CHAPITRE

5

Le système d'exploitation Unix/Linux

Les systèmes et réseaux informatiques sont gérés par un système d'exploitation (SE) qui cherche à coordonner les opérations d'exécution de logiciels et d'entrées/sorties, de sauvegarde, de communication, etc. Le SE Unix nous intéressera davantage : il a été rédigé en C, donc intègre parfaitement la fonctionalité de ce langage. Il définit par ailleurs l'environnement de travail de plusieurs utilisateurs travaillant en parallèle (simultanément).

Le but ici est de faire un survol de ce SE pour en saisir l'esprit et nous permettre de travailler efficacement en TP.

5.1 Caractéristiques principales d'Unix

1. configuration libre : l'utilisateur est libre d'adapter son environnement de travail à ses propres besoins.
2. multi-tâches, temps partagé entre utilisateurs
3. multi-utilisateurs : protocole de gestion de comptes individuels
4. mode interactif ou dormant
5. stable

6. versions pérennes (Linux, solaris, etc ..)

Unix permet à la fois une interactivité par interface graphique (surtout Xwindows), mais aussi un environnement de langage de commandes, écrites et lancées par l'utilisateur : il s'agit de travailler sous un logiciel dit 'interpréteur de commandes' ou *shell*.

Il existe différents shell, donc l'interprétation et la syntaxe des commandes varient en fonction du shell utilisé :

- Bourne shell : 'Bourne Again Shell' ou bash
- Korn Shell
- C-shell : logiciel csh
- TC-shell : logiciel tcsh

Les deux derniers shells incorporent plusieurs opérations de contrôle du C d'où il tirent leur nom. Par exemple, les structures *if ... else* . On peut ainsi (presque) composer un programme sans compilation : on parlera de *scripts du shell*.

5.2 Outils de travail : logiciels

Les principaux outils que nous utiliserons sous Unix seront par exemple

- Editeurs de texte : nedit, vi, emacs, sed, awk
- Compilateurs pour le développement de logiciels : gcc, g++, cc, autres ..
- Services réseaux : navigateurs web, courriel, transfert de fichiers ..
- Documentation : pour se dépanner, le manuel en ligne *man*

et bien d'autres.

5.3 Système de fichiers

> Définition : le fichier est un volume d'informations sauvegardé sur un médium non volatile, identifié par une structure (ou i-node) sur ce médium.

Sous Unix, tout est fichier, c'est-à-dire un volume de données. On reconnaitra quatre types de fichiers :

1. ordinaires : par exemple, un programme, des données numériques
2. répertoires : ils contiennent des fichiers ordinaires ou des (sous-)répertoires
3. lien symbolique : pointe vers un fichier sans le dupliquer
4. spéciaux : ils permettent l'accès à un périphérique (imprimante, modem, souris ..) ou à un réseau

Le i-node contient les informations essentielles d'un fichier : son nom, son volume, droits d'accès, date de création, propriétaire, etc.

Le nom d'un fichier $\longrightarrow$ pointe vers le i-node de ce fichier.

Les fichiers ordinaires : flot de caractères (codés 8 ou 16 bits). Les fichiers texte (ASCII) sont un cas particulier, dont le flot contient le caractère de formattage 'carriage return' (\r, pour 'retour de chariot', qui sépare les lignes de texte).

5.4 Fichiers et répertoires

L'utilisateur qui accède au système Unix fait partie d'une arborescence représentant tout le système. La position absolue d'un utilisateur dans cette arborescence se définie à partir du répertoire qu'il/elle occupe par défaut.

- Le **répertoire courant** marque la position actuelle, à partir de laquelle s'effectue toute opération de manipulation ou recherche de fichiers.
- le *chemin absolu* d'un fichier se lit à partir d'un point de référence universel du SE, soit la *racine* du système, symbolisée par "/".
- le caractère "/" sépare différents répertoires de l'arborescence, par exemple
  /home/boily/Cours/STUSM/fichier.tex
  indique la localisation exacte (universelle) de 'fichier.tex' sur l'ordinateur.
- le *chemin relatif* indique un chemin défini à partir de la position courante.
- Les fichiers : majuscules et minuscules sont distinguées
- pour les noms : tous les caractères sont permis sauf /, &
- par de contrainte pratique sur la longueur permise des noms de fichiers..

