Unix et son shell

1 Généralités sur Unix

Unix n'est pas un système d'exploitation mais une famille de systèmes reposant sur les mêmes idées.

Le premier Unix date de 1969 (Ritchie & Thompson chez AT&T) ; depuis, presque chaque constructeur a sa propre version. Exemple : Linux sur PC, MacOS X sur Macintosh…

- **Caractéristiques**
  - Multi-utilisateurs : cela provient d'une organisation ancienne de la forme UC+terminaux où plusieurs utilisateurs pouvaient travailler sur la même machine en même temps à partir de terminaux différents. Maintenant, c'est plutôt une personnalisation du système pour chaque utilisateur.
    Ainsi, chaque utilisateur a un login, un mot de passe et un espace de stockage sur le disque qui lui est réservé (dont la racine est son répertoire personnel) ; de plus, son utilisation est personnalisée par des fichiers de configuration.
    Un utilisateur est distingué, c'est l'administrateur système (*root* ou super-utilisateur). Il a accès à tout le système et à tous les fichiers sans garde-fous.
  - Multitâches : c'est la possibilité d'exécuter plusieurs programmes (appelés processus) « en même temps ».

- **Comment interagir avec le système ?**
  - Interface graphique *(GUI)* : par exemple Unity, GNOME ou KDE sous Linux, MacOS X sur les Macintosh ; c'est intuitif et pratique mais on est limité aux commandes prévues.
  - Ligne de commande : un programme spécial (*shell*) permet de taper des commandes au clavier ; elles sont alors exécutées par le système. Dans le cas d'une interface graphique, un programme *(terminal)* ouvre une fenêtre dans laquelle un shell est lancé, permettant de taper les commandes. C'est évidemment plus compliqué (mémorisation des commandes, syntaxe…) mais beaucoup plus puissant car cela va permettre de multiples possibilités de combinaisons.

Exercice 1

Une fois logué, ouvrez une fenêtre terminal en cliquant sur l'icône du terminal dans la barre de gauche. Essayez les commandes suivantes :

```
whoami
cal
date
date -u
date -R
date -I
```

La première commande à connaître est celle donnant les pages d'aide !

- `man` ***(manual)***

`man nom_commande` permet d'afficher la page du manuel *online* de la commande. On peut faire défiler les pages avec la souris ; on tape ensuite `q` pour quitter et revenir au shell.

Par défaut, `man` donne l'aide concernant une commande du shell.
`man n nom_commande` affiche la page du manuel de la commande, se trouvant dans la section `n`. C'est utile lorsqu'une commande existe dans plusieurs langages (C, perl, shell...).

Exercice 2

1. *Regardez l'aide de la commande `date` et vérifiez les explications concernant les options testées dans l'exercice précédent.*
2. *Testez la différence entre `man printf` et `man 3 printf`. Laquelle va vous intéresser ?*

Il est également possible d'ouvrir l'aide dans une fenêtre extérieure à partir de l'icône de recherche dans le lanceur :

1. lancer `yelp` dans le champ recherche du lanceur d'application (icône en bas à gauche) ;
2. taper `^L` (appui des touches [Ctrl] et [l]) pour ouvrir un champ recherche en haut de la fenêtre ;
3. effacer le texte présent et taper `man:nom_commande` pour avoir l'aide correspondante (ex : `man:date`) ;
4. si on veut une section particulière, on la met entre parenthèse immédiatement après le nom de la commande (ex: `man: printf(3)`).

1.1 Le système de fichiers

Un fichier est un ensemble d'informations auquel est associé un nom. Un système de fichiers permet à l'utilisateur de conserver des informations en les nommant. Il doit supporter les pannes matérielles et doit comporter un mécanisme de sécurité contre les accès illicites. Le système de fichiers d'Unix est dit arborescent car il peut être représenté par un arbre. Cet arbre est constitué d'une part de feuilles qui symbolisent les fichiers et d'autre part de branches qui symbolisent les répertoires. Ainsi un répertoire peut contenir soit des fichiers soit des sous-répertoires.

Le répertoire ou dossier *(directory)* situé au sommet de l'arborescence s'appelle la racine et est désignée par `/`. Voici un exemple d'arborescence en figure 1 :

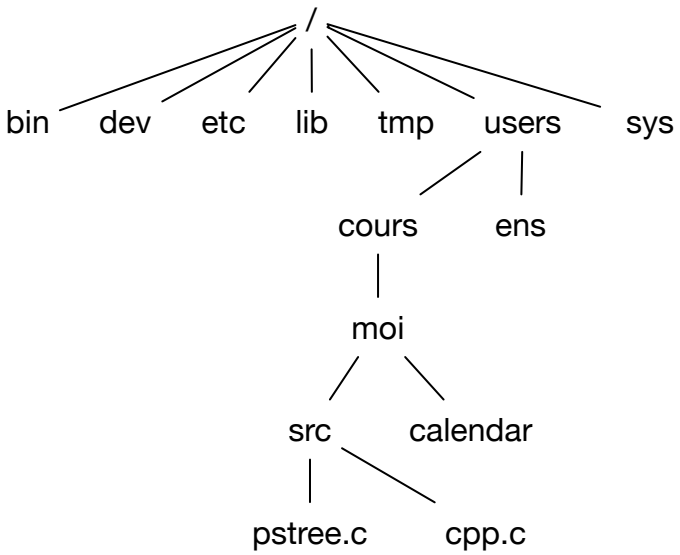

Fig. 1 – *Un exemple d'arborescence*

1.1.1 Les chemins absolus

Pour accéder à un fichier, il faut connaître son nom. Le nom d'un fichier dans l'arborescence correspond au chemin que l'on prend pour y arriver. Par exemple, pour accéder au fichier `cpp.c` du schéma, il faut partir de la racine `/`, puis aller dans le répertoire des utilisateurs (`users`), puis dans le répertoire du groupe auquel appartient l'utilisateur (`cours`), puis aller dans le répertoire de l'utilisateur (`moi`). Enfin, ce dernier a créé deux sous-répertoires pour ses besoins : il faut choisir le dossier contenant les sources des programmes (`src`), puis désigner enfin le fichier (`cpp.c`).

