR
    fi
    echo -e "\x1B[49;34;1m
=====
Copyleft (C) 2010 <tv>
=====
\x1B[0m\n"
}

# main :
say_hello true
say_hello false

root_check
if [ $? = "1" ]
then
    echo -e "\x1B[49;31;1mERREUR: ce script ne peut etre execute que sous le compte root !\
\x1B[0m"
    exit 1;
fi

exit 0
```

*Remarque : on utilise ici des commandes d'échappement (**Echap** ou **Esc** dont le code ascii est **0x1B**) qui permet de personnaliser le terminal notamment en utilisant des couleurs.*

Annexe 1 : Une liste de commandes de base

Voici quelques commandes usuels :

`dpkg` : un gestionnaire de paquet pour Debian
`apt-get` : utilitaire APT pour la gestion des paquets (voir aussi `aptitude`)
`alias` : crée ou supprime des alias de commandes
`pwd` : affiche le chemin d'accès au répertoire courant
`man` : permet de consulter les manuels de référence
`clear` : efface l'écran
`echo` : affiche une ligne de texte (et aussi des variables)
`cd` : permet de se déplacer dans une arborescence
`ls` : liste le contenu d'un répertoire
`rm` : supprime un fichier (voir aussi `rmdir`)
`cp` : permet la copie de fichier (voir aussi `cp -a`)
`mv` : déplace ou renomme une partie d'une arborescence
`mkdir` : crée un répertoire dans une arborescence
`touch` : modifie l'horodatage d'un fichier (permet aussi de créer un fichier vide)
`file` : affiche le type des fichiers
`type` : indique le type pour une commande
`locate` : localise un fichier
`find` : recherche des fichiers sur le système
`cat` : affiche et/ou concatène le(s) fichier(s) sur la sortie standard
`more` : affiche à l'écran l'entrée standard (page par page)
`less` : idem avec possibilité de retour en arrière
`cut` : permet d'isoler des colonnes dans un fichier
`head` : affiche les n première lignes
`join` : joint les lignes de deux fichiers en fonction d'un champ commun
`sort` : trie les lignes de texte en entrée
`paste` : concatène les lignes des fichiers
`tail` : affiche les n dernières lignes d'un fichier
`tac` : concatène les fichiers en inversant l'ordre des lignes
`uniq` : élimine les doublons d'un fichier trié
`rev` : inverse l'ordre des lignes d'un fichier
`diff` : compare des fichiers texte
`cmp` : compare deux fichiers octet par octet
`tr` : remplace ou efface des caractères
`grep` : recherche des chaînes de caractères dans des fichiers
`sed` : éditeur de flux pour le filtrage et la transformation de texte (soit un éditeur de texte non-interactif)
`awk` : manipulation de fichiers texte pour des opérations de recherches, de remplacement et de transformations complexes
`md5sum` : génère et vérifie un hachage MD5
`whereis` : permet de trouver l'emplacement d'une commande
`whatis` : donne une description d'une commande
`which` : donne le chemin complet d'une commande
`du` : affiche une arborescence et sa taille (`du -h`)
`df` : fournit la quantité d'espace occupé par les systèmes de fichiers (`df -Th`)
`od` : affiche le dump d'un fichier (voir aussi `hexdump`)
`wc` : compte les caractères, les mots et les lignes en entrée
`date` : affiche et modifie la date et l'heure
`cal` : affiche le calendrier
`bc` : calculatrice
`ln` : crée des liens physiques et symboliques

mkfs : crée un FS (file system)
fsck : vérifie un FS
mount : monte un FS
umount : démonte un FS
mke2fs : crée un FS ext2 (voir aussi dumpe2fs)
e2fsck : vérifie un FS ext2
tune2fs : paramètre un FS ext2
debugfs : débogue un FS ext2
lsattr : liste les attributs des fichiers
chattr : change les attributs des fichiers
stat : affiche des informations sur un fichier ou un système de fichier
lsof : affiche des informations sur les fichiers ouverts
fuser : identifie les processus utilisant des fichiers
fdisk : gère les tables de partitions pour Linux
cfdisk : manipule les table de partitions pour Linux (voir aussi sfdisk)