5.5 Les utilisateurs

Chaque utilisateur du SE possède

1. Un nom et un mot de passe (code secret) : gestion des *login*
2. un identifiant numérique (uid) et groupe (gid)
3. un répertoire de travail
4. un shell : langage de commandes, e.g. tcsh

Toutes ces informations : stockées dans la base de données du logiciel *passwd* (qui permet à l'utilisateur de changer son mot de passe).

La notion de groupe : permet à l'utilisateur de partager certains fichiers avec d'autres membres du même groupe.

Le super utilisateur *root* gère tout le système, a des droits d'accès interdits aux utilisateurs normaux. C'est l'ingénieur de tout le système informatique.

5.6 Les droits d'accès

Il y a trois catégories d'utilisateurs :

le propriétaire	u
le groupe	g
les autres	o
tous	a

et trois types de droits d'accès

lecture	r
écriture	w
exécution	x

De plus il faut reconnaitre le type du fichier : ordinaire (-) , répertoire (d), lien symbolique (l) ou spécial (c ou b). Donc : il faut au total $3 \times 3 + 1 = 10$ caractères pour identifier tous les droits d'accès d'un fichier.

On représentera ces droits par la chaîne

tuuugggooo

où tour à tour apparaissent le type (t), les droits de propriétaire (u), etc ..

5.7 Visualiser l'info sur les fichiers

La commande Unix `ls` permet d'obtenir l'information de base sur un ou des fichiers.

Par exemple : ls -l [chemin] [nom/s de fichier/s]

est la commande Unix permettant d'obtenir à l'écran des informations tel

```
-rw-rw---- 1 boily stusm 18 Nov 15 12 :30 toto.tex
```

à propos du fichier toto.tex. Notons que '-l' est une **option** de la commande unix ls. Ici le résultat indique que : toto.tex est un fichier ordinaire (-) ; son propriétaire est boily ; l'utilisateur (auquel appartient le shell, pas le fichier) a les droits de lire (r) et écrire (w) ; et enfin, que les membres du groupe stusm ont ces droits également. Tout autre utilisateur n'a aucun droit d'accès au fichier toto.tex (représenté par les trois dernier traits - - -). La date de création est également indiquée de même que l'heure.

5.8 Changer les droits d'accès

Pour changer les droits d'accès à un fichier ou répertoire, il faut évoquer la commande Unix *chmod*. La syntaxe de cette commande est :

chmod u/g/o/a +/- r/w/x *nom de fichier*

où le séparateur '/' indique une alternative : le premier groupe indique le choix d'utilisateur visé, le deuxième ajoute (+) ou soustrait (-) le ou les droits r,w, ou x au fichier nommé.

Par exemple la commande

```
chmod a-rwx toto.*
```

permettrait d'un seul coup de soustraire (-) les droits de lecture, écriture et exécution à tous les utilisateurs (a = u,g et o ensembles) pour tout fichier dont le nom débute par la chaîne 'toto'. Le caractère spécial '$*$' est un caractère **d'échappement** : il remplace toutes les combinaisons possibles de caractères et symboles, c'est-à-dire qu'il complète la suite toto ... par tout symbole ascii permis.

5.9 Les commandes les plus courantes

Sous Unix, les commandes acceptent la syntaxe générale

nom_commande [-option1] [-option2] ... [variables] [nom_fichier ou répertoire]

et nous en avons vu des exemples avec chmod et ls d'options et de variables passées en arguments à la commande. Les crochets [] indiquent des arguments qui peuvent être facultatifs à l'exécution de la commande. Deux types de commandes vont nous intéresser davantage : celles se rapportant à la gestion des fichiers ; et celles relatives à la gestion de notre compte.

1. Les fichiers : les créer, les nommer, etc ....

   On connait déjà les 2 commandes les plus importantes se rapportant aux fichiers : soit *ls* et *chmod*. Pour déplacer un fichier n'importe où sur le système, on peut utiliser *mv* (pour *move*) comme ceci

   mv [-i] nom_fichier [chemin_absolu]nom_fichier2

   Ici le 'fichier' peut être un répertoire entier ; l'option -i force une interrogation si nom_fichier2 existe déjà (pratique pour éviter les pertes de données). On utilise mv pour renommer un fichier, par exemple.