Le nom complet est donc la mise bout à bout des noms constituant le chemin. La barre oblique `/` a été choisie comme convention pour séparer ces noms (sauf le premier `/` qui symbolise la racine). Le nom complet du fichier, c'est-à-dire son chemin absolu, est :

```
/users/cours/moi/src/cpp.c
```

De même, le chemin absolu du fichier `calendar` est `/users/cours/moi/calendar`

1.1.2 Le répertoire courant

À tout moment de sa session, chaque utilisateur se trouve dans un répertoire de l'arborescence du système. Ce répertoire peut être représenté comme un nom absolu. Par exemple `/users/cours/moi`.

On peut ainsi désigner un fichier dans ce répertoire, il suffit de donner le nom du fichier. Par exemple, si le catalogue de travail est `/users/cours/moi`, le fichier `calendar` peut être désigné simplement par `calendar`.

On désigne donc le fichier directement par son nom. De même, si le catalogue de travail est le répertoire `/users/cours/moi/src`, on peut désigner plus simplement les deux fichiers qui y résident par `cpp.c` et `pstree.c`.

~ (tilde) : Ce répertoire a une signification particulière, c'est le répertoire de connexion. C'est celui sur lequel on se retrouve en début de session. Il est souvent appelé aussi le *home directory (`HOME`).*

1.1.3 Les chemins relatifs

Les chemins absolus sont complexes et longs à taper. Pour simplifier, on utilise souvent les chemins relatifs. Cette notion repose sur celle de catalogue de travail, c'est-à-dire du répertoire courant. En effet, si nous sommes dans le répertoire `/users/cours/moi` et que nous voulions désigner le fichier `pstree.c` situé dans le répertoire `src`, le chemin sera simplement (on décrit uniquement le chemin restant à effectuer) `src/pstree.c`

Si le catalogue de travail est `/users`, le chemin relatif pour désigner le fichier `calendar` est `cours/moi/calendar`

Très important : les chemins relatifs ne commencent jamais par le caractère barre oblique `/`. C'est la seule manière de distinguer un chemin relatif d'un chemin absolu.

Remarque : lorsqu'un nom absolu est utilisé pour désigner un fichier, il y a unicité de ce fichier (on suit le chemin depuis la racine, on est donc bien sûr d'atteindre le bon fichier). En revanche, lorsqu'on utilise des noms relatifs, on ne peut plus affirmer que le fichier est unique. Par exemple le fichier `src/cpp.c` peut représenter un fichier depuis `/users/cours/moi` ou depuis `/users/cours/lui` ou encore `/users/ens/u1/src/cpp.c`...

1.1.4 Les conventions . et ..

Le symbole `.` (point) représente le répertoire de travail. Ceci évite de taper le nom absolu du répertoire courant dans certaines commandes.

Le symbole `..` (deux points) représente le nom du répertoire situé juste au-dessus (répertoire père). Ainsi, si le répertoire courant est `/users/cours/moi` alors `..` représente le répertoire `/users/cours`.

1.1.5 Noms de fichiers

Un nom de fichier peut être composé d'un nombre quelconque de caractères quelconques. Cependant, il vaut mieux respecter un certain nombre de conventions pour ne pas risquer de rencontrer de soucis lors de la désignation ou l'ouverture de fichiers :

- ne pas utiliser de caractères accentués ; les noms pourraient être mal retranscrits lors du transfert des fichiers d'un système à un autre ;
- ne pas utiliser d'espace dans les noms, cela complique son écriture (il faut mettre le nom entre guillemets ou faire précéder l'espace d'un `\` et certains logiciels ne savent pas les gérer. On peut remplacer l'espace par le caractère _ (souligné) ;
- toujours ajouter l'extension de fichiers (`.c` pour un fichier C, `.py` pour un programme Python, `.txt` pour un fichier texte...) ; cela permet au système d'associer plus facilement un logiciel à un fichier lors de son ouverture.

1.2 Divers

1.2.1 Autocomplétion

La touche TAB permet de compléter automatiquement une commande ou un nom de fichier.

- Dans le cas d'une commande, si une seule complétion est possible, la commande est affichée complétée. Par exemple, si on tape `clea` suivi de TAB, la commande est complétée en `clear`.
- Toujours pour une commande, si plusieurs complétions sont possibles, un bip retentit ; un second appui sur TAB affiche alors toutes les commandes possibles.

> **Exercice 3**
> *Tapez `clea` puis TAB pour compléter.*
> *Tapez pw suivi de deux fois TAB pour voir toutes les complétions possible de la commande.*

- Le même mécanisme peut être utilisé pour les noms de fichiers : TAB complète un nom de fichier (utilisé dans une commande) s'il n'y a qu'une possibilité et sinon deux appuis permettent de lister toutes les complétions.

1.2.2 Historique

Sous le shell *bash*, les touches flèche-haut et flèche-bas permettent de naviguer avec facilité dans l'historique des commandes.

2 Commandes de base

2.1 Commandes sur les fichiers

La syntaxe générale d'une commande Unix est :

```
commande [-options [args_option]] args_commande
```

où les crochets indiquent le caractère optionnel (et on peut souvent mettre plusieurs options). Les arguments d'une commande sont dans la plupart des cas un nom ou une liste de noms de fichiers.

- `ls` ***(list)***
  `ls` liste les répertoires et fichiers du répertoire courant.
  `ls -a` inclut les fichiers cachés (dont le nom commence par '`.`').
  `ls -l` affiche la liste au format long, en incluant des informations pour chaque fichier : type, droits d'accès, nombre de liens, propriétaires, taille, date de modification et nom.
  On peut combiner les deux options avec `ls -l -a` équivalent à `ls -la`.
  Si on fait suivre la commande `ls` d'un nom de fichier ou de répertoire, les informations se limitent à ce fichier ou au contenu de ce répertoire.