dd : convertit et copie un fichier physiquement
sync : vider les tampons du système de fichiers (finalise les opérations d'écriture)
id : affiche les identifiants d'utilisateur et de groupe effectifs et réels
whoami : affiche l'identifiant d'utilisateur
who : montre qui est connecté (voir aussi w et users)
last : affiche une liste des utilisateurs dernièrement connectés
su : change l'identifiant d'utilisateur ou permet de devenir un superutilisateur (root)
sudo : exécute une commande sous un autre compte (voir /etc/sudoers)
uname : affiche des informations sur le système
ps : affiche les processus en cours
top : affiche les tâches
kill : envoie un signal à un processus
at : permet d'exécuter ultérieurement des commandes (voir aussi batch, atq, atrm et cron)
time : exécute un programme et affiche un résumé des ressources utilisées
uptime : indique depuis quand le système a été mis en route
free : affiche les quantités de mémoire libre et utilisée du système
vmstat : affiche des statistiques sur la mémoire virtuelle
env : exécute un programme dans un environnement modifié, liste les variables d'environnement
printenv : affiche l'ensemble ou une partie des variables d'environnement

Exemples (on suppose le fichier `bonjour.txt` non vide) :

- a) `$ wc -l bonjour.txt` : compte le nombre de lignes
- b) `$ sort bonjour.txt` : trie les lignes d'un fichier texte
- c) `$ tac bonjour.txt` : affiche le fichier « à l'envers »
- d) `$ head -1 bonjour.txt` : affiche la première ligne
- e) `$ tail -2 bonjour.txt` : affiche les deux dernières lignes
- f) `$ md5sum bonjour.txt > bonjour.md5` : génère un hachage MD5
- g) `$ md5sum -c bonjour.md5` : vérifie un hachage MD5
- h) `$ echo "fin" >> bonjour.txt` : ajoute la chaîne "fin" à la fin du fichier `bonjour.txt`
- i) `$ md5sum -c bonjour.md5` : vérifie un hachage MD5
- j) `$ touch bonjour.txt` : met à jour l'horodatage du fichier `bonjour.txt`
- k) `$ cat bonjour.txt | tr -s ' ' '.'` : affiche le contenu du fichier `bonjour.txt` en remplaçant tous les espaces par des points

Annexe 2 : L'arborescence Unix/Linux

Voici quelques répertoires usuels à la racine d'un système GNU/Linux :

`/bin` : commandes accessibles à tous les utilisateurs nécessaires au démarrage et au fonctionnement minimum du système

`/boot` : fichiers statiques du chargeur de démarrage (un fichier `\texttt{vmlinuz*}` est une image compressée du noyau et un fichier `\texttt{initrd.img*}` contient des modules du noyau (gestion des systèmes de fichiers, drivers, ...))

`/dev` : fichiers spéciaux d'accès aux périphériques

`/etc` : fichiers de configuration système spécifique à la machine (ce sont tous des fichiers textes ASCII)

`/home` : répertoires personnels des utilisateurs

`/lib` : bibliothèques logicielles partagées nécessaires pour les exécutables de `bin` et `sbin`

`/mnt` : point de montage pour des systèmes de fichiers temporaires

`/media` : point de montage pour des systèmes de fichiers amovibles (cdrom, clé usb, ...)

`/opt` : paquets de logiciels applicatifs supplémentaires et optionnels (non inclus dans la distribution)

`/proc` : système de fichiers virtuel donnant accès aux variables du noyau et des différents processus

`/root` : répertoire personnel du superutilisateur

`/sbin` : commandes systèmes réservées au superutilisateur

`/tmp` : fichiers temporaires

`/usr` : hiérarchie secondaire (on retrouve des sous-répertoires comme `bin`, `lib` ...)

`/usr/local/` : hiérarchie tertiaire pour les données locales, spécifiques à l'ordinateur (on retrouve des sous-répertoires comme `bin`, `lib` ...)

`/var` : données variables de la machine sous forme de fichiers (base de données, logs, boîte aux lettres de messagerie, ...)

Voir fr.wikipedia.org/wiki/Filesystem_Hierarchy_Standard