   Pour faire une copie conforme d'un fichier, il suffit d'évoquer *cp* comme ceci

   cp [-i] nom_fichier1 [chemin_absolu ou relatif]nom_fichier2

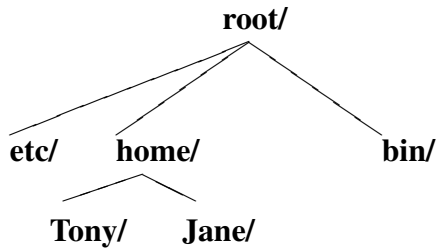


FIG. 5.1 – Diagramme montrant une partie de l'arborescence d'un système Unix.

permet une copie conforme dans le répertorie courant (si le chemin est omis), sinon dans le répertoire indiqué par le chemin, au nom choisi (nom_fichier2).

2. Les répertoires

   Pour se faire une idée de la disposition des fichiers, il est utile de consulter le diagramme de la Fig. 5.1. Ici **Tony** et **Jane** représenteront deux utlisateurs qui ont des répertoires spécifiques dans **l'arborescence** du système représentée par le diagramme. Ainsi après s'être *loggé* (connecté), Tony se retrouve au point

   /home/Tony

   du système, c'est-à-dire dans un répertoire appelé Tony, lui-même sous-répertoire du répertoire home, à son tour sous-répertoire de la racine (root), soit '/'. Si Tony veut connaitre son répertoire actuel, il lance la commande

   `pwd`

   qui lui affichera le chemin absolu où il se trouve. S'il désire changer de répertoire, par exemple pour se retrouver sous /bin, il fait

   `cd /bin`

   et s'il désire revenir à son répertoire d'origine, il lance

   `cd ~`

   ou encore

   `cd /home/Tony`

   qui donne le même résultat : le symbole '~' représente le chemin absolu de l'espace de travail de l'utilisateur : on réfère souvent à cet espace somme le répertoire *home* de l'utilisateur.

   Autres arguments utiles de cd :

   – on retourne au répertoire précédent avec `cd -` (cd suivit de 'moins').
   – on remonte d'une unité relative dans l'arborescence avec `cd ..` ; de deux unités avec `cd ../../` ; etc ..

3. Recherche et comparaison de fichiers

   Si on désire comparer le contenu de deux fichiers (par exemple pour vérifier

les différences apportées lors d'une édition), on peut utiliser la commande `diff` ainsi :

diff [-n] fichier1 fichier2

ce qui aura pour résultat d'afficher à l'écran les lignes de chaque fichier si elles diffèrent entre elles. Si l'option '-n' est donnée, alors le numéro de ces lignes apparait en première colonne.

On peut recherche une variable, voire une phrase, dans un ou plusieurs fichiers, à l'aide de la commande `grep`. Ainsi

grep [-i] nom toto.tex chap1.tex

aurait pour effet de 'fouiller' chacun des fichiers nommés et imprimer à l'écran chaque occurrence du mot *nom*. Avec l'option '-i' : ignore les majuscules.

4. Contenu d'un fichier : commandes `more` et `less`, renvoient à l'écran le contenu d'un fichier, une 'page' à la fois. En pressant la barre du clavier on passe à la page suivante.
5. Tout le fichier : on peut envoyer à l'écran un fichier entier avec la commande `cat`. Ceci est utile avec de petits fichiers, mais pas pratique sinon.

Finalement, on obtiendra une liste de toutes les options ainsi que la syntaxe en consultant le mode d'emploi d'Unix au moyen de la commande `man` :

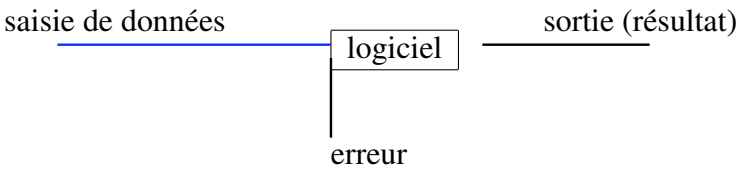
man [-k] nom_commande ou thème

placera l'utilisateur à la page pertinente du manuel où la définition et l'utlisation de la commande nom_commande apparait. L'option '-k' permet de faire la recherche d'un thème dans le manuel d'Unix.

5.10 Enchaînement des commandes

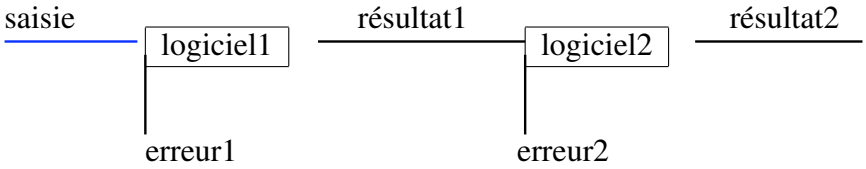
Une propriété intéressante d'Unix est la gestion de flot de données par l'entremise de fichiers **standards**, définie pour tout logiciel - y compris ceux que nous développerons. Il s'agit de fichiers standard d'entrée (*stdin*), de sortie (*stdout*) et d'erreur (*stderr*).