- `cd` ***(change directory)***
  `cd` sans argument fait revenir au répertoire de départ de l'utilisateur (`$HOME`).
  `cd ..` fait revenir au répertoire père du répertoire courant.
  `cd nom_r` change le répertoire courant pour le faire pointer sur le répertoire indiqué en argument.

- `pwd` ***(print working directory)***
  `pwd` affiche le nom absolu du répertoire courant.

- `mkdir` ***(make directory)***
  `mkdir nom_r` crée le répertoire portant le nom donné en argument dans le répertoire courant.

- `rm` ***(remove)***
  `rm fichiers` supprime les fichiers indiqués.
  `rm -i fichiers` supprime les fichiers en mode interactif, c'est-à-dire en questionnant l'utilisateur avant chaque suppression.
  `rm -d noms` supprime les éléments indiqués par leur nom, qu'il s'agisse de fichiers ou de répertoires vides.
  `rm -r noms` supprime les éléments indiqués et pour chaque répertoire, travaille récursivement (c'est-à-dire effectue la commande pour chaque fichier et répertoire inclus). Attention, cette commande est dangereuse puisqu'elle efface tous les fichiers ainsi que le contenu de tous les répertoires nommés.

- `rmdir` ***(remove directory)***
  `rmdir nom_r` efface le répertoire indiqué si celui-ci est vide (si ce n'est pas le cas, il faut utiliser la commande `rm -r`). Cette commande est équivalente à `rm -d nom_r`.

- `cp/mv` ***(copy/move)***
  `cp fichier_1 fichier_2` effectue la copie du premier fichier et lui attribue le nom `fichier_2`.
  `cp fichier_1 ... fichier_n nom_r` effectue la copie de chacun des fichiers nommés et met chaque copie dans le répertoire indiqué.

`mv fichier_1 fichier_2` renomme le premier fichier en `fichier_2` ; cela permet également de le déplacer si le second nom inclut un chemin d'accès à un répertoire.

`mv fichier_1 ... fichier_n nom_r` déplace chacun des fichiers nommés dans le répertoire indiqué.

Dans tous les cas, l'option `-i` permet d'effectuer les opérations en mode interactif.

Exercice 4

Ces exercices ont pour but de vous familiariser avec les commandes Unix. Il n'est donc pas indispensable de les respecter à la lettre mais au contraire d'expérimenter ! Bien sûr, on ne fera aucune opération à l'aide de l'interface graphique (sauf indication contraire) mais on utilisera au maximum les commandes du shell dans le terminal.

1. *Listez les fichiers de votre répertoire, listez les fichiers cachés.*
2. *Créez un répertoire `TP` et s'y placer.*
3. *À l'aide d'un éditeur de texte graphique (`gedit` par exemple), créez 4 fichiers `texte.c`, `src.c`, `doc.txt` et `config`. On lancera `gedit` à partir champ recherche du lanceur d'application (icône en bas à gauche) ;*
4. *Créez 3 sous-répertoires en une commande : `src1`, `texte` et `cfg`.*
5. *Déplacez tous les fichiers `.c` dans `src1`, le fichier `doc.txt` dans `texte` et `config` dans `cfg`.*
6. *Créez un sous-répertoire `src2` et copiez-y tous les fichiers de `src1`.*
7. *Supprimez le contenu de `src1` puis le répertoire `src1`.*

2.2 Commandes sur les processus

Un processus est l'image mémoire d'un programme disque s'exécutant en mémoire. C'est la combinaison :

- d'une configuration mémoire (code du programme, état des variables statiques, état de la pile) ;
- d'une configuration des registres du processeur (adresse de la prochaine instruction, valeurs des registres).

Sous Unix, toute activité s'exécute dans le cadre d'un processus. Unix étant un système d'exploitation multitâches, les processus peuvent s'exécuter en parallèle selon leurs priorités. Chaque processus peut évoluer dynamiquement en mémoire après sa phase de création suivant certains critères tels que : temps passé en exécution ou priorité du processus. À intervalles réguliers (quantum), le processus actif est interrompu et le contrôle de l'unité centrale est donné au processus de plus haute priorité qui est en attente.

L'exécution d'une commande depuis le shell génère la création d'un processus, ce n'est pas le shell qui exécute la commande. Le shell se contente de créer un processus (son fils) qui exécutera la commande à sa place. Pendant le temps d'exécution le shell attend sans rien faire (jusqu'à la fin de son fils).

- `ps` ***(process status)***

  `ps` permet de lister tous les processus ayant un terminal, lancés par l'utilisateur.

  `ps -x` liste tous les processus de l'utilisateur, même ceux lancés sans terminal.

  `ps -a` liste tous les processus lancés depuis le terminal, même ceux d'autres utilisateurs.

  `ps -j` affiche plus d'informations : utilisateur, numéro du processus, état, temps d'exécution...

  Une information importante est le PID *(Process identification)* du processus, c'est-à-dire son numéro d'identification. Il va permettre de désigner le processus dans une commande.

La commande `ps -axj` permet donc d'avoir la liste étendue (option `-j`) de tous les processus lancés sur le système, attachés ou non à un terminal (option `-x`) et de l'utilisateur ou pas (option `-a`).

Lancement en arrière-plan
Le caractère **&** placé à la fin d'une commande permet de la lancer en arrière-plan, le shell renvoie alors un numéro de processus et attend une nouvelle commande de l'utilisateur pendant que la commande s'exécute. La commande `jobs` donne la liste des processus en arrière-plan (l'option `-l` permet d'obtenir en plus leur PID).

- `kill` ***(kill process)***

En temps normal, lorsqu'un processus est lancé, il s'arrête avec la fin de son exécution. Il se termine alors normalement. Cependant, il arrive qu'on désire interrompre un processus avant sa fin pour diverses raisons.

Un premier moyen est d'interrompre le processus par `^C` (appui des touches [Ctrl] et [c]).

Cependant, `^C` n'est pas toujours efficace, surtout pour des processus lancés en arrière-plan qui ne reçoivent pas de commandes du clavier. Unix propose la commande `kill` pour tuer un processus, c'est-à-dire le terminer prématurément. `kill` admet une option correspondant à des signaux de -1 à -15, mais nous n'aborderons pas ce détail pour nous souvenir uniquement de l'option -9 qui permet de tuer un processus efficacement en indiquant son PID.

`kill -9 PID` tue le processus de numéro indiqué.

- `bg` ***(background)*** **et** `fg` ***(foreground)***

Il arrive parfois que l'on lance une commande au premier plan pour s'apercevoir ensuite qu'elle dure trop longtemps et que l'on souhaite la faire passer à l'arrière-plan. Il n'est alors pas nécessaire de l'arrêter et de la relancer.

On peut interrompre une commande au premier-plan avec `^Z` (appui des touches [Ctrl] et [z]). La commande est alors suspendue en attendant que l'utilisateur la reprenne. Pour ce faire, on peut utiliser la commande `fg` qui débloque la commande et la remet au premier-plan ou `bg` qui la reprend mais en la détachant du terminal pour la lancer en arrière-plan.

Chacune de ces commandes a un numéro de tâche en option : il ne s'agit pas du PID du processus mais du numéro obtenu avec la commande `jobs`. Voici un exemple :

```
$ sleep 1000
^Z
[1]+ Stopped       sleep 1000
$ jobs
[1]+ Stopped       sleep 1000
$ bg 1
[1]+ sleep 1000 &
```

Exercice 5

1. *Quels processus voit-on avec `ps` ? `ps -j` ? `ps -jx` ?*
2. *Listez tous les processus vous appartenant sur la machine.*
3. *Lancez la commande `sleep 100` et l'arrêter.*
4. *Lancez la commande `sleep 100` en arrière-plan. Comment faire pour l'arrêter ?*
5. *Lancez la commande `sleep 100`, la suspendre et la reprendre en arrière-plan ; arrêtez-là.*

Exercice 6

1. *Écrire un programme C qui fait une boucle infinie en affichant* `"coucou !"` *toutes les secondes (*`sleep(1);`*, défini dans* `<unistd.h>`*, permet d'attendre une seconde).*
2. *Lancez-le et arrêtez-le ; lancez-le en arrière-plan et arrêtez-le.*
3. *Lancez-le, suspendez-le et reprenez-le en arrière-plan ; arrêtez-le.*

Attention, pour être sûr que le système affiche votre chaîne, il faut forcer l'affichage en terminant la chaîne à afficher par un `'\n'` (et écrire donc `printf('coucou !\n')`). Si vous ne le faites pas, le système met la chaîne à afficher en mémoire temporaire mais n'affiche rien à l'écran tant qu'il n'a pas besoin.

2.3 Commandes diverses

- `wc` ***(word count)***
  `wc fichier` compte et affiche le nombre de lignes, mots et octets du fichier indiqué.
  L'option `-c` ne compte que le nombre d'octets.
  L'option `-l` ne compte que le nombre de lignes.
  L'option `-w` ne compte que le nombre de mots.

- `grep`
  `grep` permet de rechercher une chaîne de caractères dans un ou plusieurs fichiers en affichant la ou les lignes contenant cette chaîne.
  La syntaxe est :

```
grep [options] chaîne [fichier ...]
```

  Les options sont :

`-i` ignore les majuscules/minuscules ;
`-h` supprime le nom de fichier devant chaque ligne ;
`-n` affiche le numéro de ligne dans le fichier ;
`-v` affiche toutes les lignes sauf celles contenant la chaîne.

  La chaîne de caractère peut être une chaîne simple. Ainsi
  `grep abc src/poirot.c` : affiche les lignes contenant `abc` dans le fichier `src/poirot.c`
  `grep b2 games/chess*.txt` : affiche les lignes contenant `b2` dans les fichiers texte commençant par `chess` (voir plus loin les jokers).

  La chaîne peut aussi contenir des caractères spéciaux :

`^` indique que la chaîne ne peut se trouver qu'en début de ligne ;
`$` indique que la chaîne ne peut se trouver qu'en fin de ligne ;
`[...]` liste une série de caractères dont un devra se trouver dans la chaîne.

  Il faudra dans ce cas mettre la chaîne entre guillemets pour éviter que ces caractères ne soient interprétés par le shell.

  Exemple :

`grep "^e" /etc/passwd`	affiche les lignes commençant par `e` dans `/etc/passwd`
`grep -v ";" *.c`	affiche les lignes ne contenant pas le caractère `;`
`grep "sh$" /etc/passwd`	affiche les lignes se terminant par la chaîne `sh`
`grep "[123][ab]" ici.txt`	affiche les lignes contenant `1`, `2` ou `3` suivi de `a` ou `b`

Remarque : La commande `grep`, si elle est appelée avec plusieurs noms de fichiers, place devant chaque ligne trouvée le nom du fichier. Dans le cas d'un seul fichier ou de l'entrée standard, seule la ligne trouvée est affichée.

2.4 Les protections

Le système Unix connaît tous les utilisateurs. Chaque utilisateur appartient à un groupe. Ceux qui n'appartiennent pas au groupe d'un utilisateur sont les autres. Chaque fichier garde la trace du propriétaire et du groupe auquel il appartient. Le système permet de contrôler trois opérations sur les fichiers : lecture, écriture et exécution.