On peut se représenter l'exécution d'un logiciel sous Unix comme un flot de données comme ceci (de gauche à droite)



La saisie de données est fonction du logiciel : par exemple il peut être nécessaire d'initialiser une variable du programme. La sortie est le résultat attendu, il est envoyé à l'écran par défaut.

Il est clair que les commandes Unix suivent ce schéma : nous avons vu comment leur passer des paramètres en entrée, voire des options, donc il y a un flot de données saisies. Il est alors possible d'imaginer une **chaîne** de logiciels (ou commandes Unix) s'emboitant l'une l'autre comme ceci :



En procédant ainsi avec 2,3,4 .. logiciels, on construira une chaîne de plus en plus longue, où le résultat d'une commande devient le paramètre saisie par la commande suivante, et ainsi de suite. Sous Unix cette opération de briquetage se fait avec l'opérateur '|'.

Exemple : combiner les commande cat et grep. La commande cat demande un nom de fichier (saisie), puis renvoie tout le fichier à l'écran en résultat. La commande grep cherche une chaine de caractères dans un flot de texte. Une manière équivalente de faire cette recherche serait d'écrire

```
cat fichier | grep -i prof
```

On imagine alors le 'flot' de texte, résultat de `cat fichier`, être bifurqué vers l'entrée standard de `grep`, laquelle recherche, dans ce flot, le mot 'prof'. Si ce mot est trouvé, alors comme résultat la ligne où il apparait sera reproduite à l'écran (résultat2 de notre schéma ci-haut).

5.11 Les scriptes shell

Les commandes Unix peuvent se lancer l'une après l'autre. Dans la pratique, une chaîne de travail peut amener plusieurs tâches différentes, chacune exécutée

par un logiciel différent. C'est souvent le cas lorsque nous avons à manipuler des fichiers (par exemples, compter le nombre de ligne de tous les fichiers textes d'un répertoire, sélectionner les 10 plus petits, ensuite concaténer les cinq premiers, etc ..). Les commandes Unix offrent une palette intéressante de logiciels qui permettent des manipulations simples ; les **scriptes shell** vont permettre d'agencer ces appels de commandes et nous sauver du temps. On peut penser aux scriptes shell comme des programmes en toute lettre, puisqu'il exécutent un algorithme.

Ce qui suit est une présentation sommaire de quelques scriptes shell, l'objectif est de permettre à ceux que cela peut intéresser de construire leurs propres scriptes ; une recherche sur internet permettra rapidement de parfaire ses connaissances en la matière. Les quelques ouvrages cités dans la bibliographie touchent tous aux scriptes shell et vont beaucoup plus loin.

Pour la suite, le shell de travail est présumé être CSH ou encore TCSH. (Voir à ce propos le commentaire à la fin de cette section.)

5.11.1 Set, echo, cat

Avant de débuter il est bon de se rappeler que le shell admet des définitions de variables avec la commande Unix SET. Par exemple

```
commande unix> set toto = 1
```

définit une variable 'toto' reconnue par le shell de travail :

```
commande unix> echo $toto
1
commande unix>
```

La commande ECHO permet de lire le contenu d'une variable. Notons que les valeurs numériques peuvent être quelconques (des entiers ou des flottants), toutefois le C-shell n'admet que l'arithmétique des entiers. Ainsi

```
commande unix> set toto = 1
commande unix> @ toto = ($toto + 1 )
commande unix> echo $toto
2
commande unix>
```

va permettre de créer un compteur ou, plus généralement, d'effectuer des opérations arithmétiques simples. L'affectation d'une valeur à une variable shell ne se limite pas aux nombres simples. Par exemple la série de commandes suivantes crée un tableau de valeurs.

```
commande unix> set toto = `wc *.csh`
commande unix> echo $toto
34 145 765 reshuffle.csh
commande unix>   echo $toto[1]
34
commande unix>
```

Ici 'toto' est affecté du résultat de la commande WC *.CSH, qui fait le décompte de tous les fichiers d'extension .csh. (Il n'y en a qu'un dans cet exemple.) Le résultat est quatre valeurs (nombre de lignes, mots et caractères dans le fichier, puis son nom). Chaque valeur est lu de toto comme pour la lecture d'un tableau dans un programme C. Toutefois, les indices courent de 1 à N pour une variable de N dimensions.