Ainsi pour tout objet, on connaît le type de l'objet (cf. `ls -l`), les droits d'accès sur cet objet pour le propriétaire, les utilisateurs du même groupe que le propriétaire et les autres. Ces droits sont représentés par trois ensembles de trois attributs `rwx` :

- `r` pour lecture (read),
- `w` pour écriture (write),
- `x` pour exécution (execute),
- `-` (tiret) signifie l'absence de droit.

- Le premier ensemble concerne l'utilisateur.
- Le deuxième ensemble concerne le groupe.
- Le troisième ensemble tous les autres utilisateurs (n'appartenant pas au groupe).

Voici un extrait de l'affichage de la commande `ls -l` :

```
drwx------   7 lazard  staff   224  9 fév 23:21 Desktop
drwxr-xr-x  15 lazard  staff   480 24 jan 16:28 Documents
-rwxr-xr-x   1 lazard  staff  8432 13 fév 10:48 TP
-rw-r--r--   1 lazard  staff    74 13 fév 10:48 TP.c
-rw-r--r--   1 lazard  staff  3408 30 jan 17:27 total.txt
```

Les deux premiers noms correspondent à des répertoires (petit `d` en tête de ligne) tandis que les trois autres sont des fichiers. `Desktop` ne peut être ouvert que par le propriétaire tandis que tous les utilisateurs peuvent voir ce qu'il y a dans `Documents` (mais seul le propriétaire peut y écrire).

Le propriétaire peut modifier les deux derniers fichiers et les autres seulement les lire (il s'agit de fichiers textes, ils ne sont donc pas exécutables) ; `TP` peut être lu (sans grand intérêt, c'est un fichier binaire) et exécuté par tout le monde.

Remarque : Droits des répertoires

Ces droits n'ont pas tout à fait la même signification pour les répertoires (un répertoire est un fichier avec une structure particulière). En effet, le droit de lecture permet de connaître le nom de chaque objet contenu dans le répertoire, mais sans plus d'informations (on ne sait si ce sont des fichiers ou d'autres sous-répertoires). Pour avoir accès à ces renseignements complémentaires il faut avoir le droit exécution. Ce droit autorise l'accès aux informations concernant tous les éléments contenus dans le répertoire. Pour pouvoir atteindre un objet dans un répertoire il faut donc avoir le droit exécution sur tous les répertoires des niveaux supérieurs.

Le droit écriture dans un répertoire symbolise la possibilité de modifier un fichier. Pour effacer, modifier les droits d'accès d'un fichier, il faut avoir le droit écriture sur le répertoire, même si éventuellement on n'a aucun droit sur le fichier lui même (l'accès est fait par le répertoire).

Remarque : si on a un droit d'écriture sur un fichier, sans avoir le droit d'écriture sur le répertoire le contenant, il est impossible d'effacer celui-ci. En revanche on peut le modifier et éventuellement le remettre à zéro (avec la commande copie par exemple).

- `chmod` ***(change mode)***

Unix permet pour tout fichier de positionner les droits d'accès désirés selon le format ci-dessus. On spécifie ainsi les droits d'accès du fichier pour le propriétaire du fichier *(User)*, les utilisateurs appartenant au même groupe *(Group)* que le propriétaire et les autres utilisateurs *(Other)*.

La commande réalisant cette fonction est `chmod`. qui s'écrit :

```
chmod u|g|o +|- r|w|x nom_fichier
```

Avec les exemples suivants :

`chmod o-r toto`	retire le droit de lecture pour les autres utilisateurs sur le fichier `toto`
`chmod ug+w toto`	ajoute le droit d'écriture pour le propriétaire et le groupe
`chmod -rwx toto`	enlève tous les droits pour tout le monde
`chmod +x s.sh`	rend le fichier `s.sh` exécutable par tout le monde

2.5 Jokers

Les jokers ou métacaractères sont des caractères ayant des significations spéciales lorsqu'ils sont utilisés dans des noms de fichiers.

`?`	ce caractère représente un et un seul caractère quelconque.
`*`	ce caractère représente n'importe quelle suite (éventuellement nulle) de n'importe quel caractère.
`[...]`	les crochets définissent un ensemble de caractères possibles. Un `-` définit un intervalle.

Exemples :

`truc*`	représente tous les noms commençant par `truc`.
`*ici*`	représente tous les noms ayant la chaîne `ici` comme sous-chaîne.
`?x*`	représente tous les noms ayant un `x` en deuxième caractère.
`*.c`	représente tous les noms se terminant par `.c`
`?[abc]?`	représente tous les noms d'exactement trois caractères ayant un `a`, `b` ou `c` comme deuxième caractère.
`*_v[0-9]`	représente tous les noms se terminant par `_v` suivi d'un chiffre.

Exercice 7 *Placez-vous dans le répertoire `/usr/include`. Listez tous les fichiers de ce répertoire ayant `'q'` comme deuxième caractère dans leur nom. Même question avec `'q'` ou `'g'`. Revenez à votre dossier personnel et recommencez l'affichage de la liste sans vous placer dans le répertoire en question.*

Lorsque la commande `ls` est utilisée avec un nom de fichier correspondant à un répertoire, le contenu de ce dernier est affiché. L'option `-d` permet alors de ne pas afficher le contenu mais simplement le nom du répertoire.

Exercice 8

1. *Placez-vous dans le répertoire `/usr/include`. Listez tous les fichiers/dossiers de ce répertoire ayant `'z'` comme deuxième caractère dans leur nom.*

2. *On voit qu'un répertoire correspond à l'écriture `?z*` et est donc listé intégralement. Corrigez la commande pour ne lister que les noms de fichiers/dossiers correspondants.*
3. *Listez tous les fichiers/dossiers du répertoire `/usr/include` ayant `'q'` ou `'y'` comme deuxième caractère dans leur nom, sans que le contenu de ces dossiers ne s'affiche.*

Exercice 9

1. *Extraire de tous les fichiers `.h` du répertoire `/usr/include` les lignes contenant 1983. Même question avec 1987 puis 1988. Comment répondre aux trois questions (les lignes contenant 1983 ou 1987 ou 1988) en une commande ?*

2.6 Redirections

Chaque commande possède un canal d'entrée (par défaut celui du shell, c'est-à-dire le clavier) et un canal de sortie (par défaut l'écran). On peut rediriger l'un ou/et l'autre lors du lancement de la commande.

`commande >fichier` redirige la sortie par défaut dans le fichier.

`commande <fichier` redirige l'entrée par défaut depuis le fichier.

`commande <fichier_1 >fichier_2` redirige les deux.

C'est le shell qui effectue ces redirections ; autrement dit, le programmeur de la commande (dans le cas d'un script shell par exemple) n'a absolument pas besoin de s'en occuper. C'est l'utilisateur de la commande qui choisit cette possibilité. Cela permet par exemple de sauvegarder le résultat d'une commande (s'affichant normalement à l'écran) dans un fichier.

`ls >temp` liste le répertoire courant et met le résultat dans le fichier `temp` (et rien ne s'affiche à l'écran).

`wc -l <truc` compte les lignes du fichier `truc`. En effet, `wc -l` sans autre argument (c'est-à-dire sans nom de fichier) compte les lignes du canal d'entrée (par défaut le clavier), donc du fichier `truc` avec la redirection.

• **Sortie d'erreur**

Chaque commande possède en fait deux canaux de sortie : le canal normal d'affichage des résultats (par défaut l'écran et qu'on peut rediriger avec `>`) mais également le canal d'affichage des messages d'erreur qui est par défaut également l'écran mais séparé du canal d'affichage des résultats. On peut ainsi rediriger indépendamment la sortie des résultats (avec `>`) et les messages d'erreur (dont la redirection se fait avec `2>`). On a par exemple deux usages classiques :

- rediriger la sortie normal dans un fichier pour récupérer les résultats mais laisser les erreurs à l'écran (pour les voir et ne pas polluer le fichier qui pourra ainsi être réutilisé par une autre commande) ;
- rediriger la sortie d'erreur dans un fichier pour n'avoir que les résultats à l'écran.

Exercice 10 *Comparez le résultat des commandes :*

```
ls /r* -d
ls /r*
ls /r* >l.txt
ls /r* 2>err.txt
ls /r* >l.txt 2>err.txt
```

Lorsqu'on n'est pas intéressé par les messages d'erreur d'une commande, il est classique de les rediriger dans le « néant » avec la redirection `2>/dev/null`.

2.7 Tubes *(pipes)*

Plusieurs commandes peuvent être chaînées. Il peut être intéressant de faire que la sortie d'une première commande soit l'entrée d'une autre. Les redirections permettent de faire ça avec l'aide d'un fichier temporaire :

```
ls >temp
wc -l <temp
```

permet de compter le nombre de fichiers dans le répertoire courant mais cela oblige à utiliser un fichier temporaire. Il est possible de chaîner directement les commandes en tapant le caractère '|' entre les commandes.

```
commande_1 | commande_2 | commande_3
```

indique au shell que la sortie de la première commande doit être envoyée comme entrée de la deuxième commande dont la sortie est l'entrée de la troisième. L'exemple précédent s'écrit alors :

```
ls | wc -l
```

Exercice 11

1. *Enregistrez dans un fichier la liste au format long de vos fichiers, fichiers cachés et répertoires de votre répertoire racine.*
2. *À l'aide la commande* `"sort -n -k 5"`*, triez les lignes de ce fichier (*`-k 5`* indique d'utiliser la taille de chacun des fichiers/répertoires comme clef de tri (cinquième champ affiché) et* `-n` *indique de trier à l'aide la valeur numérique), et sauvegardez le résultat dans un autre fichier.*
3. *Refaire les deux tâches précédentes en une commande à l'aide d'un tube.*
4. *À l'aide la commande* `wc`*, comptez le nombre de fichiers et répertoires de votre répertoire racine. Enregistrez cette commande dans un fichier et rendez-le exécutable.*
5. *Comptez le nombre de processus dont vous êtes le propriétaire.*
6. *Listez les processus et trier-les par utilisateur (d'abord avec* `ps` *puis avec* `sort`*).*
7. *Combien de lignes contiennent* `#ifdef` *dans tous les fichiers* `.h` *de* `/usr/include` *?*

3 Shell et environnement

Il n'existe pas un shell mais plusieurs qui se différencient par la syntaxe des commandes et par leurs possibilités (historique, complétion de commandes, commandes d'édition...). on peut citer :

- Bourne shell (*sh*) 1970 (Unix version 1, puis version 7)
- C-shell (*csh*) 1982 (Unix BSD)
- Korn shell (*ksh*) 1983 (Unix system V)
- Turbo C shell (*tcsh*) 1989 (FreeBSD, premier Unix en *open source*)
- Bourne again shell (*bash*) 1990 (Linux)

Chaque utilisateur a un shell par défaut qu'il peut changer dans ses fichiers de configuration. *bash* est le plus classique car c'est le shell par défaut sur Linux et MacOS.

Un shell est en même temps un interpréteur interactif de commandes et un interpréteur d'un langage de programmation de commandes. Un programme shell est appelé « script ». C'est un fichier texte contenant des variables, des commandes et des structures de contrôle.