Il est possible d'affecter l'ensemble d'un fichier (son contenu en entier) à une variable comme 'toto' : dans l'exemple précédant, il suffirait de remplacer la commande WC par CAT. La variable 'toto' contient maintenant tous les mots du fichier reshuffle.csh.

5.11.2 Branchement conditionnel : foreach, while

Comme pour le langage C, les scriptes C-shell admettent des constructions de branchement avec conditions. Étudions l'exemple suivant.

```
commande unix> csh
commande unix> set toto = `cat *.csh`
commande unix> set N = 0
commande unix> foreach t ($toto)
@ N = ( $N + 1 )
   if( $N > 100 ) then
      echo Grand fichier ..
      exit
   else
   endif
end
commande unix>   Grand fichier ..
commande unix>
```

Dans cet exemple nous avons utilisé la construction FOREACH … END. Ici nous avons créer un compteur, N, et nous cherchons à déterminer si les fichiers .csh (tous ensembles) contiennent plus de cent mots. (Il y aurait une façon plus simple de faire avec WC.) La déclaration de 'toto' contient tous les mots, séparés d'un

espace blanc. L'argument de la boucle FOREACH, 't', accepte tour à tour les mots contenu dans 'toto'. La construction IF .. ELSE .. ENDIF ressemble beacoup à celle du langage C, sans lui être identique. Ainsi, l'utilisation des points virgules n'est maintenant plus nécessaire (il n'est pas faux de les utiliser, par contre).

Remarque : la commande EXIT agit comme en C. Elle va causer la fin du processus shell. C'est pour cela que nous avons d'abord lancer un cshell (commande CSH) avant l'exécution de la boucle, par anticipation. En créant puis en arrêtant un processus csh, nous revenons au shell de départ (qui pouvait être un csh, tcsh ou autre).

5.11.3 Exemple d'un C-shell

Jusqu'à présent, nous avons travaillé à la ligne de commandes. Il est bien entendu que le but des scriptes est de pouvoir répéter une chaîne de commandes à souhait.

Pour cela, il suffit éditer un fichier de texte et d'y recopier notre scripte shell. Ensuite, il faudra donner les droits d'exécution (avec CHMOD) pour pouvoir lancer le scripte, qui agit alors comme une nouvelle commande Unix.

Pour isoler le scripte, on cherchera à lui accorder son propre shell de travail. La première ligne du texte suivant fait exactement cela ; donc lorsque la commande EXIT est exécutée, le scripte quitte le shell (ici, un csh) et nous libère la ligne de commandes.

```
#!/bin/csh -f

# Les commentaires sont maintenant precedes
# du diese.

# Creer la liste de tous les fichiers .f :
set liste = `ls *.f`

# Pour chaque fichier, changer la fin .f -> .F :
foreach i ($liste)

# En utilisant le point comme separateur, imprimer
# le champ 1 (qui le precede); puis ajouter .F :
   set fic=`echo $i | cut -d. -f1`.F
   echo $i $fic

# Deplacer le fichier vers le nouveau nom ($fic) cree :
   mv $i $fic
```

```
# compiler le nouveau fichier :

g77 $fic

# lancer l'executable, etc etc ..

end

exit
```

Le scripte C-shell ci-dessus permet de faire une gestion efficace de fichiers et de les manipuler pour ensuite lancer un calcul (s'il s'agit de fichiers sources, de programmation).

Les scriptes shell permettent beaucoup de flexibilité, toutefois chaque type de shell aura une syntaxe qui lui sera particulière. Ainsi les instructions de branchement sous un C-shell n'acceptent pas les mêmes instructions que sous un shell Bourne (bash), ou encore zsh.

Lorsque les scriptes sont courts, il peut être très avantageux de les adapter à ses besoins et d'apprendre à en composer un minimum.

Toutefois, dans les cas où les scriptes demandent des opérations complexes, il est préférable de se tourner vers des langages de programmation interprétés, comme PERL, ou encore PYTHON. En particulier, PERL est un langage intermédiaire entre les commandes Unix et la programmation proprement dite ; pour cette raison c'est un des langages de choix pour la mise en place de chaîne d'analyse et de manipulation de données (post-processing).