- **Comment exécuter un script shell ?**

1. `sh nom_script`
   Lance une copie du programme `sh` et lui fait exécuter le script. On peut bien sûr indiquer n'importe quel shell disponible sur le système.
2. `. nom_script`
   Contrairement au cas précédent, cela ne lance pas de nouveau shell mais exécute le script dans le shell courant (il n'a alors pas besoin d'être exécutable). Cela permet de modifier des variables d'environnement du shell courant (et devrait être limité à cet usage).
3. `./nom_script`
   Si le fichier script est **exécutable** (par exemple avec `chmod +x nom_script`), ceci lance un copie du shell actuel et exécute le script. C'est la manière la plus classique d'exécuter un script. Attention, la syntaxe du script ne correspond pas forcément à celle du shell si elles sont de deux types différents.
   On peut forcer le type de shell à lancer en mettant en première ligne du script le nom de l'interpréteur : `#!/bin/sh` ou `#!/bin/csh`... On est alors sûr que c'est le bon shell qui sera lancé (c'est-à-dire que la syntaxe du script correspondra) sous réserve que celui-ci soit disponible sur la machine...
   De façon plus générale, une telle commande en début de fichier texte (appelée *shebang*) permet, lors de son « exécution », de lancer le bon programme qui va interpréter les commandes dans le fichier. On peut le faire avec un shell comme ci-dessus mais aussi avec d'autres langages : `#!/usr/bin/perl`, `#!/usr/bin/python`...
   On peut remarquer qu'on ne lance pas le script simplement en indiquant son nom relatif mais bien avec son chemin absolu (grâce au `./` placé devant son nom). L'explication, liée à la variable PATH, se trouve un peu plus loin.

- **Variables**
  Un shell utilise des variables qui peuvent être définies en ligne de commande ou dans un script.
  - Variables locales (au shell)

    `x=truc` (sh/ksh) `set x=truc` (csh)

  - Variables globales

    `x=truc`
    `export x` (sh/ksh) `setenv x truc` (csh)

    Les variables globales sont accessibles par le shell et par tout processus lancé par le shell.
    Le préfixe `$` permet de récupérer la valeur de la variable :

    `x=$truc` (`x` prend la valeur de la variable `truc` et pas la valeur *truc* directement)
    `echo $x` (permet d'afficher la valeur de la variable `x`)
    La commande `printenv` (sh/csh) ou `setenv` (csh) sans argument permet d'afficher toutes les variables globales connues du shell. Voici quelques variables classiques :

    SHELL Contient le nom du shell de départ.

    HOME Contient le chemin d'accès au répertoire de départ de l'utilisateur

    HOSTNAME Contient le nom de la machine ; HOST sous *csh*.

    USER/LOGNAME Contient le nom de l'utilisateur.

    PWD Contient le chemin d'accès du répertoire courant.

    PATH Contient la liste des répertoires dans lesquels le shell cherche les programmes à exécuter.

Ce dernier ensemble de répertoires est important puisque c'est grâce à lui que le shell va trouver un programme à partir de son nom relatif. Il est classiquement formé des répertoires `/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin` et autres.

On remarque que le répertoire courant « `.` » n'apparaît pas dans la liste ci-dessus. Cela veut dire que le shell ne pourra pas exécuter un programme qui se trouve dans le répertoire courant même si on le lance en tapant son nom ! En effet, le nom étant un nom relatif, le shell va parcourir l'ensemble des répertoires du PATH pour le trouver et donc ne parcourt pas le répertoire courant. On peut remédier à ce problème en lançant systématiquement les programmes par `./nom_programme`, ce qui revient à donner le nom absolu mais ce n'est pas pratique. On peut aussi changer les fichiers de configuration (`.bash_profile` par exemple) pour inclure « `.` » dans le PATH (par la commande `PATH=$PATH:.`).

Attention, le shell parcourt les répertoires du PATH dans l'ordre indiqué. Ainsi les commandes `PATH=$PATH:.` et `PATH=.:$PATH` n'ont pas le même effet. Dans cette dernière, le répertoire courant est parcouru en premier pour trouver le nom de la commande : imaginons que votre programme s'appelle `test` (ou tout autre nom identique au nom d'une commande Unix) ; dans le dernier cas, votre répertoire étant parcouru d'abord, c'est bien votre programme qui est exécuté (mais dans ce répertoire, vous n'aurez plus facilement accès à la commande Unix de même nom) ; dans le premier cas, ce sera toujours la commande Unix qui sera exécutée et jamais votre programme (à moins de donner son nom absolu) !

4 Écriture de scripts shell

Un script shell permet de faire une extension des commandes de base. Il s'agit d'un fichier texte que le script va exécuter. On va pouvoir y mettre des variables, des commandes et des structures de contrôle pour gérer l'exécution du script. Pour lancer l'exécution d'un script `nom_Scr`, on peut soit taper `sh nom_Scr` (qui lance une copie de `sh` [ou d'un autre shell indiqué] et exécute le script), soit écrire `. nom_Scr` qui l'exécute dans le shell courant mais le plus pratique est de pouvoir lancer le script simplement en indiquant son nom `nom_Scr`. Il faut alors que le script soit exécutable en modifiant explicitement les droits d'accès du fichier texte contenant le script et que le chemin d'accès soit dans la variable `PATH` (sinon, il faut le lancer avec la commande `./nom_Scr`).

- **Arguments**

Un script étant une commande, on peut le lancer en ajoutant des arguments après son nom. Il faut alors pouvoir les récupérer dans le script. Des variables spéciales permettent de faire ça :

`$#` : renvoie le nombre d'arguments passés avec la commande.
`$n` : renvoie le n^{e} argument.
`$*` : renvoie une liste composée de tous les arguments (utilisable par exemple dans un `for`).

- **Commentaires**

Des commentaires peuvent être mis dans le script en les faisant précéder de `#`. Un `#!` sur la première ligne indique le nom du shell à lancer pour exécuter le script.

- **Chaînes de caractères**

Une chaîne de caractères indiquée entre apostrophes (`'`) est prise de façon littérale.

Une chaîne de caractères entre guillemets (`"`) est utilisée après substitution des variables : `"Bonjour $1"` correspondra à une chaîne où le `$1` aura été remplacé par le premier argument de la commande.

Une chaîne de caractères entre apostrophes inverses (`` ` ``) est remplacée par le résultat de l'exécution de la commande placée dans la chaîne : `echo` `` `ls | wc -l` `` affiche le nombre de fichiers du répertoire courant.

Si la variable `$R` contient la valeur `nom_rep` :
`echo 'ls $R | wc -l'` affiche la chaîne `ls $R | wc -l` *(affichage littéral)* ;
`echo "ls $R | wc -l"` affiche la chaîne `ls nom_rep | wc -l` *(remplacement)* ;
`echo` `` `ls $R | wc -l` `` affiche le nombre d'éléments du répertoire `nom_rep` *(exécution).*

- **Gestion des espaces**

Le shell est très strict sur la gestion des espaces dans un script. Chaque commande a des arguments et ceux-ci doivent être séparés par des espaces. De même, le caractère `[` (resp. `]`), indiquant un test doit être suivi (resp. précédé) d'une espace. En revanche, lors d'une affectation à une variable, le symbole = ne doit être ni précédé ni suivi par des espaces.

- **Conditions**

Une condition est encadrée par les caractères `[` et `]` (séparés de l'expression par des espaces). En voici quelques exemples :

– Conditions sur les fichiers

`-e nom`	vraie si `nom` est un fichier ou répertoire existant
`-f nom`	vraie si `nom` est un fichier ordinaire
`-d nom`	vraie si `nom` est un répertoire ordinaire
`-r nom`	vraie si `nom` est accessible en mode lecture
`-w nom`	vraie si `nom` est accessible en mode écriture

– Conditions sur les chaînes

`ch1`	vraie si la chaîne `ch1` n'est pas la chaîne nulle
`-z ch1`	vraie si la chaîne `ch1` est nulle
e `ch1 = ch2`	vraie si les deux chaînes sont égales
`ch1 != ch2`	vraie si les deux chaînes sont différentes

– Conditions sur les nombres

`nbr1 op nbr2`	compare les deux nombres indiqués

`op` peut prendre comme valeurs possibles :

`-eq`	*(equal)* égal
`-ne`	*(not equal)* différent
`-gt`	*(greater than)* strictement plus grand
`-ge`	*(greater or equal)* plus grand ou égal
`-le`	*(less or equal)* plus petit ou égal
`-lt`	*(less than)* strictement plus petit

– Combinaisons logiques

`! cond`	négation de la condition
`cond1 -a cond2`	*(and)* et logique entre deux conditions
`cond1 -o cond2`	*(or)* ou logique entre deux conditions
`(cond)`	regroupe une condition

Ainsi, la condition `[ -f $1 -o -d $1 ]` est vraie si l'argument est un fichier ou un répertoire ; la condition `[$# -ge 1 -a -f $1]` est vraie s'il y a au moins un argument et que le premier est un fichier.

- **Tests et boucles**

Un test simple ou une boucle s'écrit :

```
if [ expr ]
then
  commandes
fi
```

```
while [ expr ]
do
  commandes
done
```

```
for var in liste
do
  commandes
done
```

Ainsi, le script suivant teste si un fichier (dont le nom est passé en paramètre) existe et le déplace en le renommant avant de créer un fichier vide de même nom.

```
#!/bin/bash

if [ -e $1 ]
then
    mv $1 $1.backup
fi
touch $1
```

Et le script suivant ne fait rien d'autre que réafficher chacun des arguments passés à la commande.

```
#!/bin/bash

for arg in $*
do
  echo $arg
done
```

4.1 Script compteur

On a vu qu'avec `ls | wc -l`, on pouvait compter le nombre de fichiers dans le répertoire courant. On souhaite maintenant restreindre ce comptage aux fichiers ayant une extension donnée (par exemple `ls *.c | wc -l` va compter touts les fichiers ayant `.c` comme extension).

Exercice 12
Écrire un script qui prend un argument (l'extension) et renvoie le nombre de fichiers ayant cette extension.

4.2 Script Verlan

Les commandes `head` et `tail` extraient les premières et dernières lignes d'un fichier :
`head -n i fichier` extrait les *i* premières lignes du fichier (on écrit aussi `head -i`).
`tail -n i fichier` extrait les *i* dernières lignes du fichier (on écrit aussi `tail -i`).
`tail -n +i fichier` extrait les lignes du fichier en partant de la i^{e}.

Par exemple, `tail -n +2` extrait les lignes à partir de la deuxième, c'est-à-dire affiche le fichier sans la première ligne.
On peut les combiner pour extraire une ligne quelconque :
`head -n 5 fichier | tail -n 1` va isoler la cinquième ligne du fichier.

On peut alors définir une fonction qui a pour but de renvoyer la i^{e} ligne d'un fichier dont le nom est dans une variable globale :

```
lire_ligne() {
    ligne=`head -n $i $file | tail -n 1`
}
```

Exercice 13 *Écrire maintenant un script qui liste un fichier à l'envers en extrayant la dernière ligne, puis l'avant-dernière...*

Il faut faire attention que `wc` compte les lignes en comptant les *retours-chariot*. Si le fichier que l'on veut lister ne se termine pas par un retour-chariot, le script va compter une ligne de moins et ne va pas afficher la dernière ligne. On peut donc le modifier en incrémentant `i` de 1 avant la boucle.

4.3 Script liste

Exercice 14
Écrire un script qui prend un répertoire en argument et liste son contenu en indiquant le type pour chaque élément.

4.4 Script liste2

Exercice 15
Étendre le script précédent à un nombre quelconque d'arguments, chacun devant être listé. On pourra utiliser la variable `$*` *qui renvoie une liste formée de tous les arguments.*

4.5 Script décryptage MD5

Exercice 16 *Écrire un script qui compare la version cryptée d'une liste de mots, avec un mot de passe crypté grâce à la commande* `md5sum`*. Utilisez la liste mots disponible dans* `/usr/share/dict/words` *pour décoder le mot de passe suivant: 351a7b5994f2b820a91812722119afca*